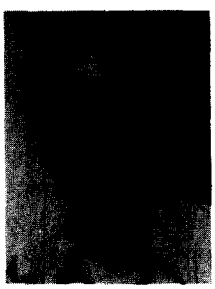
Specifications of a Simplified Transport Protocol Using Different Formal Description Techniques

Gregor V. BOCHMANN
Département d'informatique et de recherche opérationnelle, Université de Montréal, Montréal, Canada H3C 3J7

Abstract. Formal description techniques have been developed for the specification of OSI communication protocols and services, and can also be used as specification languages for other application areas. This paper presents two complete example specifications of a simplified Transport protocol (class 2) written in Estelle and LOTOS, and the outline of a similar specification in SDL. These examples are of sufficient complexity to demonstrate the difficulties encountered in the development of formal specifications. They may also be taken as a basis for a comparative evaluation of the three languages, Estelle, LOTOS and SDL, bearing in mind that they are particular examples.

Keywords. Communication protocols, formal description techniques, transport protocol, Estelle, LOTOS, SDL, formal specifications, protocol specifications, specification development.



Gregor v. Bochmann received the Diploma degree in Physics from the University of Munich, Munich, West 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, compiler design, communication protocols, and software engineering and has published many papers in these areas. He is currently a Professor in the Département d'Informatique et de Récherche Opérationelle, Université de Montréal, Montréal. His present work is aimed at design models for communication protocols and distributed systems. He has been actively involved in the standardization of formal description techniques for OSI. From 1977 to 1978 he was a Visiting Professor at the Ecole Polytechnique Federale, Lausanne, Switzerland. From 1979 to 1980 he was a Visiting Professor in the Computer Systems Laboratory. Stanford University, Stanford, CA. From 1986 to 1987 he was a Visiting Researcher at Siemens, Munich.

North-Holland
Computer Networks and ISDN Systems 18 (1989/90) 335–377

1. Introduction

The orderly introduction of new communication protocols, for proprietary systems or Open Systems Interconnection (OSI) [39], requires a careful analysis of the proposed protocols and services, and much effort for the development and testing of protocol implementations. Much research effort has gone into improving the working methods for these activities. In this context, the use of formal description techniques (FDTs) for the specification of communication protocols and services has received much attention, since such techniques allow a more systematic approach for protocol validation, implementation and testing, as compared to the traditional use of protocol specifications given in natural language (see for instance [6] or [4]).

Three FDTs are presently considered for application in this area, namely Estelle [13,26], LOTOS [11,27] and SDL [15,35]. Estelle and LOTOS are developed within ISO for application to OSI, but can also be used in other areas of application. Estelle is based on a finite state machine model which is extended by Pascal data structures, expressions and statements for the description of interaction parameters, additional state variables and related processing. A specified system may consist of a large number of interconnected state machine modules. LOTOS is a combination of a variation of Milner's CCS formalism [32] with a particular notation for abstract data types called ACT ONE [19]. Similar to the other FDTs, it allows the construction of a specification from several smaller components. SDL was originally developed by CCITT for the description of switching systems, but can also be used in other areas of application. Like Estelle, it is based on an extended finite state machine model. It is largely oriented towards a graphical representation. The original language has been considerably extended during the past years, also including facilities for

defining data structures, interaction parameters and additional state variables. Abstract data types are also supported.

The purpose of this paper is two-fold. First, we try to demonstrate the difficulties that arise in the development of formal specifications of protocols as they appear in real systems by discussing an example of sufficient complexity and detail. A simplified version of the OSI class 2 Transport protocol is chosen for this purpose. It includes, in particular, the provision of multiple parallel connections, multiplexing and flow control. Complete specifications in Estelle and LOTOS for this example are given in Annexes 1 and 2. A detailed explanation of these specifications is given in Section 3 and highlights the similarities and differences of the different specifications. A sketch of a specification in SDL is given in Annex 3.

In order to provide a basis for the comparison of the different FDTs, an effort was made to present specifications that are similar to one another, as much as this was possible and reasonable. For example, similar specification structures, design choices, and identifier names were chosen wherever possible. However, certain structural and other differences between the specifications remained and are largely due to the nature of the underlying FDTs. Some of these differences are discussed in Section 4. The discussion in Section 4 concentrates on those aspects of the specifications and underlying FDTs which appear to be most important for a comparative evaluation of the different FDTs. Section 5 provides some concluding remarks.

2. The Role of Protocol Specifications

Protocol specifications play an important role in the development life cycle of distributed systems. During the design phase of a distributed system, the protocol specifications are developed in relation with the communication service to be provided by the system and the service available from the system layer below, as indicated in Fig. 1. It is important to thoroughly validate the protocol specification, since it is the reference for the implementations in all the system components. During the implementation phase, it is not only used for deriving large parts of the implementation code, but should also serve as the basis for the selection of test cases for conformance testing and for the evaluation of test results. A more detailed discussion of these issues can be found in [4,6].

If formal specification languages are used in the design and implementation phases, different descriptions can be used in the successive stages of the development process. Starting with abstract service and protocol specifications, which correspond for instance to the OSI service and protocol standards, successively more detailed and implementation-oriented protocol specifications can be developed, which finally lead to the implemented program code [9,38]. It seems that these successive specifications differ from one another in respect to two aspects:

(a) *The specified behavior*: Certain behavior aspects, not defined in the original specification, are determined during the implementation process. This may include the addition of behavioral possibilities which were originally not included, such as the reaction to certain invalid inputs, or the selection of "implementation choices" which reduce the number of behavior possibilities, such as the selection of options to be implemented. The comparison of behaviors can be formalized through various equivalences and "implement" relations, as described for instance in [33] and [12].

(b) *The structure of the specification*: The structure of the more detailed specification reflects in certain ways the structure of the intended implementation. A structural comparison of specifications is difficult to formalize. Vissers et al. identify four major specification styles [17] which can be classified into extensional specifications, which define only "what" the specified system should do, and intentional specifications, which also define to some extent "how" the specified behavior is obtained by giving "implementation hints". The specifications given in this paper are partly intentional. For complex systems, it is often difficult to write human-readable specifications without introducing some form of internal structure. The internal structure of the Transport protocol specifications, as shown in Figs. 2–5, clearly indicate more detail than the "black box" structure of the protocol entity shown in Fig. 1.

In addition to the above two aspects, specifications can be distinguished by a third aspect [18], namely the language in which they are written. This is the aspect on which this paper focuses.

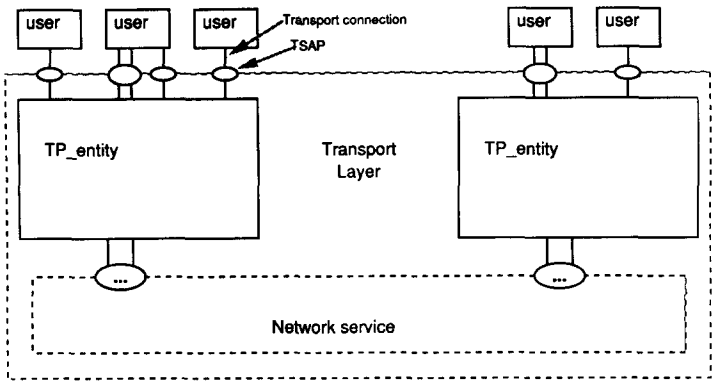


Fig. 1. Structure of the Transport layer.

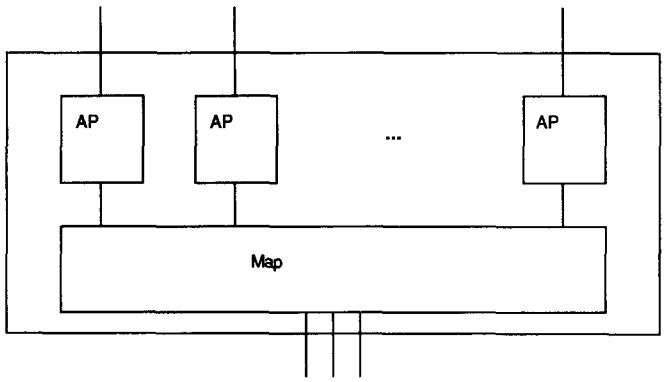


Fig. 2. Structure of Estelle specification.

Section 4 also includes some discussion of the impact of specification languages on the style of specifications.

3. Explanation of the Transport Protocol Specifications

This section contains a detailed explanation of the specifications given in the annexes. Annexes 1 and 2 contain complete specifications of a simplified OSI class 2 Transport protocol [29]. The functions supported by this simplified example are described in Section 3.2.1 below. The specification of Annex 1 was written first (an early draft in 1984) and has been influenced by an earlier complete OSI class 0/2 protocol specification [3]. The specification of Annex 2 was written afterwards (first version late 1986) and was modelled to same extent after the specification of Annex 1. The experience of writing this LOTOS specification lead to some revisions of the specification in Annex 1. Influences from the formal specifications developed within ISO [24,28] must also be acknowledged.

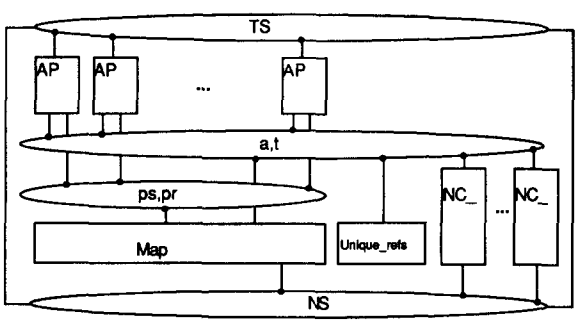


Fig. 3. Structure of LOTOS specification.

The overall structure of the Transport layer is shown in Fig. 1. The users of the Transport service access this service through the Transport service access points (TSAP), as shown in the figure. The Transport service is provided by a collection of Transport protocol entities, named *TP-entity*, which in turn use the Network service for the

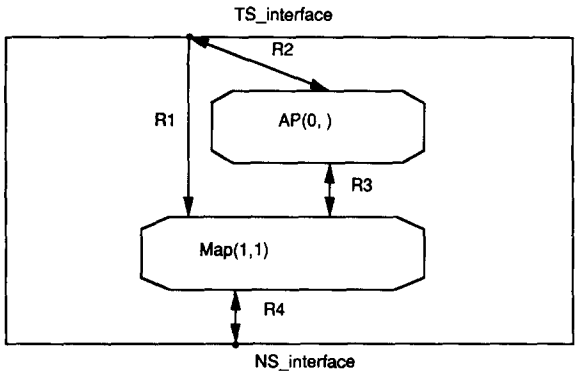


Fig. 4. Structure of SDL specification (using SDL graphic symbols).

exchange of protocol data units (PDUs) between one another. A protocol specification is the specification which must be satisfied by all protocol entities in the layer.

The Transport service provided by the Transport entities allows the users to establish Transport connections between one another. Several Transport connections can be established by a given user through a given service access point, usually leading to different destinations. At a given access point, the different connections are distinguished by so-called connection end-point identifiers (TCEP indentifiers). The different destinations are distinguished by so-called Transport addresses: Each address identifies a single access point at the service level.

For a given Transport connection, the following three phases of operation can be distinguished: (1) connection establishment, (2) data transfer, and (3) disconnection. During the data transfer phase, the users at both ends of the connection can send and receive simultaneously data in both direction. Data transfer is interrupted when either user issues a disconnect request. The disconnection phase may also be initiated by one of the Transport entities involved, or through the failure of the underlying Network connection.

The following explanation of the specifications given in the annexes is written as two versions of text. Most parts of the versions are identical and only written once. Where the corresponding text for the Estelle and LOTOS versions are different (because of differences in the specifications) the two versions are written one after the other in the following from "...common text... {Estelle:... text for Estelle version...} {LOTOS:...text for LOTOS version...} ... continuation of common text...".

The outline of the SDL specification given in Annex 3 follows largely the structure of the Estelle specification in Annex 1. Certain differences are discussed in Section 4.

3.1. Internal Structure of a Protocol Entity

3.1.1. Addressing Conventions

It is assumed that each protocol entity uses only a single Network service access point (NSAP),

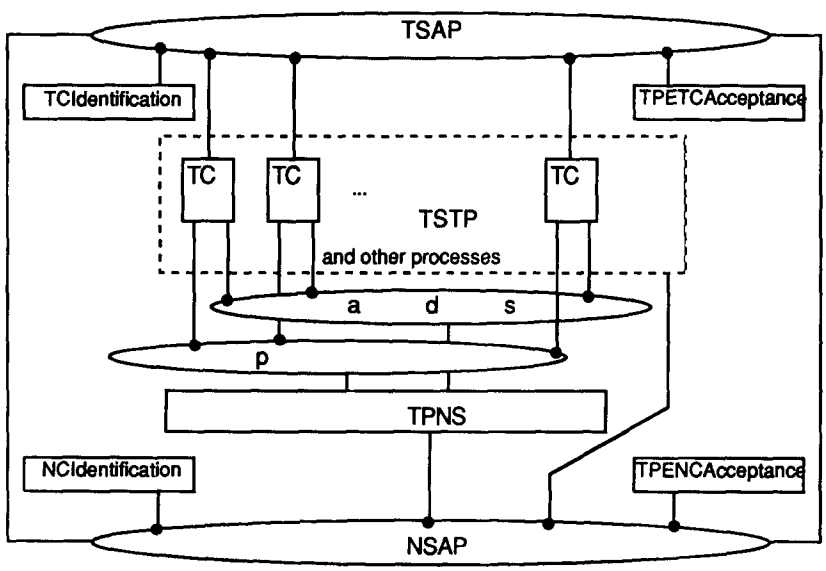


Fig. 5. Structure of ISO LOTOS specification.

identified by a particular Network address. This implies a hierarchical addressing scheme where each Transport address determines the Network address over which the TSAP is accessible. In such a case, it is convenient to partition the Transport address into two parts: the "Network address" prefix, and a suffix identifying the TSAP within the Transport protocol entity servicing the NSAP identified by the prefix. This address structure is defined by the *T_address_type* definition.

3.1.2. Service Access Points

As shown in Fig. 1, a common specification of the Transport service access points is used in the specifications of (a) the Transport service, (b) the Transport protocol entity, and (c) the Transport service user, i.e. the Session protocol entity. Such a common specification defines the possible interactions, also called "service primitives", their parameters, and the local rules determining the possible order of execution. (In the case of the LOTOS specification in [28], this common part is represented by the subprocesses TCEP which are part of the TC processes shown, in Fig. 5.) It is noted, however, that the local rules are often not explicitly defined. This is also the case for the specifications given in the annexes. The Transport service primitives and their parameters are defined by the part of the specification entitled {Estelle: Definition of Transport Service Primitives} {LOTOS: TCEP_primitives}.

A given Transport connection is identified, within the supporting Transport entity, by the address suffix of the supporting TSAP and the TCEP identifier. {Estelle: The array of interaction points *TS* of the *PT_entity* module represents the service access points of the Transport entity. For each interaction taking place at one of these interaction points, the first index of type *T_suffix-type* indicates to which TSAP the interaction pertains; the second index indicates the TCEP identifier of the connection.} {LOTOS: All Transport service interactions take place at the gate *TS*. Each interaction includes three parameters, namely the address suffix of the TSAP to which the interaction pertains, the TCEP identifier of the connection, and the service primitive exchanged.}

A similar specification applies to the Network access points. The Network service primitives are defined in the part entitled {Estelle: Definition of Network Primitives} {LOTOS: NCEP_primitives}. Since only a single Network access point is used by a given Transport entity, a Network connection is identified by the value of its NCEP identifier. The Network access point is represented by the {Estelle: interaction point array} {LOTOS: gate *NS* of the *TP_entity*}.

3.1.3. Submodules of an Entity

The behavior of the *TP_entity* is described in the following by first defining an internal substructure of the entity in terms of submodules and their interconnection. A static structure shown in Figs. 2–5 is assumed. This structure foresees one *AP* module per possible Transport connection, and a single *Map* module which provides the multiplexing function and looks after the sending and receiving of the TPDUs. This substructure can be defined as follows.

{Estelle: Written at the end of the specification, an array of *AP* module variables and a *Map* module variable are declared. These modules are initialized through the execution of Estelle INIT statements which are part of the initialization code for the Transport entity. The CONNECT and ATTACH statements then create the interconnection structure shown in Fig. 2.}

{LOTOS: The process structure shown in Fig. 3 is defined by the text of the *TP_entity* module. The parameters *tc_ids* and *nc_ids* determine what the possible connection end-points at the different service access points can be. As defined by the text of the processes *AP_modules* and *NC_managers*, their invocation by the *TP_entity* is equivalent to the parallel invocation of a number of *AP_closed* and *NC_manager* modules, respectively. A *AP_modules* process, for instance, receives as parameter a set of ⟨*T_suffix, TCEP_id*⟩ pairs which represent a set of possible Transport connection identifications. The defined process is equivalent to the invocation of a new *AP_closed* process for a selected ⟨*T_suffix, TCEP_id*⟩ pair and its replacement (so to speak) by another invocation of *AP_modules* where the selected pair is deleted from the parameter set.}

The above substructure of a protocol entity is one among many possibilities. The choice of a particular substructure for the formal specification of a protocol entity does not imply that the same substructure must be manifest in any implementation of the protocol. (An implementation should only exhibit the behavior defined by the specifica-

tion.) However, the choice has certain implications about what properties are evidently satisfied by the defined behavior. Important considerations for the choice of a substructure are the following:

(a) Which parts of the specified system operate independently of one another? Such parts could usually operate "in parallel".

(b) If two parts of the specified system share some common information, these parts may either be represented as a single module which contains this information as an internal state, or as two submodules which exchange their knowledge about this information by means of interactions.

The *AP* submodules in the protocol entity refinement above operate independently from one another, since they deal with separate connections. However, the different connections interfere with one another when they are multiplexed over a single, shared Network connection. The latter aspect is handled by the {Estelle: *Map* module.} {LOTOS: *Map* and *Unique_refs* modules.} {Estelle: The fact that the *Map* module, by definition, executes one transition at a time, and each transition deals with a single connection, implies that mutual exclusion is established between operations for different connections. It is to be noted that the sequential execution of the transitions of the *Map* module exhibits less parallelism than would be possible. In fact, among the transitions of the *Map* module, there are many that have no conflict, nor do they operate on shared information. An implementation could therefore execute these transitions in parallel.} {LOTOS: The *Map* module is further subdivided into *PDU_handlers*, one for each Transport connection. The mutual exclusion of access to the Network connection is realized by the gate *NS*. A specification describing PDU concatenation would probably include a subprocess per Network connection.}

3.2. Functional Decomposition and Inter-module Communication

While the specification substructure shown in Figs. 2–5 was obtained by an overall consideration of independent Transport connections and their multiplexing over shared Network connections, more detailed design choices must be made in order to determine what functions each of the submodules should realize. A useful design criteria is to minimize the required communication between the submodules. These issues, as they relate to the Transport protocol, are discussed in this section.

3.2.1. Functional Decomposition

The following functions of a Transport protocol entity can be identified [8,36]:

(a) *Addressing*: To select the correct TSAP for remotely initiated connection requests, and selecting Network connections with appropriate remote Network addresses for connection requests initiated by local users.

(b) *Local identification of Transport connections*: To identify Transport connections based either on locally selected reference numbers (for incoming PDUs), or on the local Transport address suffix and the TCEP identifier (for service primitives received from the users).

(c) *Connection establishment, and clearing*: To be able to establish, and clear Transport connections, as requested from the remote peer protocol entity or the local user.

(d) *Data transfer*: To be able to transfer user data over established Transport connections. This includes the flow control at the local interfaces of the Network and Transport SAPs. (Note that the principle of flow control is to make the sender side wait until the receiver side is ready for reception.)

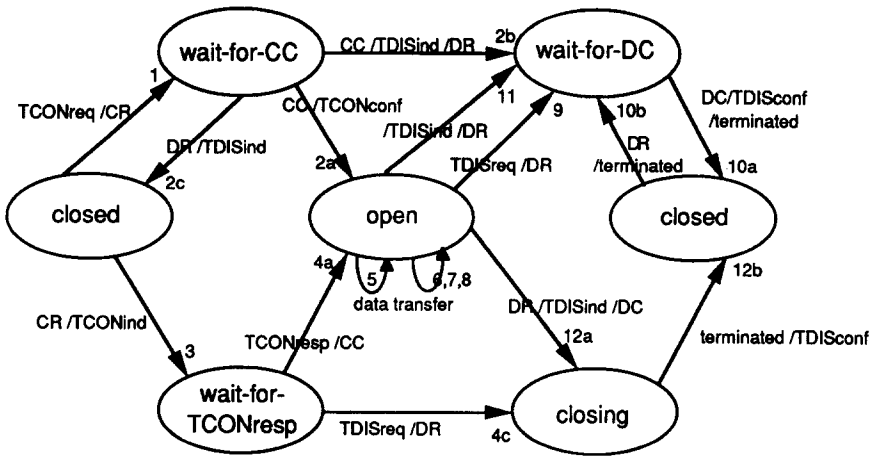
(e) *End-to-end flow control*: To exchange control information with the peer protocol entity (in the form of acknowledge (AK) PDUs and certain parameters of other PDUs) in order to control the flow of data between the two protocol entities.

(f) *Option negotiation*: To be able to negotiate the values of options to be used over a connection to be established, based on options proposed by the users at the two ends of the connection.

(g) *Connection parameter negotiation*: To be able to negotiate values of certain parameters to be used over a connection to be established. In contrast to option negotiation, these parameters are selected by the two protocol entities largely based on their implementation parameters, and possibly also on performance parameters provided by the users.

(h) *Multiplexing*: The use of a single Network connection for several Transport connections (provided that the Network prefixes of the Transport addresses are the same).

(j) *Segmentation*: To be able to handle arbitrarily long Transport service data units (SDUs).



(Notation: /X means "output of X")

Fig. 6. State diagram for a single Transport connection.

It is assumed by the specifications in the annexes that the SDU segments provided by the user of the service interface are already of such a length that they can be included in a single PDU.

(k) *Network connection management*: This aspect is largely simplified for the specifications in the Annexes. It is assumed that the Network connections are initially established and remain always open.

3.2.2. *Allocation of Functions to Submodules*

The functions above are allocated to the two types of submodules of the *TP-entity* based on the question whether they can be executed for each Transport connection independently of other Transport connections, or not. In the former case, the function is performed by the *AP* module (which handles a single Transport connection). The functions handled by the *AP* module are data transfer, end-to-end flow control, option negotiation, segmentation. Connection establishment and clearing, as well as connection parameter negotiation involves the *AP* as well as *Map* modules. The other functions are handled by the *Map* module.

This partitioning is reflected in the declaration of the {Estelle: local variables} {LOTOS: parameters} of the submodules. The variable/parameter *opt* of the *AP* modules records the options selected for the connection. The other variables/parameters are sequence counters and credit values for window-oriented flow control for the two directions of data transfer.

{Estelle: The variable **state** represents the major state, as shown in Fig. 6} {LOTOS: The major states shown in Fig. 6 are represented partly by the different *AP_xxxx* processes, where *xxxx* represents the state where the process starts its execution, and partly by the "locus of execution" within the active *AP_xxxx* process. Note that only one of the *AP_xxxx* processes is active at a given time for a given connection.}

{Estelle: The body of the *Map* module contains two arrays of state information. The array *TC* records for each Transport connection, identified by an index in the array, certain information required for the functions performed by the *Map* module. In particular, the variable *assigned_NC* indicates which Network connection is used by the Transport connection for the transmission of PDUs. The value *undefined* means that no Network connection is assigned. (This value is used when the Transport connection is in the *closed* or *wait_for_CC* state.)}

{LOTOS: The state information required by the *Map* process for the Transport connections is kept as parameters of the *PDU_handler* subprocesses. A *PDU_handler* process is created when a connection is established and contains in particular the *NC_id* parameter which determines the associated Network connection.}

Since the Network connections are assumed to be always open, only minimal information is required, namely the Network address of the remote Transport entity, which is checked when a new

Transport connection is to be allocated. {Estelle: The array *NC* records for each Network connection this information.} {LOTOS: The *NC_manager* process includes this information as parameter.}

The only global information, pertaining to all connections, is the set of local references in use. {Estelle: This information is held in the variable *active_references* of the *Map* module.} {LOTOS: This information is held as parameter of the process *Unique_refs*.}

3.2.3. Communication Between the Submodules

The communication between the *AP* and *Map* modules is mainly concerned with information about the PDUs exchanged with the remote protocol entity. {Estelle: This information is included in the data type *TPDU_and_control_information* which also contains some "control information" (as indicated by the comments). For instance, the field *full* is used to indicate whether a PDU buffer contains a PDU or not.} {LOTOS: This information is defined by the data type *p_info*.} This information is exchanged between an *AP* module and the *Map* module {Estelle: through a channel of type *PDU_and_control*.} {LOTOS: at the gates *pr* and *ps*, respectively. The gate *pr* is used for reception of PDUs and *ps* for sending.}

The {Estelle: interaction *terminated*} {LOTOS: gate *t*} is used by the modules to indicate to one another that a connection is closed after the last PDU was received or sent. Additional communication is required for flow control (see Section 3.5).

{LOTOS: An additional gate *a* is used for assigning a new Transport connection to a given Network connection. Interactions at this gate, as well as for gate *t*, involve all submodules of the TP_entity (see Fig. 3).}

3.3. The AP Module

The specification of the *AP* module defines the order for exchange of PDUs with the peer Transport protocol entity and for exchange of service primitives with the local user of the Transport service. This order is shown in Fig. 6 in the form of a state transition diagram. The specification of the *AP* module also defines the allowed interaction parameters values, not shown in the figure. Also not shown in the figure are functions such as interface and end-to-end flow control. In order to facilitate the dicussions below, the transitions shown in the figure are labeled with names T_1 through T_{11}. The corresponding parts of the formal specifications are indicated in the annexes.

For the establishment of a Transport connections, there are transitions for the case that the connection is initiated by the local user (transitions T_1 and T_2), and for the case initiated by the peer entity (transitions T_3 and T_4). The first transition is executed when the user initiates a *TCONreq* primitive, which contains as parameters the destination address and a proposal for the options to be used over the connection. It is assumed that the entity is ready to receive such a request for the *AP* in question. The action of the *AP* module is simply to forward a CR PDU to the *Map* module. *CR_PDU* is a function which is introduced to simplify the notation for sending PDUs. It returns as result a {Estelle: record of type *TPDU_and_control_information*} {LOTOS: value of sort *CR_PDU*} which represents a CR PDU including the parameters passed to the function as arguments, not including, however, the local and remote reference numbers which are added by the *Map* module. Similar functions are defined for forming other kinds of PDUs. The definition of these functions can be found in the part entitled *PDU definitions* of the specifications. {Estelle: The proposed options are also recorded in the local *opt* variable because their value is needed for checking the option parameter of the CC PDU to be received.}

The second transition (T_{2a}) describes the action to be taken when a CC PDU arrives (in response to the CR sent by the first transition), and checks whether the accepted options mentioned in the PDU are among those proposed initially. (Note that it is possible that the remote user does not accept all of the options proposed by the initiator.) The *AP* module records the accepted options, which will be used for the established connection, and sends a *TCONconf* primitive to the user indicating the successful establishment of the connection. The CC PDU also contains a *credit_value* parameter which indicates the credits given by the remote entity for data transfer from the local user to the remote side. {Estelle: The variables} {LOTOS: The parameters of the *AP_open* process named} *TSseq* and *TRseq* are the sequence coun-

ters for the DT PDUs to be sent and received, respectively. They are set to zero.

Another transition (T_{2b}) is executed when a CC PDU is received which contains options not originally requested. Such a response is not accepted and leads to the termination of the connection.

The establishment of a connection in response to a CR PDU received from the peer Transport entity is described similarly by the transitions T_3 and T_4.

There are many different cases of connection termination that may occur, in addition to the case above. These cases are described by different transitions of the *AP* modules which are mentioned below.

Transition T_{4b} is executed when the user refuses an incoming connection request, given in the form of a *TCONind*, by responding with a *TDIS-req*. It leads to the sending of a DR PDU. Transition T_{2c} is executed when such a DR PDU arrives in response to a CR PDU sent to the peer protocol entity.

Transition T_9 corresponds to a termination requested by the local user during the data transfer phase. Transition T_{12} covers the case that the termination originates on the other end of the connection. In contrast to the initial refusal of a connection (transitions T_{2c} or T_{4c}), the termination in the data transfer phase involves the additional exchange of a DC PDU. The reception of such a DC PDU is described by transition T_{10a}. Transition T_{11}, finally, is a spontaneous transition which leads to the termination of the connection without user initiative. It may be executed by a protocol entity in exceptional situations, for instance in case of congestion.

The data transfer phase of the protocol is described by the transitions T_5 through T_8. The first two transitions handle the sending and receiving of DT PDUs, respectively. The sequence counter and credit variables/parameters are updated accordingly. Note that on reception, the sequence number and the available credit are checked. The flow control issues (transitions T_{7x} and T_{8x}) are discussed in Sections 3.5.

3.4. The Map Module

The other functions, such as addressing, PDU en- and de-coding, multiplexing, etc. are mainly handled by the *Map* module. Its operation is explained in the following.

{Estelle: The "state" variables of the *Map* module contains information about all Transport connections and Network connections.} {LOTOS: The *Map* process creates a *PDU_handler* process for each established Transport connection. The parameters of this process contain the required information on that connection. For each Network connection, there is also a *NC_manager* process, which includes as parameter the necessary information about that connection. The latter processes are only involved in interactions on the *a* and *t* gates at the beginning and end of a Transport connection. The *PDU_handler* processes perform the sending and reception of PDUs.}

The sending of PDUs is described in two steps. The PDUs received from an *AP* module are first stored in a *PDU_buffer* (see part T_b). {Estelle: There is one buffer for each kind of PDU, for each Transport connection.} {LOTOS: The *buffer* parameter of the *PDU_handler* process is a sequence of PDUs. The corresponding data type *PDU_buffer_type* is based on the predefined *string* type of LOTOS.}

These stored PDUs may be sent in coded form as data fragment in a Network NDATA service primitive, as described in part T_s. {Estelle: This is done by a (spontaneous) transition which is only executed when, for a given Transport connection and PDU kind, the PDU buffer contains a PDU, and the Network connection assigned to the Transport connection is ready to receive more data (flow control at the Network service interface).} The PDU is coded and sent as a Network SDU. Before the PDU is encoded, the information received from the *AP* module is complemented with certain PDU parameters which are determined from the information available in the *Map* module.

It is important to note that the order in which the PDUs for a given Transport connection are sent over the assigned Network connection is not necessarily the same in which they were submitted by the *AP* module. It may happen that several PDUs are stored some time before they are sent (possibly due to Network flow control). For example, expedited data PDUs (not described here) or AK PDUs may overtake normal DT PDUs. In order to allow for such overtaking, but disallow

overtaking of DT over expedited or CC PDUs, each kind of PDU is assigned an *order* attribute which assigns a kind of priority to each type of PDU ({Estelle: see assignment in functions *CR_PDU*, etc. in the *AP* module} {LOTOS: see type *PDU_ordering*}). The predicate *order_constraint* ({Estelle: a function declared in the *Map* module} {LOTOS: defined in the *PDU_ordering* type}) is used to check that for each PDU sent the order in consistent with the priority rules. In addition, DR and DC PDUs may overtake other PDUs by destroying them, since the Transport termination phase is "destructive". {Estelle: This is modeled by statement S_1 in transition T_3.} {LOTOS: The operation *drop* (see *PDU_ordering* type definition) is used for destructive overtaking.}

A Network connection must be "assigned" to a newly requested Transport connection before any PDU can be sent. {Estelle: This is done by executing transition T_a.} {LOTOS: This is done by an event at the *a* gate, and involves the *AP, Map, Unique_refs,* and *NC_manager* processes.} This operation is executed when for a given Transport connection a CR PDU {Estelle: has been} {LOTOS: is} forwarded by the corresponding *AP* module, and a Network connection exists which is connected to a remote entity with has the address corresponding to the destination address requested in the PDU. {LOTOS: This latter condition is checked by the *NC_manager* process.} During this operation, a local reference is also selected which is used to identify the Transport connection among all the other connections handled by the Transport entity. {Estelle: The selection of a new reference is performed by the procedure *assign_new_ref* which also updates the set of active reference numbers. A specification of a possible algorithm is given by the procedure definition of the annex.} {LOTOS: The selection of a new reference is performed by the event at the *a* gate (which generates the new value) since none of the participating processes determines the value. The condition that the new reference should presently not be in use is imposed by the *Unique_refs* process.}

The reception of PDUs is described by part T_r. This part is executed when a NDATA primitive is received from the Network containing the PDU in question.

{Estelle: The received PDU is decoded and the function *exists_TP* determines whether a corresponding Transport connection already exists. If such a connection does not exists, the PDU should be a CR requesting the establishment of a new Transport connection. It is assumed that a free local Transport end point identifier *EP_id* can always be found for the new connection. The value for the local variables of the new connection are determined, and the PDU is forwarded to the corresponding *AP* module.}

{LOTOS: PDUs with values for *NC_id* and *dest_ref* for which no corresponding *PDU_handler* is active must be received directly by the *Map* process. It accepts only CR PDU's (see part T_a.). After the assignment of a local reference (event at gate *a*, see discussion above) a corresponding *PDU_handler* is created and operates in parallel with the *Map* until the Transport connection is terminated through an event at the *t* gate. The CR PDU is a parameter of the assign operation and is passed during that operation to the assigned *AP* process which processes it further. The *AP* process participating in the assign operation imposes the condition that no Transport connection is active at the given end point and the Transport address suffix of the end point corresponds to the one requested in the CR PDU.}

If a Transport connection already exists for the PDU received, the PDU is immediately forwarded to the corresponding *AP* module. {LOTOS: For all PDUs, except CR, this is done by the corresponding *PDU_handler* process (see part T_r).} In the case that a CC PDU is received, the remote reference is recorded in order to be used as reference in the PDUs to be sent for this connection {LOTOS: (see function *update_source_ref* defined within the *PDU_type* definition)}.

3.5. Flow Control and Synchronization Issues

In the case that a source module transfers data to a destination module, it is evident that the reception process in the destination process cannot go faster than the data output from the source. However, buffering problems may occur when a fast source does not wait for a slow reception process. The purpose of "flow control" is to have the source process restrain from sending when either the destination or the transmission medium is not capable of handling the data at the speed of the source.

Flow control can be described through different mechanisms. This section discusses how these issues are described in the specifications in the annexes. The end-to-end flow control mechanism performed between the peer protocol entities is usually described precisely by the protocol specification, the aspects of interface flow control are sometimes ignored [24] or sometimes described in a very abstract manner [28]. From the pragmatic standpoint, there is clearly a relation between the end-to-end flow control and the flow control at the interfaces related to the limited buffering within the protocol entities. In the specifications of the annexes, a particular design choice was taken by limiting the amount of buffering within the entities.

3.5.1. Flow Control for Sending User Data

Transitions T_{8a} through T_{8c} deal with end-to-end flow control for sending user data. Transition T_{8a} describes the action of the protocol entity when an AK PDU is received. The *S_credit* variable is updated according to the credit value received. A *TDATreq* interaction is only accepted when the *S_credit* variable/parameter has a value greater or equal to 1. The received user data blocks are stored in the PDU buffer of the *Map* module until they are sent over the network. {Estelle: Since the PDU buffer is limited to one block only, there is the additional condition that the DT PDU buffer should be empty. This fact is conveyed by the *Map* module by sending the *ready* interaction to the *AP* module (see statement S_2 of transition T_s). The *AP* module uses the *map_ready* variable to remember whether the PDU buffer is empty (see transition T_{8b}). When both conditions are satisfied a *READY* interaction is sent to the user module (see transition T_{8c}). This gives permission to send one *TDATreq*.} {LOTOS: The *TDATreq* is received from the user by the *AP_open* process through transition 5. The guard *S-credit ge 1* and the rendezvous nature of the interaction effect the flow control.}

3.5.2. Flow Control for Receiving User Data

The specifications are written in such a manner that credit is only given to the remote Transport entity for as much data as the user is ready to receive. The user indicates his willingness to receive data blocks by invoking the *U_READY* service primitive which contains as parameter the number of additional blocks to be received. This information is stored in the *R_credit* variable/parameter of the *AP* module (see {Estelle: transition T_{7a}} {LOTOS: transitions T_{7a} and $T_{7a'}$}). The value of *R_credit* is used to determine how many credits are sent in acknowledgment PDUs to the remote peer Transport entity (see transition T_{7b}). This transition may be executed any time. It is assumed that it is executed often enough to obtain a reasonable throughput for the data transfer from the remote site to the local user.

3.5.3. Synchronization of Connection Termination

It is important to note that after a *TDISreq* or a *TDISind* for a given pair of *T_suf* and *EP_id*, the user should not immediately send a new connect request, since a DC or DR PDU may still have to be sent. It is therefore necessary to advise the user when a new *TCONreq* can be sent. {Estelle: For this purpose the *TDISconf* service primitive and an intermediate state *closing* are introduced. The *TDISconf* primitive is invoked by the Transport entity after each *TDISreq* or *TDISind* when the transition to the *closed* state occurs.} {LOTOS: The fact that the *TCONreq* interaction (see transition T_1 of the *AP_open* process) is only possible at the beginning of the *AP* process (which is reinitialized after the complete termination of the previous connection, see transition T_{12} of the *AP_disc* process), and the rendezvous nature of the *TCONreq* interaction force the user to wait.}

The *AP* and *Map* modules synchronize the complete termination of a connection through {Estelle: the *terminated* interaction} {LOTOS: an event at the *t* gate}.

3.6. Protocol Functions Not Described

The following list indicates functions of the ISO/CCITT class 2 Transport protocol [29] which are not supported by the simplified protocol described in this paper.

- Only a single Network service access point (NSAP) is supported by the specified entity.
- TPDU size negotiation does not exist.
- It is assumed that Network connections always remain open. No Network connection establishment, disconnection nor reset are considered.
- There is no user data in TCON req primitives.
- Expedited data transfer is not specified.

- The handling of procedure errors committed by the remote entity is not described.
- The possible limitation of local resources is not considered.
- There are no address parameters in CC PDUs.
- Simple hierarchical addressing is assumed, where a Transport address consists of the Network address prefix and a Transport suffix.
- The Transport user initiating a connection cannot disconnect until the response from the peer has been received.
- There is only limited buffering of user data in the Transport entity.
- Concatenation of several PDUs into a single Network service data unit is not described.
- It is assumed that the user processes do the necessary segmentation of longer Transport service data units.

4. Comparison of the Specification Languages

As pointed out before, the specifications in the annexes have been written with the objective to make their structure and form as similar to one another as seemed reasonable, given the constraints of the different languages. Differences between the specifications, which became partly evident from the description in Section 3, are therefore largely related to differences between the specification languages. The purpose of this section is to discuss the specification differences and at the same time discuss certain important differences between the languages.

It is important to note that the following discussion does not pretend to address all important differences between these specification languages. Additional aspects of comparison between the specification languages can be found in [5,40,41].

4.1. Synchronous Versus Asynchronous Inter-module Communication

In SDL and Estelle, interaction between two modules is through message passing. One module generates an output interaction, including its parameters and places it into a queue from where it is subsequently taken as input by the receiving module. This mode of communication is sometimes called "asynchronous" because the outputting module usually continues its processing before the output is processed by the receiving module.

In LOTOS an interaction between several modules can only be initiated if all participating modules agree; each of them determine some parameter values, or impose conditions on these values. This mode of communication is also called "synchronous" or "rendezvous", because all modules participating in an interaction do so at the same time, and the execution of an interaction implies a simultaneous state change for all participating processes.

This difference in the nature of inter-module communication has a strong impact on the way the specification languages can be used to define communication between different system modules. Two particular aspects are discussed in the subsections below. It is noted that Estelle also allows for the possibility that variables are shared between modules. This introduces some form of "synchronous" communication. Also certain dialects of Estelle have introduced rendezvous communication as an option [16,22,30]. On the other hand, asynchronous communication can be modelled in LOTOS by introducing queues explicitly between the communicating system modules.

4.1.1. Flow Control

With rendezvous interactions, each module participating in an interaction may pose his own conditions for the execution of the interaction or its parameter values. This may be used for defining flow control and other conditions. For example, the interface flow control by which the Transport entity restrains the user from sending more data is described by this mechanism (see Section 3.5.1). Similarly, the readiness for a new Transport connection is indicated in this manner (see Section 3.5.3). The specification of these issues with asynchronous interactions requires additional interactions (e.g. the *READY* and *TDISconf* Transport service primitives in Annex 1). The resulting specification is more complex.

It is important to note that not all flow control issues can be handled in this manner. For instance, data flow control in the other direction is handled also in the LOTOS specification of Annex 2 with an additional service primitive (see Section 3.5.2). It is noted that these aspects are sometimes not defined formally [24], but only

informally. They are, however, an important aspect of a system, and should be specified.

4.1.2. Interaction Cross-over at Interfaces

With rendezvous interactions, all participating modules are aware of the execution of an interaction; with queued interactions, the receiving module may not be aware of the readiness of the next input when it produces an output. This may lead to unintended cross-overs of interactions between two communicating modules, sometimes called "collisions". For the specification of Annex 1, for instance, a user may output a *TCONreq* interaction which is entered into the queue of an *AP* module, while at the same time, the same *AP* module executes a T_3 transition and enters a *TCONind* interaction into the queue of the same user module. The possible occurrence of this situation is not foreseen in the specification of Annex 1. (A similar cross-over of *CR* PDUs may occur at the interface between an *AP* module and the *Map* module.) It is important to note that not all cross-overs in corresponding queues lead to such problems, e.g. the cross-over of *TDATAreq* and *TDATAind* interactions at the user interface pose no particular problems.

There seem to be the following approaches to solving cross-over problems of asynchronous communications:

(1) To ignore them, as in Annex 1 (in general not satisfactory).

(2) To write the specification in such a manner that the cross-over situations are taken care of. For the example of Annex 1, one may write the specification of the *AP* module in such a manner that an incoming *TCONreq* interaction in the *wait_for_TCONresp* state (after sending a *TCONind*, see Fig. 6) will be dropped. Inversely, the *AP* module may be defined in such a manner that it goes back to the *closed* state and processes the *TCONreq* normally. These two design choices correspond to giving priority to the incoming or outgoing calls, respectively. (A general approach for handling such conflicts based on priorities is described in [20]; cross-over problems at the Transport interface are further discussed in [10].) It is important to note, however, that such decisions are adequate for interface standards, but not for the abstract interface definitions of protocol standards, where such decisions should be left to the implementation phase.

(3) To assure that they cannot occur due to the structure of the system specification. For example, any hand-shake oriented communication structure avoids cross-overs; therefore, a cross-over of *TCONresp* and *TCONconf* cannot occur. This approach largely limits the kind of possible communication structures, and therefore is not generally applicable.

(4) To impose a specific run-time environment of interpretation rule such that cross-overs cannot occur. An example of such a rule is a system-wide priority of input transitions over spontaneous transitions. Together with a restriction that a transition generates at most one output, this rule assures that any waiting input will be processed before another spontaneous transition may produce an additional queued interaction. If the system contains initially no queued interaction and receives none from its environment (or only a single one after all its internally queued interactions are processed) then the system contains at most one queued interaction at any given time, and no cross-over can occur.

4.2. The Concept of "Transition"

The languages SDL and Estelle use the concept of "transition" to model a state transition of a module which is initiated by an available input interaction. When executed, a transition leads from a given major state through updating of state variables and output generation to a new module state, in which further input is expected. An example is given in Fig. 6, where each arrow represents a transition of the AP module. It is interesting to note that the same "transitions" can also be identified within the LOTOS specification.

In SDL, there is (at most) one transition per major state and possible kind of input interaction. The next state in which input is expected may depend on decisions which are part of the actions associated with the transition. A transition, once started, cannot be interrupted by other modules. The situation is similar in Estelle. However, there may be several "Estelle transitions" for a given major state and kind of input. Any one of them may be executed, unless the associated PROVIDED clauses define additional conditions. As an example, Annex 3 shows how the Estelle transitions T_{2a} and T_{2b} are written in SDL as a single "SDL transition".

In Estelle, there is also the concept of a spontaneous transition, which has the properties described above, except that there is not associated input interaction, Instead, a spontaneous transition can be selected for execution only based on the present major state and other variables as expressed in the FROM and PROVIDED clauses.

A transition has an atomicity property, which means that, once started, any output foreseen by the transition must be completed before the same module can initiate another transition. In the case of asynchronous communication, this poses no problem, since output is queued in the input queue(s) of the receiving module. In the case of models of communicating finite state machines without queuing (e.g. [2] or [1]) or Estelle dialects using rendezvous interactions, the system designer must take care that no deadlock situations are possible where, for instance, two communicating modules are in the midst of a transition and want to send output to the other. Note that this situation, in the presence of queues, leads to the cross-over discussed in Section 4.1.2.

One way to avoid these deadlock possibilities is by restricting transitions to the following kinds of *basic transitions*:

(a) pure input transition, effecting no output, and

(b) spontaneous output transitions, which are initiated independently of any waiting input.

Many finite state reachability tools are based on such a model (e.g. [7]). An SDL or Estelle transition can easily be decomposed into several basic transitions by the introduction of additional intermediate states. It is noted that the *Map* module of the Estelle specification in Annex 1 uses only basic transitions for the processing of PDUs to be sent. With this specification style, it is necessary to store the information received with an input in appropriate state variables (e.g. the *PDU_buffer*) for use by subsequent output transitions. However, such intermediate storage is necessary anyway if flow control considerations may disallow the production of output, which is the case in our example.

4.3. Assertional Specifications and Abstract Data Types

Estelle uses the Pascal data type definitions, procedures and other statements for the definition of the operations on interaction parameters and state variables. This favors an "algorithmical" specification style, which is appropriate for most parts of protocol specifications, as our example shows. However, there are certain parts in many specifications, where a more "assertional" specification style is more appropriate. In the latter style, the specification would define what properties the module would have, not what algorithm it executes in order to obtain these properties. A typical example is the procedure *assign_new_ref* in Annex 1, which finds a new local reference for a new Transport connection. The required property is that this new reference is not yet in use. The Estelle specification uses the EXITS expression for this purpose which, however, has no straightforward efficient implementation. In the LOTOS specification of Annex 2, this property is expressed in the guard of the *Unique-refs* process participating in the assign *a* operation.

For implementation purposes, both of these specifications would probably have to be changed in order to include some more efficient algorithm for the selection of a new reference number, such as the following:

```
var next_ref: reference_type;
begin new_ref := next_ref;
      active-refs := active_refs + [new_ref];
repeat
if next_ref = max_ref − 1 then next_ref := 1
   else next_ref := next_ref + 1
   until not (next_ref in active_refs) or next_ref
   = new_ref;
if next_ref = new_ref then * * * * * (* too many
   connections *)
end
```

LOTOS uses an abstract data type formalism for the definition of operations related to interaction parameters and "state variables", represented as process parameters. This formalism favors an assertional specification style. On the other hand, the present version of LOTOS [27] lacks abbreviations for defining simple data structures, e.g. the equivalent of Pascal RECORDS, ARRAYS, and enumeration types. Such abbreviations [23] have already been used in Annex 2 in order to avoid unreasonably lengthy and trivial text in the specification. The latest version of SDL [15] also includes an abstract data type formalism, and predefined abbreviations for records and array data

structures. SDL also allows the declaration of variables and the use of assignment statements, in contrast to LOTOS which has the flavor of a functional language.

An example related to the use of LOTOS' abstract data type formalism is the PDU buffer of the *Map* module. The Estelle specification defines an array with space for one PDU per kind of PDU. This form of specification was chosen in order to simplify the description where DT PDUs can be overtaken by AK and DR PDUs, and still to avoid the complexity of defining a PDU buffer containing arbitrarily many PDUs. The LOTOS specification defines such an unlimited PDU buffer using the *string* concept, a predefined abstract data type. An equivalent specification in Estelle would require an "algorithmical" definition of a queue which usually leads to some implementation-oriented choices. Examples of "assertional" and "algorithmical" queue specifications can be found in [21].

It is important to note that Lotos does not preclude algorithmic specifications. For implementation purposes, for instance, the *Unique_refs* process of Annex 2 could be replaced by the following definition which is similar to the Pascal algorithm above and allows for efficient implementation as discussed in [31]:

```
process Unique_refs [a, t]
        (ref_set: ref_set_sort,
         next_ref: reference_sort): noexit :=
a ?NC_id: NCEP_id_sort
  ?T_suf: T_suffix_sort
  ?EP_id: TCEP_id_sort
  ?PDU: p_info
  ?remote_N_addr: N address_sort
  !next_ref
  ?d: direction ?accepted: Bool;
  find_next-ref [a, t] (Insert (next_ref, ref_set),
                        next_ref, next_ref)
[ ] * * * * * (* termination *)
where process find_next_ref [a, t]
   (ref-set: ref_set_sort,
    ref: reference_sort,
    last-ref: reference_sort): noexit :=
(       [ref eq max_ref] → exit(1)
 [ ]    [not(ref eq max_ref)] → exit(succ(ref)))
        >> accept r: reference_sort in
(       [r Not_In ref_set] → Unique_refs [a, t]
          (ref_set,r)
 [ ]    [r eq last_ref] → * * * * *(* too many
          connections *)
 [ ]    [r In ref_set] → find_next_ref(ref_set, r,
          last_ref))
end proc  end proc
```

4.4. Process Structures

4.4.1. Static and Dynamic Structures

The Estelle module instances and the SDL processes (in the following simply called "processes") usually represent a somehow stable processing entity. They can naturally be mapped to implementation structures, e.g. "tasks" in operating systems or Ada programs, or "processes" in Modula. A given specification may either use the process creation and deletion facilities of the language for creating dynamically changing process structures, or use a static structure of processes which can be established during the "initialization" phase. This latter approach has been taken in the example Transport protocol specifications. The static part of the specification structures is shown in Figs. 2–5.

A specification structure with dynamically created processes has been taken in the Estelle Transport protocol specification of [24]. Here the protocol entity module creates a new *AP_closed* submodule (called *General-TPM_body* in [24]) for each new connection to be established. The created *AP_closed* submodule looks after the connection establishment phase and is replaced by a class-specific submodule *AP_open_i* (called *class_i_TPM_body*) which looks after the data transfer and disconnection phase according to the selected protocol class. The latter submodule is deleted at the end of the disconnection phase.

In LOTOS, the dynamic invocation of modules is an essential feature, since it is the only mechanism in the language for describing loops. For instance, the T_5 transition in the *AP_open* process ends with the invocation of a new version of an *AP-open* process replacing the exiting one. Dynamic process creation is used by the *Map* module which creates a *PDU_handler* process for each new connection. This substructure within the *Map* module could not be used for the Estelle specification since *one* module (here the *Map*) requires information concerning multiplexing for several connections. In the LOTOS specification of Annex 2, this information is shared among the diffent submodules through the global gates *a*, *t*, *ps*, and *pr*.

In SDL, the possible module structuring meth-

ods are similar to those of Estelle. For instance, the signal routes R2, R3 and R4 shown in Fig. 4 correspond to the connections shown in Fig. 2. However, an important difference is the fact that, in SDL, the destination module (i.e. SDL process) must sometimes be addressed by its identifier (or name), while in Estelle the local name of the interaction point of the sending module is always used for this purpose. The destination modules is determined indirectly through the interconnection structure between the Estelle modules. This allows for a more modular specification style, since the sending module does not need to know the name of the destination. In the SDL example of Annex 3, this difference requires for instance three additional variables in the *AP* process, and corresponding parameters in the interactions which are used for providing the identity of the sending or receiving processes.

4.4.2. Constraint-oriented Process Structures

The multi-way rendezvous interaction provided by LOTOS is a very powerful mechanism which allows the specification of an interaction involving more than two processes. Each of the processes may add its own constraints for the interaction, such as conditions about parameter values or particular ordering between different interactions. A simple example is the "assign" interaction at the *a* gate in Annex 2 which involves the following processes: One *AP* process (order constraint), the *Map* process (no constraint), the *Unique_refs* process (constraint on *local_ref* parameter), and one *NC_handler* process (constraint on remote Network address in PDU parameter). All constraints must be satisfied for such an interaction to occur.

The use of constraint-oriented process structures for LOTOS specifications has been advocated in [37,17] and is systematically applied in [28]. The idea is related to path expressions [14] which also allow the separate specification of constraints on the access of shared ressources, and the order in which the user processes wish to access these ressources. As shown by a comparison of Fig. 3 with Fig. 2, the constraint-oriented specification style provided by LOTOS, combined with the introduction of internal events at the gates *a*, *t*, *ps* and *pr*, leads to a quite modular specification structure.

5. Concluding Remarks

Implementations in software or hardware are usually obtained through a process of step-wise refinement which leads from requirements specifications, possibly through several stages of design or implementation specifications, to the final product. An important attribute of a specification language is its ability to express abstract specification which can be used as requirement or design specification without implying any design or implementation choices which would be left open at that stage.

The following properties of LOTOS make it particularly suitable for writing abstract specifications:

- assertional specification (see Section 4.3),
- synchronous communication (see Section 4.1),
- process structure without an implementation model (see Section 4.4.1), and
- multi-way rendezvous interactions, allowing a constraint-oriented specification style (see Section 4.4.2)

On the other hand, these same properties also make it more difficult to generate implementations from LOTOS specifications, as compared to specifications written in Estelle or SDL. In fact, the assertional parts of specifications must be replaced by an equivalent algorithmical part in order to proceed to implementation. The implementation of the queued inter-module communication of Estelle and SDL can be implemented in a straightforward manner by message passing primitives within a multi-programming operating system or a truly distributed system; the implementation of rendezvous is more complex in such environments. The mapping of SDL and Estelle modules into software structures can be performed in different manners. Each implementation support environment for these languages [4] usually provides such a mapping. In the case of LOTOS, it seems necessary to provide several different mappings which are selected depending on the particular specification structure.

The example specifications of the simplified Transport protocol in the annexes show that specifications with similar structure can be written using the different FDTs, Estelle, LOTOS and SDL. However, these examples also indicate the important differences mentioned above. In fact, it

seems that these languages emphasize different stages within the implementation process. While LOTOS is oriented towards abstract specifications, Estelle and SDL specifications tend to be closer to implementation.

The usefulness of a specification language not only depends on its ability to express the systems properties at the appropriate level of detail, but also on the availability of tools for the development of specifications, their validation and implementation. The development of such tools for these languages, which are still relatively new, is an area of much activity [4]. Further experience with these languages and related tools is necessary for obtaining a complete evaluation of the languages.

Acknowledgment

The author would like to thank many people, and in particular Michel Deslauriers and Daniel Ouimet, who contributed to improving various versions of the example specifications. The original version of this paper was written in 1986–87 during a sabbatical year at Siemens AG, Munich, FRG. Financial support from the Natural Sciences and Engineering Research Council of Canada is also acknowledged.

Annex 1. Simplified Transport Protocol Specification in Estelle

Author: G. v. Bochmann (Originally written February 1984, updated January 1987 and March 1989)

```
specification simple_TP;

default  individual queue; (* all queues are individual by default *)

const  maxdata = any integer;
   (* the maximum size of any piece of data
      that the specification may handle *)
   max_TCEP_id = any integer;
   max_NCEP_id = any integer;
   max_T_suffix = any integer;

type
   octet = 0 .. 255;
   len_type = 0 .. maxdata;
   data_type = record
      l : len_type; (* length of data *)
      d : array [1 .. maxdata] of octet; (* the actual data *)
   end;
   (* the standard routines for *******)

(*** Definition of Transport Service Primitives ***)
(************************************************)

    option_type = ...; (* expedited data, etc. *)
    options_type = set of option_type;
    reason_type = (TS_user_initiated, error, procedure_error (* etc. *) );
    N_address_type = ...;
    T_suffix_type = 1..max_T_suffix;
    T_address_type = record
       N_prefix : N_address_type;
       T_suffix : T_suffix_type;
       end;
    seq_number_type = 0 .. 127;
    credit_type = 0..15;
```

```
channel TCEP_primitives (user, provider);
   by user :
      TCONreq  (dest_address : T_address_type;
                proposed_options : options_type);
                    (* connect request *)
      TCONresp (accepted_options : options_type);
                    (* connect response *)
      TDISreq  ;  (* disconnect request *)
      TDATAreq (TS_user_data : data_type;
                EoSDU : boolean); (* sending data *)
      U_READY (credits : credit_type);
                  (* ready for n additional blocks *)
   by provider :
      TCONind  (source_address : T_address_type;
                proposed_options : options_type);
                    (* connect indication *)
      TCONconf (accepted_options : options_type);
                    (* connect confirmation *)
      TDISind  (DIS_reason : reason_type);
                    (* disconnect indication *)
      TDISconf; (* connection is terminated *)
      TDATAind (TS_user_data : data_type;
                EoSDU : boolean); (* receiving data *)
      READY;   (* ready for one additional block *)


(* Definition of Network Service Primitives ***)
(*********************************************)

channel NCEP_primitives (user, provider);
   (* similar to TCEP_primitives, in particular the primitives *)
   by user :    NDATAreq (NSDU_fragment : data_type;
                  is_last_fragment_of_NSDU : boolean); (* sending data *)
   by provider: NDATAind (NSDU_fragment : data_type;
                  is_last_fragment_of_NSDU : boolean); (* receiving data *)

(*** Definition of the Transport Entity ***)
(*****************************************)

type  NCEP_id_type = 1..max_NCEP_id;
      TCEP_id_type = 1..max_TCEP_id;

module TP_entity;
   ip  TS : array [T_suffix_type, TCEP_id_type] of
                                    TCEP_primitives (provider);
       NS : array [NCEP_id_type] of  NCEP_primitives (user);
   end;


body TP_body for TP_entity;

(*** Definitions Internal to the Transport Entity ***)
(***************************************************)

const
    max_ref = 65535; (* 2**16 - 1 *)

type

    reference_type = 0 .. max_ref  (* 0 .. (2**16 - 1)  *);
    order_type = (first, fast, normal, destructive);

    TPDU_code_type = (CR, CC, DR, DC, DT, AK, undefined_code);
```

```
    TPDU_and_control_information = record
       (* control information *)
       full : boolean;
       order : order_type;
       peer_address : T_address_type;

       (* fields of TPDU *)
       credit_value : credit_type; (* used for CR, CC, AK *)
       dest_ref : reference_type;
           (* used for CC, DR, DC, DT (class 2 only),
                       EDT, AK, EAK, ERR *)
       source_ref : reference_type; (* used for CR, CC, DR, DC *)
       user_data : (* optional *) data_type; (* see TS *)
           (* used for CR, CC, DR (not in this version
              of the protocol), DT, EDT *)
       case kind : TPDU_code_type of
       CR, CC : (
          options_ind : options_type; (* see TS *)
          TSAP_id_calling,
          TSAP_id_called (* used only for CR *) : T_suffix_type);
       DR : (
          is_last_PDU : boolean; (* control information *)
          disconnect_reason : reason_type);
       DC :();
       DT : (
          send_sequence : seq_number_type;
          end_of_TSDU : boolean );
       AK : (
          expected_send_sequence : seq_number_type);
       undefined_code : ();
       end;

channel  PDU_and_control (protocol, mapping);
      by protocol, mapping :
         transfer (PDU : TPDU_and_control_information);
         terminated;
      by mapping :
         ready; (* ready for one more block *)

   (* end PDU_and_control *)
(*** Definition of the Map Module ***)
(************************************)

module Map systemactivity ;
   ip   AP : array [T_suffix_type, TCEP_id_type] of
             PDU_and_control (mapping);
        NS : array [NCEP_id_type] of NCEP_primitives (user);
   end;

body Map_body for Map;

 type
    reference_set_type = set of reference_type;

 var
    TC : array [T_suffix_type, TCEP_id_type] of record
       local_ref : reference_type;
       remote_ref : reference_type;
       assigned_NC : (* optional *) NCEP_id_type;
       PDU_buffer : array [TPDU_code_type] of
                    TPDU_and_control_information
       end;
```

```
    NC : array [NCEP_id_type] of record
       remote_N_addr : N_address_type; (* see NS *)
       end;

    active_references : reference_set_type;

   (* The following variables are not contributing
      to the state space of the module *)
   T_suf : T_suffix_type;    EP_id : TCEP_id_type;
   (* used in "when AP[T_suf, EP_id]..." clause to receive on any value
      of T_suf and EP_id *)
   NC_id : NCEP_id_type;
   (* used in "when NS[NC_id]..." clause to receive on any value of NC_id. *)

(*** Definition of Local Functions and Procedures ***)

function exists_TC (NC_id : NCEP_id_type;
                    ref : reference_type) : boolean;
         primitive;
    (* determines whether a TC already exists, i.e. such that
       TC[TC].assigned_NC = NC_id  and  TC[TC].local_ref = ref *)

function find_T_suffix (NC_id : NCEP_id_type;
                    ref : reference_type) : T_suffix_type;
         primitive;

function find_EP_id (NC_id : NCEP_id_type;
                    ref : reference_type) : TCEP_id_type;
         primitive;
procedure assign_new_ref (var new_ref : reference_type;
                          var active_refs : reference_set_type);
   var ref : reference_type;
   begin
      if exist ref : reference_type
         suchthat not(ref in active_refs)
      then begin new_ref := ref;
                 active_refs := active_refs + [new_ref];
           end
      else *****  (* too many connections *)
   end;

function form_T_address (N_address: N_address_type;
                         T_suffix: T_suffix_type) : T_address_type;
         primitive;
         (* forms a Transport address from N_address and T_suffix. *)

procedure assign_new_TCEP_id (var new_EP_id : TCEP_id_type);
         primitive;

function order_constraint (T_suf : T_suffix_type;
                           EP_id : TCEP_id_type;
                           kind : TPDU_code_type) : boolean;
   var OK : boolean;
   begin
      OK := true;
      with TC[T_suf, EP_id] do all k : TPDU_code_type do
         if (k <> kind)
            and PDU_buffer[k].full
            and (PDU_buffer[k].order < PDU_buffer[kind].order)
         then OK := false;
      order_constraint := OK;
      end;
procedure close_and_clear_buffers
       (T_suf : T_suffix_type;  EP_id : TCEP_id_type);
    const undefined = any NCEP_id_type;
```

```
    var  kind : TPDU_code_type;
    begin with TC[T_suf, EP_id] do begin
       assigned_NC := undefined;
       active_references := active_references - [local_ref];
       for kind := CR to AK do PDU_buffer [kind]. full := false;
       end end;

procedure encode_PDU (PDU : TPDU_and_control_information;
                      d: data_type);
         primitive;
  procedure decode (d: data_type;
                    PDU : TPDU_and_control_information);
           primitive;


  initialize
     var kind : TPDU_code_type;
     begin
     all T_suf : T_suffix_type do
     all EP_id : TCEP_id_type do  begin
        close_and_clear_buffers (T_suf, EP_id);
        with TC[T_suf, EP_id] do
           begin
           for kind := CR to AK do
              PDU_buffer[kind].is_last_PDU := false;
           PDU_buffer[DC].is_last_PDU := true;
           end;
        end;
     end;

  (*** Transitions of the Map Module ***)
  (*************************************)

  (* handling interactions from the AP module *)
  trans
  when AP[T_suf, EP_id].transfer (* PDU *)
        (* this input may occur with ANY value of T_suf, EP_id *)
     begin
        TC[T_suf, EP_id]. PDU_buffer [PDU.kind] := PDU;
        with TC[T_suf, EP_id]. PDU_buffer [PDU.kind] do begin
           full := true;
     end end;

  when AP [T_suf, EP_id].terminated
     begin close_and_clear_buffers (T_suf, EP_id)  end;

  (* assignment of Network connections for outgoing calls *)
  trans
  any T_suf : T_suffix_type;
      EP_id : TCEP_id_type;
      NC_id : NCEP_id_type do
      provided  TC[T_suf, EP_id].PDU_buffer[CR].full
            and (TC[T_suf, EP_id].PDU_buffer [CR].
                        peer_address.N_prefix =
                                       NC[NC_id].remote_N_addr)
        begin with TC[T_suf, EP_id], NC[NC_id] do begin
           assigned_NC := NC_id;
           assign_new_ref (local_ref, active_references);
        end  end;

  (* similarly: in the case that no suitable Network
  connection exists, a TDISind must be returned to the user; this
  may be initiated by sending a DR PDU to the AP module *)
  (* send a TPDU *)
```

T_b

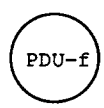
T_a

```
trans
any NC_id : NCEP_id_type;
    T_suf : T_suffix_type;
    EP_id : TCEP_id_type;
    kind : TPDU_code_type  do
    provided TC[T_suf, EP_id].PDU_buffer[kind].full
       and   (TC[T_suf, EP_id].assigned_NC = NC_id)
    (* and   flow control to Network ready *)
       and order_constraint (T_suf, EP_id, kind)
          var NSDU : data_type;
    begin with TC[T_suf, EP_id] do begin
       with PDU_buffer[kind] do begin
          if kind = CR
          then begin
             TSAP_id_calling := T_suf;
             TSAP_id_called := peer_address.T_suffix;
             dest_ref := 0;
             end
          else dest_ref := TC[T_suf, EP_id].remote_ref;
          if kind in [CR, CC, DR, DC]
             then source_ref := TC[T_suf, EP_id].local_ref;
          end;
       encode_PDU (PDU_buffer [kind], NSDU);
       output NS[NC_id]. NDATAreq (NSDU, true);
       PDU_buffer[kind].full := false;
       if PDU_buffer[kind].order = destructive
          then all k : TPDU_code_type do PDU_buffer[k].full :=
                                                        false;
       if (kind = DC) or
          ((kind = DR)  and  PDU_buffer[kind].is_last_PDU)
       then begin
          close_and_clear_buffers (T_suf, EP_id);
          output AP [T_suf, EP_id]. terminated
          end;
       if kind = DT
       then output AP [T_suf, EP_id] . ready;
       end  end;

(* handling of incoming PDU's *)
trans
when NS[NC_id]. NDATAind
                  (* NSDU_fragment, is_last_fragment_of_NSDU *)
       (* Assumption: the fragment contains exactly one TSDU  *)
       (* Note: flow control to the Transport entity is always ready *)
    var
       received_PDU : TPDU_and_control_information;
       (* used to decode NSDU fragments into received_PDU *)
    begin
       decode (NSDU_fragment, received_PDU);
       with received_PDU do begin
          if exists_TC (NC_id, dest_ref)
          then begin
             T_suf := find_T_suffix (NC_id, dest_ref);
             EP_id := find_EP_id (NC_id, dest_ref)  end

          else (* a new TC must be created if PDU is CR *)
             if kind = CR
             then begin
                peer_address := form_T_address
                      (NC[NC_id].remote_N_addr, TSAP_id_calling);
                (* Assumption: the address is valid,
                   and another connection to that address
                   can be supported *)
                T_suf := TSAP_id_called;
```

T_s S_1 S_2 T_r

```
                assign_new_TCEP_id (EP_id); (* such that
                                not AP[T_suf, EP_id].in_use *)
                with TC[T_suf, EP_id] do begin
                   assign_new_ref (local_ref, active_references);
                   remote_ref := source_ref;
                   assigned_NC := NC_id;
                   end;
             end
             else  (* error *);
          case kind of
             CR: (* error *);
             CC : TC[T_suf, EP_id].remote_ref := source_ref;
             DR,DC,DT,AK: ;
             end;
          output AP [T_suf, EP_id].transfer(received_PDU) ;
          end;
   end;

end (* Map_body *);


 (*** Definition of the AP Module ***)
 (***********************************)

 module AP_type systemactivity;
    ip  TS : TCEP_primitives (provider);
        Map : PDU_and_control (protocol) ;
    end;

 body AP_body for AP_type;

 var
    opt : options_type;
    TRseq,
    TSseq : seq_number_type;
    R_credit,
    S_credit : credit_type;
    user_ready : integer;
    map_ready : boolean;
    state  closed, wait_for_CC, wait_for_TCONresp,
           open, wait_for_DC, closing;

 stateset
    any_state = [ closed, wait_for_CC, wait_for_TCONresp,
                  open, wait_for_DC, closing ];

 (*** Definition of Local Functions and Procedures ***)

 function impl_choice : boolean;
          primitive;

 (* PDU definitions:    The following functions have merely the role
             of assembling their parameters into a record data structure *)

function CR_PDU (to_adr : T_address_type;
                 o : options_type;
                 c : seq_number_type)
                              : TPDU_and_control_information;
   var PDU : TPDU_and_control_information;
   begin
      with PDU do begin
         kind := CR;          peer_address := to_adr;
         options_ind := o;    credit_value := c;
```


```
         order := first;       end;
      CR_PDU := PDU;
      end;

function CC_PDU (o : options_type;
                 c : seq_number_type)
                                 : TPDU_and_control_information;
   var PDU : TPDU_and_control_information;
   begin
      with PDU do begin
         kind := CC;           options_ind := o;
         credit_value := c;    order := first;     end;
      CC_PDU := PDU;
      end;

function DR_PDU (r : reason_type;
                 last_PDU : boolean)
                                 : TPDU_and_control_information;
   var PDU : TPDU_and_control_information;
   begin
      with PDU do begin
         kind := DR;                disconnect_reason := r;
         is_last_PDU := last_PDU;   order := destructive;   end;
      DR_PDU := PDU;
      end;

function DC_PDU : TPDU_and_control_information;
   var PDU : TPDU_and_control_information;
   begin
      with PDU do begin
         kind := DC;     order := destructive;   end;
      DC_PDU := PDU;
      end;
```




```
function DT_PDU (s : seq_number_type;
                 d : data_type;
                 e : boolean)
                                 : TPDU_and_control_information;
   var PDU : TPDU_and_control_information;
   begin
      with PDU do begin
         kind := DT;           send_sequence := s;
         user_data := d;       end_of_TSDU := e;
         order := normal;      end;
      DT_PDU := PDU;
      end;

function AK_PDU (s : seq_number_type;
                 c : seq_number_type)
                                 : TPDU_and_control_information;
   var PDU : TPDU_and_control_information;
   begin
      with PDU do begin
         kind := AK;           expected_send_sequence := s;
         credit_value := c;    order := fast;    end;
      AK_PDU := PDU;
      end;

initialize to closed begin end;
(*** Transitions of the AP Module ***)
(************************************)

(*** Connection Establishment ***)
```

```
trans
when TS.TCONreq   (* dest_address, proposed_options *)
   from closed  to wait_for_CC
   begin
      opt := proposed_options;
      output Map.transfer
                 (CR_PDU (dest_address, opt, R_credit));
      end;
```

T_1

```
when Map.transfer  (* PDU *)  provided ( PDU.kind = CC )
      and ( PDU.options_ind  <= opt )
   from wait_for_CC  to open
   begin
      opt := PDU.options_ind;
      TRseq := 0;
      TSseq := 0;
      S_credit := PDU.credit_value;
      output TS.TCONconf (opt);
      end;
```

T_{2a}

```
when Map.transfer (* PDU *)  provided (PDU.kind = CC)
      and not (PDU.options_ind  <=  opt)
   from wait_for_CC  to wait_for_DC
   begin
      output TS.TDISind (procedure_error);
      output Map.transfer (DR_PDU (procedure_error, false));
      end;
```

T_{2b}

```
when Map.transfer   provided PDU.kind = DR
   from wait_for_CC
      to closed
      begin
         output TS.TDISind (PDU.disconnect_reason);
         output Map.terminated;
         end;
```

T_{2c}

```
when Map.transfer  (* PDU *)  provided  (PDU.kind = CR)
      and impl_choice (* assumption: requested options
                  are supported by implementation.
                  The opposite case is not considered here;
                  it could be described by another transition *)
   from closed  to wait_for_TCONresp
   begin
      opt := PDU.options_ind;
      S_credit := PDU.credit_value;
      output TS.TCONind (PDU.peer_address, opt);
      end;
```

T_3

```
when TS.TCONresp (* accepted_options *)
   from wait_for_TCONresp  to open
   provided accepted_options  <=  opt
   begin
      opt := accepted_options;
      TRseq := 0;
      TSseq := 0;
      output Map.transfer (CC_PDU (opt, R_credit));
      end;
```

T_{4a}

```
(* refusal by the user of a connection indication *)
when TS.TDISreq
   from wait_for_TCONresp
      to closing
      begin
        output Map.transfer (DR_PDU (TS_user_initiated, true));
        end;
```

T_{4c}

```
(*** Connection Termination ***)

(* disconnect initiative by local user *)
when TS.TDISreq
   from open
         to wait_for_DC
         begin
            output Map.transfer
                        (DR_PDU (TS_user_initiated, false));
            end;
```

T_9

```
when Map.transfer   provided PDU.kind = DC
   from wait_for_DC  to closed
       begin
          output Map.terminated;
          output TS. TDISconf;
       end;
```

T_{10a}

```
(* disconnect collision *)
when Map.transfer   provided PDU.kind = DR
   from wait_for_DC  to closed
      begin
         output Map.terminated
      end;
```

T_{10b}

```
   (* disconnect initiative by Transport entity *)

trans
any reason : reason_type do
from open to wait_for_DC
      provided reason <> TS_user_initiated
   begin
      output TS.TDISind (reason);
      output Map.transfer (DR_PDU (reason, false));
   end;
```

T_{11}

```
(* remote disconnect initiative *)
trans
when Map.transfer   provided PDU.kind = DR
   from open
      to closing
      begin
         output TS.TDISind (PDU.disconnect_reason);
         output Map.transfer (DC_PDU);
         end;
```

T_{12a}

```
when Map. terminated
   from closing to closed
      begin
         output TS. TDISconf
      end;
```

T_{12b}

```
(*** Normal Data Transfer ***)

(* sending data *)
trans
when TS.TDATAreq (* TS_user_data, EoSDU *)
     (* Note: the user determines the size of the DT PDU
        which will contain the complete TDATAreq data fragment *)
   provided S_credit > 0
   from open to same
      begin
         S_credit := S_credit - 1;
         output Map.transfer (DT_PDU (TSseq, TS_user_data, EoSDU));
```

T_5

```
            TSseq := (TSseq + 1) mod 128;
            map_ready := false;
            end;

(* receiving data *)
trans
when Map.transfer (* PDU *)
   provided PDU.kind = DT
   from open  to same
      begin
         if (R_credit <> 0)  and  (PDU.send_sequence = TRseq)
         then begin
            TRseq := (TRseq + 1) mod 128;
            R_credit := R_credit - 1;
            output TS.TDATAind (PDU.user_data, PDU.end_of_TSDU);
            end
         else  (* error *)
         end;

(* acknowledgements *)

trans
when TS. U_READY (* credits *)
   begin  R_credit := R_credit + credits   end;

trans
from open  to same
   begin
      output Map.transfer (AK_PDU (TRseq, R_credit));
      end;

trans
when Map.transfer (* PDU *)   provided PDU.kind = AK
   from open  to same
         var new_credit : integer;
   begin
         if TSseq < PDU.expected_send_sequence
         then new_credit := PDU.credit_value + PDU.expected_send_sequence
                          - (TSseq + 128)
         else new_credit := PDU.credit_value + PDU.expected_send_sequence
                          - TSseq;
         if (new_credit >=0) and (new_credit <= 15)
         then S_credit := new_credit
         else  (* error *);
      end;

when Map. ready
   begin map_ready := true  end;

trans
provided  map_ready and (S_credit > 0)
   begin  output TS.READY  end;

end; (* AP_body *)


(*** Part of Transport Entity Body: Creation of Submodules ***)
(*************************************************************)

modvar
   m : Map;
   aps : array [T_suffix_type, TCEP_id_type] of AP_type;
```

T_6 T_{7a} T_{7b} T_{8a} T_{8b} T_{8c}

```
initialize begin
   init m with Map_body;
   all T_suf : T_suffix_type do
   all EP_id : TCEP_id_type do begin
      init aps [T_suf, EP_id] with AP_body;
      connect m.AP [T_suf, EP_id]  to  aps [T_suf, EP_id] . Map;
      attach TS [T_suf, EP_id]  to  aps [T_suf, EP_id] . TS;
      end;
   all NC_id : NCEP_id_type do
      attach  NS [NC_id]  to  m . NS [NC_id];
   end;

end; (* TP_body *)

end. (* of specification *)
```

Annex 2. Simplified Transport Protocol Specification in LOTOS

Author: G. v. Bochmann (April 1987, revised July 1988 and March 1989)

```
(*

Notes:

(1) The abbreviated notations described in "Potential Enhancements to LOTOS"
(ISO 97/21 N1540) are used for the description of data types.
This simplifies the notation, compared with what is allowed according
to standard Lotos.

(2) Certain parts of the protocol, in particular error cases are not
completed. These parts are indicated by *****.

(3) A corresponding (complete) specification of this simplified Transport
protocol in standard LOTOS is available from the author.

*)

specification simple_TP [TS, NS] (tc_ids : TCid_set, nc_ids : NCid_set)
                                 : noexit

(* Library definitions *)
library
        Boolean, Set, String, OctetString, DecNatRepr,
        NaturalNumber (* assumed to include the minus operation
                         and the constants 1 and 15 *),
        Element, BasicNonEmptyString
endlib

(*** Global Type Definitions ***)
(*******************************)

type T_suffix_type is Boolean
     sorts T_suffix_sort
     opns  _eq_ : T_suffix_sort, T_suffix_sort -> Bool
           T_suffix_1 : -> T_suffix_sort
           (* assumed to include the eq equations *)
endtype

type N_address_type is
     sorts N_address_sort
     opns  find_remote_N_addr :  -> N_address_sort
endtype
```

```
 type T_address_sort is Tuple make_T_addr comp  (* abbreviated notation *)
      N_prefix : N_address_sort,
      T_suffix : T_suffix_sort

 (* can be written in standard Lotos as
 type T_address_sort is N_address_type, T_suffix_type
      sorts T_address_sort
      opns
        make_T_addr : N_address_sort, T_suffix_sort -> T_address_sort
        N_prefix    : T_address_sort   ->  N_address_sort
        T_suffix    : T_address_sort   ->  T_suffix_sort

      eqns forall N_p : N_address_sort,  T_s : T_suffix_sort
           ofsort N_address_sort
                  N_prefix (make_T_addr (N_p, T_s)) = N_p
           ofsort T_suffix_sort
                  T_suffix (make_T_addr (N_p, T_s)) = T_s
 *)
 endtype

 type reason_type is
      sorts reason_sort
      opns
        TS_user_initiated : -> reason_sort
        remote_initiated  : -> reason_sort
        (* ***** *)
 endtype

 type direction is
      sorts direction
      opns
        up   : -> direction
        down : -> direction
 endtype

 type credit_type is DecNatRepr
      renamedby sortnames credit_sort for Nat
 endtype

 type TCEP_id_type is
      sorts TCEP_id_sort
      (* for execution, some constant values must be defined *)
 endtype

 type option_type is
      sorts option_sort
      opns expedited_data : -> option_sort
        (* ***** *)
 endtype

 type options_sort is SetOf option_type  (* abbreviated notation *)
 (* can be written in standard Lotos as
 type options_sort_1 is Set
      actualizedby option_type, Boolean,
      using sortnames option_sort for Element
                      Bool        for FBool
 endtype
 type options_type is options_sort_1
      renamedby sortnames options_sort for Set
 *)
 endtype

(*** Definition of Service Primitives ***)
(****************************************)

(* TCEP_primitives, using abbreviated notation as for T_address_sort; "for user"
```

```
means that this service primitive is created by the service user only *)
type TCONreq is Tuple make_TCONreq (* for user *) comp
     dest_address     : T_address_sort,
     proposed_options : options_sort
endtype

type TCONind is Tuple make_TCONind comp
     source_address   : T_address_sort,
     proposed_options : options_sort
endtype

type TCONresp is Tuple make_TCONresp (* for user *) comp
     accepted_options : option_sort
endtype

type TCONconf is Tuple make_TCONconf comp
     accepted_options : options_sort
endtype

type TDISreq is
     sorts TDISreq
     opns
       make_TDISreq : -> TDISreq (* for user *)
endtype

type TDISind is Tuple make_TDISind comp
     DIS_reason : reason_sort
endtype

type TDATAreq is Tuple make_TDATAreq (* for user *) comp
     TS_user_data : data_sort,
     EoTSDU       : boolean
endtype

type TDATAind is Tuple make_TDATAind comp
     TS_user_data : data_sort,
     EoTSDU       : boolean
endtype

type U_READY is Tuple make_U_READY (* for user *) comp
     credits : credit_or_seq_sort
endtype


(* NCEP primitives *)

type NDATAreq is Tuple make_NDATAreq comp
     NS_user_data : OctetString,
     EoNSDU       : Bool
endtype

type NDATAind is Tuple make_NDATAind comp
     NS_user_data : OctetString,
     EoNSDU       : Bool
endtype
(*** Definition of Transport Entity ***)
(**************************************)

type NCEP_id_type is
     sorts NCEP_id_sort
     (* for execution, some constant values must be defined *)
endtype
```

```
type NCid_set is SetOf NCEP_id_sort    (* abbreviated notation *)
endtype

type TCid_set is SetOf TCid_pair    (* abbreviated notation *)
endtype

type TCid_pair is Tuple make_T_id comp    (* abbreviated notation *)
     comp1 : T_suffix_sort,
     comp2 : TCEP_id_sort
endtype

type TCid_pair_set is SetOf TCid_set    (* abbreviated notation *)
endtype


behavior TP_entity [TS, NS] (tc_ids, nc_ids)
where

process TP_entity [TS, NS] (tc_ids : TCid_set, nc_ids : NCid_set) : noexit :=

(* Notes :

(1) The TP entity contains a single Map process which contains a
    PDU_handler per active Transport connection.

(2) A transport connection is identified either by the pair of
    T_suffix and TCEP identifier or by the pair NCEP identifier and
    local reference.

(3) There is one AP_closed process (or AP_open, AP_disc, AP_wait_for_DC
    into which the AP_closed process transforms) per pair of T_suffix and
    TCEP identifier.

(4) There is one NC_manager process per NCEP identifier (i.e.
    per Network connection).

(5) The following types of parameters are exchanged during the
    interactions at the gates :
TS : T_suf : T_suffix_sort
     EP_id : TCEP_id_sort
     TCONreq | TCONind | TCONresp  | TCONconf | TDISreq | *** etc.

a :  NC_id : NCEP_id_sort
     T_suf : T_suffix_sort
     EP_id : TCEP_id_sort
     PDU   : p_info
     N_addr: N_address_sort
     local_ref : reference_sort
     d : direction
     accepted  : boolean

t :  T_suf : T_suffix_sort
     EP_id : TCEP_id_sort
     NC_id : NCEP_id_sort
     ref   : reference_sort

ps : T_suf : T_suffix_sort
     EP_id : TCEP_id_sort
     PDU   : p_info

pr : T_suf : T_suffix_sort
     EP_id : TCEP_id_sort
     PDU   : p_info
```

```
NS : NC_id : NCEP_id_sort
     NDATA_sort | *****

(6)  A single Network service access point (NSAP) is assumed.
*)

hide pr,ps,a,t  in
   (AP_modules [TS, pr, ps, a, t] (tc_ids)
     |[pr,ps,a,t]|
    Map [NS, pr, ps, a, t]
     |[a,t]|
    NC_managers[a,t](nc_ids)
    |[a,t]|
    Unique_refs [a, t] ({} of ref_set_sort)
   )
where

process AP_modules [TS, pr, ps, a, t] (tc_ids : TCid_set) : noexit :=
   choice tc_id : TCid_pair [] [tc_id IsIn tc_ids] ->
       (AP_closed [TS, pr, ps, a, t]
                  (comp1 (tc_id), comp2 (tc_id), 0 of credit_sort)
        |||
        i; AP_modules [TS, pr, ps, a, t] (Remove (tc_id, tc_ids)))
endproc

process NC_managers [a, t] (nc_ids : NCid_set) : noexit :=
   choice NC_id : NCEP_id_sort [] [NC_id IsIn nc_ids] ->
       (NC_manager [a, t] (NC_id, find_remote_N_addr)
        |||
        i; NC_managers [a, t] (Remove (NC_id, nc_ids)))
endproc


(*** Type Definitions Internal to the Transport Entity ***)
(*********************************************************)

type reference_type is NaturalNumber
     renamedby sortnames reference_sort for Nat
endtype

type ref_set_sort is SetOf reference_sort    (* abbreviated notation *)
endtype

type seq_number_type_1 is DecNatRepr
     renamedby sortnames seq_number_sort  for Nat
endtype

type seq_number_type is seq_number_type_1, credit_type
     (* has properties of natural numbers; modulo arithmetic and the minus
        operation are defined; conversion to credit type is required
        for type checking consistency of expressions describing the
        reception of AK PDU's *)
     opns
       convert_to_credit : seq_number_sort                  -> credit_sort
       Pred              : seq_number_sort                  -> seq_number_sort
       _-_               : seq_number_sort, seq_number_sort -> seq_number_sort
       _mod_             : seq_number_sort, seq_number_sort -> seq_number_sort
       128               :                                  -> seq_number_sort
       1                 :                                  -> seq_number_sort
     eqns  forall m, n : seq_number_sort
           ofsort credit_sort
                  convert_to_credit (0)       = 0;
                  convert_to_credit (Succ(n)) = Succ (convert_to_credit (n))
```

```
            ofsort seq_number_sort
                   Pred (Succ (m)) = m;
                   Succ (Pred (m)) = m;
                   m -  0          = m;
                   m -  Succ (n)   = Pred (m) - n;
                   m -  Pred (n)   = Succ (m) - n;
                   m +  Pred (n)   = Pred (m) + n;
                   m *  Pred (n)   = (m * n)  - m;
                   m lt n => m mod n = m;
                   m ge n => m mod n = (m - n) mod n;
                   128 = NatNum (Dec (1) ++ Dec (2) ++ Dec (8));
                   1   = Succ (0)
endtype

type order_type is [destructive, normal, fast, first]
     (* abbreviated notation, can be written in standard Lotos
        through representation as integer constants *)
endtype

(*** PDU definitions ***) (* using abbreviated notation as for T_address_sort *)

type CR_PDU is Tuple CR_PDU comp
     TSAP_id_called  : T_suffix_sort,
     TSAP_id_calling : T_suffix_sort,
     option_ind      : options_sort,
     credit_value    : credit_sort
endtype

type CC_PDU is Tuple CC_PDU comp
     option_ind      : options_sort,
     credit_value    : credit_sort
endtype

type DR_PDU is Tuple DR_PDU comp
    disconnect_reason : reason_sort
endtype

type DT_PDU is Tuple DT_PDU comp
     send_sequence : seq_number_sort
     user_data : data_sort
endtype

type AK_PDU is Tuple AK_PDU comp
     expected_send_sequence : seq_number_sort
     credit_value           : credit_sort
endtype

type p_info is EitherOf
     CR_PDU, CC_PDU, DR_PDU, DC_PDU, DT_PDU, AK_PDU
endtype

(* The above abbreviations can be written in standard Lotos
   in the following form:
type p_info is Boolean
     sorts p_info
     opns
       IsCR_PDU        : p_info -> Bool
       IsCC_PDU        : p_info -> Bool
       IsDR_PDU        : p_info -> Bool
       IsDC_PDU        : p_info -> Bool
       IsDT_PDU        : p_info -> Bool
       IsAK_PDU        : p_info -> Bool
endtype
```

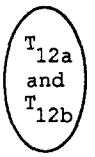
PDU-f

PDU-f

```
type CR_PDU is T_suffix_type, options_type, credit_type, p_info
     opns
       CR_PDU          : T_suffix_sort, T_suffix_sort,
                         options_sort, credit_sort   -> p_info
       TSAP_id_called  : p_info  -> T_suffix_sort
       TSAP_id_calling : p_info  -> T_suffix_sort
       option_ind      : p_info  -> options_sort
       credit_value    : p_info  -> credit_sort

     eqns forall Called, Calling : T_suffix_sort,  Opt : options_sort,
                 Cred    : credit_sort

          ofsort T_suffix_sort
          TSAP_id_called  (CR_PDU (Called, Calling, Opt, Cred)) = Called;
          TSAP_id_calling (CR_PDU (Called, Calling, Opt, Cred)) = Calling

          ofsort options_sort
          option_ind (CR_PDU (Called, Calling, Opt, Cred)) = Opt

          ofsort credit_sort
          credit_value (CR_PDU (Called, Calling, Opt, Cred)) = Cred

          ofsort Bool
          IsCR_PDU  (CR_PDU (Called, Calling, Opt, Cred)) = true;
          IsDC_PDU  (CR_PDU (Called, Calling, Opt, Cred)) = false
endtype

type CC_PDU is ***** etc.
*)

type PDU_type is CR_PDU, CC_PDU, DR_PDU, DC_PDU, DT_PDU, AK_PDU, reference_type
     sorts  PDU_sort
     opns
       decode      : OctetString  ->  PDU_sort
       encode      : PDU_sort     ->  OctetString
       make_PDU    : p_info, reference_sort, reference_sort  -> PDU_sort
       source_ref  : PDU_sort     ->  reference_sort
       dest_ref    : PDU_sort     ->  reference_sort
       p_PDU       : PDU_sort     ->  p_info
       checks      : PDU_sort     ->  Bool
       update_source_ref : reference_sort, PDU_sort  ->  reference_sort

     eqns forall p     : p_info, r1,r2 : reference_sort, PDU   : PDU_sort

          ofsort PDU_sort
          decode (encode (PDU)) = PDU

          ofsort p_info
          p_PDU (make_PDU (p, r1, r2)) = p

          ofsort reference_sort
          dest_ref   (make_PDU (p, r1, r2)) = r1;
          source_ref (make_PDU (p, r1, r2)) = r2;
          r1 ne (0 of reference_sort) => update_source_ref (r1, PDU) = r1;
          r1 eq (0 of reference_sort) => update_source_ref (r1, PDU)
                                                  = source_ref (PDU);

          (* ***** for check (verification of received PDU's) *)
endtype

type PDU_buffer_sort is StringOf p_info  (* abbreviated notation *)
(* can be written in standard Lotos similarly to the "SetOf" construction *)
endtype
```

```
(*** Definition of the "AP Module" ***)
(************************************)

process AP_closed [TS, pr (* receive PDU *), ps (* send PDU *),
                       a (* assign *),       t (* terminate *)  ]
                  (T_suf : T_suffix_sort,   EP_id : TCEP_id_sort,
                   R_credit : credit_sort)    : noexit  :=

(* connection establishment, user initiated *)

TS  !T_suf !EP_id ?tcr : TCONreq;
a   ?NC_id : NCEP_id_sort !T_suf !EP_id                          (T1)
    !CR_PDU (T_suffix (dest_address (tcr)),
             T_suf, proposed_options (tcr), R_credit)
    !N_prefix (dest_address (tcr))
    ?ref : reference_sort !down ?accepted : Bool;
    ([accepted] ->
     (pr !T_suf !EP_id ?PDU : p_info [IsCC_PDU (PDU)];
         ([option_ind_CC (PDU) IsSubsetOf proposed_options (tcr)]
         ->
              TS  !T_suf !EP_id                                  (T2a)
                  !make_TCONconf (option_ind_CC (PDU));
              (AP_open [TS, pr, ps]
                  (T_suf, EP_id, option_ind_CC (PDU), 0 of seq_number_sort,
                      0 of seq_number_sort, R_credit, credit_value_CC (PDU))
              [>  AP_disc [TS, pr, ps, a, t] (T_suf, EP_id)))

         [] pr  !T_suf !EP_id ?PDU : p_info [IsDR_PDU (PDU)];
            TS  !T_suf !EP_id                                    (T2b)
                !make_TDISind (disconnect_reason (PDU));
            t   (* terminate *) !T_suf !EP_id ?id : NCEP_id_sort
                                 ?r : reference_sort;
            AP_closed [TS, pr, ps, a, t] (T_suf, EP_id, 0 of credit_sort)
         (* [] [ ***** otherwise ] -> ***** *)
     )
    (* [] [not accepted] -> ***** *)
    )
[]
TS  !T_suf !EP_id ?tsp : U_READY;                                (T7a)
AP_closed [TS, pr, ps, a, t] (T_suf, EP_id, R_credit + credits (tsp))

[]  (* connection establishment, initiated by peer entity *)

a  ?NC_id : NCEP_id_sort  !T_suf  !EP_id  ?PDU : p_info
   ?remote_N_addr : N_address_sort  ?ref : reference_sort !up ?accepted : Bool
   [IsCR_PDU (PDU) and (TSAP_id_called (PDU) eq T_suf)];
   ([accepted] ->
     TS  !T_suf  !EP_id
         !make_TCONind (make_T_addr (remote_N_addr, TSAP_id_calling (PDU)),
                        option_ind (PDU));
     TS  !T_suf  !EP_id  ?tcr : TCONresp
         [accepted_options (tcr) IsSubsetOf option_ind (PDU)];
     ps  !T_suf  !EP_id
         !CC_PDU (accepted_options (tcr), R_credit);
      (AP_open [TS, pr, ps] (T_suf, EP_id, accepted_options (tcr),
                             0 of seq_number_sort, 0 of seq_number_sort,
                             R_credit, credit_value (PDU))
       [> AP_disc [TS, pr, ps, a, t] (T_suf, EP_id))
   )
where

process AP_open [TS, pr, ps]
     (T_suf : T_suffix_sort, EP_id : TCEP_id_sort, opt : options_sort,
```

```
        TRseq, TSseq : seq_number_sort, R_credit, S_credit : credit_sort)
      : noexit :=

(* receiving data *)

pr  !T_suf  !EP_id  ?PDU : p_info  [IsDT_PDU (PDU)];
    ([ (R_credit ne 0) and (send_sequence (PDU) eq TRseq) ] ->
T6  TS  !T_suf  !EP_id
        !make_TDATAind (user_data (PDU), EoTSDU (PDU));
    AP_open [TS, pr, ps] (T_suf, EP_id, opt, (TRseq + 1) mod 128, TSseq,
              R_credit - 1, S_credit)
    (* []  [ ***** otherwise ] *****  *)
    )

[]  (* receiving credits from user *)

TS  !T_suf  !EP_id  ?tsp : U_READY;
T7a AP_open [TS, pr, ps] (T_suf, EP_id, opt, TRseq, TSseq,
                      R_credit + credits (tsp), S_credit)

[]  (* sending data *)

[ S_credit ge 1 ] -> (TS  !T_suf  !EP_id  ?tdt : TDATAreq;
T5 ps  !T_suf  !EP_id  !DT_PDU (TSseq, TS_user_data (tdt), EoTSDU (tdt));
AP_open [TS, pr, ps] (T_suf, EP_id, opt, TRseq, (TSseq + 1) mod 128,
         R_credit, S_credit - 1))

[]  (* receiving an AK PDU *)

pr  !T_suf  !EP_id  ?PDU : p_info  [IsAK_PDU (PDU)];

(  (    [ TSseq lt expected_send_sequence (PDU) ]  ->  exit (TSseq +128)
     [] [ not (TSseq lt expected_send_sequence (PDU))] -> exit (TSseq)
   ) >> accept s:seq_number_sort in
T8a   (let new_credit : credit_sort = credit_value_AK (PDU) +
           convert_to_credit (expected_send_sequence (PDU) - s)  in
      ( [ (new_credit ge S_credit) and (new_credit le 15 ] ->
        AP_open [TS, pr, ps]
            (T_suf, EP_id, opt, TRseq, TSseq, R_credit, new_credit)
      (* []  [ ***** otherwise ] ***** *)
 )   ))

[]  (* sending an AK PDU *)

i;
T7b ps  !T_suf  !EP_id  !AK_PDU (TRseq, R_credit);
AP_open [TS, pr, ps] (T_suf, EP_id, opt, TRseq, TSseq, R_credit, S_credit)

endproc (* AP_open *)

process AP_disc [TS, pr, ps, a, t] (T_suf : T_suffix_sort, EP_id : TCEP_id_sort)
              : noexit :=

(* disconnect initiated by local user *)

TS  !T_suf  !EP_id  ?tdr : TDISreq;
Ta  ps  !T_suf  !EP_id  !DR_PDU (TS_user_initiated);
AP_wait_for_DC [TS, pr, ps, a, t] (T_suf, EP_id)

[]  (* disconnect initiated by remote entity *)
```




```
pr  !T_suf  !EP_id  ?PDU : p_info [IsDR_PDU (PDU)];
TS  !T_suf  !EP_id  !make_TDISind (disconnect_reason (PDU));
ps  !T_suf  !EP_id  !DC_PDU;
t (* terminate *) !T_suf !EP_id ?NC_id : NCEP_id_sort ?ref : reference_sort;
AP_closed [TS, pr, ps, a, t] (T_suf, EP_id, 0 of credit_sort)

(* [] ***** *)

where

process AP_wait_for_DC [TS, pr, ps, a, t] (T_suf : T_suffix_sort,
                                           EP_id : TCEP_id_sort) : noexit :=
```


```
pr  !T_suf  !EP_id  ?PDU : p_info [IsDC_PDU (PDU)];
t (* terminate *) !T_suf  !EP_id ?id : NCEP_id_sort ?r : reference_sort;
AP_closed [TS, pr, ps, a, t] (T_suf, EP_id, 0 of credit_sort)

[]  (* other PDU's are ignored *)

pr  !T_suf  !EP_id  ?PDU : p_info [not (IsDC_PDU (PDU))];
AP_wait_for_DC [TS, pr, ps, a, t] (T_suf, EP_id)

endproc (* AP_wait_for_DC *)
endproc (* AP_disc        *)
endproc (* AP_closed      *)


(*** Definition of the "Map Module" ***)
(**************************************)

process Map [NS, pr, ps, a, t] : noexit :=

(* new connection requested by the user:
if accepted, a new PDU_handler process is created *)
a  ?NC_id : NCEP_id_sort  ?T_suf : T_suffix_sort  ?EP_id : TCEP_id_sort
   ?PDU : p_info  ?N_addr : N_address_sort
   ?local_ref : reference_sort  !down ?accepted : Bool;
   ([accepted] ->
      (PDU_handler [NS, pr, ps, t]
          (NC_id, T_suf, EP_id, local_ref, 0 of reference_sort,
           String (PDU) of PDU_buffer_sort)
       |||
       Map [NS, pr, ps, a, t]
      )
    (* [] [ ***** otherwise ] ***** *)
   )
```


```
[]
(* new connection requested by the remote entity:
if accepted, a new PDU_handler process is created *)
NS ?NC_id : NCEP_id_sort
   ?ndt : NDATAind [IsCR_PDU (p_PDU (decode (NS_user_data(ndt))))];
a  !NC_id  ?T_suf : T_suffix_sort  ?EP_id : TCEP_id_sort
   !p_PDU(decode (NS_user_data(ndt)))  ?N_addr : N_address_sort
   ?local_ref : reference_sort  !up ?accepted : Bool;
   ([accepted] ->
      (PDU_handler [NS, pr, ps, t]
          (NC_id, T_suf, EP_id, local_ref,
           source_ref (decode (NS_user_data(ndt))),
           <> of PDU_buffer_sort)
       |||
       Map [NS, pr, ps, a, t]
      )
    (* [] [ ***** otherwise ] ***** *)
```

```
where

process PDU_handler [NS, pr, ps, t]
     (NC_id : NCEP_id_sort,T_suf : T_suffix_sort, EP_id : TCEP_id_sort,
      local_ref, remote_ref : reference_sort, buffer : PDU_buffer_sort)
     : exit :=

(* accepting PDU's from the assigned AP_xxxx module *)
ps  !T_suf  !EP_id  ?PDU : p_info;
PDU_handler [NS, pr, ps, t]
     (NC_id, T_suf, EP_id, local_ref, remote_ref, buffer ++ String (PDU))

[]  (* advance fast PDU's over normal ones in the buffer *)
i; PDU_handler [NS, pr, ps, t] (NC_id, T_suf, EP_id, local_ref,
                                remote_ref, advance (buffer))

[]  (* drop PDU's during disconnection phase *)
i; PDU_handler [NS, pr, ps, t] (NC_id, T_suf, EP_id, local_ref,
                                remote_ref, drop (buffer))

[]  (* sending a PDU *)
[buffer ne <>] ->
   NS  !NC_id  !make_NDATAreq (encode (make_PDU (first(buffer),
                              remote_ref, local_ref)),true);
   PDU_handler [NS, pr, ps, t]
        (NC_id, T_suf, EP_id, local_ref, remote_ref, tail (buffer))

[]  (* receiving a PDU *)
NS  !NC_id
    ?ndt : NDATA_sort  [ (dest_ref (decode (NS_user_data(ndt))) eq local_ref)
                         and EoNSDU (ndt) ];
    ([checks (decode (NS_user_data(ndt)))] ->
        pr  !T_suf  !EP_id  !p_PDU (decode (NS_user_data(ndt)));
        PDU_handler [NS, pr, ps, t]
              (NC_id, T_suf, EP_id, local_ref,
               update_source_ref (remote_ref, decode (NS_user_data(ndt))),buffer)
     []
     [not (checks (decode (NS_user_data(ndt))))] ->
        (* ***** *)
        PDU_handler [NS, pr, ps, t]
              (NC_id, T_suf, EP_id, local_ref, remote_ref, buffer)
    )

[]  (* termination of Transport connection *)
t  !T_suf  !EP_id  !NC_id  !local_ref;
exit
where

type PDU_ordering is order_type, PDU_buffer_type
     opns
       order            : p_info          -> order_sort
       drop             : PDU_buffer_sort -> PDU_buffer_sort
       advance          : PDU_buffer_sort -> PDU_buffer_sort

     eqns forall p : p_info , s : PDU_buffer_sort

          ofsort order_sort
          IsCR_PDU (p) => order (p) = first;
          IsCC_PDU (p) => order (p) = first;
          IsDR_PDU (p) => order (p) = destructive;
          IsDC_PDU (p) => order (p) = destructive;
          IsDT_PDU (p) => order (p) = normal;
          IsAK_PDU (p) => order (p) = fast;
```

```
    ofsort PDU_buffer_sort
           drop (<>) = <>;
           order (first (s)) eq destructive => drop (p + s) = s;
           order (first (s)) ne destructive => drop (p + s) = p + drop (s);
           advance (<>) = <>;
           order (p) eq normal, order (first (s)) eq fast =>
                                advance (p + s) = first (s) + (p + tail (s));
           advance (p + s) = p + advance (s);  (* This rule should be applied
                                                  if the above rule is not
                                                  applicable *)
endtype

endtype
endproc (*PDU_handler *)
endproc (* Map *)


(*** Network Connection Management ***)
(************************************)

process NC_manager [a (* assign *),  t (* terminate *)]
             (NC_id : NCEP_id_sort,  remote_N_addr : N_address_sort )
             : noexit :=

(* assignment of new Transport connection *)
a  !NC_id  ?T_suf : T_suffix_sort  ?EP_id : TCEP_id_sort  ?PDU : p_info
   !remote_N_addr  ?local_ref : reference_sort  ?d : direction ?accepted : Bool;
NC_manager [a, t] (NC_id, remote_N_addr)

[]
(* termination of a Transport connection *)
t  ?T_suf : T_suffix_sort  ?EP_id : TCEP_id_sort
   !NC_id  ?ref : reference_sort;
NC_manager [a, t] (NC_id, remote_N_addr)

endproc (* NC_manager *)

(*** Unique Reference Numbers ***)
(*********************************)

process Unique_refs [a, t] (ref_set: ref_set_sort) : noexit :=

(* assignment of new Transport connection *)
a  ?NC_id : NCEP_id_sort  ?T_suf : T_suffix_sort  ?EP_id : TCEP_id_sort
   ?PDU : p_info  ?remote_N_addr : N_address_sort
   ?local_ref : reference_sort  ?d : direction ?accepted : Bool
   [local_ref NotIn ref_set];
Unique_refs [a, t] (Insert (local_ref, ref_set))

[]
(* termination of a Transport connection *)
t  ?T_suf : T_suffix_sort  ?EP_id : TCEP_id_sort
   ?NC_id : NCEP_id_sort  ?ref : reference_sort;
Unique_refs [a, t] (Remove (ref, ref_set))

endproc (* Unique_refs *)
endproc (* TP_entity   *)
endspec (* simple_TP   *)
```

Annex 3. Simplified Transport Protocol Specification in SDL

```
SYSTEM TP_protocol;

***** type definitions *****


SIGNAL   TCONreq (T_address, options_type, Pid),
         TCONind (T_address, type, options_type, Pid),
         TCONresp (options_type, Pid),
         TCONconf (options_type, Pid),
         TDISreq,
         TDISind (reason_type),
         TDATAreq (data_type, boolean),
         TDATAind (data_type, boolean),
         U_READY (credit_type),
         READY;

SIGNAL   NDATAreq (data_type, boolean),
         NDATAind (data_type, boolean);


CHANNEL  TS_interface

    FROM     ENV TO TP_entity WITH TCONreq, TCONreap, TDISreq, DTATAreq,
             U_READY;

    FROM     TP_entity TO ENV WITH TCONind, TCONconf, TDISind, TDATAind,
             READY;

    ENDCHANNEL  TS_interface;



CHANNEL  NS_interface

    FROM       ENV TO TP_entity WITH NDATAind;

    FROM       TP_entity TO ENV WITH NDATAind;

    ENDCHANNEL  TS_interface;
BLOCK TP_entity;

***** type definitions *****

CONNECT    TS_interface AND R1, R2;

CONNECT    NS_interface AND R4;

SIGNALROUTE  R1  FROM  ENV  TO  Map  WITH  TCONreq;
SIGNALROUTE  R2  FROM  ENV  TO  AP  WITH  TCONresp, TCONconf,
    TDISreq, TDISind, TDISconf, TDATAreq, TDATAind, U_READY, READY;
SIGNALROUTE  R3  FROM  MAP  to  AP  with  transfer, terminated_from_Map,
    terminated_from_AP, ready;
SIGNALROUTE  R4  FROM  ENV  TO  MAP  WITH  NDATAreq, NDATAind;

SIGNAL
    transfer (PDU_and_control_information, T_suffix-type, TCEP_id_type),
    terminated_from_Map,
```

```
    terminated_from_AP (T_suffix_type, TCEP_id_type),
    ready (T_suffix_type, TCEP_id_type);

PROCESS    Map (1,1)    REFERENCED;

PROCESS    AP (0,)    REFERENCED;

ENDBLOCK    TP_entity;

ENDSYSTEM    TP_protocol;
```

PROCESS Map (1,1);

***** similar to Map in Annex 1
In addition it performs the following actions:

(1) It receives TCONreq from user processes and creates a corresponding AP process instance, to which it then forwards the TCONreq interaction. The TCONreq includes as additional parameter the Pid of the user process to which the created AP process returns the TCONconf directly. The TCONconf also includes as additional parameter the Pid value of the AP process which the user process has to use to send the subsequent service interactions directly to the responsible AP process.

(2) A similar operation is performed for connection requests coming from the remote side.

ENDPROCESS Map;

PROCESS AP (0,);

```
DCL         ****** as in Annex 1, plus the following ******
            user_Pid  Pid,
            T_suf  T_suffix_type,
            EP_id  TCEP_id_type;
            /* the latter two additional variables are used for communication with the Map
            process to indicate the identity of the AP process instance */

***** transition T1, similar to Annex 1 *****

/* the following SDL transition corresponds to transitions T2a, T2b, and T2c of Figure 3 */

STATE   wait_for CC;
INPUT   transfer (PDU);
    DECISION PDU! kind
    (CC):  DECISION (PDU! option_ind <= options)

            (true):     options:= PDU! option_ind;
                        TRseq := 0;
                        TSseq := 0;
                        S_credit := PDU! credit_value
                        OUPUT   TCONconf (options, SELF)   TO   user-Pid;
                        NEXTSTATE    open;

            (false):    OUPUT   TDISind (procedure_error)   To   user-Pid;
                        OUTPUT   transfer (DR_PDU (procedure_error),
```

```
                    T_suf, EP_id)   TO   MAP;
        NEXTSTATE   wait_for_DC;

    ENDDECISION;

(DR):
        OUTPUT   TDISind (PDU! disconnect-reason)  TO  user_Pid;
        OUTPUT   terminated_from_AP (T_suf, EP_id)  TO  MAP;
        NEXTSTATE   closed;

ENDDECISION,

***** transitions similar to Annex 1 *****

ENDPROCESS AP;
```

References

[1] S. Aggarwal, D. Barbara and K.Z. Meth, SPANNER: A Tool for the Specification, Analysis, and Evaluation of Protocols, *IEEE Trans. Software Engrg.* **13** (1987) 1218–1237.

[2] G. v. Bochmann, Finite State Description of Communication Protocols, *Comput. Networks* **2** (1978) 361–372.

[3] G. v. Bochmann, Examples of Transport Service Specifications, Doc. de travail #145, Dept. d'IRO, Univ. de Montreal; also submitted to ISO and CCITT working groups on FDT, 1983.

[4] G. v. Bochmann, Usage of Protocol Development Tools: The Results of a Survey (invited paper), in: *Proc. 7th IFIP Symposium on Protocol Specification, Testing and Verification*, Zurich (1987).

[5] G. v. Bochmann, Comparison of SDL and Estelle in View of Finding a Usable Language Subset and Compatible Tools, prepared for Siemens, Munchen, 1987.

[6] G. v. Bochmann, Protocol Specification for OSI, *Comput. Networks ISDN Systems* **18** (3) (1990) 167–184.

[7] P. Zafiropulo, C.H. West, H. Rudin, D.D. Cowan and D. Brand, Towards Analyzing and Synthesizing Protocols, *IEEE Trans. Comm.* **28** (4) (1980) 651–660.

[8] G. v. Bochmann, E. Cerny, M. Maksud and B. Sarikaya, Testing of Transport Protocol Implementations, in: *Proc. CIPS Conference*, Ottawa (1983) 123–129.

[9] G. v. Bochmann, G. Gerber and J.M. Serre, Semiautomatic Implementation of Communication Protocols, *IEEE Trans. Software Engrg.* **13** (9) (1987) 989–1000; reprinted in: D.P. Sidhu, ed., *Automatic Implementation and Conformance Testing of OSI Protocols* (IEEE, New York, 1989).

[10] G. v. Bochmann and A. Finkel, Impact of Queued Interaction on Protocol Specification and Verification, in: *Proc. International Symposium on Interoperable Informatics Systems (ISIIS)*, Tokyo (1988) 371–382.

[11] T. Bolognesi and E. Brinksma, Introduction to the ISO Specification Language Lotos, *Comput. Networks ISDN Systems* **14** (1) (1987) 25–59.

[12] E. Brinksma, G. Scollo and C. Steenbergen, Lotos Specifications, Their Implementations and Their Tests, in: *Proc. IFIP Workshop on Protocol Specification, Testing and Verification* VI, Montreal (North-Holland, Amsterdam, 1986) 349–360.

[13] S. Budkowski and P. Dembinski, An Introduction to Estelle: A Specification Language for Distributed Systems, *Comput. Networks ISDN Systems* **14** (1) (1987) 3–23.

[14] R.H. Campbell and A.N. Habermann, The Specification of Process Synchronization by Path Expressions, in: Lecture Notes in Computer Science **16** (Springer, Berlin, 1974).

[15] CCITT SG XI, Recommendation Z.100, 1987.

[16] J.P. Courtiat, Estelle *: A Powerful Dialect of Estelle for OSI Protocol Description, in: *Proc. IFIP Symposium on Protocol Specification, Testing and Verification*, Atlantic City (1988).

[17] C. Vissers, G. Scollo and M. v. Sinderen, Architecture and Specification Style in Formal Descriptions of Distributed Systems, in: *Proc. IFIP Symposium on Protocol Specification, Testing and Verification*, Atlantic City (1988).

[18] E. Dubois et al., A Framework for the Taxonomy of Synthesis and Analysis Activities in Distributed System Design, in: R. Speth, ed., *Research into Networks and Distributed Applications* (North-Holland, Amsterdam, 1988) 859–872.

[19] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specifications 1* (Springer, Berlin, 1985).

[20] M.G. Gouda and Y.-T. Yu, Synthesis of Communicating Finite State Machines with Guaranteed Progress, *IEEE Trans. Comm.* **32** (7) (1984) 779–788.

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

[22] ISO TC97/SC16 N1347, A FDT Based on an Extended State Transition Model, revised July 1983.

[23] ISO 97/21 N1540, Potential Enhancement to Lotos, 1986.

[24] ISO TC97/SC6 N 3576, Formal specification of Transport protocol in Estelle, 1986.

[25] ISO TC97/SC6 N..., Formal specification of Transport protocol in Lotos, 1986.

[26] ISO DIS9074, Estelle: A Formal Description Technique Based on an Extended State Transition Model, 1987.

[27] ISO DIS8807, LOTOS; A Formal Description Technique, 1987.

[28] ISO TC97/SC6/WG4N ad hoc group on Formal Description of Transport in Lotos, Formal Description of ISO 8073 in Lotos, 1987.

[29] ISO TC97/SC6, IS 8073, OSI – connection Oriented Transport Protocol Specification.

[30] C. Jard, J.F. Monin and R. Groz, Experience in Implementing Estelle-X.250 in VEDA, in: M. Diaz, ed., *Protocol Specification, Testing and Verification V* (North-Holland, Amsterdam, 1985).

[31] G. Karjoth, Implementing Process Algebra Specifications by State Machines, in: *Proc. IFIP Symposium on Protocol Specification, Testing and Verification*, Atlantic City (1988).

[32] R. Milner, A Calculus of Communicating Systems, Lecture Notes in Computer Science **92** (Springer, Berlin, 1980).

[33] R. De Nicola, Extensional Equivalences for Transition Systems, *Acta Inform.* **24** (1987) 211–237.

[34] D. Rayner, OSI Conformance Testing, *Comput. Networks ISDN Systems* **14** (1987) 79–98.

[35] F. Belina and D. Hogrefe, The CCITT-Specification and Description Language SDL, *Comput. Networks ISDN Systems* **16** (1989) 311–341.

[36] B. Sarikaya, G. v. Bochmann and E. Cerny, A Test Design Methodology for Protocol Testing, *IEEE Trans. Software Engrg.* (April 1987) 518–531.

[37] G. Scollo and M. Sinderen, On the Architectural Design of the Formal Specification of the Session Standards in LOTOS, in: *Proc. IFIP Symposium on Protocol Specification, Testing and Verification*, Grag Rocks (North-Holland, Amsterdam, 1986).

[38] D.P. Sidhu and T.P. Blumer, Semi-automatic Implementation of OSI Protocols, *Comput. Networks ISDN Systems*, **18** (3) (1990) 221–238.

[39] Special Issue on Open Systems Interworking, *Proc. IEEE* (December 1983).

[40] M. Steinacker, Comparison of Two Specification Languages, in: L. Csaba et al., eds., *Computer Network Usage* (North-Holland, Amsterdam, 1985).

[41] The SPECS Consortium and J. Bruyning, Evaluation and Integration of Specification Languages, *Comput. Networks ISDN Systems* **13** (1987) 75–89.