Development and Structure of an X.25 Implementation

GREGOR V. BOCHMANN AND TANKOANO JOACHIM

Abstract—**This paper describes experience with an implementation of the X.25 communication protocols for accessing public data networks. The implementation effort is characterized by: 1) the development of a formalized protocol specification on which all further implementation work is based, and 2) the use of Concurrent Pascal as the implementation language. The main features of the formalized protocol specification are given, and a method for deriving a protocol implementation based on parallel processes, *monitors*, and *classes* is explained. The overall structure of the system and the step-wise refinements leading to the complete implementation are discussed. Some comments on the possible implementation on multiple microprocessors are also given.**

Index Terms—**Communications software, Concurrent Pascal, formal specification, process structuring, protocol implementation, step-wise refinement, structured programming, X.25 protocol.**

I. Introduction

X.25 [1] is a standard access protocol for using virtual circuits (VC's) provided by public data networks. This paper describes certain aspects of the experience gained from the implementation of this protocol in a host computer [2]. For the implementation of most communication protocols, the following points must be considered:

1) ensuring the compatibility of the implementation with the remote communication partner,

2) implementing several parallel activities, which is usual for real-time systems, and

3) a step-wise refinement of the system design, which is a useful discipline for any software development project.

We have used a high-level implementation language [3] which provides the concepts of abstract data types (i.e., *class*), parallel processes, and *monitors* (for process interaction). These concepts support points 2) and 3) above. In view of point 1), we have used a formalized specification of the X.25 protocol. Part of our project was the development of this specification. More precise and more algorithmical in nature than the original specification of the protocol, given in natural language, it has been used as the basis for deriving the implementation in a more or less straightforward manner, as described in Section III.

Section II describes the main features of the formalized X.25 specification as used in our project. (The complete specification is contained in [2].) Section III explains how such a formalized specification may be transformed into an implementation, taking one component of the X.25 link level as an example. In Section IV, we describe the overall structure of our X.25 implementation as far as the organization of parallel activity is concerned, and the interfaces between the different system parts, including the user of the VC communication facility provided. In Section V, we make some remarks on the step-wise refinement of our system, and discuss in some detail the problems of buffer management and message coding. We finish with some general conclusions from our implementation experience. The complete text of our formalized specification of X.25, and its implementation in Concurrent Pascal, is contained in [2].

We assume in the following some familiarity with the X.25 protocol [1], the concepts of classes, processes, and monitors as realized in Concurrent Pascal [3], and the unified protocol specification method of Bochmann and Gecsei [4].

II. A Formalized Specification of X.25

The X.25 specification contains three procedure layers:

1) the physical layer, specifying bit transmission between the subscriber and network equipments,

2) the link layer, specifying frame formats, transmission error detection, and error recovery procedures, and

3) the packet layer, specifying packet formats and procedures for the use of VC's.

A basic decomposition of the X.25 protocol is shown in Fig. 1, where the different modules communicate by exchanging packets or frames, respectively. The *VC control* modules implement the packet level procedures separately for each VC, and the *Packet sender* and *receiver* modules implement the link level procedures. These procedures have been considered for the formalized specification. The other modules of Fig. 1 have essentially a (de-) multiplexing function, and are relatively simple. The *Frame input* and *output* modules also handle transmission error detection and transparency coding, as well as physical input/output. We note that the X.25 link level (we consider the original LAP A standard [1]) distinguishes primary and secondary functions which, relatively independent of one another, perform the sending and receiving of frames,

Manuscript received July 12, 1978; revised February 16, 1979. This work was performed at the Université de Montréal, P.Q., Canada, and was supported by the Ministère de l'Education du Québec and the Canadian International Development Agency.

G. V. Bochmann is with the Département d'I.R.O., Université de Montréal, Montreal, P.Q., Canada. In 1978 he was on leave at the Département de Mathématiques, Ecole Polytechnique Fédérale, Lausanne, Switzerland.

T. Joachim is with the Centre National du Traitement de l'Information, Upper Volta.

0098-5589/79/0900-0429$00.75

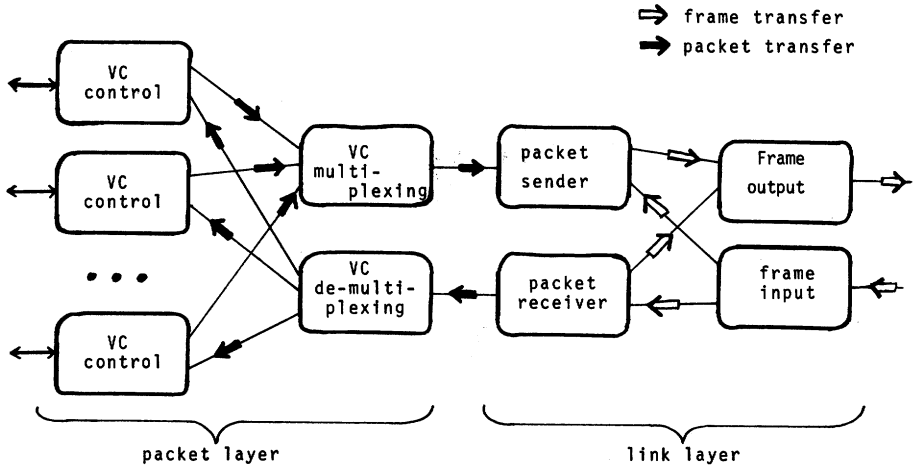


Fig. 1. Decomposition of an X.25 implementation into modules interacting by exchange of messages.

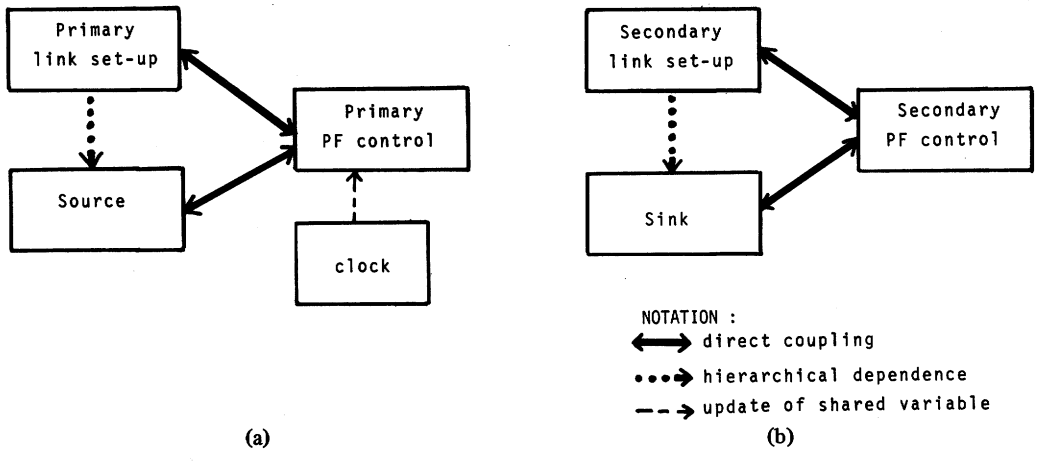


Fig. 2. (a) Component structure of the *Packet sender* module. (b) Component structure of the *Packet receiver* module.

respectively. This is reflected by separate *Packet sender* and *receiver* modules.

A. The Link Layer

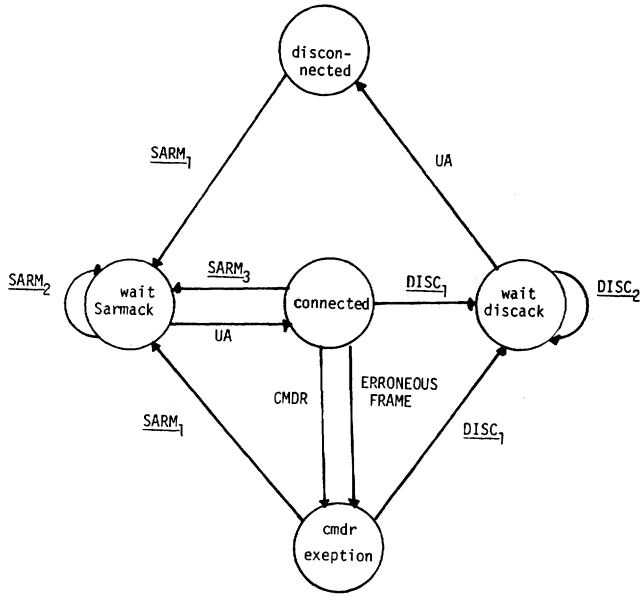
The link level procedures describe a particular class of HDLC procedures. A formalized specification of HDLC procedures, in general, has been described elsewhere [5]. Our formalized specification of the X.25 link level is based, as far as possible, on that specification, and therefore uses the same specification formalism.

The HDLC procedures may be considered [5] to be composed of several different interrelated components, as shown in Fig. 2. The link between the computer and the network is set up (and disconnected) separately for each direction of frame transmission by the *Link setup* components. The *Source* and *Sink* components perform the frame transmission during the *connected* state; and the *PF control* components determine the exchange of poll/final (PF) bits [1]. The *Clock* component provides a time-out mechanism for retransmission.

In the formalized specification, each component is characterized by program variables, a transition diagram, and enabling predicates and actions for each transition. All transitions exclude one another in time, and a given transition may only be executed when its enabling predicate, which depends on the variables, is true. When executed, the transition action may update the variables and thus enable or disable other transitions of the same and other components (for more detail, see [4]). As an example, we show in Fig. 3 the specification of the *Primary link setup* component. The transition diagram of Fig. 3(a) shows the possible transitions. Fig. 3(c) shows, for each transition, when it may be executed and what its action is. Enabling predicates, as well as actions, may involve variables of other components, which are written in the form "<component name>.<variable name>". The local variables of the *Link setup* component are listed in Fig. 3(b).

There are certain differences between our formalized specification of the X.25 link level procedures and the specification of HDLC given in [5]. They may be attributed to the following two factors.

1) The X.25 procedures operate in a particular configuration including a primary and a secondary station, and in asynchronous response mode only.



(a)

```
variables

ERRCOUNT:  integer;

HIGHLEVEL:  interface of Link manager;
   CONNECT:  booléan;
   DISCONNECT:  booléan;
   REPORTCMDR;
   ERROR;
```

(b)

TRANSITION	ENABLING PREDICATE	ACTION	MEANING
$\underline{SARM_1}$	HIGHLEVEL.CONNECT	ERRCOUNT:=0 PFCONTROLPRIMARY.BIT:=1 INIT (TRANSMIT,SARM); *Send* (TRANSMIT);	invites the DCE to establish the link
$\underline{SARM_3}$	LINKSOURCE.ERRCOUNT=MAXERRCOUNT	- idem -	- idem - (case of retransmission)
$\underline{SARM_2}$	LINKCLOCK.TIMEOUT ∧ ERRCOUNT < MAXERRCOUNT	ERRCOUNT:=ERRCOUNT + 1; PFCONTROLPRIMARY.BIT:=1; INIT(TRANSMIT,SARM); *Send*(TRANSMIT);	- idem -
$\underline{DISC_1}$	HIGHLEVEL.DISCONNECT	ERRCOUNT:=0 PFCONTROLPRIMARY.BIT:=1 INIT(TRANSMIT,DISC); *Send*(TRANSMIT);	invites the DCE to disconnect the link
$\underline{DISC_2}$	LINKCLOCK.TIMEOUT ∧ ERRCOUNT < MAXERRCOUNT	ERRCOUNT.=ERRCOUNT + 1; PFCONTROLPRIMARY.BIT:=1; INIT(TRANSMIT,DISC); *Send*(TRANSMIT);	- idem - (case of retransmission)
UA	RECEIVED.KIND=UA ∧ RECEIVED.FBIT = 1	LINKSOURCE.*initialisation*;	initializes the LINKSOURCE component
CMDR	RECEIVED.KIND = CMDR	HIGHLEVEL.REPORTCMDR;	a frame has been rejected by the DCE
ERRONEOUSFRAME	RECEIVED.KIND =ERRONEOUSFRAME	HIGHLEVEL.ERROR;	an erroneous frame has been received

(c)

Fig. 3. Specification of the *Primary link setup* component. (a) Transition diagram (underlined transition names indicate a sending transition; nonunderlined names a receiving transition). (b) Local variables. (c) Definition of the transitions.

2) One objective of the specifications in [5] was to include only those aspects that are necessary to ensure the compatibility between the communicating system parts. For the X.25 specification, we have included additional aspects, not essential for compatibility. These aspects include points described in the standard, points adopted for the subscriber equipment by analogy with the specifications for the network equipment, and an interface to a higher level link manager module.

A comparison between the two formalized specifications may be made comparing Fig. 3(c) and (d). Finally, Fig. 3(e)

TRANSITION	ENABLING PREDICATE	ACTION	MEANING
SARM	PF-control.bit = 1	send-unnumbered (SARM) ;	
UA	received.kind = UA	init (source) ; init (sink) ; init (transmission)	initialize the source and sink components
DISC	PF-control.bit = 1	send-unnumbered (DISC)	
CMDR	received.kind = CMDR	init (transmission) ;	
ERROR	status ∈ [invalid-control-field invalid-info, invalid-size, invalid-NR]	init (transmission) ;	frame received contained an error to be resolved by a higher level recovery procedure at Primary

(d)

2.3.4.5 *Set Asynchronous Response Mode (SARM) Command*

The SARM unnumbered command is used to place the addressed secondary in the Asynchronous Response Mode (ARM).

No information field is permitted with the SARM command. A secondary confirms acceptance of SARM by the transmission at the first opportunity of a UA response. Upon acceptance of this command, the secondary receive state variable is set to zero.

Previously transmitted frames that are unacknowledged when this command is actioned remain unacknowledged.

2.4.3.1 *Link Setup*

The DCE will indicate that it is able to set up the link by transmitting contiguous flags (active channel state).

The DTE shall indicate a request for setting up the link by transmitting a SARM command to the DCE.

Whenever receiving a SARM command, the DCE will return a UA response to the DTE and set its receive state variable V(R) to zero.

Should the DCE wish to indicate a request for setting up the link, or when receiving from the DTE a first SARM command as a request for setting up the link, the DCE will transmit a SARM command to the DTE and start timer T1 (see Section 2.4.7). The DTE will confirm the reception of the SARM command by transmitting a UA response.

When receiving the UA response, the DCE will set its send state variable V(S) to zero and stop its timer T1. If timer T1 runs out before the UA response is received by the DCE, the DCE will retransmit a SARM command and restart timer T1.

After transmission of SARM N2 times by the DCE, appropriate recovery action will be initiated.

The value of N2 is defined in Section 2.4.7.

2.3.5.6 *Rejection Condition*

A rejection condition is established upon the receipt of an error-free frame which contains an invalid command/response in the control field, an invalid frame format, an invalid N(R) count, or an information field which exceeded the maximum information field length which can be accommodated.

At the primary this exception is subject to recovery/resolution at a higher function level.

2.4.5.5 If the DCE transmits a CMDR response, it enters the command rejection condition. This command rejection condition is cleared when the DCE receives a SARM or DISC command. Any other command received while in the command rejection condition will cause the DCE to retransmit this CMDR response. The coding of the CMDR response will be as described in Section 2.3.4.8. In the case of an invalid N(S), bits 4, 5, 6, and 7 of octet 3 will be set to zero.

(e)

Fig. 3(cont'd). (d) Definition of the transitions, taken from [5] (the same transition diagram (a) applies, but there are no local variables). (e) Some pieces of text from the X.25 standard; relevant to the *Link setup* component.

shows some pieces of text describing the use of the SARM command (one of the topics relevant to this component) extracted from the standard specification [1].

B. The Packet Layer

We found that the same specification techniques used for the link layer could be easily applied to the description of the packet level procedures. We adopted the decomposition of the layer into the components shown in Fig. 4, with a hierarchical dependence [5] between the different components. The *restart* component is the hierarchically highest component on which all VC's depend; the components of only one VC are shown. A timer component seems to be necessary for a realistic system, although this aspect has been ignored in the standard.

As in the case of the link layer, each component is described by variables, transition diagrams, and transitions. Most of the transition diagrams given in the annex of the standard have been adapted, and completed with an *error* state and corresponding transitions. As an example, we show the transition diagram of the *Reset* component in Fig. 5.

III. Implementation Transformations

We now explain how the formalized protocol specification discussed above may be transformed into an implementation in terms of processes, monitors, and classes. As mentioned above, a system component is characterized by variables, a transition diagram, and enabling predicates and actions for each transition. A straightforward realization of a component could be obtained using conditional critical regions, for which an efficient implementation, however, is not always easy to obtain [6]. We have chosen an implementation pattern where a component is generally implemented by a monitor and some processes. The monitor contains the component variables, a variable representing the state of the transition diagram, and procedures which, when called, effect the component transitions. The processes represent different external events and call these procedures. The transitions of the *Primary link setup* component, for example, are activated by two processes representing the reception and sending of frames over the network access circuit, as shown in Fig. 6.

This implementation approach works for independent components, such as the *Primary* and *Secondary link setup* components of the X.25 link layer. In the case of component dependences, we have adopted the following implementation patterns.

1) Variables shared between several components: the monitor parts of all components are merged into a single monitor to ensure mutual exclusion between the transitions of different components.

2) A component X is hierarchically dependent on a com-

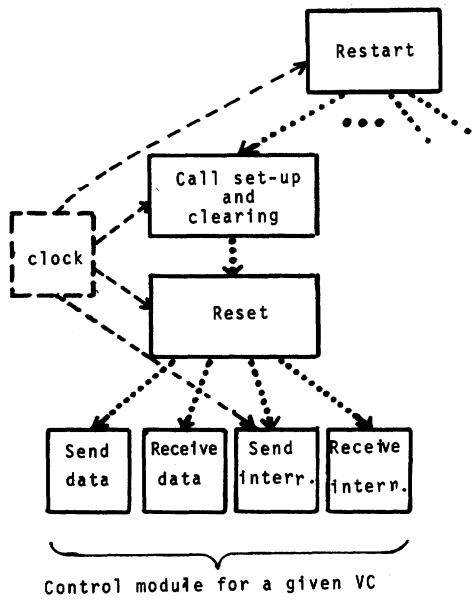


Fig. 4. Component structure of the *VC control* modules (see explanations in Fig. 2).

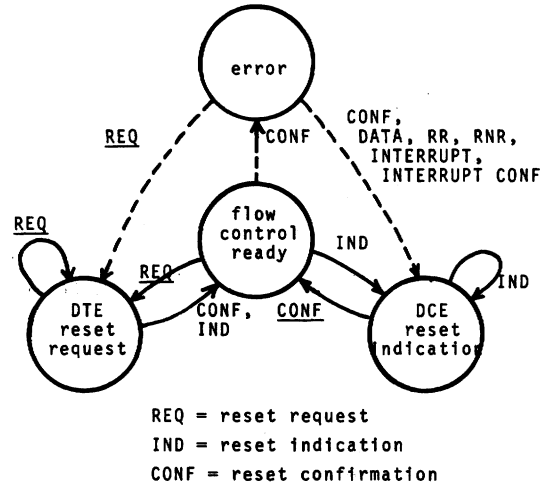


Fig. 5. Transition diagram for the *Reset* component (X.25 packet level).

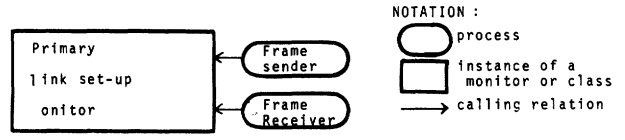


Fig. 6. The *Primary link setup* component realized by a monitor and two processes activating the transitions defined in Fig. 3.

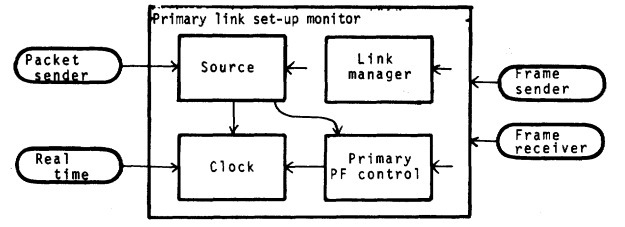


Fig. 7. The realization of the *Packet sender* module in terms of monitors, classes, and processes (see explanation in Fig. 6).

ponent Y (i.e., transitions of X are only possible when Y is in a particular state; see [5]): the monitor part of X is realized as a class declared as local variable or parameter inside the monitor part of Y. The process part of X accesses this class via the monitor part of Y.

3) Two components X and Y are directly coupled (i.e., certain transitions of X may only be executed in parallel with certain transitions of Y; see [5]): the monitor part of one component is realized as a class declared inside the monitor part of the other component, similarly to the case above.

As an example, Fig. 7 shows the inner structure of the packet sender module. In addition to the *Primary link setup* component, already shown in Fig. 6, this figure also shows the realization of the other components of the module (see Fig. 2), and the *Link manager* monitor (see Section IV). To explain the relations shown in the figure, we note that a sending transition, for instance, is activated by the *Frame sender* process calling an operation of the *Primary link setup* monitor. The latter performs a link setup, reset, or disconnection transition, if appropriate (depending on its own state and the *Link manager*), and otherwise calls an operation of the *Source* class which, in turn, may perform a sending transition. Any transition performed is coordinated with the *PF control* class which sets the poll/final bit of the frame to be sent. Appendix A shows the detailed coding of the *Primary link setup* monitor in Concurrent Pascal.

The transformation rules for obtaining a protocol implementation from its formalized specification should be straightforward in order to avoid programming errors. This is the case for the rules discussed so far. However, we found that the following two aspects of the transformation involved more complex decisions, and are therefore more subject to errors.

1) The nondeterminism inherent in the transition diagram must be eliminated, which implies an ordering of the transitions and some rearrangement of the enabling predicates in order to obtain efficient test sequences. The transition actions may also be rearranged in order to eliminate redundancy.

2) To avoid busy waiting in the case when no transition is enabled, a calling process must *wait* in the monitor until another process changes the component state. This change must be *signaled* to the waiting process. It is not always easy to decide when, and to which process, a signal must be sent (for an example, see Appendix A).

An example of nondeterminism is given by the transitions SARM and DISC possible in the *connected* state of the *Primary link setup* component [see Fig. 3(a)]. While the choice between these two transitions is left completely open by the formalized specification of [5] [see Fig. 3(d)], the choice is largely determined by the enabling predicates in our formalized specification [see Fig. 3(c)]. However, a system state is possible for which both transitions are enabled. In our implementation (see the Appendix), we have given a priority to the DISC transition.

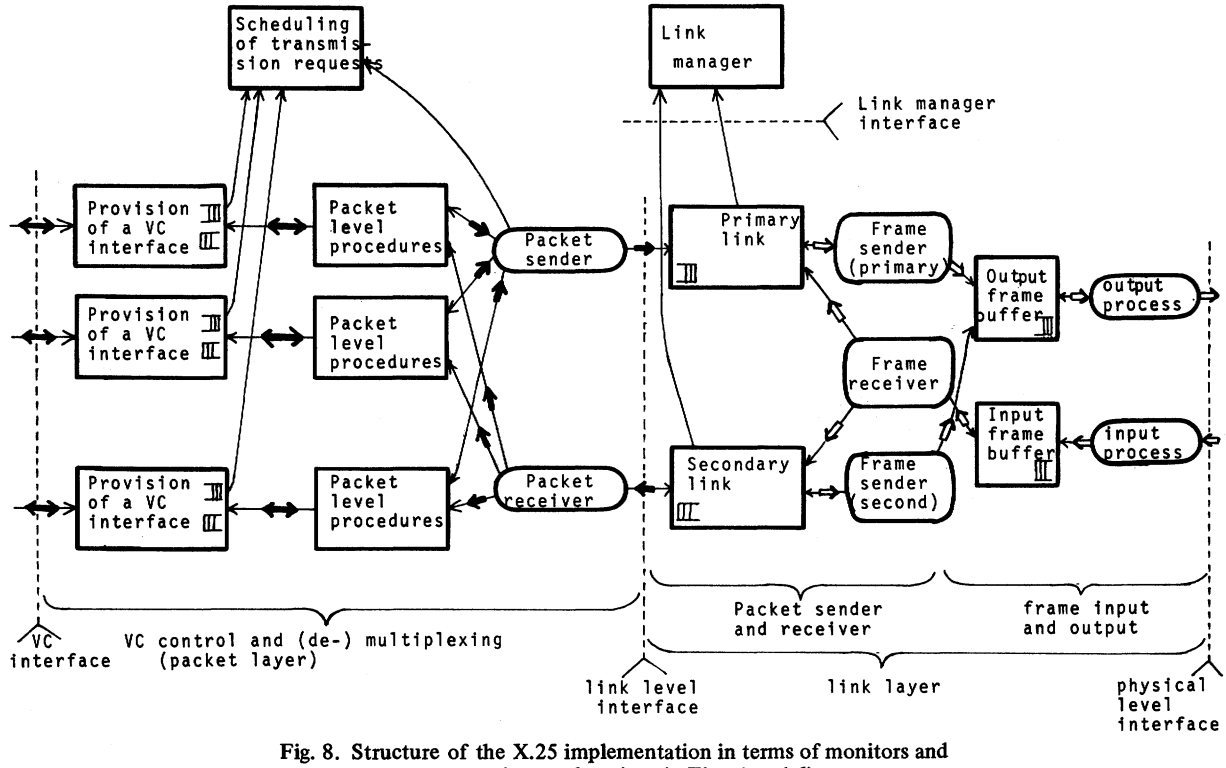


Fig. 8. Structure of the X.25 implementation in terms of monitors and processes (see explanations in Figs. 1 and 6).

IV. The Structure of the X.25 Implementation

The general structure of the X.25 implementation is shown in Fig. 8. The physical layer of X.25 is implemented in the line controller hardware, and is not shown.

The structure of the link layer is obtained by applying the transformations discussed above to the structure of Fig. 1. The three *Frame sender* and *receiver* processes activate the transitions of the primary and secondary link components. The piggybacking of acknowledgments is performed in the *Output frame buffer*, which also performs the multiplexing of frames from the primary and secondary link components over the output circuit. The demultiplexing of incoming frames on to the primary and secondary link components is performed by the *Frame receiver* process. This process activates the receiving transitions of both components. Two separate *receiver* processes could have been used to allow for full parallelism between the sending and receiving of packets. The *Input* and *Output* processes activate the frame input and output, and perform the transmission error detection, frame delimitation, and transparency functions. In our implementation, these functions are mainly realized in software by the Concurrent Pascal system kernel [7] via IO commands executed by the *Input* and *Output* processes. Clearly, these functions would be more efficiently implemented by a separate hardware processor.

The operation of the link layer is supervised by a *Link manager*. It determines whether the link to the network should be established, disconnected, or reset, and coordinates the operation of the primary and secondary components. The latter, in turn, report to the link manager those errors which cannot be recovered by the link level procedures. The interface between the *Link manager* and the *Primary link* component, for instance, is described in Fig. 3(b), and its use is shown in Appendix A.

The interface between the link and packet layers is very simple. It consists of two primitives for sending and receiving a packet, respectively. We note that the calling processes may be delayed due to flow control considerations (see Section V-B below).

The transformation principles described above were also applied to the *VC control* module of the packet level. As in the case of the link layer, a single process, the *Packet receiver* (see Fig. 8), performs the demultiplexing of incoming packets into the different VC's, and activates the receiving transitions of all *VC control* monitors. For the multiplexing of outgoing packets, an approach different from the link layer was adopted. Instead of having independent packet sending processes, one for each VC, a single *Packet sender* process looks after all VC's and receives requests for packet transmission through a *Scheduling* monitor. This monitor is the place where different priorities may be introduced for the different VC's. The control of each VC is partitioned into a module responsible for observing the X.25 packet level procedures, and a module which provides a VC interface to the next higher layers of the computer system. In particular, the latter module provides flow control functions, automatic answering of clear, reset, and interrupt indication packets, and a time-out function for call, clear, and reset requests and interrupts [8].

We have tried to design a reasonable VC interface to the higher layers following the X.25 specifications as closely as possible. The resulting interface may be characterized by the

following primitives:

restart-request
call-request (· · ·)
wait-for-incoming-call (· · ·)
accept-call
clear-request
reset-request
send-interrupt (· · ·)
send-data (· · ·)
receive-data (· · ·)
get-new-status.

Each of these primitives, called by the higher layer, returns VC status information, which includes

1) information about the present state of the interface, such as

restarted by DTE or DCE,
connected by DTE or DCE,
disconnected by DTE or DCE,
reset by DTE or DCE,
interrupt sent by DTE or received from DCE,
time out, i.e., the primitive returned control to the higher level before the system received an appropriate packet from the network (DCE) in response to a request from the system;

2) flow control, i.e., indication that received data are available, or no buffer space is available for sending more data;

3) error indications, such as

procedure errors of the network
invalidity of a request from the higher layer in the present interface state.

V. Step-Wise Refinement and Implementation Choices

A. General Remarks

Our X.25 implementation effort may be considered as an exercise in step-wise refinement. The first step is the establishment of the formalized protocol specification described in Section II. Further steps, some of which are described in Sections III and IV, lead towards the implemented system which is described in full in [2]. In Sections III and IV, we have described the choices that lead from the system structure of Fig. 1, which consists of message-driven modules, the operation of which is described by the formalized protocol specification, to the structure of Fig. 8, which is based on the monitor, class, and process primitives available in the implementation language.

However, there are many more implementation choices to be made. They mainly concern the implementation of classes and monitors for which, so far, only the interfaces have been defined. Examples are the *Link manager* component, which in our system is implemented as a monitor and process interacting with the operator, and the buffer management described below. For both modules, the interface has been used in the formalized protocol specification. A complete list of all program components is given in Appendix B.

Our effort for obtaining the X.25 implementation may be subdivided into the following steps, each of which took about one man month of work:

to derive the formalized specification of the link and packet level procedures (given the specification in [5]),
to design the structure of the system, such as shown in Figs. 7-9 and in Appendix B (this includes the development of the implementation transformations described in Section III),
to write the program components in Concurrent Pascal, and
to test and debug the system.

B. Buffer Management and Flow Control

Buffer queues for the intermediate storage of packets or frames between any pair of cooperating processes have been foreseen in the system as indicated in Fig. 8. These queues control the information flow within the system, and synchronize the relative speeds of the different processes in the system, since a process accessing a queue has to wait until it is not empty or not full respectively. The only exception is the *Input* process which is not delayed when the *Input frame buffer* is full. Instead, the last frame is lost.

In order to avoid unnecessary copying of data packets from one queue to another during the processing of the packets within the system, the frames coming in from the network, as well as the data packets from the higher system layers, are stored within a centrally managed buffer space and subsequently referred to by pointers. Therefore, the information exchanged between the system components shown in Fig. 8 includes these pointers, together with other control information, but not the copies of data packets.

In order to simplify the avoidance of deadlocks, a fixed number of packets or frames, respectively, is allocated as the maximum length for each of the queues. The total space required may be determined according to the equation

$$\left.\begin{array}{l}\text{total number}\\ \text{of blocks}\end{array}\right\} = \sum_i \text{maximum number of blocks in queue } i$$

$$+ \sum_j \text{number of blocks not in a queue and being processed by process } j.$$

The structure of the buffer management facility is shown in Fig. 9, which shows the central buffer manager (a monitor) and the different buffer queues (classes). The queue of the *Primary link* is completed by a class providing additional management facilities needed for packet retransmission. The central buffer manager may also be directly accessed, to obtain a new block, change or read the information stored in a block, or free a block.

C. Message Coding

For compatibility with the remote communication partner, a protocol specifies the exact layout of information fields within the exchanged messages. This message format must be implemented by the communications software, and involves the specification of memory layout of structured data, bit packing, etc. It is not possible to describe these details in a single soft-

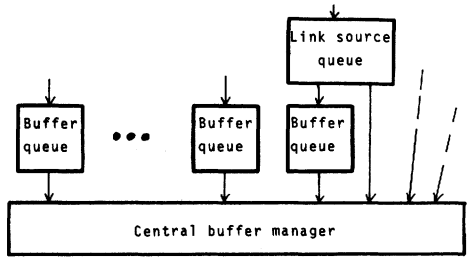


Fig. 9. Structure of the buffer management facility (see explanation in Fig. 6).

ware module, since each protocol layer, separately, specifies the layout of the corresponding message header. An implementation language with facilities for specifying memory layout of packed data structures would be convenient for this purpose.

Our implementation language did not provide this facility; therefore, the coding and decoding of the packet and frame headers are implemented in several different procedures. The central buffer manager provides operations for reading and writing selected octets of a given data block. These operations may also be used by higher level protocols. Specific procedures are included in the *Packet sender* and *receiver* processes (for packet header (de-)coding), and in the *Frame receiver* and *Output frame buffer* (for HDLC header (de-)coding).

VI. Concluding Remarks

A. The Use of a Formalized Protocol Specification

As explained in Sections II and III, we have developed a formalized specification of X.25 which served as the basis for the implementation. We would have appreciated a more formalized specification of the X.25 standard which could have saved us this effort. A formalized protocol specification not only has the advantage of simplifying the implementation, but is also useful during the protocol design, verification, and evaluation phase (see, for example, [9]).

B. The Use of a High-Level Implementation Language

We conclude from our experience that the following properties of the implementation language were most valuable for the project.

1) Facilities for step-wise refinement, in particular, the *class* concept.

2) Facilities for describing parallel activities. We used the *processes* and *monitors* of Concurrent Pascal; however, we would have appreciated a language construct (see, for example, [2]) closely related to the *component* structure described in Section II-A.

3) The facilities for type definition and checking, common to most Pascal-like languages.

Other aspects of our language implementation were not entirely satisfactory, such as, for example, its low efficiency and the inability to interwork with the standard computer operating system.

An advantage of using a high-level implementation language is the reduction of the programming and testing effort required. The testing of each protocol layer was done in two phases. First the system was embedded, on the same computer, in a testing environment, also written in Concurrent Pascal. Second, the system was checked with an X.25 protocol tester equipment which was connected to the computer via the data network access line. Both phases were effective.

We believe that a high-level language implementation such as ours is useful even when the high-level programming language is not implemented on the target computer, or when the efficiency or operating system interfaces of the implementation are insufficient. Efficiency may be increased by reprogramming the critical procedures in machine language, or the whole program may be used as a "blueprint" for an implementation in a suitable language. We note that Belsnes [10] comes to similar conclusions, describing an implementation of X.25 in Simula.

C. The VC Interface

In Section IV, we described in some detail the VC interface, which is the interface between the X.25 network access module and the remaining part of the computer system. In deriving this interface from the X.25 packet level specifications, we were astonished by the great complexity of the resulting interface. We wonder whether an interface to an end-to-end transport service [11] would be simpler in nature. A criterion for the delimitation of major system modules is the simplicity of the resulting interfaces. The experience with our X.25 implementation has not convinced us that the X.25 VC is a natural system interface.

D. Implementation on Multiple Microprocessors

In a microprocessor-based implementation of X.25, the different protocol layers may be distributed over several microprocessors [12], [13]. To avoid memory bus congestion, each microprocessor usually has its own local memory, which contains the program code and processed data, and may exchange messages via a system bus with the other microprocessors in the system. A system described in terms of processes and monitors, such as shown in Fig. 8, is suitable for distribution over a multimicroprocessor system. A possible distribution method, called "split process organization" by Cavers [12], proceeds as follows. First each monitor of the system is allocated to a suitable microprocessor. Then the processes are allocated. Processes accessing the monitors in one microprocessor are allocated to that microprocessor. Processes accessing monitors in more than one microprocessor are split into subprocesses, one for each microprocessor involved and allocated to it. The subprocesses communicate by message exchange via the system bus. This organization is particularly appropriate when most processing in the system is done in the monitors, and the processes have essentially the role of passing information. This is the case in the X.25 system of Fig. 8.

Appendix A

Implementation of the *Primary Link Setup* Component

In the following, we give the details of the *Primary link setup* component as implemented in Concurrent Pascal [3]. The implementation follows the structure of Fig. 7 and is based on the formalized protocol specification given in Fig. 3(a)-(c). The underlying method for deriving the implementation from the formalized specification is explained in Section III.

The *Primary link setup* component is a monitor called by the processes shown in Fig. 7. The monitor has access to the central buffer manager *Buffer* and a typewriter resource *Typuse*, which is used by the link manager *Myoperator* for interacting with the operator. The other parameters of the monitor are constants. The local variables of the monitor include the protocol components shown in Fig. 7, and a link source queue (Section V-B and Fig. 9) *Bufq* which contains the packets to be transmitted. The implementation details of these components, used by the *Primary link setup* component, are not included in the monitor, but are described in separate program components of the Concurrent Pascal implementation.

The *Frame sender* process calls the *Sendevent* operation of the monitor. This operation realizes the SARM and DISC transitions according to the diagram of Fig. 3(c), and certain transitions of the *Source* and *Primary PF control* components. The local monitor variable *State* records the active state of the diagram, and the link manager *Myoperator* is used to decide between different transition possibilities. In the connected state, for instance, the link manager may decide a disconnection, or the *Link setup* component itself may execute a reset (SARM transition) if there were too many unsuccessful retransmissions of information frames. Otherwise, the *Source* component is called upon to transmit an information frame. The *Primary PF control* component, which is directly coupled to the *Link setup* component, is called upon at the end of the operation. The parameter *Transmitframe* of the operation contains information about the frame to be sent. This information is passed onto the *Output frame buffer* (see Fig. 8) where it is coded in the HDLC format.

The *Frame receiver* process calls the *Rcvevent* operation which, similarly, realizes the UA, CMDR, and ERRONEOUS-FRAME transitions according to the diagram of Fig. 3(c), and the reception transitions of the *Source* and *PF control* components. The *Packet sender* process calls the *Usersendevent* operation, which enters a packet into the link source queue, provided the link is not disconnected. This operation, together with a corresponding operation of the *Secondary link* component (see Fig. 8), forms the interface between the link and packet layers of X.25. The *Clockinterrupt* operation is called at regular intervals by the *Real time* process.

LISTING OF THE *PRIMARY LINK SETUP* COMPONENT

```
0009 TYPE LINKSETUPPRIMARY =
0010 MONITOR(BUFFER : SNAPBUFFERTYPE ; TYPEUSE : TYPERESOURCE ;
0011          TIMERT1 : INTEGER ; MAXERRCOUNT : INTEGER ;
0012          BUFLENGTH : INTEGER) ;
0013
0014 VAR
0015
0016 STATE : PRIMARYSTATETYPE ;
0017
0018 ERRCOUNT : INTEGER ;
0019
0020 SENDERQ,USERQ : QUEUE ;
0021
0022 BUFQ : SOURCEQUEING ;
0023
0024 CLOCK : LINKCLOCK ;
0025
0026 PFCONTROL : PFCONTROLPRIMARY ;
0027
0028 SOURCE : LINKSOURCE ;
0029
0030 MYOPERATOR : PRIMARYTERMINAL ;
0031
0032 PROCEDURE EXECSARMACTION(VAR KIND: COMMANDKIND) ;
0033 BEGIN
0034   KIND := SARM ;
0035   STATE := WAITSARMACK ;
0036   ERRCOUNT := 0 ;
0037   PFCONTROL.SETBIT ;
0038 END ;
0039
0040 PROCEDURE EXECDISCACTION(VAR KIND : COMMANDKIND) ;
0041 BEGIN
0042   STATE := WAITDISCACK ;
0043   KIND := DISC ;
0044   ERRCOUNT := 0 ;
0045   PFCONTROL.SETBIT ;
0046 END ;
0047
0048 PROCEDURE ENTRY SENDEVENT(VAR TRANSMITFRAME : COMMANDFRAME) ;
0049 VAR CONTROL: SENDCONTROL ;
0050 BEGIN
0051   WITH TRANSMITFRAME
0052   DO REPEAT
0053      CONTROL := EXIT ;
0054      CASE STATE OF
0055 DISCONNECTED :
0056        IF MYOPERATOR.CONNECT THEN EXECSARMACTION(KIND)
0057        ELSE CONTROL := WAITFOR ;
0058 WAITSARMACK,WAITDISCACK :
0059        IF CLOCK.TIMEOUT
0060        THEN IF ERRCOUNT < MAXERRCOUNT
0061             THEN BEGIN
0062                    ERRCOUNT := ERRCOUNT + 1 ;
0063                    IF STATE = WAITSARMACK
0064                    THEN KIND := SARM ELSE KIND := DISC ;
0065                    PFCONTROL.SETBIT ;
0066                  END
0067             ELSE BEGIN
0068                    STATE := DISCONNECTED ;
0069                    PFCONTROL.RESET ;
0070                    MYOPERATOR.ERROR(INOPERABLECIRCUIT) ;
0071                    CONTROL := TRYAGAIN ;
0072                  END
0073        ELSE CONTROL := WAITFOR ;
0074 PCMDREXEPTION :
0075        IF MYOPERATOR.RESET THEN EXECSARMACTION(KIND)
0076        ELSE EXECDISCACTION(KIND) ;
0077 PCONNECTED :
0078        IF MYOPERATOR.DISCONNECT THEN
0079           EXECDISCACTION(KIND) ELSE
0080        IF SOURCE.ERRCOUNT = MAXERRCOUNT
0081        THEN BEGIN
0082               EXECSARMACTION(KIND) ;
0083               MYOPERATOR.ERROR(RETRANSMISSIONFAIL)
0084             END
0085        ELSE SOURCE.SENDEVENT(TRANSMITFRAME,CONTROL)
0086
0087      END ;
0088      IF CONTROL = WAITFOR THEN DELAY(SENDERQ)
0089   UNTIL CONTROL = EXIT ;
0090   TRANSMITFRAME.PBIT := PFCONTROL.BIT ;
0091   PFCONTROL.SENDEVENT ;
0092   CONTINUE(USERQ)
0093 END ;
0094
0095 PROCEDURE ENTRY RCVEVENT(VAR RECEIVED : RESPONSEFRAME) ;
0096 BEGIN
0097   IF RECEIVED.KIND <> ERRONEOUSRESPONSE
0098   THEN BEGIN
0099      PFCONTROL.VALIDATEFBIT(RECEIVED) ;
0100      IF RECEIVED.KIND IN (.IRR,RR,RNR,REJ.)
0101      THEN SOURCE.VALIDATENR(RECEIVED)
0102   END ;
0103   WITH RECEIVED
0104   DO BEGIN
0105      CASE STATE OF
0106 PDISCONNECTED,PCMDREXEPTION : ;
0107 WAITSARMACK,WAITDISCACK :
0108        IF (KIND = UA) AND (FBIT = 1)
0109        THEN BEGIN
0110               SOURCE.INITIALISATION ;
0111               PFCONTROL.RCVEVENT(0,0,0,FBIT) ;
0112               IF STATE = WAITSARMACK
0113               THEN BEGIN STATE := PCONNECTED ; BUFQ.RESET
0114                    END
0115               ELSE BEGIN STATE := PDISCONNECTED ; BUFQ.CLEAR
0116                    END
0117        END ;
0118 PCONNECTED :
0119        IF KIND = CMDR
0120        THEN BEGIN
0121               STATE := PCMDREXEPTION ;
0122               PFCONTROL.RCVEVENT(0,0,0,FBIT) ;
0123               MYOPERATOR.CMDRREPORT(INFOPOINTER) ;
0124             END ELSE
0125        IF KIND = ERRONEOUSRESPONSE
0126        THEN BEGIN
0127               STATE := PCMDREXEPTION ;
0128               MYOPERATOR.STATUSREPORT(STATUS) ;
0129             END ELSE
0130        IF KIND IN (.IRR,RR,RNR,REJ.)
0131        THEN SOURCE.RCVEVENT(RECEIVED)
0132      END ;
0133      IF INFOPOINTER <> NUL THEN BUFFER.FREE(INFOPOINTER) ;
0134   END ;
0135   CONTINUE(SENDERQ)
0136 END ;
0137
0138 PROCEDURE ENTRY USERSENDEVENT(MESSPTR : SNAPBUFFERINDEX ;
0139                               VAR XSTATE : PRIMARYSTATETYPE) ;
0140 BEGIN
0141   WHILE (STATE <> PDISCONNECTED) AND BUFQ.FULL
0142   DO DELAY(USERQ) ;
0143   IF (STATE <> PDISCONNECTED) AND NOT BUFQ.FULL
0144   THEN BUFQ.INTO(MESSPTR) ;
0145   XSTATE := STATE ;
0146   CONTINUE(SENDERQ)
0147 END ;
0148
0149 PROCEDURE ENTRY CLOCKINTERRUPT ;
0150 VAR OK : BOOLEAN ;
0151 BEGIN
0152   CLOCK.INTERRUPT(OK) ;
0153   IF OK THEN CONTINUE(SENDERQ)
0154 END ;
0155
0156 BEGIN
0157   INIT
0158     BUFQ(BUFFER,BUFLENGTH),CLOCK(TIMERT1),PFCONTROL(CLOCK),
0159     SOURCE(BUFFER,CLOCK,PFCONTROL,BUFQ),MYOPERATOR(TYPEUSE,BUFFER) ;
0160   STATE := PDISCONNECTED ;  ERRCOUNT := 0
0161 END ;
0162
0163
```

APPENDIX B

THE PROGRAM COMPONENTS OF THE X.25 IMPLEMENTATION. THIS IS A COMPLETE LIST OF ALL *CLASSES*, *MONITORS*, AND *PROCESSES* OF THE X.25 SYSTEM

Name	Number of occurrences	Referenced in the paper	Main function
Fifo	several		FIFO queue for scheduling processes waiting for a ressource.
Resource	several		Monitor providing mutual exclusion for resource access.
Type resource	1		Idem, for shared operator's console.
Typewriter	several		Line-oriented text input-output for the operator's console.
Terminal	several		"Typewriter" with shared access to operator's console.
Terminal stream	several		Character stream input-output through "Terminals".
Snap buffer type	1	Fig. 9; sect. 5.2	*Central buffer manager.*
Buffifo	several	Fig. 9; sect. 5.2	*Buffer queue*, FIFO queue of buffer blocks.
Source queing	1	Fig. 9	Augmented *Buffer queue* for packet retransmission.
Circuit send process	1	Fig. 8; sect. 4	*Output process*, performs the physical output of frames.
Circuit rcv process	1	Fig. 8; sect. 4	*Input process*, performs the physical reception of frames.
Circuit send buffer	1	Fig. 8; sect. 4	*Output frame buffer*, also performs the coding of the frame header.
Circuit rcv buffer	1	Fig. 8	*Input frame buffer* (very simple).
Link receiver process	1	Fig. 6,7,8; sect. 4	*Frame receiver*, also performs the decoding of the received frames.
Primary sender process	1	Fig. 6,7,8; sect. 4	*Frame sender* (primary) (very simple).
Secondary sender process	1	Fig. 8; sect. 4	*Frame sender* (secondary) (very simple).
Primary terminal	1	Fig. 7, 8; sect. 4.	*Link manager.*
Secondary terminal	1		
Linkclock	1	Fig. 7	Time-out service for the *Primary link.*
Clock process	1	Fig. 7	*Real time*, activates the time-out facilities for the link and packet level.
PF control primary *	1	Fig. 7; sect. 3	*Primary PF control* of the *Primary link*, sets the poll bit of outgoing frames and checks the final bit of incoming ones.
PF control secondary *	1		Similar, part of the *Secondary link* (Very simple)
Link source *	1	Fig. 7; sect. 3	*Source* component of the *Primary link* performing packet transmission.
Link sink *	1		Performs packet reception in the *Secondary link.*
Link set up primary *	1	Fig. 7, 8; sect 3; appendix A	*Primary link.*
Link set up secondary *	1	Fig. 8	*Secondary link.*
Event monitor	1	Fig. 8; sect. 4	*Scheduling of transmission requests.*
VC sender process	1	Fig. 8; sect. 4	*Packet sender*, passes the packets to be transmitted to the link layer; also codes the packet header.
VC receiver process	1	Fig. 8; sect. 4	*Packet receiver*, distributes the received packets to the different VC's; also decodes the packet header.
VC clock component	1 for each VC		Time-out service for the packet level.
VC restart component*	1		Handles the X.25 restart procedure.
VC data transfer component*	1 for each VC		Handles the transmission of data packets and interrupts.
VC reset component *	1 for each VC		Handles the X.25 reset procedure (packet level).
VC set up component*	1 for each VC	Fig. 8; sect. 4	*Packet level procedures*, handles the packet level establishment and clearing procedure, and includes the other packet level components.
Packet level interface	1 for each VC	Fig. 8; sect. 4	*Provision of a VC interface* to the next higher system layer.

* Component derived from the formalized specification of X.25

Explicit scheduling is necessary for the *Frame sender* and *Packet sender* processes, which are delayed when no frame sending transition is possible, or the link source queue is full, respectively. This is programmed with the monitor primitives *wait* and *signal* (*delay* and *continue* in Concurrent Pascal). In order to simplify the decision as to when to wake up a waiting process, we have chosen to wake up processes more often than necessary. The *Frame sender* is woken up after the reception of a frame from the *Frame receiver* or of a packet from the *Packet sender* or after a time-out, and the *Packet sender* is woken up after a frame has been sent by the *Frame sender*.

Acknowledgment

We thank P. Desjardins for many useful discussions, and the Concurrent Pascal implementation on the Xerox Sigma-6 computer used for our implementation. We are grateful to the Computer Communications Group of Bell Canada for letting us use their X.25 tester equipment. Finally, we thank S. Waddell for a revision of the manuscript, and Mme. Luyet for the careful typing.

References

[1] CCITT Recommendation X.25, 1976.

[2] T. Joachim, "Implantation du protocole standard X.25 à partir d'un modèle de formalisation et de mécanismes abstraits de programmation," Master's thesis, Dep. I.R.O., Univ. Montreal, Montreal, P.Q., Canada, Dec. 1977.

[3] P. Brinch Hansen, "The programming language Concurrent Pascal," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 199–207, 1975.

[4] G. V. Bochmann and J. Gecsei, "A unified model for the specification and verification of protocols," in *Proc. IFIP Congr. 1977.* Amsterdam: North Holland, pp. 229–234.

[5] G. V. Bochmann and R. J. Chung, "A formalized description of HDLC classes of procedures," in *Proc. IEEE Nat. Telecommun. Conf.*, 1977, pp. 03A 2-1–2-11.

[6] H. A. Schmid, "On the efficient implementation of conditional critical regions and the construction of monitors," *Acta Inform.*, vol. 6, pp. 227–249, 1976.

[7] P. Desjardins, "Un pilote pour controleur de communication dans Solo-Sigma," I.R.O., Univ. Montreal, Montreal, P.Q., Canada, Tech. Rep., in preparation.

[8] A. M. Rybczynski, "Collection of questions and answers on X.25," working document, 1977.

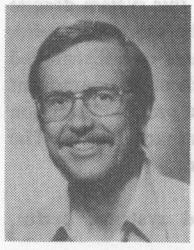
[9] G. V. Bochmann, "Specification and verification of computer communication protocols," I.R.O., Univ. Montreal, Montreal, P.Q., Canada, Publ. 294, 1978.

[10] D. Belsnes, "X.25 DTE implement in Simula," in *Proc. Eurocomp 78*, Online, England.

[11] IFIP Working Group 6.1, "Proposal for an internetwork end-to-end transport protocol," INWG Note 96.X; also in *Proc. Comput. Network Protocols Symp.*, Univ. Liège, Liege, Belgium, 1978.

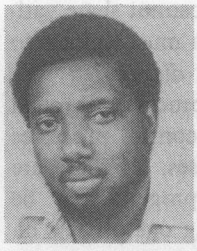
[12] J. K. Cavers, "Implementation of X.25 on a multiple microprocessor system," in *Proc. Int. Commun. Conf.*, 1978.

[13] D. L. A. Barber, T. Kalin, and C. Solomonides, "An implementation of the X.25 interface in a datagram network," in *Proc. Comput. Network Protocols Symp.*, Univ. Liège, Liege, Belgium, 1978, pp. E6-1–E6-5.



Gregor V. Bochmann received the Diplom in physics from the University of Munich, Munich, Germany, in 1968, and the Ph.D. degree from McGill University, Montreal, P.Q., Canada, in 1971.

He has worked in the areas of programming languages and compiler design, communication protocols, and software engineering. He is currently Associate Professor in the Département d'Informatique et de Recherche Operationnelle, Université de Montreal. His present work is aimed at design methods for communication protocols and distributed systems. In 1977–1978 he was a Visiting Professor at the Ecole Polytechnique Fédérale, Lausanne, Switzerland.



Tankoano Joachim was born in Fada N'Gourma, Upper Volta, on April 14, 1951.

He received the B. Sc. and M.Sc. degrees in computer science from the Université de Montréal, Montreal, P.Q., Canada, in 1976 and 1978, respectively.

He currently leads the Division of Systems at the Centre National pour le Traitement de l'Information (CENATRIN), Ouagadougou, Upper Volta.