Incremental development of an HDLC protocol in Esterel

Gérard Berry, Georges Gonthier

INRIA

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 1031

Programme 1

INCREMENTAL DEVELOPMENT OF AN HDLC PROTOCOL IN ESTEREL

Gérard BERRY
Georges GONTHIER

Mai 1989

RR_1031

Développement Incrémental d'un Protocole HDLC en ESTEREL

Incremental Development of an HDLC Protocol in ESTEREL

Gérard Berry
Ecole des Mines de Paris
Centre de Mathématiques Appliquées
06565 Valbonne, France
INRIA
06565 Valbonne, France

Georges Gonthier
AT&T Bell Laboratories
Murray Hill, New Jersey 07974, USA

RESUME

Esterel est un langage de programmation parallèle fondé sur un mécanisme de paralléllisme et de communication synchrone. La gestion du parallélisme et de la synchronisation est complètement résolue par le compilateur Esterel, qui produit un code séquentiel ne contenant plus que les opérations de manipulation des données qui doivent se faire à l'exécution. On peut ainsi librement découper une tâche en sous-tâches parallèles et utiliser des signaux pour synchroniser ces sous tâches, sans que cela n'induise de pénalité à l'exécution: sous-tâches et signaux disparaissent après compilation.

Ce papier est un exemple avancé de programmation Esterel, prenant comme exemple un protocole HDLC. Nous développons le programme de façon modulaire et en plusieurs étapes. Nous partons d'une spécification réduite du protocole, en insistant sur la façon de concevoir l'*architecture* d'une application de ce type. Puis nous ajoutons graduellement de nouveaux éléments de spécification juqu'à atteindre le protocole complet. Dans cette seconde phase, nous insistons sur l'aspect *maintenance* de programme Esterel.

Mots–clef: *Parallélisme, Langages Synchrones, Protocoles, Automates*

ABSTRACT

Esterel is a parallel programming language based on a model of synchronous parallelism and communication. All the parallelism, process handling and synchronization are performed by the Esterel compiler, which produces efficient sequential code containing only the data handling required at run-time. Separating a task into parallel components for better modularity, adding signals for better synchronization incur no run-time overhead as they are compiled away. This paper is an advanced tutorial on the new programming style that is made possible by these features. Our working example is an HDLC protocol driver, for which we will develop a modular program in gradual steps. We start from a simple, restricted case, and then add complexity and detail while retaining most of the initial code. The emphasis is first on *design* as we develop a program architecture that captures the basic structure of the driver, and the shifts to *maintenance* as we add new features into the initial architecture.

Keywords: *Concurrent Programming, Synchronous Languages, Protocols, Automata*

Incremental Development of an HDLC Protocol in ESTEREL

Gérard Berry

CMA-ENSMP and INRIA
Sophia-Antipolis, 06565 Valbonne – France

Georges Gonthier

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

The Esterel language was designed [1] for writing *reactive* programs, i.e., communicating programs whose correct operation depend on the ordering of inputs and outputs events. As an example, real-time controllers are reactive, while computer operating systems that start and stop tasks asynchronously are not. Communication protocol drivers, which rely on precise exchanges of messages and acknowledgements, are also a good example of reactive systems. The object of this paper is to show how the special features of Esterel can be used to derive a much cleaner and modular description of an HDLC protocol.

The main Esterel feature is *synchrony*: Esterel programs communicate *as if* their basic control and data operations took *no time*. As we will demonstrate in this paper, this paradigm allows one to program reactive applications very cleanly and naturally. Aside from synchrony, Esterel has fairly common imperative primitives, with a PASCAL-like syntax, and broadcast communication primitives. We shall not give a precise description of Esterel here, as this paper is meant as programming style and not a language tutorial. Instead we refer the reader to [2] for a simple introduction, [1] for a complete presentation of the language's semantics, [3] for reference manuals, [4] for some simple commented program examples, and [5] for a detailed study of the semantics and the implementation of the language.

This paper is a tutorial on advanced Esterel programming techniques. The example we will be elaborating is a protocol driver for a set of HDLC procedures (the BAC 2,8 procedures). Our goal here is to develop a parallel, modular, and incremental program for this driver. We will base our modular style of programming on the synchronous parallelism and communication provided by Esterel. Modular programming does not only mean carving up a large program into small pieces. Its main purpose must be to uncover a simple and robust *architecture* whose basic building blocks must be individually meaningful, have clearly identified functions, and be easily adaptable and reusable. Meeting this objective is the only way to ensure that the programming of an application is *flexible* enough that local changes in the specification only induce local changes in the architecture and the code. Following the precepts of Object Oriented Programming, we will stick to the rule *one function — one module*. However, the synchronous parallelism will make it considerably easier for us to obey this rule than any of the asynchronous primitives in classical Object Oriented Languages.

An essential quality of well-written programs is *the absence of code duplication*. This should be a part of any good programming style, since code duplication multiplies the risk of errors, especially during maintenance. Here, this property will follow from the one function — one module rule. Contrast this with hand-coded automata, where duplication of actions on several transitions is the rule. The Esterel compiler automatically transforms our parallel programs into sequential automata, and thus realizes the needed duplications with no risk of error.

We shall illustrate these points by programming the protocol incrementally, going through five increasingly complex versions:

- **Version 1:** Transmission of numbered messages on a perfect "instantaneous" line for which

frame emission is instantaneous (end-to-end transmission can take time, though). The numbering of received messages is checked, and there is input flow control, but no error recovery is attempted.

- **Version 2:** We handle a non-instantaneous line, where frame emission by the line handler takes time; thus, we need to buffer input.
- **Version 3:** We handle the reception of checkpointing request frames (*P*-bit set). The responses to these frames must have priority over normal information frames.
- **Version 4:** We handle the reception of *REJ*ect control frames from the line. This entails retransmission of buffered input. We also add in transmission of **REJ** frames as a response to out-of-sequence messages.
- **Version 5:** Finally we add a timeout error recovery mechanism that resynchronizes both ends with checkpointing frames.

In the first two simple versions we set up the basic structure of the HDLC driver program; the main concern there is *architecture*. The next three versions build more complexity into the structures of version 2. Thus they mostly deal with code *maintenance*, extending previous code with as few modifications as possible.

Each step in this progression will be relatively easy, and illustrate the use of new Esterel constructions. The whole paper gives a rather complete tour of the Esterel programming style features. We will also briefly review the code generated by the Esterel v3 compiler.

2. A Short Description of the HDLC Protocol

HDLC is a complex full-fledge protocol that specifies the data exchanges right down to the bit encoding schemes. We will not go into such detail, concentrating on th logical aspects instead. A complete description of the HDLC protocol can be found in [6], for example. Also, we will only implement the procedures for asynchronous balanced (ABM) data transfer, thus putting aside the connection/disconnection procedures and the other transfer modes. PP In the set-up we consider, the protocol is to ensure reliable full-duplex transmission of data between two users *A* and *B* through an established, but unreliable full-duplex line. This is done by inserting a protocol *driver* between each end-user and the line, as shown in figure 1. The two drivers will exchange control information along with user messages, so that reliable transmission is achieved. Although the line and users are asynchronous the protocol drivers are reactive, and it is *one* of these drivers that we shall be programming.

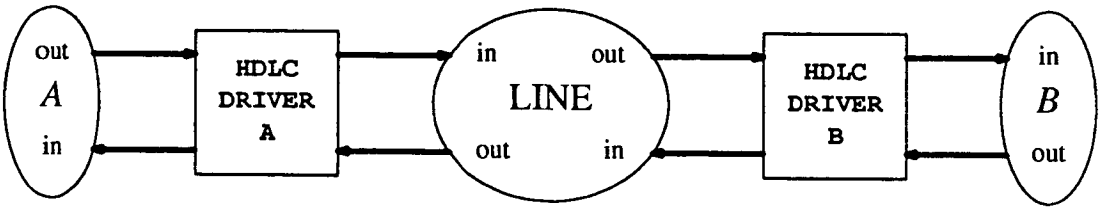

Figure 1: Protocol Set-Up

HDLC is a typical sliding-window protocol. Let us describe how it operates in a transmission from *A* to *B*:

- The **A** driver attributes a sequence number $N(S)$ to each message that user *A* hands it over, packages each message and its $N(S)$ in a data *frame* (an *I*nformation-frame), and transmits that to **B**.
- The **B** driver accepts *I*-frames from **A** in ascending $N(S)$ order, and forwards their data contents to user *B*.
- The **B** driver also periodically echoes back to **A** the *next* message number it is expecting, $N(R)$. Thus when **A** receives an $N(R)$ value, it knows that *all* messages with an $N(S) \leq N(R)$ have been received correctly by *B* (they are *acknowledged*).
- To cater to finite frame formats, message numbers are actually counted mod 8 (or 128 in some variants). To avoid number confusion, **A** keeps all pending message numbers within an *emission*

window whose length is less than the modulus. Thus when there are too many unacknowledged messages the **A** driver performs *flow control* to stop input from user *A*. When messages finally are acknowledged, the window can be moved and new input will be accepted.

Transmission from *B* to *A* works exactly the same, and both will be running concurrently. There are two ways **B** can send *N* (*R*) "acknowledgements" to **A**:

- either in an *RR*-type *S*-frame (*R*eceive *R*eady *S*upervisory frame),
- or as a *piggyback*, as part of an *I*-frame that it is sending to **A**.

Thus in normal full-duplex operation there is very little need to exchange *S*-frames for acknowledgements. Their use is mostly confined to error recovery:

- A *REJ*-type (*REJ*ect) frame from **B** indicates that **B** has received an out-of-sequence frame from **A**, so that **A** should retransmit all unacknowledged frames since a frame has been lost (this is called a "go-back-to-*N*" discipline).
- A *command* frame from **B** (see below) with the *P* (*P*olling) bit set is an enquiry into **A**'s reception status. It requires a preemptive response by **A**: a *response* frame with the *F* (*F*inal) bit set. This *checkpointing* mechanism is typically used after a timeout or abnormal event.

There are other HDLC features and related frame types that we will not consider here: connection/disconnection procedures and unnumbered frames, output flow control and *RNR* (*N*ot *R*eady) frames, *SREJ* (*S*elective *REJ*ect) frames. All could be added to our basic design, but the additional complexity would get beyond the scope of this paper.

The different frame types we will be using are summed up in figure 2. All frames bear an *address*. If it is the address of the recipient, the frame is a *command*, otherwise it is the address of the emitter, and the frame is a *response* (Figure 2 shows frames sent by **A**). Only commands have a *P* bit, and only responses have an *F* bit. All frames carry an *N* (*R*) value and a *P* or *F* bit. *I*-frames carry a message and *N* (*S*) value, and *S*-frames carry only their type (*RR* or *REJ*, in our case). There are additional restrictions on command and responses: *REJ* frames can only be responses, and *I* frames can only be commands (this is option 8 of the BAC 2,8 procedures we are coding; option 2 is the use of *REJ* frames).

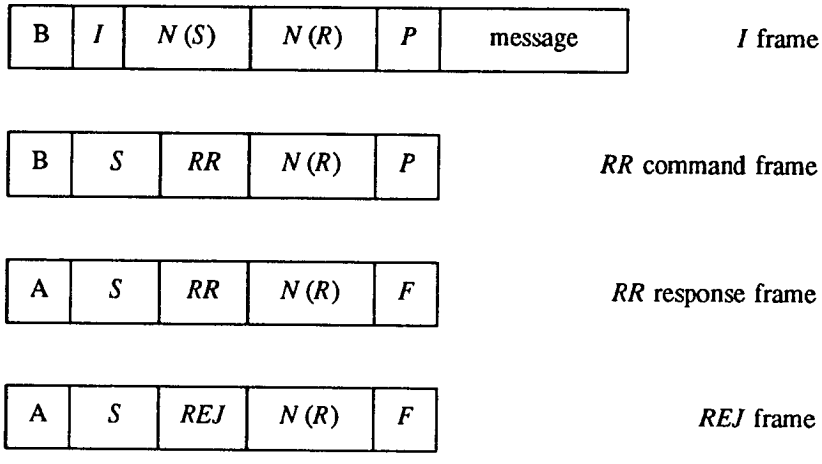

Figure 2: HDLC Frame Types

3. The Main Program Interface

The interface for the main program will be the same for all five versions, although some parts of it will only be used in some of the later versions: e.g., the time signal **MS** (millisecond) is only used in the last version. We will describe this interface in four parts: the end user and time interface, the transmission line interface, the input signal relations, and the data interface for the manipulation of

abstract data types. The latter is not really part of the main module text, but is a summary of all the operations that will be needed throughout the whole program.

The signal interface is illustrated by figure 3.

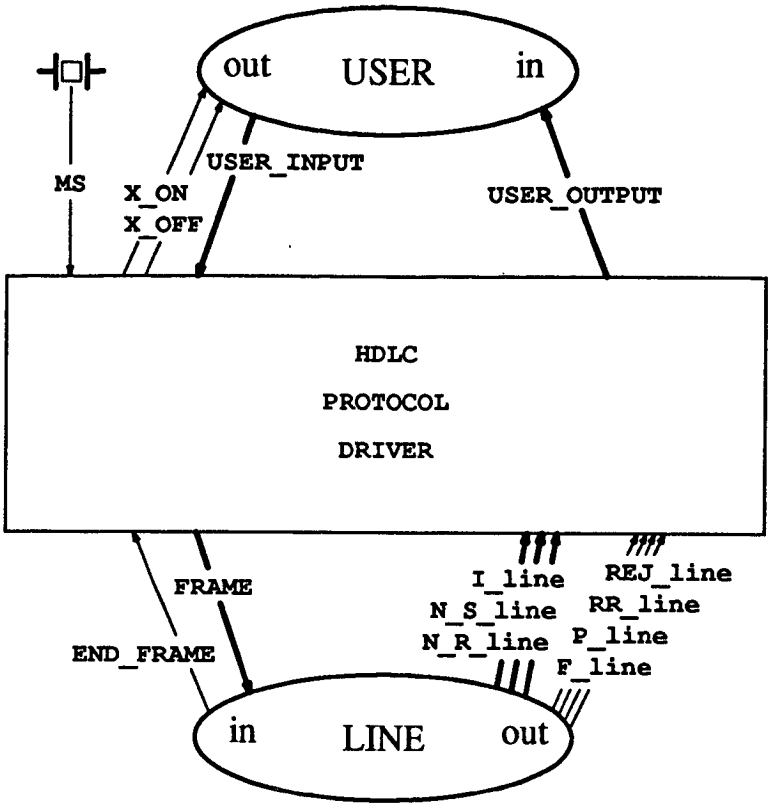

Figure 3: The HDLC Driver Interface

3.1. The End User and Time Interface

There are two kinds of information exchanged between the user and the program: data messages, both input and output, and input control flow information. The contents of messages is irrelevant here, so we just encapsulate them in an abstract data type `MESSAGE`, whose implementation is relegated to the *host language* in which the automaton produced by Esterel will be implemented (e.g., C).

Since we will be concerned solely by the driver program, we discuss user input and output from the protocol driver's point of view; thus the user *provides* input to the driver through a `USER_INPUT` signal. This input is subject to flow control. The program will emit `X_ON` and `X_OFF` signals to prompt and stop input according to its emission window.

The driver transmits messages to the user through the `USER_OUTPUT` signal. We do not consider output control flow (it could be added as an extension). Finally, we use an absolute time `MS` (millisecond) signal for timeout recovery.

Summing up, we have the following user and time interface

```
type MESSAGE;
input USER_INPUT (MESSAGE);
output X_ON, X_OFF;
output USER_OUTPUT (MESSAGE);
input MS;
```

3.2. The Line Interface

The line interface of the program handles frame emission and reception. We do not care for the actual bit structure of the frames, so we encapsulate them in an abstract data type **FRAME** whose constructor functions will be given later in 3.4.

3.2.1. The Output Line Interface

The frames are output on the line through a signal **FRAME**.* Except in the simplified first version, this signal will have to be interpreted by the host system who will then initiate actual line emission. During the emission the program will be able to react to other events, e.g., user inputs or incoming frames. When the actual emission is completed and the line is free for another transmission, the system sends back an **END_FRAME** signal to the program. This gives the following interface:

```
output FRAME (FRAME);
input END_FRAME;
```

3.2.2. The Input Line Interface

We could similarly reduce the input line interface to a single input signal of type **FRAME**. However, since we are especially concerned with the *control information* associated with the frames, it will be easier to suppose that these informations have already been extracted from the frame as separate signals. Thus the reception of an $N(R)$ counter value or a P bit can be handled uniformly, regardless of the type of the message carrying it. This frame split-up could be carried out in Esterel, but it is better handled in the host language (e.g., see the C Language Interface Manual in [3]). We introduce an abstract data type **NUMBER** for the message sequence numbers (integers mod 8, 128, or whatever). The input signal names are based on the usual HDLC notations, and suffixed by **_line** to flag their external origin (some internal counters will bear the same name):

- **I_line**: indicates *I*-frames, and carries a **MESSAGE** value.
- **RR_line** (resp **REJ_line**): indicates an *RR*-type (resp. *REJ*-type) *S*-frame.
- **N_S_line** (resp. **N_R_line**): carries the message sequence numbers $N(S)$ (resp. the expected sequence number $N(R)$), of type **NUMBER**.
- **P_line** (resp. **F_line**): indicates a frame with the P (resp. F) control bit set.

This gives the following declarations

```
input I_line (MESSAGE), N_S_line (NUMBER),
      N_R_line (NUMBER),
      P_line, F_line,
      RR_line, REJ_line;
```

Those signals are not temporally independent. Esterel allows us to express very nicely the correlations between those signals through *synchrony relations*. We have the following basic relations:

- messages and message numbers are always simultaneous
- the three frame types are incompatible
- the two checkpointing bits are incompatible
- all frames carry acknowledges $N(R)$

* In Esterel, variables, signals, and types have separated name spaces, so it is possible to give the same name to several objects in different classes.

This gives us the following relations

```
relation I_line => N_S_line, N_S_line => I_line,
         I_line # RR_line # REJ_line,
         P_line # F_line,
         I_line => N_R_line, RR_line => N_R_line, REJ_line => N_R_line,
         P_line => N_R_line, F_line => N_R_line;
```

In addition, *REJ*-frames can only be responses and so are incompatible with *P* bits. Also, option 8 says that *I*-frames can only be commands. We thus have the two additional relations:

```
relation REJ_line # P_line, % a REJ is always a response
         I_line # F_line;   % option 8
```

Note how relations are a powerful way to express dependencies between signals. The only correlation missing here is that any frame has to be either an *I*-frame, an *RR*-frame, or an *REJ*-frame. Since the `RR_line` signal is never actually used by the program (it is included here for completeness), this omission does not affect the code generated by the compiler!

3.3. The Global Serializing Relation

In addition to the line interface described above, we shall be assuming a global *serializing* relation between the different types of input to the program. This type of relation is commonly used in Esterel to reduce the size of the generated automaton code; it is automatically verified when the underlying operating system relies itself on serialization. Since all line input frames carry an $N(R)$ acknowledge, the following relation expresses the fact that time, user and line input, and line feedback are serialized:

```
relation USER_INPUT # MS # N_R_line # END_FRAME;
```

3.4. The Data Interface

As it is relatively short and standard, we shall describe here the complete abstract data type interface used by the five versions of the protocol, rather than describing pieces of it as they are needed.

For the message number type `NUMBER`, which will be represented by integers mod some k, we will need the following operations:

- constant `ZERO : NUMBER`. The initial number.
- procedure `INCREMENT (NUMBER) ()`. Increments its argument mod k.
- function `SPAN (NUMBER, NUMBER) : integer`. Computes the distance between its arguments, on the ring of integers mod k. `SPAN`(a, b) computes $(b-a) \bmod k$, so it is always a non-negative integer less than k.
- procedure `GUARD_UNDERFLOW (NUMBER) (NUMBER, NUMBER)`. A call `GUARD_UNDERFLOW` (X) (a, b) checks whether the value of variable X is between a and b ($X \in [a,b[\bmod k$), and resets it to a if it is not. This procedure is used to prevent underflow during retransmission, hence its name.

We also use equality tests on `NUMBER`s.

We just transmit `MESSAGE`s without care for their contents, so there are no special operations for this data type. Starting with version 2, we will need to buffer messages, so we will introduce a type `BUFFER`, which would probably be represented by an array with start and end indexes. We will need the following constants and functions:

- constant `WINDOW_SIZE : integer`. The transmission window size, and also the maximum buffer size.
- function `EMPTY_BUFFER (integer) : BUFFER`. Returns an empty buffer, given a maximal size.
- procedure `ENQUEUE (BUFFER) (MESSAGE)`. Add a message in the next available slot of the

buffer.

- procedure **PURGE (BUFFER) (integer)**. A call **PURGE (*B*) (*n*)** removes older messages from buffer *B*, leaving only *n* messages in *B* (so $n=0$ completely clears *B*).
- function **ACCESS (BUFFER, integer) : MESSAGE**. The call **ACCESS (*B*, *n*)** returns the n^{th} most recent message in *B* (so $n=1$ returns the last message entered in *B*).

To handle *S*-frame creation, we will need an auxiliary type **S_FRAME_TYPE**, with constants **RR_COMMAND**, **RR_RESPONSE**, and **REJ**. Frames are created using two functions

- function **I_FRAME (NUMBER, NUMBER, boolean, MESSAGE) : FRAME**. The arguments are respectively the message number $N(S)$, the expected number $N(R)$, the *P* bit, and the message.
- function **S_FRAME (S_FRAME_TYPE, NUMBER, boolean) : FRAME**. The arguments are respectively the supervisory frame type the acknowledge number $N(R)$, and the *P*/*F* bit. In the case of an *RR* frame, the frame type also determines the frame's address, selecting between command and response.

Finally, we will need two integer timing constants **ACKNOWLEDGE_TIME_LIMIT** and **RESYNCH_TIME_LIMIT** in the final version of the program, for timeouts (they are numbers of milliseconds).

4. HDLC Version 1 : Instantaneous Line with No Error Recovery

In this first version we shall be assuming that messages can be output instantly to the transmitting line and that no error recovery needs to be performed by the protocol program. This first simplified problem will allow us to lay out our architecture and code our first general-purpose modules. It is obviously very crude, since it is incapable of retransmission; also, it depends exclusively on available input messages for sending acknowledgements.

This version illustrates the use of valued signals to ensure consistent broadcasting of shared data, and the simplicity afforded by the Esterel synchronous communication model.

4.1. The Protocol Counters

Although this version will not include any error recovery, it will perform the basic message numbering and checking implied by the window protocol. Informal descriptions of the protocol usually mention only two counters, the $N(S)$ emission counter and the $N(R)$ reception counter, but to manage the protocol window we also need to keep track of the acknowledged messages and the user input numbering. Thus we need four counters to completely describe a protocol driver state:

- **N_S**: the sequence number of the *next* message to be sent out.
- **N_R**: the sequence number of the *next* message the driver expects to receive from the line.
- **N_A**: the greatest $N(R)$ value received as yet, that is, the sequence number of the *next* message the *distant* driver expects, as far as this driver knows.
- **N_U**: the sequence number to be attributed to the *next* user-input message.

Since all of these counters hold future values (they always represent the *next* value of something), we can initialize them with the same constant **ZERO** when there are no messages in the system.

Although **N_S** can be kept local to the *I*-frame generation, the values of the other counters are used throughout the program: **N_A** and **N_U** define the span of the *emission window*, and **N_R** is generated by the reception but fed back in the emission as the acknowledgement value $N(R)$. Thus these three counters are *shared values*, and in Esterel this means they have to be broadcast by *valued signals* and thus have a precisely defined value *at each instant*. The very definition of synchronous communication ensures that *every* part of the program will use the proper and timely counter values: for example, the Esterel expression "**?N_U**" always denotes the sequence number of the next awaited input message wherever it appears, and even at the very instant a new input message is received. As a rule, in a synchronous environment, the shared data is *always available*, and *always up to date!*

In addition to a broadcasted value, Esterel signals also carry a *control* information indicating when their value is changed. Since **N_R**, **N_A**, and **N_U** are counters that are never reset, their value can only

increase; we make the further commitment that their signals should *only* be emitted when their value *actually increases*. Thus:

- the emission of **N_U** will coincide *exactly* with the intake of user input,
- the emission of **N_R** will flag the successful reception of a new message,
- the emission of **N_A** will signal the acknowledgement of a pending message.

Also, **N_S** is incremented exactly each time an *I*-frame is transmitted, although it will be reset for retransmissions in later versions.

4.2. General Architecture

In this simplified version, the protocol driver has only four functions to perform: sending *I*-frames, receiving *I*-frames, accepting user input, and tallying acknowledgements. According to our modularity principle, each of these functions should be described by a separate module. These modules must implement *concurrent tasks*, as several functions might be activated simultaneously. For example, the driver can receive a data frame carrying a piggybacked acknowledgement while its emission window is full. It then has to transmit the *I*-frame contents to the user, tally the acknowledgement, and unblock the user input, thus executing three functions simultaneously.

We have carefully chosen the set of driver variables so that each can be maintained by exactly *one* of these modules (some functions may *use* the values of several counters, though). Thus our architecture will have four modules in parallel, each implementing a separate function, each updating a protocol counter according to its internal algorithm and exporting it, alike a *monitor*:

- Module **USER_INPUT_HANDLER** does user input flow management; it emits **X_ON** and **X_OFF** according to the emission window scheme, and increments and emits **N_U** each time new user input is accepted.
- Module **ACKNOWLEDGE_HANDLER** tallies acknowledgement. It updates and emits **N_A** each time an $N(R)$ value within the emission window acknowledges a pending message.
- Module **RECEPTION_HANDLER** accepts *I*-frames with the expected $N(S)$ value by forwarding their contents to the user and by incrementing and emitting **N_R**.
- Module **EMISSION_MANAGER** sends out *I*-frames each time user input is accepted. The **N_S** counter is local to this module.

While the emission function is rather trivial here, it will get increasingly complex in later versions; thus the simplistic **EMISSION_MANAGER** in this version is essentially a placeholder for upcoming extensions. Conversely, all three other modules are definitive and will remain *unchanged* through all the versions.

The **USER_INPUT_HANDLER** and **ACKNOWLEDGE_HANDLER** modules are tightly linked: they depend on each other for the values of **N_A** and **N_U**. Together they encapsulate the driver's emission window management. Thus, it is convenient to group them in a **WINDOW_MANAGER** intermediate module. The message buffering of version 2 will fit nicely in this module, which can also be used as a separate compilation unit, as we will see in section 9.

The version 1 architecture is illustrated by the block diagrams in figure 4, which depict the data/control relations between the various submodules and the driver environment. We must issue a word of caution on this kind of diagram, though: although they seem very clear and concise in simple cases, they rapidly become unmanageable as complexity increases.

The Esterel notation, on the other hand, although a little terse in this case, will remain readable even in the more complex versions. Since Esterel signals are broadcasted, the complex block diagram networks never have to be programmed: they are implicit in the use pattern of each signal. Thus the code for the main driver simply declares the three broadcasted protocol counters and runs the three main submodules in parallel:

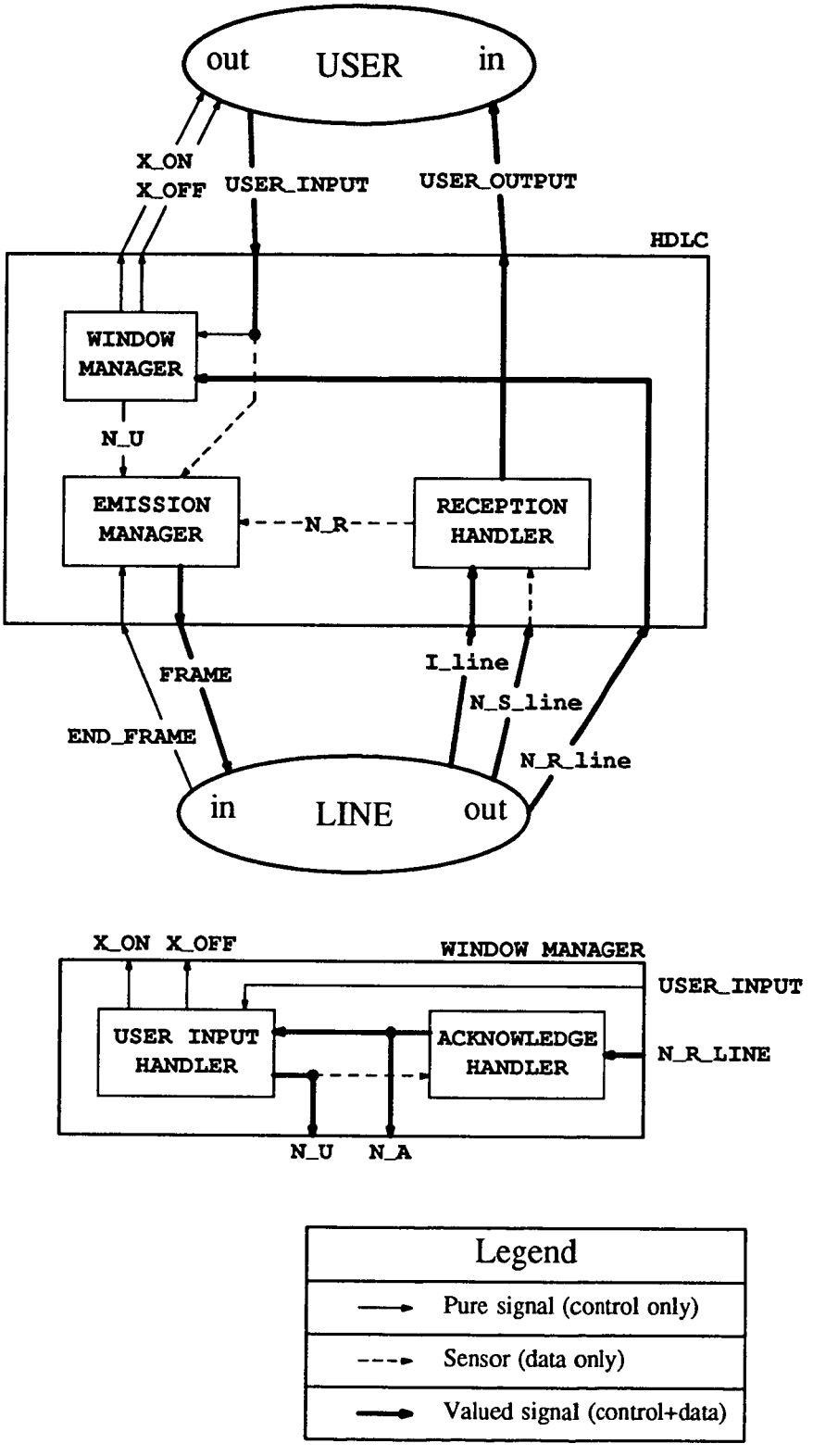

Figure 4: The Version 1 HDLC Driver Architecture

```
signal N_U (NUMBER), N_A (NUMBER), N_R (NUMBER) in
   copymodule WINDOW_MANAGER
||
   copymodule RECEPTION_HANDLER
||
   copymodule EMISSION_MANAGER
end
```

The complete code for the protocol version 1 can be found in Annex 1. We will now detail the programming of the submodules.

4.3. The WINDOW_MANAGER Module

It is simply runs submodules USER_INPUT_HANDLER and ACKNOWLEDGE_HANDLER modules, and makes the union of their interface. See Annex 1.

4.3.1. The USER_INPUT_HANDLER Submodule

This module produces the control flow outputs X_ON and X_OFF and the input number counter N_U (NUMBER), using inputs USER_INPUT and N_A (NUMBER), as follows:

- input is initially opened by emitting X_ON, and N_U is initialized to ZERO.
- When some USER_INPUT is received, N_U is incremented.
- If the number of unacknowledged messages, given by SPAN(?N_A, ?N_U), is now equal to WINDOW_SIZE, input is closed by emitting X_OFF, and ignoring further input. input will be reopened when a pending message is acknowledged, i.e., N_A is emitted.

This specification translates directly into Esterel code, except for one caveat: we cannot increment *directly* the value of signal N_U. This would require referencing simultaneously the *past* and *present* value of N_U, whereas Esterel signals only have a *present* value. One cannot use the "obvious" statement*

```
emit N_U (PLUS_1(?N_U))
```

This is *nonsense* in Esterel: it asserts that the current value of N_U is equal to the current (*not* the past) value of N_U plus one.

The solution is to declare a *sequential* local variable N_U, which we will be able to increment, to *shadow* the value of the signal. Such a *local internal variable / exported signal* pair is a frequent Esterel construction, and all the protocol counters described in 4.1 will be handled this way.

* Assuming appropriate declarations

With this construction, the rest of the code is straightforward:

```
var N_U:=ZERO : NUMBER in
   emit N_U (ZERO);
   emit X_ON;
   loop
      await USER_INPUT do
         call INCREMENT (N_U)();
         emit N_U (N_U);
         if SPAN(?N_A,?N_U) = WINDOW_SIZE then
            emit X_OFF;
            await N_A;
            emit X_ON
         end
      end
   end
end
```

4.3.2. The ACKNOWLEDGE_HANDLER Submodule

This module maintains the acknowledgment counter **N_A(NUMBER)**, according to its input **N_R_line(NUMBER)**, and the *value* of **N_U**.†

The value of **N_A** should be updated to reflect **?N_R_line** each time this value acknowledges pending messages, that is, each time it decreases the number of pending messages. This is easily coded, using a shadow variable:

```
var N_A:=ZERO : NUMBER in
   emit N_A (ZERO);
   every N_R_line do
      if SPAN (?N_R_line,?N_U) < SPAN (N_A,?N_U) then
         N_A := ?N_R_line;
         emit N_A (N_A)
      end
   end
end
```

4.4. The RECEPTION_HANDLER Module

This module handles the **I_line(MESSAGE)** data frame inputs and the associated **N_S_line(NUMBER)** sequence number values. It produces the **USER_OUTPUT(MESSAGE)** signal accordingly, and maintains the **N_R(NUMBER)** reception counter.

We accept line messages only in ascending number sequence; out-of-sequence messages are simply ignored. When a message is accepted, we forward it and increment **N_R**. This is readily coded, using a shadow variable:

† The fact that only the value of **N_U** is used is formalized in the interface by declaring it as a **sensor**.

```
var N_R:=ZERO : NUMBER in
   emit N_R (ZERO);
   every I_line do
      if ?N_S_line = N_R then
         emit USER_OUTPUT (?I_line);
         call INCREMENT (N_R)();
         emit N_R (N_R)
      end
   end
end
```

4.5. The EMISSION_MANAGER Module

This module composes and emits frames on the line. It is elementary here: each time an input message is accepted (N_U emitted), an *I*-frame is composed with the current values of N_S, N_R, and USER_INPUT, and is emitted on FRAME(FRAME). The N_S counter is simply a sequential variable local to this module.

```
var N_S:=ZERO : NUMBER in
   every N_U do
      emit FRAME (I_FRAME (N_S, ?N_R, false, ?USER_INPUT));
      call INCREMENT (N_S) ()
   end
end
```

5. HDLC Version 2: the Non Instantaneous Line

We shall now assume that sending a frame to the line takes time. Output is initiated by the emission of the FRAME signal, and the host system returns the END_FRAME signal when output is completed. We cannot send out all messages as soon as we receive them from the user. This calls for the following extensions to our version 1 program:

- An *Input Buffer* must be maintained to hold messages prior to their transmission.
- *Frame Emission* is now considerably more complex, since message input and transmission are not synchronized any more.

Following the one function–one module rule, the input buffer will be managed in an BUFFER_MANAGER module that we will add to the WINDOW_MANAGER submodule. The buffer will be output to the EMISSION_MANAGER module on a BUFFER signal. The declaration of this signal will be the only actual modification to the HDLC master module.

Messages will be kept in the buffer until they are actually acknowledged, as opposed to until they are transmitted. In this way, the BUFFER_MANAGER module will be independent of the transmission logic in the EMISSION_MANAGER module. Thus when we add in error recovery in versions 4 and 5, the latter module will find the messages it must retransmit in the current buffer, and we will not need to change the BUFFER_MANAGER module.

The new complex frame emission logic calls for a complete rewriting of the EMISSION_MANAGER module. In fact we need to develop a complete *sub-architecture* for this task, that is flexible enough to support all the extensions of the following versions. We devote the following section to its analysis. The resulting global architecture is illustrated in the figures on the following pages.

This version illustrates two important points of the Esterel programming style: the free use of pure signals to foster a good modular decomposition of a task, and the use of instantaneous dialogues to communicate state information across modules.

5.1. The EMISSION_MANAGER Architecture

The emission algorithm can be specified as follows:

> "Whenever there is a message to transmit and no transmission is occurring, transmit the message and increment the transmission counter N_S"

Since Esterel is event-driven, we will be using the following equivalent formulation: send the current message and increment N_S each time

- a new input message is accepted while the line is free,
- or the line frees while there are still messages to transmit.

To translate this into Esterel code, however, we must have a way to determine when the line is free. The most straightforward way to do this is to track the line state inside the emission management code. However this goes against our one function–one module rule, and indeed yields very *poor* modularity. It amounts to build into the *I*-frame emission code the hypothesis that the line must be free when no *I*-frame emission is occurring, and this will actually be *false* in every subsequent version of the driver!

Thus the good design decision is to have a LINE_STATE module track the line state, and an I_FRAME_MANAGER module do the *I*-frame emission according to the line state information provided by LINE_STATE.

Now it is easy to deduce the line state from the interleaving of FRAME and END_FRAME signals. The LINE_STATE submodule will signal a free line by emitting a LINE_FREE signal to I_FRAME_MANAGER. Conversely, I_FRAME_MANAGER will be able to probe the line status by emitting a TEST_LINE probe, to which LINE_STATE should answer by reemitting LINE_FREE if the line is free. With this interface I_FRAME_MANAGER can directly implement the emission specification.

Note that because of the Esterel synchrony hypothesis, all the *dialogues* to exchange line status information between the two submodules are *instantaneous*. This simplifies considerably their description and coding: for example the I_FRAME_MANAGER module can actually *test* for an immediate response to its TEST_LINE probe, and act accordingly. Furthermore, because all the pure signal exchanges are performed at compile-time, they incur *no run-time cost!* The automaton generated by the compiler simply looks up its state and the current input event, and performs *immediately* the correct action, without any intervening internal steps.

The global communication architecture is illustrated in the figure 5. The following sections describe the actual programming of these new modules.

5.2. The WINDOW_MANAGER Module

We simply add the BUFFER_MANAGER submodule to the parallel of the USER_INPUT_HANDLER and ACKNOWLEDGE_HANDLER submodules. Those are left unchanged. We also add the BUFFER signal to the output interface.

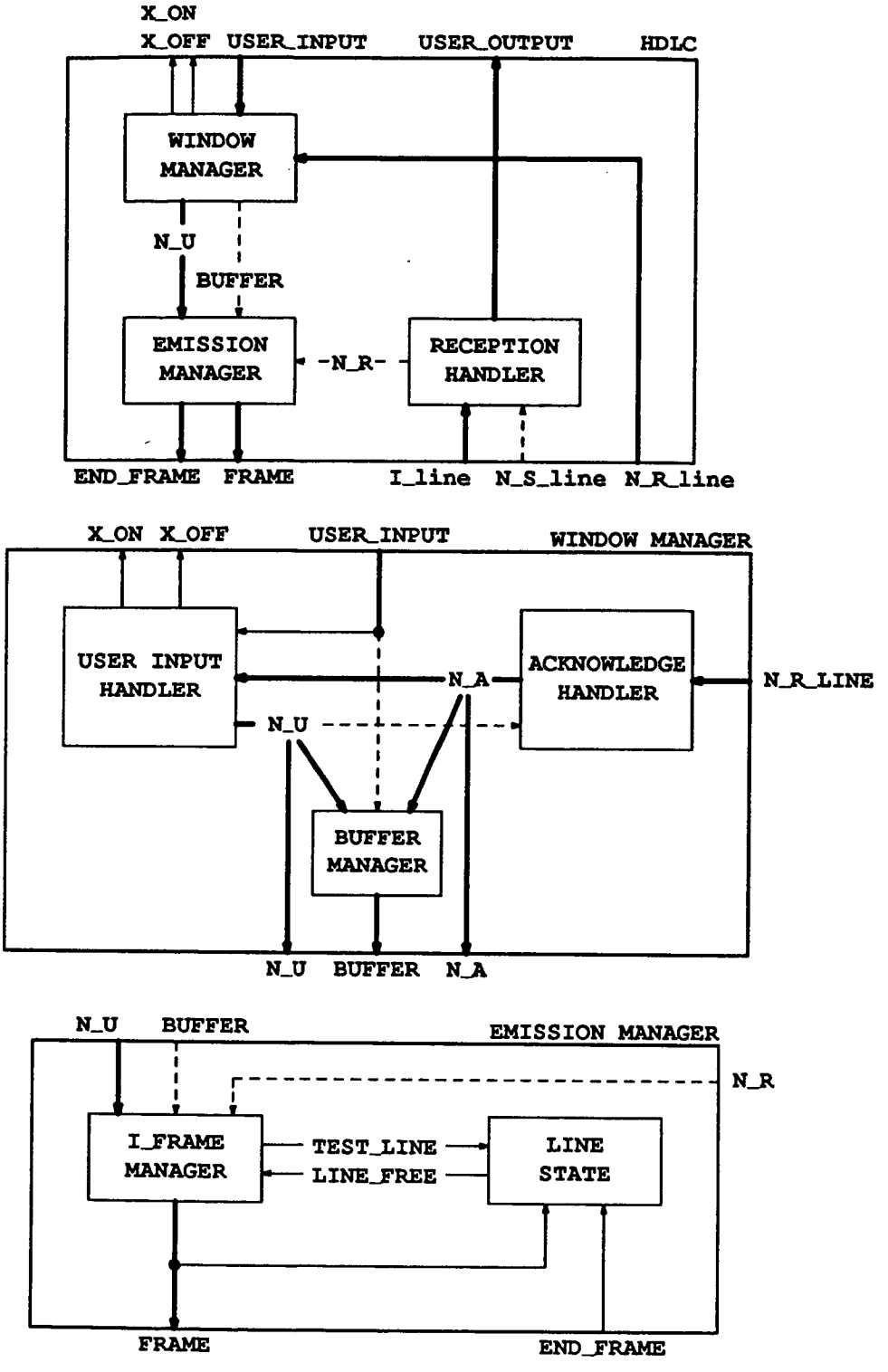

Figure 5: The Version 2 HDLC Driver Architecture

5.2.1. The BUFFER_MANAGER Submodule

This module maintains the message buffer `BUFFER(BUFFER)`, using a shadow variable. A `?USER_INPUT` message is entered when it is accepted (`N_U` emitted), and acknowledged messages are purged each time `N_A` is emitted:

```
var BUFFER:=EMPTY_BUFFER(WINDOW_SIZE) : BUFFER in
   loop
      emit BUFFER (BUFFER);
      await
         case N_U do
            call ENQUEUE (BUFFER)(?USER_INPUT)
         case N_A do
            call PURGE (BUFFER)(SPAN (?N_A,?N_U))
      end
   end
end
```

This module is a typical use of the Esterel **await-case** construct, where the branches are *exclusive* and operate on a common sequential variable (note that the **N_A#N_U** relation is included in the interface, since it is *necessary* for the proper functioning of this module).

5.3. The EMISSION_MANAGER Module

This module's interface is composed of the protocol's counters and buffer, and the line's output interface, with all the relations that can be derived from the global serializing and line relations, since this module is intended as a separate compilation unit.

The **TEST_LINE** and **LINE_FREE** signals are declared locally to this module. The body of the **EMISSION_MANAGER** module simply consists of the parallel composition of the **I_FRAME_MANAGER** and **LINE_STATE** submodules (see annex 2).

5.3.1. The I_FRAME_MANAGER Submodule

This module has the same interface as **EMISSION_MANAGER**, except that it gets the line status by the **LINE_FREE** signal (instead of **END_FRAME**), and can inquire about it with a **TEST_LINE**.

The **I_FRAME_MANAGER** has actually two functions to carry out: determining when a message frame should be sent, and actually shipping out this frame. It would not be convenient to code those two function in parallel, because they are coupled by shared data: the value of the emission counter **N_S** determines if a new message is available, but **N_S** is also used and incremented when a message is sent out. So, we compose these functions sequentially, using the Esterel exception construct. The first function waits for an opportunity to send out a frame, and then calls the emission function by raising a **SEND_FRAME** exception. The emission is performed (instantly) by the exception handler, after which the module simply loops back to the first function. Thus the code outline for **I_FRAME_MANAGER** is:

```
var N_S:=ZERO : NUMBER in
   loop
      trap SEND_FRAME in
         % decide when to send an I-frame
      handle SEND_FRAME do
         % actually compose and emit the frame; increment N_S
      end
   end
end
```

The decision function uses the greedy algorithm described in 5.1. It is coded as an **await-case** statement within a **loop** because the two signals it monitors are *not* independent. It is quite possible that **LINE_FREE** and **N_U** be simultaneously present, and in this case we do not want to test for message availability, since we know that at least the message we just buffered in must be available. Thus the handling of **N_U** must have *priority* over that of **LINE_FREE**, and this is conveniently expressed by the order of the **case** clauses.

The message composition is straightforward with our choice of data interface. The distance from message number **N_S** to the end of the buffer is given by **SPAN(N_S, ?N_U)**, and can be used directly

to index the message in the buffer, using the ACCESS function. The acknowledgement counter value is just read off N_R, as in the first version.

The complete code for the I_FRAME_MANAGER module is thus:

```
var N_S:=ZERO : NUMBER in
   loop
      trap SEND_FRAME in
         loop
            await
               case N_U do
                  emit TEST_LINE;
                  present LINE_FREE then exit SEND_FRAME end
               case LINE_FREE do
                  if N_S <> ?N_U then exit SEND_FRAME end
            end
         end
      handle SEND_FRAME do
         emit FRAME (I_FRAME (N_S, ?N_R, false,
                              ACCESS (?BUFFER, SPAN(N_S,?N_U))));
         call INCREMENT (N_S)()
      end
   end
end
```

5.3.2. The LINE_STATE Submodule

This module must send the LINE_FREE signal each time the line becomes free, and as a response to the TEST_LINE signal if and only if the line is free. This is done by monitoring the FRAME and END_FRAME signals, as follows:

```
loop
   trap LINE_BUSY in
      await immediate FRAME do exit LINE_BUSY end
   ||
      loop emit LINE_FREE each TEST_LINE
   handle LINE_BUSY do
      await END_FRAME
   end
end
```

After a frame has been emitted, control stays in the handle statement until END_FRAME is received, so TEST_LINE requests are ignored. The internal loop-each ensures that all requests are answered when the line is free. Note that a frame may be sent immediately after the line frees, so that an immediate delay is needed here. Note also that the following simpler code will *NOT* work:

```
loop
   do
      loop emit LINE_FREE each TEST_LINE
   watching immediate FRAME
   timeout await END_FRAME end
end
```

Here the reception of FRAME *inhibits* emission of LINE_FREE, which the I_FRAME_EMITTER module is precisely *waiting* for to send out FRAME! This is a (superb) example of Esterel causality cycle.

6. HDLC Version 3: Checkpoint Answering

In this version we add responses to error-recovery checkpointing commands. When a command with the *P* bit set is received, our driver should issue a *response* with the *F* bit set containing the next expected message number **?N_R**, so that the distant emitter can resynchronize after an error. We will make use of this facility in our version 5 driver.

The responses must have *priority* over normal data frames; the latter can never be used as responses, because of option 8. Responses must be *S*-frames, and we choose to use *RR* (Receive Ready) frames here.

Because we have chosen a good modular structure, we only need to change the **EMISSION_MANAGER** module, and then only by *adding* two new submodules to it, and reconnecting the other submodules. We just add a **RESPONSE_MANAGER** submodule to send responses, and a **LINE_FILTER** submodule to resolve line access conflicts.

This version illustrates two new features of Esterel programming: submodule connection through renamings and the use of instantaneous presence test of a synchronous signal (here **PREEMPT_LINE**) to determine behavior (here to decide priority).

6.1. The LINE_FILTER Submodule; the Priority Discipline

To resolve priority contentions on the line, we use an auxiliary synchronous signal **PREEMPT_LINE** and the following discipline:

- A module wishing to emit a higher-priority frame also emits the **PREEMPT_LINE** signal along with it. It can do so as soon as it receives the **LINE_FREE** signal.
- A module wishing to emit a normal-priority frame must wait for a **NORMAL_LINE_FREE** signal, not just for **LINE_FREE**.

The **LINE_FREE** signal is computed by the **LINE_STATE** submodule. The **LINE_FILTER** submodule will compute the **NORMAL_LINE_FREE** signal. The latter should be emitted each time the **LINE_FREE** signal is, *except* when a preemptive frame is sent *synchronously*. This is coded trivially as

```
every LINE_FREE do
   present PREEMPT_LINE else emit NORMAL_LINE_FREE end
end
```

6.2. Changes to the EMISSION_MANAGER Module

Now that we have designed the priority mechanism, we can fit all the submodules together in the **EMISSION_MANAGER** module. Since here the normal-priority frames are *I*-frames, we rename the **NORMAL_LINE_FREE** signal in **LINE_FILTER** into **I_LINE_FREE**. According to the discipline above, we must also connect the **LINE_FREE** input of the **I_FRAME_MANAGER** submodule to this signal, and we can do this using another renaming.

Note also that the **PREEMPT_LINE** signal is local to the **LINE_FILTER** and **RESPONSE_MANAGER** submodules. The combination of these two modules actually acts as a filter on the line state seen by **I_FRAME_MANAGER**, occupying part of the line bandwidth with checkpointing responses. Thus we group them, which gives us the following structure for **EMISSION_MANAGER**

```
signal TEST_LINE, LINE_FREE, I_LINE_FREE in
   copymodule I_FRAME_MANAGER [signal I_LINE_FREE / LINE_FREE]
||
   signal PREEMPT_LINE in
      copymodule LINE_FILTER [signal I_LINE_FREE / NORMAL_LINE_FREE]
   ||
      copymodule RESPONSE_MANAGER
   end
||
   copymodule LINE_STATE
end
```

Note that we do not need to rename or merge the **TEST_LINE** queries from the two frame emission submodules; this is done naturally by the synchronous execution and broadcast.

6.3. The RESPONSE_MANAGER Submodule

This module transmits an appropriate *S*-frame each time a *P* bit is received. Of course, even if this module has priority, it must wait for the line to be available to send this frame. This is elegantly coded as

```
every P_line do
   emit TEST_LINE;
   await immediate LINE_FREE do
      emit PREEMPT_LINE;
      emit FRAME (S_FRAME (RR_RESPONSE, ?N_R, true))
   end
end
```

Note that this module illustrates a completely different way to use the **LINE_STATE** interface, which shows its flexibility (version 5 will show yet another use!).

```
await immediate LINE_FREE
```

correctly waits for the first free line, since it is instantly triggered by the response to the **TEST_LINE** probe if the line is immediately available, and by the **LINE_FREE** triggered by the end of the current transmission otherwise.

7. HDLC Version 4: Using *REJ* Frames

In this version we add an active form of error recovery, using *REJ* supervisory frames (option 2). We will actually add two mechanisms, one to handle incoming *REJ* frames, and the other to actually send out such frames when out-of-sequence *I*-frames are received.

All the decisions about receiving or sending *REJ* frames will be done in a **REJECT_HANDLER** module that we will add to the top **HDLC** module. This module will issue two command signals to the **EMISSION_MANAGER** module:

- **RETRANSMIT** signals that the message buffer should be retransmitted
- **SEND_REJECT** indicates that a *REJ* frame should be sent as soon as possible

These command signals are declared in the top **HDLC** module. Handling these new commands requires only local changes to the **EMISSION_MANAGER** module. The **I_FRAME_MANAGER** submodule will handle the **RETRANSMIT** command with a few changes, and the **RESPONSE_MANAGER** submodule must be rewritten in order to handle the **SEND_REJECT** request.

This version illustrates the use of synchrony to monitor parallel behavior and the combination of sequential variables and parallel exits to accumulate data.

7.1. The REJECT_HANDLER Module

This module is actually an exception to our one function — one module rule, since it handles both sending and receiving *REJ*-frames. However, since the two functions are related and are implemented by very short code, our module is still coherent. It is thus made up of two independent pieces of code in parallel; its interface is just the union of the interfaces of the two parts.

As a rule, the reception of a *REJ* frame will cause retransmission; however it should not do so if the $N(R)$ value it carries is outdated, or if the buffer is empty. This translates into the following code

```
every REJ_line do
   if not ((?N_R_line <> ?N_A) or (?N_A = ?N_U))
   then emit RETRANSMIT end
end
```

Note that we do not need to specify that retransmission starts at the $N(R)$ number carried by the *REJ* frame, since the **N_A** counter is *instantly updated* with this value.

Reject frames should be sent out when out-of-sequence frames are received. We could easily do this inside the **RECEPTION_HANDLER** module, but this would require changing it. Instead, we will detect bad frames in this module by *observing* the **RECEPTION_HANDLER** module. We simply note that an **I_line** message is out of sequence *if and only if* it does not cause *instantly* the emission of a new **N_R** value by **RECEPTION_HANDLER**, and furthermore that this is *testable* in Esterel!! In fact code for rejecting each out-of-sequence message is trivial:

```
every I_line do
   present N_R else emit SEND_REJECT end
end
```

However, it is unadvisable to generate a *REJ* frame for each incorrect frame that is received, since this would lead to multiple retransmissions that would clutter up the line (if there are n messages buffered in the line, recovering from a single error will require $n^2/2$ retransmissions!). This is easily avoided by sending only *one REJ per expected frame*. This is easily coded as*

```
loop
   await I_line do
      present N_R else emit SEND_REJECT end
   end
each N_R
```

7.2. Changes to the I_FRAME_MANAGER Submodule

A **RETRANSMIT** command should be handled by resetting the **N_S** counter to **?N_A**, and by trying to send the corresponding frame if the line is available. All of this is conveniently handled by simply adding a **RETRANSMIT** case to the multiple **await** statement. Note that this case shares the "emit a frame if the line is available" function with the **N_U** case. We can avoid code duplication by setting this function apart in parallel, and by triggering it by a **TRY_TO_SEND** local signal. Such factoring cleans up the source with no change to the compiled code, as it is automatically expanded by the compiling process.

Since we have retransmissions, we must guard against transmission underflow: a late acknowledge from a previous transmission might set **N_A** *past* the current transmission point **N_S**. Since we should not attempt to transmit a message that has been purged from the buffer, we should reset **N_S** to **N_A** when this occurs. Since our data interface provides us with a **GUARD_UNDERFLOW** procedure for this purpose, we only need to call it each time **N_A** advances, in an additional clause in the multiple **await**. Note that because this procedure can only advance **N_S**, there is no need to attempt transmission after it.

* Experienced Esterel programmers will notice that the **present** test is useless here, since the external **loop-each** has priority; however, it is harmless, and does make the code clearer.

In fact, N_A *must* be incompatible with N_U and LINE_FREE† for this code to work; since a RETRANSMIT command resets N_S anyhow, its case should have priority over the underflow guard.

The complete code for I_FRAME_MANAGER is thus

```
var N_S:=ZERO : NUMBER in
   loop
      trap SEND_FRAME in
         signal TRY_TO_SEND in
            loop
               await
                  case N_U do emit TRY_TO_SEND
                  case RETRANSMIT do N_S := ?N_A; emit TRY_TO_SEND
                  case LINE_FREE do
                     if N_S <> ?N_U then exit SEND_FRAME end
                  case N_A do
                     call GUARD_UNDERFLOW (N_S) (?N_A, ?N_U)
               end
            end
         ||
            every TRY_TO_SEND do
               emit TEST_LINE;
               present LINE_FREE then exit SEND_FRAME end
            end
         end
      handle SEND_FRAME do
         emit FRAME (I_FRAME (N_S, ?N_R, false,
                              ACCESS (?BUFFER, SPAN(N_S,?N_U))));
         call INCREMENT (N_S) ()
      end
   end
end
```

7.3. The RESPONSE_MANAGER Submodule

The RESPONSE_MANAGER submodule in version 2 sends out *RR* responses to *P* bit checkpointing commands. Since *REJ* frames are also responses, they are naturally handled by this module. The two functions can actually be conveniently multiplexed, since one can send a *REJ* frame with an *F* bit set. Thus the P_line and SEND_REJECT signals should both trigger a response-sending function. We code this by triggering this function by a SEND_RESPONSE local signal, and using two **every** loops to emit SEND_RESPONSE.*

The P_line and SEND_REJECT signals not only trigger the emission of a response frame, they also determine its data contents: whether it is an *RR* or a *REJ*, whether its *F* bit is on or not. Thus the response sender not only waits for an available line, it also accumulates the data for its frame.

---

† More precisely, occurrences of LINE_FREE not synchronous with RETRANSMIT.

* In fact SEND_RESPONSE could be a command in its own right, which could be used for example to send acknowledges after a timeout if there were no available frames to piggyback on.

```
signal SEND_RESPONSE in
   every P_line do emit SEND_RESPONSE end
||
   every SEND_REJECT do emit SEND_RESPONSE end
||
   loop
      var RESPONSE_TYPE:=RR_RESPONSE : S_FRAME_TYPE,
          CHECKPOINTING:=false : boolean in
         trap SEND_FRAME in
            every P_line do CHECKPOINTING := true end
         ||
            every SEND_REJECT do RESPONSE_TYPE := REJ end
         ||
            await SEND_RESPONSE do
               emit TEST_LINE;
               await immediate LINE_FREE do exit SEND_FRAME end
            end
         handle SEND_FRAME do
            emit PREEMPT_LINE;
            emit FRAME (S_FRAME (RESPONSE_TYPE, ?N_R, CHECKPOINTING))
         end
      end
   end
end
```

Note how the data accumulated in the every loops is collected in the **SEND_FRAME** handler, and that the variables are correctly reset for a new frame after each transmission. We could have used **await** instructions instead of **every**s for the data accumulation; we will discuss our choice in section 9.

8. HDLC Version 5: Error Recovery on Timeouts

The active error recovery of section 4 does not protect against multiple errors (since we chose single retransmission) or an error on the last frame. To do this, we need a passive error recovery that depends on absolute time, not on cooperation of the receiver. This is the mechanism we will now add in this final version.

The mechanism we implement will send out a polling *RR* command (*P* bit set) if an acknowledge for a message is not received within an **ACKNOWLEDGE_TIME_LIMIT** millisecond delay. It will then resynchronize emission by blocking further output (except for its own responses, to avoid deadlocks) until an *F* response is received, and resetting the emission counter to the expected $N(R)$ when this finally occurs. If this resynchronization does not succeed within **RESYNCH_TIME_LIMIT** millisecond, it is assumed that some of the *S*-frames were lost, and the whole resynchronization process is restarted. A complete implementation would go one step further and abort the whole session if too many resynchronization attempts fail.

Because we have developed quite extensively our program architecture, this last version will require comparatively fewer changes, since most of the functions it requires are already provided by our program. We will add a **TIMEOUT_MANAGER** module to perform the timing management, and this module will itself call a **RESYNCH_MANAGER** module to perform the necessary resynchronizations. Inserting these modules in our architecture will only require a few renamings, and the addition of a **BUFFER_STATE** module to detect the empty buffer. We will also add a **BUFFER_FILTER** module that synchronizes **I_FRAME_EMITTER** with the buffer state, to improve the compiler code generation.

The new features introduced in this version are a more general use of a (non-parallel, terminating) submodule, and the issues of module synchronization and compiler state generation.

8.1. Changes to the EMISSION_MANAGER Module

The timeout manager will synchronize mainly on absolute time and acknowledges (N_A), but it also needs to know about the *message buffer state*, since it must suspend its timer when the buffer is empty. Thus we add an EMPTY_BUFFER signal, and a BUFFER_STATE module that generates it from N_A and N_U.

The functions required for our timeout error recovery are sending out polling frames, blocking message output, and resetting the emission counter. The latter is already provided by the RETRANSMIT command. The first two are easily handled by extending the line priority scheme we have developed. All we have to do is take *another copy* of the LINE_FILTER module, make it communicate with a new PREEMPT_LINE signal with the RESYNCH_MANAGER module inside EMISSION_MANAGER, and insert the resulting *line filter stage* between RESPONSE_MANAGER and I_FRAME_MANAGER. The NORMAL_LINE_FREE output of this filter stage will thus be connected to I_LINE_FREE, and its LINE_FREE input will be connected to a new RESYNCH_LINE_FREE signal, to which the NORMAL_LINE_FREE output of the response stage will be reconnected. Thus the code for the EMISSION_MANAGER module will become

```
signal TEST_LINE, LINE_FREE, RESYNCH_LINE_FREE, I_LINE_FREE,
       BUFFER_EMPTY in
   copymodule BUFFER_STATE
||
   signal ACTIVE_RETRANSMIT, ACTIVE_LINE_FREE in
      copymodule BUFFER_FILTER [signal I_LINE_FREE / LINE_FREE]
   ||
      copymodule I_FRAME_MANAGER
         [signal ACTIVE_RETRANSMIT / RETRANSMIT,
                 ACTIVE_LINE_FREE / LINE_FREE]
   end
||
   signal PREEMPT_LINE in
      copymodule LINE_FILTER [signal I_LINE_FREE / NORMAL_LINE_FREE,
                                     RESYNCH_LINE_FREE / LINE_FREE]
   ||
      copymodule TIMEOUT_MANAGER [signal RESYNCH_LINE_FREE / LINE_FREE]
   end
||
   signal PREEMPT_LINE in
      copymodule LINE_FILTER [signal RESYNCH_LINE_FREE / NORMAL_LINE_FREE]
   ||
      copymodule RESPONSE_MANAGER
   end
||
   copymodule LINE_STATE
end
```

The addition of the BUFFER_FILTER module will be discussed below in section 8.4.

8.2. The BUFFER_STATE Submodule

This module is very similar in function to the LINE_STATE submodule. The differences are that it does not need to respond to queries and that it leaves its "buffer not empty" state as soon as N_A reaches N_U. Its code is straightforward:

```
loop
   trap BUFFER_EMPTY in
      emit BUFFER_EMPTY;
      await N_U;
      every N_A do
         if ?N_A = ?N_U then exit BUFFER_EMPTY end
      end
   end
end
```

8.3. The TIMEOUT_MANAGER Submodule

The only delicate point in the timeout management is that the resynchronization management is tightly coupled with it, since it both determines and is subject to timeouts. Rather than set it off in a parallel submodule and have to design complex synchronizations, we include *inside* the TIMEOUT_MANAGER code a RESYNCH_MANAGER submodule that *terminates* when a resynchronization is completed. Given this design decision the code for TIMEOUT_MANAGER is both easy to design and to read (see annex 5 for the interface):

```
loop
   % no timer while buffer is empty
   await N_U;
   % now messages should get acknowledged regularly
   loop
      do
         await N_A % wait for a new acknowledge
      watching ACKNOWLEDGE_TIME_LIMIT MS
      timeout
         % repeat resynch attempts, until one succeeds
         trap RESYNCHRONIZED in
            loop
               copymodule RESYNCH_MANAGER; exit RESYNCHRONIZED
            each RESYNCH_TIME_LIMIT MS
         % resume transmission when counters are resynchronized
         handle RESYNCHRONIZED do emit RETRANSMIT
         end
      end
   end
each BUFFER_EMPTY
```

Note that because the `loop · · · each BUFFER_EMPTY` construct is preemptive, the RETRANSMIT command will *not* be sent when the *N* (*R*) acknowledge in the response empties the message buffer — and this is the *correct* behavior.

The RESYNCH_MANAGER module controls the actual resynchronization: it must send out a polling *RR* frame and block *I*-frame emission until the corresponding response is received. This second function is accomplished by emitting PREEMPT_LINE each time the line is free, which prevent the LINE_FILTER module from echoing the LINE_FREE signal to the I_FRAME_MANAGER. The first function is best seen as a special case of the second: on the *first* LINE_FREE, we must not only preempt the line but also actually send out a polling frame. This can be coded very classically, using a boolean variable; of course, we must first probe the line status to take advantage of an initially free line:

```
var POLL_SENT:=false : boolean in
   % hold the line until a response occurs
   % send out a polling frame on the first occasion
   do
      emit TEST_LINE;
      every immediate LINE_FREE do
         emit PREEMPT_LINE; % take the line
         if not POLL_SENT then
             emit FRAME (S_FRAME (RR_COMMAND, ?N_R, true));
             POLL_SENT := true
         end
      end
   watching F_line
   % cause a free line to be signaled to I_FRAME_MANAGER
   timeout emit TEST_LINE end
end
```

To unblock the line, we simply probe its status; if it is free a **LINE_FREE** signal will be echoed to **I_FRAME_MANAGER**.*

This module shows yet another way of using the **LINE_STATE** interface; we certainly could have adapted the **RESPONSE_MANAGER** usage here, but this would have been less efficient, as we will see in section 9.

8.4. The BUFFER_FILTER Submodule

There is an obvious logical relationship between the transmission algorithm and the message buffer state, namely that no *I*-frame can be sent out while the buffer is empty. The code given so far does respect this relationship, but only for the following reasons:

(*i*) An *I*-frame can only be emitted when either an **N_U**, **RETRANSMIT**, or **LINE_FREE** signal is received.

(*ii*) The message buffer cannot be empty when **N_U** is emitted.

(*iii*) The **RETRANSMIT** signal is not emitted by the **REJECT_HANDLER** when **?N_A=?N_U**, that is, when the buffer is empty.

(*iv*) An *I*-frame is only sent as a reaction to the **LINE_FREE** signal when **N_S<>?N_U**; since the **GUARD_UNDERFLOW** procedure keeps **N_S** between **?N_A** and **?N_U**, this can only be the case if **?N_A<>?N_U**, that is, if the buffer is not empty.

While all these arguments are sound, they are quite subtle, especially the last two, which involve reasoning about data values. Since the Esterel compiler abstracts away from these values, there is no way that it can make these points. Thus, from its point of view, it is perfectly possible for an *I*-frame emission to occur while the message buffer is empty and the timeout guards are turned off, and it will actually generate code to handle these emissions.

This is of course very inefficient, since this extra generated code is completely useless. Worse than that, it actually violates basic invariants of this program, so that the automaton-checkers applied to this code would fail this technically correct program.

The solution to this problem is to state explicitly the logical relationship between buffer and emission in the program. This will only introduce purely logical code, that will be completely removed by the compiler; better, it will actually help the compiler prune off the undesired cases, so this is a case where *adding* source code lines *decreases* the compiled code size!

---

* This is not essential in practice, since the **TIMEOUT_MANAGER** sends out synchronously a **RETRANSMIT** command that causes the **I_FRAME_MANAGER** to probe the line status anyhow; however this **emit** is certainly harmless and makes the module more general.

Thus we need to modify our code so that points (*iii*) and (*iv*) above become explicit in the code ((*i*) and (*ii*) are already explicit). Point (*iii*) can be restated as "no **RETRANSMIT** command can reach **I_FRAME_MANAGER** while the buffer is empty". This can be made explicit by inserting a submodule that *filters out* extraneous **RETRANSMIT**s when the buffer is empty. To make point (*iv*) explicit, we would need to change **I_FRAME_MANAGER** code so that it would test the buffer status first before comparing the counter values. We can avoid doing this by filtering out the **LINE_FREE** signal altogether when the buffer is empty.

To sum up, we only need to add a **BUFFER_FILTER** module that filters the **RETRANSMIT** and **LINE_FREE** inputs of **I_FRAME_MANAGER**. As we have seen in 8.1, this introduces two new local signals in **EMISSION_MANAGER**, and a few renamings:

```
signal ACTIVE_RETRANSMIT, ACTIVE_LINE_FREE in
   copymodule BUFFER_FILTER [signal I_LINE_FREE / LINE_FREE]
||
   copymodule I_FRAME_MANAGER
      [signal ACTIVE_RETRANSMIT / RETRANSMIT,
              ACTIVE_LINE_FREE / LINE_FREE]
end
```

The code for **BUFFER_FILTER** itself is obvious

```
loop
   await N_U do
      every immediate RETRANSMIT do emit ACTIVE_RETRANSMIT end
   ||
      every immediate LINE_FREE do emit ACTIVE_LINE_FREE end
   end
each BUFFER_EMPTY
```

9. The Esterel Compilation into Automata

The Esterel compiler transforms our concurrent modular program into a sequential unstructured automaton. Since the transitions of this automaton only contain *unavoidable* data manipulation statements, it is time-optimal. Thus the efficiency of the compiling process is mainly measured by the *size* of the automaton.

The Esterel compiler assigns a code to every elementary action (e.g., assignments) in the program text, and describes each transition by listing of the codes of the action this transition performs. A given code is often copied many times on many transitions; thus the best measure of the *size* of the automaton is the *total* number of call *codes* in all the transitions, and the number of states is more an auxiliary measure.

Here are the sizes of the automata generated for our five versions, together with the source code sizes. For completeness, we have included a 4a version that has timeout recovery but no rejects (obtained by removing the **REJECT_HANDLER** module from version 5).

Version	Source Lines	States	Call Codes
1	120	3	70
2	197	5	173
3	228	7	450
4	280	13	1308
4a	361	16	1952
5	382	31	5012

Note that the code sizes are in direct relation with the capabilities of the different stages. Thus version 2 generates very little code, although it has the same modular decomposition as the more elaborate

versions, and more than half the source code size: the modularity comes at no cost. What does cost, on the other hand, is functionality: the code size increases exponentially as emission priority levels are added. Thus the more useful versions have larger code tables, that might be judged too large for this kind of application.

This can be partly counteracted by noticing that the final automaton is really composed of concurrent, weakly interacting automata. For example, between versions 4a and 5 the number of states doubles and the code size nearly triples, due solely to the addition of the small `REJECT_HANDLER` module. Since it operates independently from the rest of the program, its two active states that track the emission of rejects are replicated in *all* the states of the program, as usual in the classical parallel product of automata. The code size is more than doubled because the module contains a dynamic "`if`" test that splits transitions in the current compiling algorithm, and thus entails additional replication.

The Esterel v3 compiler `-cascade` option will compile separately the independent submodules and *cascade* the resulting small automata for execution. The total code size is then only the *sum* of the sizes of the smaller automata. The following table gives the code/state sizes of the smaller automata for versions 4 and up:

Version	4	4a	5
`WINDOW_MANAGER`	47/3	47/3	47/3
`RECEPTION_HANDLER`	14/2	14/2	14/2
`REJECT_HANDLER`	37/3	*omitted*	37/3
`EMISSION_MANAGER`	126/4	627/10	627/10
Total calls:	224	688	725

This compiling technique can thus achieve substantial space savings. There is a slight time penalty, though: intermodule signals are now implemented by boolean variables that must be tested by each of the submodules, instead of being compiled away.

All the submodule automata are *minimal*, as it should be expected for Esterel programs with no code duplication. Obtaining minimal automata is not that obvious, however, as it involves a number of programming subtleties. These fall in three categories:

- precise coding of all the *logical* synchronizations between modules
- choosing the right combination of program state and data values to represent the system states
- careful coding of each module, so that its automaton is minimal

The first point is certainly the most difficult, as it requires a thorough understanding of the problem at hand. As it is achieved by stating all the implicit (data-dependent) relations between modules as explicit signals, it often leads to better-structured code. A good example of this is the management of buffer state discussed at length in section 8.4.

The second point is illustrated by the line handling in `RESPONSE_MANAGER` and `RESYNCH_MANAGER`. These two module share a common function, namely sending out a frame as soon as the line becomes available; however, this function is coded very differently in the two modules. In `RESPONSE_MANAGER` the fact that a response must be sent is represented by the fact that the program is executing the "`await immediate LINE_FREE`" statement, so testing for it involves no run-time overhead (it is compiled in the automaton state switch). On the other hand, in `RESYNCH_MANAGER`, the same information is recorded in the boolean variable `POLL_SENT`, which has to be tested at run-time each time a polling frame might be sent. In this case, the overhead is acceptable because we only enter `RESYNCH_MANAGER` when a timeout occurs, and the emission of normal frames is inhibited anyhow at that time. On the other hand, response frames can be sent at any time, so this overhead would not have been reasonable in `RESPONSE_MANAGER`.

The third point involves more technical programming tricks. For example, in the `RESPONSE_MANAGER` module in version 4, changing the `every` statements we used to `await`s would increase the number of states of the module from 3 to 5. The module would now "remember" whether it had done one of the `CHECKPOINTING:=true` or `RESPONSE_TYPE:=REJ` assignments. This is not very useful, since redoing the assignments is harmless. Such considerations may seem a little

arcane, but they can be critical for the efficient use of the current Esterel systems, just as avoiding right-recursive rules can be critical for LALR(1) parser generators.

All these optimizations become secondary when cascade compiling is used, as they affect only a few bytes then. The best way to proceed in Esterel is to compile separately each leaf module, and then try to identify the states produced by the compiler. According to the modularity principle, each should be small, so the task should be reasonable. Time-space compromises, cascade compiling, and refining inter-module synchronizations can then considered if the total size is too large.

10. Conclusion

We have shown how Esterel's synchronous parallelism can be used to write an elegant, modular, and safe program for the HDLC data transfer. However, the final program which we presented here is only part of the story. The development of the program was considerably more complex than its final code or even the detailed comment in this paper might suggest. This complexity seems to be inherent to Esterel programming, for several reasons:

- The reactive systems that Esterel is meant to describe are generally both intricate and poorly (sometimes inconsistently) specified. Thus they are *never* easy to program, in any language whatsoever.
- Esterel has powerful control and communication structures that are used not only to implement the external behavior of the system, but also to express the *logic* of the algorithm that generates it. Often two constructions will generate *exactly* the same code but have completely different structures. Such flexibility is unthinkable in most other languages, where one has to be content with implementing as simply as possible the specified behavior. It adds complexity to program development, since one has constantly to determine the *true* structure of the underlying algorithm.
- Once a program structure has finally been decided on, it has to be revised to take into account efficiency considerations such as code and state size. Thus, even though Esterel programs do not involve explicit states, the states of the program have to be identified to check the efficiency of the compiling process. This is still very different from the more classical state-table specification method: here states are *determined* by the algorithm, not the basis for describing it. Furthermore, the changes for space state optimization tend not to change the global structure of the program. In this case, we only inserted the `BUFFER_FILTER` module and made some local changes to introduce some better-compiled idioms.

In conclusion Esterel trades some programming complexity to get better readability and safety. For applications such as protocols where reliability is at such a high premium, this is certainly an excellent tradeoff.

References

1. BERRY, G. AND GONTHIER, G., "The Synchronous Programming Language ESTEREL: Design, Semantics, Implementation," INRIA report 842 (1988). to be published in Science of Computer Programming
2. BERRY, G.; COURONNE, P.; AND GONTHIER, G., "Synchronous Programming of Reactive Systems: an Introduction to ESTEREL," pp. 35-55 in *Programming of Future Generation Computers*, ed. K. Fuchi and M. Nivat, Elsevier Science Publisher B.V. (North Holland) (1988). INRIA report 647
3. BERRY, G.; BOUSSINOT, F.; COURONNE, P.; AND GONTHIER, G., "ESTEREL System Manuals," Collection of Technical Reports, Ecole des Mines / INRIA, Sophia-Antipolis (1986).
4. BERRY, G.; BOUSSINOT, F.; COURONNE, P.; AND GONTHIER, G., "ESTEREL Programming Examples," Collection of Technical Reports, Ecole des Mines / INRIA, Sophia-Antipolis (1986).
5. GONTHIER, G., "Sémantiques et modèles d'exécution des langages réactifs synchrones; application à ESTEREL," *Thèse d'Informatique*, Université d'Orsay (1988).
6. SCHWARTZ, M., *Telecommunication Networks: Protocols, Modeling, and Analysis*, Addison-Wesley (1987).

Annex 1 — HDLC Version 1: Instantaneous Line With No Error Recovery

1. The Main Module

```
module HDLC:
```

The user and time interface:

```
type MESSAGE;
input USER_INPUT (MESSAGE);
output X_ON, X_OFF;
output USER_OUTPUT (MESSAGE);
input MS;
```

The line interface:

```
type FRAME, NUMBER;
```

The output line interface:

```
output FRAME (FRAME);
input END_FRAME;
```

The input line interface:

```
input I_line (MESSAGE), N_S_line (NUMBER),
      N_R_line (NUMBER),
      P_line, F_line,
      RR_line, REJ_line;
relation I_line => N_S_line, N_S_line => I_line,
         I_line # RR_line # REJ_line,
         P_line # F_line,
         I_line => N_R_line, RR_line => N_R_line, REJ_line => N_R_line,
         P_line => N_R_line, F_line => N_R_line;
relation REJ_line # P_line, % a REJ is always a response
         I_line # F_line;   % option 8
```

The global serializing relation:

```
relation USER_INPUT # MS # N_R_line # END_FRAME;
```

The body:

```
signal N_U (NUMBER), N_A (NUMBER), N_R (NUMBER) in
   copymodule WINDOW_MANAGER
||
   copymodule RECEPTION_HANDLER
||
   copymodule EMISSION_MANAGER
end.
```

2. The WINDOW_MANAGER Module

```
module WINDOW_MANAGER:
```

The user interface:

```
type MESSAGE;
input USER_INPUT (MESSAGE);
output X_ON, X_OFF;
```

The protocol interface:

```
type NUMBER;
input N_R_line (NUMBER);
output N_U (NUMBER), N_A (NUMBER);
```

The serializing relation:

```
relation USER_INPUT # N_R_line;
```

The body:

```
[
   copymodule USER_INPUT_HANDLER
||
   copymodule ACKNOWLEDGE_HANDLER
].
```

2.1. The USER_INPUT_HANDLER Submodule

```
module USER_INPUT_HANDLER:
```

The signal interface:

```
type MESSAGE;
input USER_INPUT (MESSAGE);
output X_ON, X_OFF;
type NUMBER;
input N_A (NUMBER);
output N_U (NUMBER);
relation USER_INPUT # N_A;
```

The data interface:

```
constant WINDOW_SIZE : integer;
constant ZERO : NUMBER;
procedure INCREMENT (NUMBER)();
function SPAN (NUMBER, NUMBER) : integer;
```

The body:

```
var N_U:=ZERO : NUMBER in
   emit N_U (ZERO);
   emit X_ON;
   loop
      await USER_INPUT do
         call INCREMENT (N_U)();
         emit N_U (N_U);
         if SPAN(?N_A,?N_U) = WINDOW_SIZE then
            emit X_OFF;
            await N_A;
            emit X_ON
         end
      end
   end
end.
```

2.2. The ACKNOWLEDGE_HANDLER Submodule

```
module ACKNOWLEDGE_HANDLER:
```

The signal interface:

```
type NUMBER;
input N_R_line (NUMBER);
sensor N_U (NUMBER);
output N_A (NUMBER);
```

The data interface:

```
constant ZERO : NUMBER;
function SPAN (NUMBER, NUMBER) : integer;
```

The body:

```
var N_A:=ZERO : NUMBER in
   emit N_A (ZERO);
   every N_R_line do
      if SPAN (?N_R_line,?N_U) < SPAN (N_A,?N_U) then
         N_A := ?N_R_line;
         emit N_A (N_A)
      end
   end
end.
```

3. The RECEPTION_HANDLER Module

```
module RECEPTION_HANDLER:
```

The signal interface:

```
type MESSAGE;
input I_line (MESSAGE);
output USER_OUTPUT (MESSAGE);
type NUMBER;
sensor N_S_line (NUMBER);
output N_R (NUMBER);
```

The data interface:

```
constant ZERO : NUMBER;
procedure INCREMENT (NUMBER)();
```

The body:

```
var N_R:=ZERO : NUMBER in
   emit N_R (ZERO);
   every I_line do
      if ?N_S_line = N_R then
         emit USER_OUTPUT (?I_line);
         call INCREMENT (N_R)();
         emit N_R (N_R)
      end
   end
end.
```

4. The EMISSION_MANAGER Module

```
module EMISSION_MANAGER:
```

The signal interface:

```
type MESSAGE;
sensor USER_INPUT (MESSAGE);
type NUMBER;
input N_U (NUMBER);
sensor N_R (NUMBER);
type FRAME;
output FRAME (FRAME);
```

The data interface:

```
constant ZERO : NUMBER;
procedure INCREMENT (NUMBER)();
function I_FRAME (NUMBER, NUMBER, boolean, MESSAGE) : FRAME;
```

The body:

```
var N_S:=ZERO : NUMBER in
   every N_U do
      emit FRAME (I_FRAME (N_S, ?N_R, false, ?USER_INPUT));
      call INCREMENT (N_S)()
   end
end.
```

Annex 2 — HDLC Version 2: the Non Instantaneous Line

1. The Main Module

```
module HDLC:
```

The user, time, and line interfaces:

Unchanged

The buffer data type:

```
type BUFFER;
```

The body:

```
signal N_U (NUMBER), N_A (NUMBER), N_R (NUMBER),
       BUFFER (BUFFER) in
   copymodule WINDOW_MANAGER
||
   copymodule RECEPTION_HANDLER
||
   copymodule EMISSION_MANAGER
end.
```

2. The WINDOW_MANAGER Module

```
module WINDOW_MANAGER:
```

The user interface:

Unchanged

The protocol interface:

```
type NUMBER;
input N_R_line (NUMBER);
output N_U (NUMBER), N_A (NUMBER);
type BUFFER;
output BUFFER (BUFFER);
```

The serializing relation:

Unchanged

The body:

```
[
   copymodule USER_INPUT_HANDLER
||
   copymodule ACKNOWLEDGE_HANDLER
||
   copymodule BUFFER_MANAGER
].
```

2.1. The USER_INPUT_HANDLER Submodule

Unchanged

2.2. The ACKNOWLEDGE_HANDLER Submodule

Unchanged

2.3. The BUFFER_MANAGER Submodule

```
module BUFFER_MANAGER:
```

The signal interface

```
type MESSAGE;
sensor USER_INPUT (MESSAGE);
type NUMBER;
input N_U (NUMBER), N_A (NUMBER);
relation N_U # N_A;
type BUFFER;
output BUFFER (BUFFER);
```

The data interface:

```
constant WINDOW_SIZE : integer;
function EMPTY_BUFFER (integer) : BUFFER;
procedure ENQUEUE (BUFFER) (MESSAGE);
procedure PURGE (BUFFER) (integer);
function SPAN (NUMBER, NUMBER) : integer;
```

The body:

```
var BUFFER:=EMPTY_BUFFER(WINDOW_SIZE) : BUFFER in
   loop
      emit BUFFER (BUFFER);
      await
         case N_U do
            call ENQUEUE (BUFFER)(?USER_INPUT)
         case N_A do
            call PURGE (BUFFER)(SPAN (?N_A,?N_U))
      end
   end
end.
```

3. The RECEPTION_HANDLER Module

Unchanged

4. The EMISSION_MANAGER Module

```
module EMISSION_MANAGER:
```

The interface:

```
type BUFFER;
sensor BUFFER (BUFFER);
type NUMBER;
input N_U (NUMBER);
sensor N_R (NUMBER);
type FRAME;
output FRAME (FRAME);
input END_FRAME;
relation N_U # END_FRAME;
```

The body:

```
signal TEST_LINE, LINE_FREE in
   copymodule I_FRAME_MANAGER
||
   copymodule LINE_STATE
end.
```

4.1. The I_FRAME_MANAGER Submodule

```
module I_FRAME_MANAGER:
```

The signal interface:

```
type BUFFER;
sensor BUFFER (BUFFER);
type NUMBER;
input N_U (NUMBER);
sensor N_R (NUMBER);
output TEST_LINE;
input LINE_FREE;
type FRAME;
output FRAME (FRAME);
```

The data interface:

```
constant ZERO : NUMBER;
procedure INCREMENT (NUMBER) ();
type MESSAGE;
function ACCESS (BUFFER, integer) : MESSAGE;
function SPAN (NUMBER, NUMBER) : integer;
function I_FRAME (NUMBER, NUMBER, boolean, MESSAGE) : FRAME;
```

The body:

```
var N_S:=ZERO : NUMBER in
   loop
      trap SEND_FRAME in
         loop
            await
               case N_U do
                  emit TEST_LINE;
                  present LINE_FREE then exit SEND_FRAME end
               case LINE_FREE do
                  if N_S <> ?N_U then exit SEND_FRAME end
            end
         end
      handle SEND_FRAME do
         emit FRAME (I_FRAME (N_S, ?N_R, false,
                              ACCESS (?BUFFER, SPAN(N_S,?N_U))));
         call INCREMENT (N_S)()
      end
   end
end.
```

4.2. The LINE_STATE Submodule

```
module LINE_STATE:
```

The interface:

```
type FRAME;
input FRAME (FRAME), END_FRAME;
input TEST_LINE;
output LINE_FREE;
```

The body:

```
loop
   trap LINE_BUSY in
      await immediate FRAME do exit LINE_BUSY end
   ||
      loop emit LINE_FREE each TEST_LINE
   handle LINE_BUSY do
      await END_FRAME
   end
end.
```

Annex 3 — HDLC Version 3: Checkpoint Answering

1. The Main Module

Unchanged

2. The WINDOW_MANAGER Module

Unchanged

3. The RECEPTION_HANDLER Module

Unchanged

4. The EMISSION_MANAGER Module

```
module EMISSION_MANAGER:
```

The interface:

```
type BUFFER;
sensor BUFFER (BUFFER);
type NUMBER;
input N_U (NUMBER);
sensor N_R (NUMBER);
input P_line;
type FRAME;
output FRAME (FRAME);
input END_FRAME;
relation N_U # P_line # END_FRAME;
```

The body:

```
signal TEST_LINE, LINE_FREE, I_LINE_FREE in
   copymodule I_FRAME_MANAGER [signal I_LINE_FREE / LINE_FREE]
||
   signal PREEMPT_LINE in
      copymodule LINE_FILTER [signal I_LINE_FREE / NORMAL_LINE_FREE]
   ||
      copymodule RESPONSE_MANAGER
   end
||
   copymodule LINE_STATE
end.
```

4.1. The I_FRAME_MANAGER Submodule

Unchanged

4.2. The LINE_FILTER Submodule

```
module LINE_FILTER:
```

The interface:

```
input LINE_FREE, PREEMPT_LINE;
output NORMAL_LINE_FREE;
```

The body:

```
every LINE_FREE do
   present PREEMPT_LINE else emit NORMAL_LINE_FREE end
end.
```

4.3. The RESPONSE_MANAGER Submodule

```
module RESPONSE_MANAGER:
```

The signal interface:

```
input P_line;
output TEST_LINE, PREEMPT_LINE;
input LINE_FREE;
type NUMBER;
sensor N_R (NUMBER);
type FRAME;
output FRAME (FRAME);
```

The data interface:

```
type S_FRAME_TYPE;
constant RR_RESPONSE : S_FRAME_TYPE;
function S_FRAME (S_FRAME_TYPE, NUMBER, boolean) : FRAME;
```

The body:

```
every P_line do
   emit TEST_LINE;
   await immediate LINE_FREE do
      emit PREEMPT_LINE;
      emit FRAME (S_FRAME (RR_RESPONSE, ?N_R, true))
   end
end.
```

4.4. The LINE_STATE Submodule

Unchanged

Annex 4 — HDLC Version 4: Using *REJ* Frames

1. The Main Module

```
module HDLC:
```

The interface:

Unchanged

The body:

```
signal N_U (NUMBER), N_A (NUMBER), N_R (NUMBER),
       BUFFER (BUFFER),
       RETRANSMIT, SEND_REJECT in
   copymodule WINDOW_MANAGER
||
   copymodule RECEPTION_HANDLER
||
   copymodule REJECT_HANDLER
||
   copymodule EMISSION_MANAGER
end.
```

2. The WINDOW_MANAGER Module

Unchanged

3. The RECEPTION_HANDLER Module

Unchanged

4. The REJECT_HANDLER Module

```
module REJECT_HANDLER:
```

The interface:

```
type NUMBER;
input REJ_line;
sensor N_R_line (NUMBER), N_A (NUMBER), N_U (NUMBER);
output RETRANSMIT;
type MESSAGE;
input I_line (MESSAGE), N_R (NUMBER);
output SEND_REJECT;
```

The body:

```
[
   every REJ_line do
      if not ((?N_R_line <> ?N_A) or (?N_A = ?N_U))
      then emit RETRANSMIT end
   end
||
   loop
      await I_line do
         present N_R else emit SEND_REJECT end
      end
   each N_R
].
```

5. The EMISSION_MANAGER Module

```
module EMISSION_MANAGER:
```

The interface:

```
type BUFFER;
sensor BUFFER (BUFFER);
type NUMBER;
input N_U (NUMBER), N_A (NUMBER);
sensor N_R (NUMBER);
input P_line, SEND_REJECT, RETRANSMIT;
type FRAME;
output FRAME (FRAME);
input END_FRAME;
relation N_U # P_line # END_FRAME,
         N_U # SEND_REJECT # END_FRAME,
         N_U # RETRANSMIT # END_FRAME;
relation P_line # RETRANSMIT,       % from REJ_line # P_line
         SEND_REJECT # RETRANSMIT; % from REJ_line # I_line
```

The body:

Unchanged

5.1. The I_FRAME_MANAGER Submodule

```
module I_FRAME_MANAGER:
```

The signal interface:

```
type BUFFER;
sensor BUFFER (BUFFER);
type NUMBER;
input N_U (NUMBER), N_A (NUMBER);
sensor N_R (NUMBER);
output TEST_LINE;
input LINE_FREE;
type FRAME;
output FRAME (FRAME);
input RETRANSMIT;
relation N_U # N_A, N_U # RETRANSMIT;
```

The data interface:

```
constant ZERO : NUMBER;
procedure INCREMENT (NUMBER) ();
procedure GUARD_UNDERFLOW (NUMBER) (NUMBER,NUMBER);
type MESSAGE;
function ACCESS (BUFFER, integer) : MESSAGE;
function SPAN (NUMBER, NUMBER) : integer;
function I_FRAME (NUMBER, NUMBER, boolean, MESSAGE) : FRAME;
```

The body:

```
var N_S:=ZERO : NUMBER in
   loop
      trap SEND_FRAME in
         signal TRY_TO_SEND in
            loop
               await
                  case N_U do emit TRY_TO_SEND
                  case RETRANSMIT do N_S := ?N_A; emit TRY_TO_SEND
                  case LINE_FREE do
                     if N_S <> ?N_U then exit SEND_FRAME end
                  case N_A do
                     call GUARD_UNDERFLOW (N_S) (?N_A, ?N_U)
               end
            end
         ||
            every TRY_TO_SEND do
               emit TEST_LINE;
               present LINE_FREE then exit SEND_FRAME end
            end
         end
      handle SEND_FRAME do
         emit FRAME (I_FRAME (N_S, ?N_R, false,
                              ACCESS (?BUFFER, SPAN(N_S,?N_U))));
         call INCREMENT (N_S)()
      end
   end
end.
```

5.2. The LINE_FILTER Submodule

Unchanged

5.3. The RESPONSE_MANAGER Submodule

```
module RESPONSE_MANAGER:
```

The signal interface:

```
input P_line, SEND_REJECT;
output TEST_LINE, PREEMPT_LINE;
input LINE_FREE;
type NUMBER;
sensor N_R (NUMBER);
type FRAME;
output FRAME (FRAME);
```

The data interface:

```
type S_FRAME_TYPE;
constant RR_RESPONSE, REJ : S_FRAME_TYPE;
function S_FRAME (S_FRAME_TYPE, NUMBER, boolean) : FRAME;
```

The body:

```
signal SEND_RESPONSE in
   every P_line do emit SEND_RESPONSE end
||
   every SEND_REJECT do emit SEND_RESPONSE end
||
   loop
      var RESPONSE_TYPE:=RR_RESPONSE : S_FRAME_TYPE,
          CHECKPOINTING:=false : boolean in
         trap SEND_FRAME in
            every P_line do CHECKPOINTING := true end
         ||
            every SEND_REJECT do RESPONSE_TYPE := REJ end
         ||
            await SEND_RESPONSE do
               emit TEST_LINE;
               await immediate LINE_FREE do exit SEND_FRAME end
            end
         handle SEND_FRAME do
            emit PREEMPT_LINE;
            emit FRAME (S_FRAME (RESPONSE_TYPE, ?N_R, CHECKPOINTING))
         end
      end
   end
end.
```

5.4. The LINE_STATE Submodule

Unchanged

Annex 5 — HDLC Version 5: Error Recovery on Timeouts

1. The Main Module

Unchanged

2. The WINDOW_MANAGER Module

Unchanged

3. The RECEPTION_HANDLER Module

Unchanged

4. The REJECT_HANDLER Module

Unchanged

5. The EMISSION_MANAGER Module

```
module EMISSION_MANAGER:
```

The interface:

```
type BUFFER;
sensor BUFFER (BUFFER);
type NUMBER;
input N_U (NUMBER), N_A (NUMBER);
sensor N_R (NUMBER);
input P_line, F_line, SEND_REJECT, RETRANSMIT;
type FRAME;
output FRAME (FRAME);
input END_FRAME;
input MS;
relation N_U # N_A # END_FRAME # MS,
         N_U # P_line # F_line # END_FRAME # MS,
         N_U # SEND_REJECT # END_FRAME # MS,
         N_U # RETRANSMIT # END_FRAME # MS;
relation P_line # RETRANSMIT,       % from REJ_line # P_line
         SEND_REJECT # RETRANSMIT, % from REJ_line # I_line
         F_line # SEND_REJECT;      % from F_line # I_line
```

The body:

```
signal TEST_LINE, LINE_FREE, RESYNCH_LINE_FREE, I_LINE_FREE,
       BUFFER_EMPTY in
   copymodule BUFFER_STATE
||
   signal ACTIVE_RETRANSMIT, ACTIVE_LINE_FREE in
      copymodule BUFFER_FILTER [signal I_LINE_FREE / LINE_FREE]
   ||
      copymodule I_FRAME_MANAGER
         [signal ACTIVE_RETRANSMIT / RETRANSMIT,
                 ACTIVE_LINE_FREE / LINE_FREE]
   end
||
   signal PREEMPT_LINE in
      copymodule LINE_FILTER [signal I_LINE_FREE / NORMAL_LINE_FREE,
                                      RESYNCH_LINE_FREE / LINE_FREE]
   ||
      copymodule TIMEOUT_MANAGER [signal RESYNCH_LINE_FREE / LINE_FREE]
   end
||
   signal PREEMPT_LINE in
      copymodule LINE_FILTER [signal RESYNCH_LINE_FREE / NORMAL_LINE_FREE]
   ||
      copymodule RESPONSE_MANAGER
   end
||
   copymodule LINE_STATE
end.
```

5.1. The BUFFER_STATE Submodule

```
module BUFFER_STATE:
```

The interface:

```
type NUMBER;
input N_U (NUMBER), N_A (NUMBER);
output BUFFER_EMPTY;
```

The body:

```
loop
   trap BUFFER_EMPTY in
      emit BUFFER_EMPTY;
      await N_U;
      every N_A do
         if ?N_A = ?N_U then exit BUFFER_EMPTY end
      end
   end
end.
```

5.2. The BUFFER_FILTER Submodule

```
module BUFFER_FILTER:
```

The interface:

```
type NUMBER;
input N_U (NUMBER), BUFFER_EMPTY;
input RETRANSMIT, LINE_FREE;
output ACTIVE_RETRANSMIT, ACTIVE_LINE_FREE;
```

The body:

```
loop
   await N_U do
      every immediate RETRANSMIT do emit ACTIVE_RETRANSMIT end
   ||
      every immediate LINE_FREE do emit ACTIVE_LINE_FREE end
   end
each BUFFER_EMPTY.
```

5.3. The I_FRAME_MANAGER Submodule

Unchanged

5.4. The LINE_FILTER Submodule

Unchanged

5.5. The TIMEOUT_MANAGER Submodule

```
module TIMEOUT_MANAGER:
```

The signal interface:

```
type NUMBER;
input N_U (NUMBER), BUFFER_EMPTY;
input N_A (NUMBER), MS;
output TEST_LINE, PREEMPT_LINE;
input LINE_FREE;
type FRAME;
sensor N_R (NUMBER);
output FRAME (FRAME);
input F_line;
output RETRANSMIT;
```

The data interface:

```
constant ACKNOWLEDGE_TIME_LIMIT, RESYNCH_TIME_LIMIT : integer;
```

The body:

```
loop
   % no timer while buffer is empty
   await N_U;
   % now messages should get acknowledged regularly
   loop
      do
         await N_A % wait for a new acknowledge
      watching ACKNOWLEDGE_TIME_LIMIT MS
      timeout
         % repeat resynch attempts, until one succeeds
         trap RESYNCHRONIZED in
            loop
               copymodule RESYNCH_MANAGER; exit RESYNCHRONIZED
            each RESYNCH_TIME_LIMIT MS
         % resume transmission when counters are resynchronized
         handle RESYNCHRONIZED do emit RETRANSMIT
         end
      end
   end
each BUFFER_EMPTY.
```

5.5.1. The RESYNCH_MANAGER Submodule

```
module RESYNCH_MANAGER:
```

The signal interface:

```
output TEST_LINE, PREEMPT_LINE;
input LINE_FREE;
type NUMBER, FRAME;
sensor N_R (NUMBER);
output FRAME (FRAME);
input F_line;
```

The data interface:

```
type S_FRAME_TYPE;
constant RR_COMMAND : S_FRAME_TYPE;
function S_FRAME (S_FRAME_TYPE, NUMBER, boolean) : FRAME;
```

The body:

```
var POLL_SENT:=false : boolean in
   % hold the line until a response occurs
   % send out a polling frame on the first occasion
   do
      emit TEST_LINE;
      every immediate LINE_FREE do
         emit PREEMPT_LINE; % take the line
         if not POLL_SENT then
             emit FRAME (S_FRAME (RR_COMMAND, ?N_R, true));
             POLL_SENT := true
         end
      end
   watching F_line
   % cause a free line to be signaled to I_FRAME_MANAGER
   timeout emit TEST_LINE end
end.
```

5.6. The RESPONSE_MANAGER Submodule

Unchanged

5.7. The LINE_STATE Submodule

Unchanged

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique