JTC 1

TECHNICAL REPORT

ISO/IEC TR 9572

First edition
1989-09-15

Information technology — Open Systems Interconnection — LOTOS description of the session protocol

Traitement de l'information — Interconnexion de systèmes ouverts — Description en LOTOS du protocole de session

Reference number
ISO/IEC/TR 9572 : 1989 (E)

Contents

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) together form a system for worldwide standardization as a whole. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The main task of a technical committee is to prepare International Standards but in exceptional circumstances, the publication of a technical report of one of the following types may be proposed:

- type 1, when the necessary support within the technical committee cannot be obtained for the publication of an International Standard, despite repeated efforts;

- type 2, when the subject is still under technical development requiring wider exposure;

- type 3, when a technical committee has collected data of a different kind from that which is normally published as an International Standard ("state of the art", for example).

Technical reports of types 1 and 2 are subject to review within three years of publication, to decide whether they can be transformed into International Standards. Technical reports of type 3 do not necessarily have to be reviewed until the data they provide are considered to be no longer valid or useful.

ISO/IEC/TR 9572, which is a technical report of type 2, was prepared by ISO/IEC JTC 1, *Information technology*.

Introduction

In view of the complexity and widespread use of Open Systems Interconnection standards it is imperative to have precise and unambiguous definitions of these standards. Formal Description Techniques form an important approach for providing such definitions. The use of Formal Description Techniques in this area is however relatively new and their application on a wide scale cannot be expected overnight. Formal descriptions should be introduced gradually in standards if initially the number of Member Bodies that are able to contribute to their development is too small, thus allowing time to gain experience and to develop educational material.

An ad-hoc group for the formal description of the Session Layer, i.e. of the Session Service ISO 8326 and the Session Protocol ISO 8327, was established in November 1985. This group applied the Formal Description Technique LOTOS, defined in ISO 8807, which at that time was still under development. In September 1986 two Working Documents were produced which contained the LOTOS draft specifications of ISO 8326 and ISO 8327 respectively. As a byproduct, the group also produced a number of Defect Reports on the standards, most of which have been accepted and incorporated in the standards.

A Ballot was then issued requesting Member Bodies to state their position concerning the progression of the formal descriptions. Based on this Ballot, SC21 decided in June 1987 to progress both formal descriptions as Type 2 Technical Reports. The main reason for not incorporating them into the standards was that Member Bodies expressed their current lack of expertise on the subject. It seemed therefore appropriate that a period of time passed, during which the formal decriptions can be read and compared with the standards, and after which the status and progression of the formal descriptions can be re-evaluated.

The purpose of this Technical Report is to provide a complete, consistent and unambiguous description of ISO 8327. It forms therefore a companion document to ISO 8327. It takes account of the Defect Reports incorporated in the standard (annex D of ISO 8327), however, it does not necessarily take account of subsequent amendments or addenda to the standard.

Information technology - Open Systems Interconnection - LOTOS description of the session protocol

1 Scope

This Technical Report contains a formal description of the OSI Basic Connection Oriented Session Protocol defined in ISO 8327. The formal definitions presented in this Technical Report are expressed in the formal description technique LOTOS, which is defined in ISO 8807.

These formal definitions are related to the formal descriptions in LOTOS of the OSI Connection Oriented Session Service, ISO 8326, and of the OSI Transport Service, ISO 8072. Moreover, formal definitions of these services, contained in ISO/IEC/TR 9571 and ISO/IEC/TR 10023, respectively, are used and referenced in this Technical Report.

The formal description is not limited to a single session protocol machine, but also describes multiple session protocol machines that result from support of multiple session connections either in parallel or in sequence. Therefore, it also formalizes aspects of multiplicity which are not presented in ISO 8327 directly, but by way of reference to the OSI Basic Reference Model, ISO 7498.

2 Normative references

The following standards contain provisions which, through reference in this text, constitute provisions of this Technical Report. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreement based on this Technical Report are encouraged to investigate the possibility of applying the most recent editions of the standards listed below. Members of ISO and IEC maintain registers of currently valid International Standards.

ISO 7498: 1984, *Information processing systems - Open Systems Interconnection - Basic Reference Model.*

ISO 8072: 1986, *Information processing systems - Open Systems Interconnection - Transport service definition.*

ISO 8326: 1987, *Information processing systems - Open Systems Interconnection - Basic connection oriented session service definition.*

ISO 8327: 1987, *Information processing systems - Open Systems Interconnection - Basic connection oriented session protocol specification.*

ISO 8807: 1988, *Information processing systems - Open Systems Interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behaviour.*

ISO/IEC/TR 9571: 1989, *Information processing systems - Open Systems Interconnection - LOTOS description of the session service.*

ISO/IEC/TR 10023: - 1), *Information processing systems - Open Systems Interconnection - LOTOS description of the transport service.*

3 Definitions

For the purpose of this Technical Report the definitions given in ISO 8327 apply.

4 Symbols and abbreviations

This Technical Report uses the symbols defined in clause 6 (formal syntax) and annex A (data type library) of ISO 8807, and uses the abbreviations contained in clause 4 of ISO 8327.

The following additional abbreviations are employed in this Techical Report:

SC	session connection
SCEP	session connection endpoint
SCEI	session connection endpoint identifier
SSP	session service primitive
TC	transport connection
TS	Transport Service
TSDU	transport service data unit

5 Conventions

Clauses 6 through 10 of this Technical Report constitute LOTOS text. All informal explanations in these clauses form LOTOS comments. They are thus separated from the LOTOS specifications (of data types and dynamic behaviour) according to the rules for comment delimitation. Moreover, informal explanations precede the formal definitions to which they refer and contain a final line of only "-" characters. Informal explanations following formal definitions contain a first line of only "-" characters.

Formal definitions, as well as formal symbols and identifiers referenced in informal explanations, are printed in italics.

1) To be published.

(* start of LOTOS text --

6 Introduction to the formal description

The formal description relates to the dynamic behaviour of an unbounded number of SPMs, each of which supports the provision of a single SC or, in case of re-use of the underlying TC, a sequence of SCs. The SS boundary is formally represented by a single gate *s* and the TS boundary by a single gate *t*. Events at these gates are structured as defined in the service formal descriptions, ISO/IEC/TR 9571 and ISO/IEC/TR 10023. Constraints on connection identification, acceptance of connections and backpressure flowcontrol (from the TS-provider) apply to the Session Protocol, as well as to the related services, and the processes representing these constraints are in fact imported from the service formal descriptions. Figure 1 shows the conjunction of processes resulting from this top level decomposition.

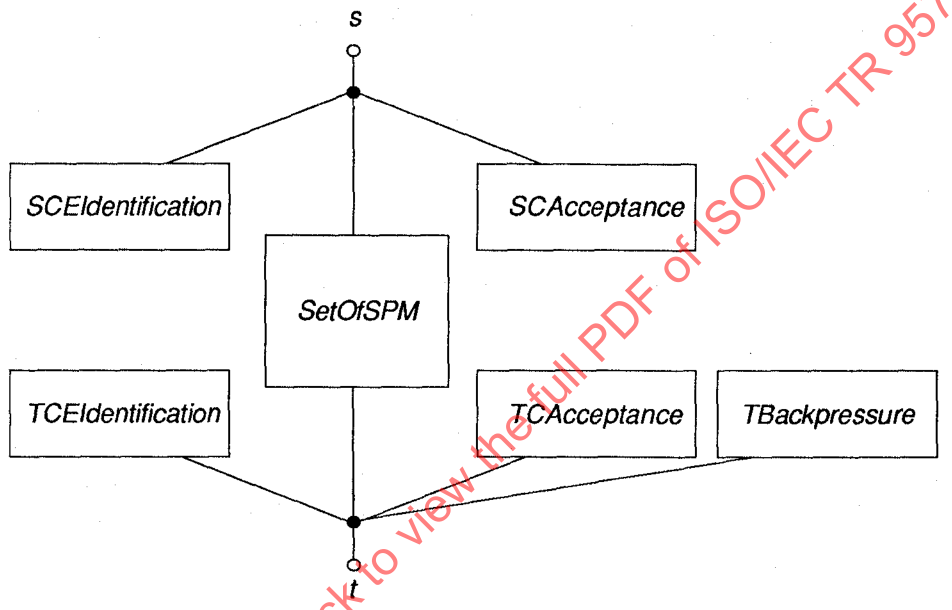

Figure 1 - (Top level) processes representing separate constraints for a Session Protocol entity

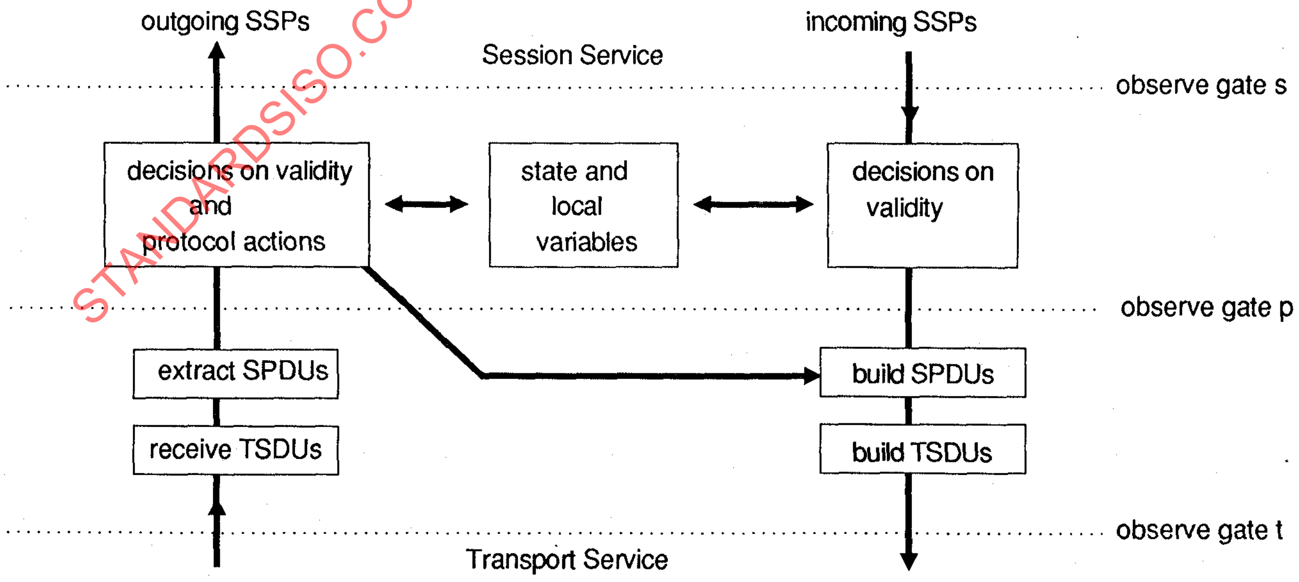

Figure 2 - Actions within a SPM

An internal gate *p* is introduced to favour separation of concerns based on the concept of SPDU. SPDUs are transferred via the TS by encoding, and possibly concatenating, them into TS data units. The interactions at gate *p* are independent of encoding and concatenation, however, thus only abstract SPDUs are to be considered at *p*. The actions of a single SPM are summarized in figure 2.

The behaviour of the SPM is described by specifying the constraints on the behaviour observed at the three gates *s*, *p* and *t*. This is obtained by a parallel composition of a process, termed *STPM*, that roughly corresponds to the state tables protocol machine, together with processes that enforce the local constraints on service primitives at connection endpoints. Again, the latter constraints can be specified with instances of processes imported from the SS and TS formal descriptions (namely *SCEP* and *TCEP*).

The next level of substructuring is found in the decomposition of *STPM* into processes representing the following separate constraints:

a) The relationship between SSPs and TSPs that does not involve abstract SPDUs (described by process *SSPTSP*). This constraint is concerned with "direct mappings" between service primitives, where no SPM generated information is needed to coordinate the interworking with the remote SPM. Also the establishment of a TC is attributed to this constraint.
b) The relationship between SSPs and abstract SPDUs, including segmentation of SSDUs (described by process *SSPSPDU*).
c) The transformation of abstract SPDUs to TSPs, and vice versa, including encoding and concatenation of SPDUs (described by process *SPDUTSP*).
d) The constraints on the ordering and contents of abstract SPDUs, corresponding to the state tables description in annex A of ISO 8327 (described by process *SPDUConstraints*).

SSPTSP and *SSPSPDU* synchronize at gate *s*, *SSPTSP* and *SPDUTSP* synchronize at gate *t*, and, finally, *SSPSPDU*, *SPDUTSP* and *SPDUConstraints* synchronize at gate *p*. This is shown in figure 3. The style of the formal description, which now also shows a limited internal structure, can be characterized as a mixture of constraint- and resource-oriented styles.

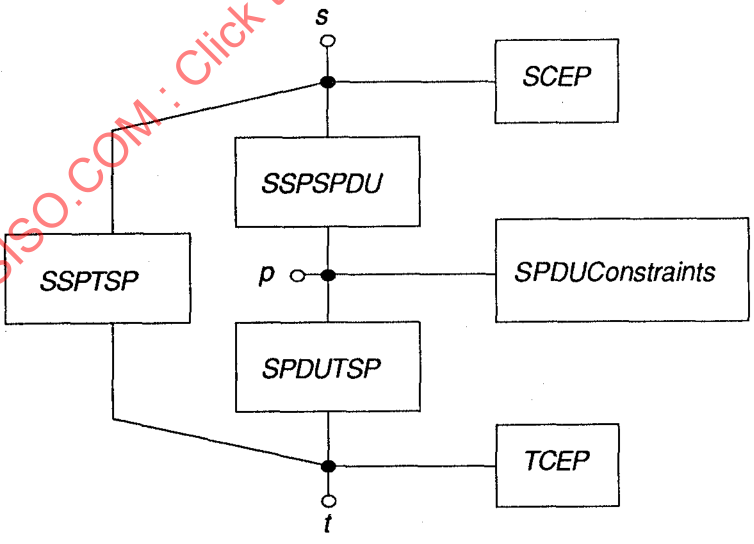

Figure 3 - Conjunction of processes representing the behaviour of a single SPM

Each of the processes that must synchronize at *p* imposes its own constraints on the exchange of abstract SPDUs. Because of this, an abstract SPDU exchange only occurs if all processes agree on that particular exchange. The event structure of the abstract SPDU exchange is as follows:

p ?pd:ASPDU ?fl:TFlow ?d:Dir ?v:Validity ?st:TrState

with components

- *pd* of sort *ASPDU*: the abstract SPDU being exchanged;
- *fl* of sort *TFlow*: the transport flow, which is either normal or expedited;
- *d* of sort *Dir*: the direction of transfer, which is either sending or receiving;
- *v* of sort *Validity*: the "status" of the abstract SPDU (only applicable for received SPDUs). According to the state tables of ISO 8327, processing of a received SPDU, and in particular determining its validity, depends upon the result of several actions (see figure 2) and/or the values of certain session variables. Relevant states that involve more than one process can be established through this component when synchronizing at *p*; and
- *st* of sort *TrState*: the "coarse-grained" state of the SPM (only applicable for received SPDUs and for sending the DN SPDU). The SPM state, known to process *SPDUConstraints*, is often also required by other processes upon exchange of a SPDU. However, rather than the fine-grained division of the data transfer phase into SPM states, certain collections of SPM states are relevant for the latter processes. Values of this component are used to represent such collections and can be communicated when synchronizing at *p*.

NOTES

1 - In order to represent flow control and segmenting, events at gates *s* and *t* may represent the exchange of single data octets instead of just executions of "complete" service primitives. Octet-wise exchange of data is only modelled for those service primitives whose user data parameter has no length restrictions (i.e., S-DATA, S-TYPED-DATA and T-DATA). Except for the need of "data octet" events, the characteristics of octet-wise exchange of data are

- two events, a start and an end event, are significant to delimitate an exchange of data;
- the state transition and predicates that are associated with a SSP request are now associated with the corresponding start event. If a colliding or overtaking, that is, "intervenient", event occurs before the end event, then either
  - a) the (unfinished) request is not disrupted (provided the intervenient event is not destructive), but will be resumed after the intervenient event, or
  - b) the (unfinished) request is disrupted, and the corresponding TSP request is cancelled or terminated;
- the state transition and predicates that are associated with a TSP indication are now associated with the corresponding end event. If a destructive or overtaking event occurs before the end event, then the (unfinished) indication is disrupted, and all previous events of the indication are discarded.

2 - The formal description extends beyond the state tables description of ISO 8327 by way of explicitly describing

- the constraints related to the multiplicity of SPMs;
- concatenation, extended concatenation and segmentation;
- the transfer of expedited data;
- the handling of invalid and unrecognizable events.

Figure 4 gives an overview of the processes used for the formal definition of the Session Protocol and their relations (a number of the lowest level processes are omitted). It also indicates the clauses where the definition of these processes can be found ("SS" means: defined in ISO/IEC/TR 9571; "TS" means: defined in ISO/IEC/TR 10023).

The definition of data types precedes the definition of the dynamic behaviour in which these types are used. A number of standard data types are imported from the LOTOS library of data types. Apart from these, and some "ad-hoc" data types, the types can roughly be divided in types for the representation of abstract SPDUs, of encoded SPDUs, and of session variables. Additionally, a number of types are related to the components of an internal event at gate *p* (besides the abstract SPDU).

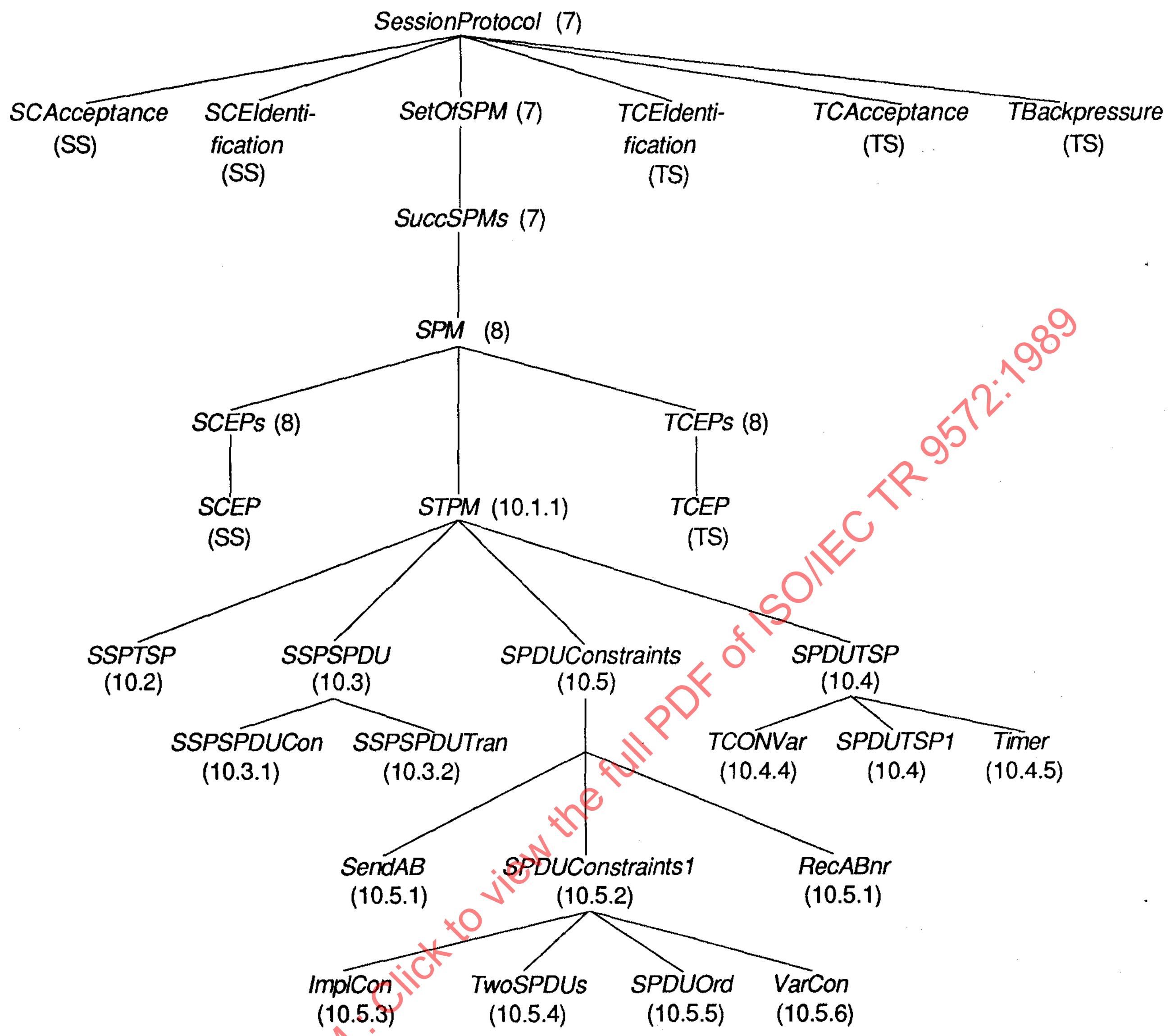

Figure 4 - Processes related by a tree structure: each "father" process contains (or is constructed from) one or more instances of its immediate "descendant" process(es)

7 Overall constraints of the session protocol

A number of standard types are imported from the LOTOS library of data types by means of the *library* construct. The specification is parameterized with a protocol implementation parameter, defined in type *SProtocolImplementationPar*. Its value indicates: 1) whether or not use of the transport expedited service is implemented, 2) what is the maximum TSDU size for both directions of data flow, 3) which functional units have been implemented and 4) whether or not extended concatenation is implemented.

The top level structure shows a full synchronization of process *SetOfSPM* with two behaviour expressions which mutually interleave, viz. *SCEIdentification* synchronized with *SCAcceptance* on the one hand, and *TCEIdentification* synchronized with *TCAcceptance* and *TBackpressure* on the other. *SetOfSPM* defines the dynamic behaviour of a potentially infinite number of independent SPMs, however disregarding any concerns related to the limited capacity of the local system and the TS-provider and the need for unique connection identification. The latter concerns are addressed by the two behaviour expressions mentioned. They formulate the overall constraints that relate to a single service boundary; the processes representing these constraints are defined in the Session Service formal description, IS/IEC/TR 9571, and the Transport Service formal description, ISO/IEC/TR 10023, respectively.

SetOfSPM represents concurrency by multiple instances of *SuccSPMs*, where each instance describes a succession of SPM behaviours. A SPM is represented by a distinct instance of *SPM*. The behaviour described by *SPM* is that of using at most one TC for support of a single SC or multiple successive SCs (based on the same TC). Termination of a TC enforces successful termination of the associated instance of *SPM*.

```
-------------------------------------------------------------------------------------------------------*)
specification SessionProtocol [s,t] (spei:SPEImplementation): noexit

library Boolean,Element, Set, String, NatRepresentations, OctetString, NaturalNumber, Bit, FBoolean,
        Octet, DecNatRepr, BitString, BitNatRepr, DecString , DecDigit
endlib

type SProtocolImplementationPar is SRequirements, TSDUSize
sorts SPEImplementation
opns
SPEImpl: Bool, SRqms, TSDUSize, Bool -> SPEImplementation
MaxTSize: SPEImplementation -> TSDUSize          ExtConc: SPEImplementation -> Bool
eqns forall tex,ExtConc:Bool, sfus:SRqms, tsduSize:TSDUSize
ofsort TSDUSize          MaxTSize(SPEImpl(tex,sfus,tsduSize,ExtConc)) = tsduSize;
ofsort Bool              ExtConc(SPEImpl(tex,sfus,tsduSize,ExtConc)) = ExtConc;
endtype

behaviour
(       ( SCEIdentification[s] || SCAcceptance[s] ) |||
        ( TCEIdentification[t] || TCAcceptance[t] || TBackpressure[t] )
)       || SetOfSPM[s,t](spei)
where

process SetOfSPM[s,t](spei:SPEImplementation): noexit := SuccSPMs[s,t](spei) |||  SetOfSPM[s,t](spei)
where
process SuccSPMs[s,t](spei:SPEImplemtation): noexit := SPM[s,t](spei) >> SuccSPMs[s,t](spei) endproc
endproc (* SetOfSPM *)
(*-----------------------------------------------------------------------------------------------------
```

8 Constraints for a single session protocol machine

Three separate constraints are distinguished with respect to the behaviour of a single SPM, namely constraints that are local to the SS boundary, constraints that are local to the TS boundary, and constraints that relate the behaviour at one boundary to that at the other. These constraints are represented by *SCEPs*, *TCEPs* and *STPM*, respectively.

SCEPs is described as the choice of two instances of *SCEP* that apply to the interaction with the Calling SS-user and with the Called SS-user, respectively, followed by either successful termination or another instance of *SCEPs*. The recursion applies only when the SPM re-uses the TC. Similarly, *TCEPs* is described as the choice of two instances of *TCEP* that apply to the interaction with the Calling TS-user and with the Called TS-user, respectively. The definitions of *SCEP* and *TCEP* are imported from ISO/IEC/TR 9571 and ISO/IEC/TR 10023 respectively.

STPM defines the mapping of SSPs to TSPs during the provision of one SC, or multiple successive SCs, supported by a single TC.

```
-----------------------------------------------------------------------------------------------*)
process SPM[s,t](spei:SPEImplementation): noexit :=
SCEPs[s] |[s]| STPM[s,t](spei) |[t]| TCEPs[t]
where
process SCEPs[s]: exit := ( SCEP[s](calling) [] SCEP[s](called) ) >> ( exit [] SCEPs[s] ) endproc
process TCEPs[t]: exit := TCEP[t](CallingRole) [] TCEP[t](CalledRole) endproc
endproc (* SPM *)
(*-----------------------------------------------------------------------------------------------
```

9 Session protocol data types

The Session Protocol data types consist of definitions for the construction of a SPDU (see 9.1), some session variables (see 9.2) for the introduction of some additional functions related to SSPs and TSPs (see 9.3), and for the construction of components of the internal event structure (see 9.5). A number of auxiliary definitions of general use are presented in 9.4.

9.1 Session protocol data unit

Two sets of definitions are distinguished for the construction of a SPDU: one for an "abstract" SPDU and the other for an encoded SPDU. This division is consistent with the observation that the elements of procedure related to SPDUs can be described independently of the encoding of SPDUs (i.e., the mapping into TS data units).

9.1.1 Abstract SPDU

The specification of the abstract SPDU type is accomplished by way of a number hierarchical type definitions. First the basic construction of SPDUs is presented (see 9.1.1.1), then a classification of SPDUs (see 9.1.1.2), and subsequently a number of additional functions on SPDUs (see 9.1.1.3 through 9.1.1.5) Concatenation of SPDUs is defined in 9.1.2; (groups of) SPDU parameters are defined in 9.1.3 (in as far as not defined in ISO/IEC/TR 9571).

9.1.1.1 SPDU basic construction

Type *BasicASPDU* defines functions, referred to as "constructor" functions, that yield (abstract) SPDU values. For each class, or "type", of SPDU a corresponding constructor function is defined with values of the parameters of that SPDU as arguments. Additionally, a function *DUM* is introduced for conveniently specifying error reporting. Definitions that relate to SPDU parameters are imported by *BasicASPDU* (most of them indirectly, by importing type *SessionServicePrimitive*).

```
-----------------------------------------------------------------------------------------------*)
type BasicASPDU is SessionServicePrimitive, SPReference, CAItem, TDISPar, Prepare,EncItem
sorts ASPDU
opns
CN: SPReference,CAItem,SFUs,SAddress,SAddress,SData -> ASPDU
AC: SPReference,CAItem,STokens,SFUs,SAddress,SAddress,SData -> ASPDU
RF: SPReference,TDISPar,SFUs,Nat (*Version*) ,DatOctStr (*Reason Code*) -> ASPDU
FN: TDISPar,SData -> ASPDU                    DN,NF: SData -> ASPDU
AB: TDISPar,Nat (*Reason Code*),DatOctStr (*Reflect Parameter Values*), SData -> ASPDU
GTC,GTA,AA,AIA,ADA: -> ASPDU                  DT,TD: EncIt,SData -> ASPDU
CD,EX,CDA: SData -> ASPDU                     GT: STokens (*Token Item parameter*) ->  ASPDU
PT: STokens,SData -> ASPDU                    MIP: SSyncType,SSPSN,SData -> ASPDU
MIA: SSPSN,SData -> ASPDU                     MAP,AE,MAA,AEA: SSPSN,SData -> ASPDU
RA: STsAss,SSPSN,SData -> ASPDU               RS : STsAss,SResynType,SSPSN,SData -> ASPDU
PR: PrepType -> ASPDU                         AS: SActId,SData -> ASPDU
```

```
AR: SCRef,SActId,SSPSN,SActId,SData -> ASPDU
AI,AD: Nat (*Reason Code*) -> ASPDU            DUM: DatOctStr -> ASPDU
ER : DatOctStr -> ASPDU                        ED : Nat (* Reason Code *), SData -> ASPDU
endtype
(*------------------------------------------------------------------------------------------------
```

9.1.1.2 SPDU classification

Type *ASPDUConstant* defines a classification of SPDUs. Each class of SPDU is represented by a constant with a name similar to the abbreviated name to denote the SPDU in table 39 of ISO 8327. The auxiliary function *h* that maps these constants to natural numbers is defined in order to simplify the definition of equality on SPDU classes.

Type *ASPDUClassifiers* is a functional enrichment of *BasicASPDU*. It defines functions, referred to a "recognizer" functions, that determine whether a given SPDU is of a certain class.

```
------------------------------------------------------------------------------------------------*)
type ASPDUConstant is NaturalNumber
sorts ASPDUConstant
opns
cn,ac,rf,fn,dn,nf,ab,aa,dt,ex,td,cd,gt,pt,gtc,gta,mip,mia,map,maa,rs,pr,ra,er,ed,as,ar,ad,ai,ada,aia,ae,aea,
      dum: -> ASPDUConstant
h: ASPDUConstant -> Nat
_eq_ , _ne_: ASPDUConstant,ASPDUConstant -> Bool
eqns forall x,y:ASPDUConstant
ofsort Nat
h(cn) = 0;                    h(ac) = Succ(0);              h(rf) = Succ(h(ac));
h(fn) = Succ(h(rf));          h(dn) = Succ(h(fn));          h(nf) = Succ(h(dn));
h(ab) = Succ(h(nf));          h(aa) = Succ(h(ab));          h(dt) = Succ(h(aa));
h(ex) = Succ(h(dt));          h(td) = Succ(h(ex));          h(cd) = Succ(h(td));
h(cda) = Succ(h(cd));         h(gt) = Succ(h(cda));         h(pt) = Succ(h(gt));
h(gtc) = Succ(h(pt));         h(gta) = Succ(h(gtc));        h(mip) = Succ(h(gta));
h(mia) = Succ(h(mip));        h(map) = Succ(h(mia));        h(maa) = Succ(h(map));
h(rs) = Succ(h(maa));         h(pr) = Succ(h(rs));          h(ra) = Succ(h(pr));
h(er) = Succ(h(ra));          h(ed) = Succ(h(er));          h(as) = Succ(h(ed));
h(ar) = Succ(h(as));          h(ad) = Succ(h(ar));          h(ai) = Succ(h(ad));
h(ada) = Succ(h(ai));         h(aia) = Succ(h(ada));        h(ae) = Succ(h(aia));
h(aea) = Succ(h(ae));         h(dum) = Succ(h(aea));
ofsort Bool
x eq y = h(x) eq h(y);        x ne y = h(x) ne h(y);
endtype

type ASPDUClassifiers is BasicASPDU, ASPDUConstant
opns
IsCN,IsAC,IsRF,IsFN,IsDN,IsNF,IsAB,IsGTC,IsGTA,IsAA,IsDT,IsTD,IsCD,IsEX,IsCDA,IsGT,IsPT,IsMIP,
IsMIA,IsMAP,IsMAA,IsAEA,IsRS,IsRA,IsPR,IsAS,IsAR,IsAI,IsAD,IsAIA,IsADA,IsAE,IsCAT21,IsACK,IsPAB,
IsER,IsED,IsDUM,IsPR_RS,IsPR_RA,IsPR_MAA,IsACT: ASPDU -> Bool
k: ASPDU -> ASPDUConstant
eqns forall spr:SPReference, sreq:SFUs, cait:CAItem, cg,cd:SAddress, d:SData, tk:STokens,
      tdpar:TDISPar, vs,reas:Nat, rcode,rpv,err:DatOctStr, sn:SSPSN, a:STsAss, et:EncIt, sntp:SSyncType,
      rt:SResynType, pt:PrepType, aid1,aid2:SActId, scr:SCRef, pd:ASPDU
ofsort ASPDUConstant
k(CN(spr,cait,sreq,cg,cd,d)) = cn;                 k(AC(spr,cait,tk,sreq,cg,cd,d)) = ac;
k(RF(spr,tdpar,sreq,vs,rcode)) = rf;
k(FN(tdpar,d)) = fn;          k(DN(d)) = dn;                k(NF(d)) = nf;
k(AB(tdpar,reas,rpv,d)) = ab; k(GTC) = gtc;                 k(GTA) = gta;
k(AA) = aa;                   k(AIA) = aia;                 k(ADA) = ada;
k(DT(et,d)) = dt;             k(TD(et,d)) = td;             k(CD(d)) = cd;
k(EX(d)) = ex;                k(CDA(d)) = cda;              k(GT(tk)) = gt;
```

```
k(PT(tk,d)) = pt;              k(MIP(sntp,sn,d)) = mip;        k(MIA(sn,d)) = mia;
k(MAP(sn,d)) = map;            k(AE(sn,d)) = ae;               k(MAA(sn,d)) = maa;
k(AEA(sn,d)) = aea;            k(RA(a,sn,d)) = ra;             k(RS(a,rt,sn,d)) = rs;
k(PR(pt)) = pr;                k(AS(aid1,d)) = as;             k(AR(scr,aid1,sn,aid2,d)) = ar;
k(AI(reas)) = ai;              k(AD(reas)) = ad;               k(ER(err)) = er;
k(ED(reas,d)) = ed;            k(DUM(err)) = dum
ofsort Bool
IsCN(pd) = k(pd) eq cn;        IsAC(pd) = k(pd) eq ac;         IsRF(pd) = k(pd) eq rf;
IsFN(pd) = k(pd) eq fn;        IsNF(pd) = k(pd) eq nf;         IsDN(pd) = k(pd) eq dn;
IsAB(pd) = k(pd) eq ab;        IsGTC(pd) = k(pd) eq gtc;       IsGTA(pd) = k(pd) eq gta;
IsAA(pd) = k(pd) eq aa;        IsDT(pd) = k(pd) eq dt;         IsCDA(pd) = k(pd) eq cda;
IsTD(pd) = k(pd) eq td;        IsCD(pd) = k(pd) eq cd;         IsEX(pd) = k(pd) eq ex;
IsMIP(pd) = k(pd) eq mip;      IsGT(pd) = k(pd) eq gt;         IsPT(pd) = k(pd) eq pt;
IsMIA(pd) = k(pd) eq mia;      IsMAP(pd) = k(pd) eq map;       IsMAA(pd) = k(pd) eq maa;
IsAEA(pd) = k(pd) eq aea;      IsRS(pd) = k(pd) eq rs;         IsRA(pd) = k(pd) eq ra;
IsPR(pd) = k(pd) eq pr;        IsER(pd) = k(pd) eq er;         IsED(pd) = k(pd) eq ed;
IsAS(pd) = k(pd) eq as;        IsAR(pd) = k(pd) eq ar;         IsAI(pd) = k(pd) eq ai;
IsAD(pd) = k(pd) eq ad;        IsAIA(pd) = k(pd) eq aia;       IsADA(pd) = k(pd) eq ada;
IsAE(pd) = k(pd) eq ae;        IsDUM(pd) = k(pd) eq dum;
IsACK(pd) = (IsMIA(pd) or IsMAA(pd) or IsAEA(pd) or IsRA(pd) or IsAIA(pd) or IsADA(pd) or IsCDA(pd));
IsCAT21(pd) = (IsRS(pd) or IsRA(pd) or IsAD(pd) or  IsADA(pd) or IsAI(pd) or IsAIA(pd) or IsER(pd) or
    IsED(pd) or IsCD(pd) or IsCDA(pd));
not(IsAB(pd)) => IsPAB(pd) = false;
IsPAB(AB(tdpar,reas,rpv,d)) =
    ((reas eq Succ(Succ(0))) and (Length(rpv) le NatNum(Dec(9)))) or
    ((reas eq NatNum(Dec(3))) and Empty(rpv)) and Empty(d);
IsPR_RS(PR(pt)) = pt eq res;
IsPR_RA(PR(pt)) = pt eq rack;
IsPR_MAA(PR(pt)) = pt eq mack;
IsACT(pd) = IsAR(pd) or IsAS(pd) or IsAI(pd) or IsAD(pd) or IsAE(pd) or IsAIA(pd) or IsADA(pd) or
    sAEA(pd);
endtype
(*-----------------------------------------------------------------------------------------------------------
```

9.1.1.3 SPDU parameter selectors

Type *ASPDUParameterSelectors* defines functions, referred to "extractor" functions, that allow to determine the value of individual SPDU parameters. It imports *SLokalTokensState* from ISO/IEC/TR 9571 and *Compare*, an auxiliary type (see 9.4.2).

```
-----------------------------------------------------------------------------------------------------------*)
type ASPDUParameterSelectors is ASPDUClassifiers,SLocalTokensState,Compare
opns
ErrStr,RPV: ASPDU-> DatOctStr                  SPSN: ASPDU -> SSPSN
SPRef: ASPDU -> SPReference                    CgAddr,CdAddr: ASPDU -> SAddress
Version: ASPDU -> Nat                          ResynType: ASPDU -> SResynType
TDISPar: ASPDU -> TDISPar                      Reason: ASPDU -> Nat
Preptype: ASPDU -> PrepType                    MtsRecFlow,MtsSendFlow: ASPDU -> Nat
MTSDUPar: ASPDU -> TSDUSize                    UserInf: ASPDU -> SData
FUs: ASPDU -> SFUs                             EncItem: ASPDU -> EncIt
TokenSet,InvTokenSet: ASPDU -> STsAss          InitialAss: ASPDU -> SLTsState
Tokens: ASPDU -> STokens                       ProtOptions: ASPDU -> Bool
```

```
eqns forall sn:SSPSN, cait:CAItem, spr:SPReference, vs:Nat, sdat:SData,prt:PrepType, rpv:DatOctStr,
        srqms:SFUs, sa1,sa2:SAddress, srf:SCRef, mt,mtr,mts:Nat, tkass:STsAss, rtyp:SResynType,
        tsize:TSDUSize, actid1,actid2:SActId, reas:Nat,str:DatOctStr, tdpar:TDISPar, encit:EncIt,
        stok,CgTokens,CdTokens:STokens, tk:SToken, styp:SSyncType
ofsort DatOctStr
ErrStr(DUM(str)) = str;
ofsort SSPSN
SPSN(CN(spr,cait,srqms,sa1,sa2,sdat)) = SSPSN(cait);
SPSN(AC(spr,cait,stok,fus,sa1,sa2,sdat)) = SSPSN(cait);
SPSN(MIP(styp,sn,sdat)) = sn;          SPSN(MIA(sn,sdat)) =sn;
SPSN(MAP(sn,sdat)) =sn;   SPSN(MAA(sn,sdat)) =sn;
SPSN(AEA(sn,sdat)) =sn;   SPSN(AE(sn,sdat)) = sn;
SPSN(RS(tkass,rtyp,sn,sdat)) = sn;
SPSN(RA(tkass,sn,sdat)) = sn;          SPSN(AR(srf,actid1,sn,actid2,sdat)) = sn;
IsAS(pd) => SPSN(pd) = s(Succ(0)); (* s maps a natural number on a value of sort SSPSN *)
ofsort SPReference
SPRef(CN(spr,cait,srqms,sa1,sa2,sdat)) = spr;
SPRef(AC(spr,cait,stok,srqms,sa1,sa2,sdat)) = spr;
ofsort SAddress
CgAddr(CN(spr,cait,srqms,sa1,sa2,sdat)) = sa1;
CdAddr(CN(spr,cait,srqms,sa1,sa2,sdat)) = sa2;
CgAddr(AC(spref,cait,stok,srqms,sa1,sa2,sdat)) = sa1;
CdAddr(AC(spr,cait,stok,srqms,sa1,sa2,sdat)) = sa2;
ofsort Nat
Version(CN(spr,cait,srqms,sa1,sa2,sdat)) = Version(cait);
Version(AC(spr,cait,stok,srqms,sa1,sa2,sdat)) = Version(cait);
ofsort SResynType
ResynType(RS(tkass,rtyp,sn,sdat)) = rtyp;
ofsort TDISPar
TDISPar(RF(spr,tdpar,srqms,vs,str)) = tdpar;     TDISPar(FN(tdpar,sdat)) = tdpar;
TDISPar(AB(tdpar,reas,rpv,sdat)) = tdpar;
ofsort Nat
Reason(RF(spr,tdpar,srqms,vs,str)) = ToNatNum(First(str));
Reason(AB(tdpar,reas,rpv,sdat)) = reas;          Reason(AI(reas)) = reas;
Reason(AD(reas)) = reas;    Reason(ED(reas,sdat)) = reas;
ofsort TSDUSize
MTSDUPar(CN(spr,cait,sqrms,sa1,sa2,sdat)) = TSDUSize(cait);
MTSDUPar(AC(spr,cait,stok,srqms,sa1,sa2,sdat)) = TSDUSize(cait);
ofsort Nat
MtsSendFlow(CN(spr,cait,sqrms,sa1,sa2,sdat)) = inresp(TSDUSize(cait));
MtsRecFlow(CN(spr,cait,sqrms,sa1,sa2,sdat)) = respin(TSDUSize(cait));
MtsSendFlow(AC(spr,cait,stok,sqrms,sa1,sa2,sdat)) = inresp(TSDUSize(cait));
MtsRecFlow(AC(spr,cait,stok,sqrms,sa1,sa2,sdat)) = respin(TSDUSize(cait));
ofsort SFUs
FUs(CN(spr,cait,srqms,sa1,sa2,sdat)) = srqms; FUs(AC(spr,cait,stok,srqms,sa1,sa2,sdat)) = srqms;
FUs(RF(spr,tdpar,srqms,vs,str)) = srqms;
ofsort DatOctStr
RPV(AB(tdpar,reas,str,sdat)) = str;     RPV(ER(str)) = str;
ofsort STsAss
TokenSet(CN(spr,cait,srqms,sa1,sa2,sdat)) = STsAss(cait);
TokenSet(AC(spr,cait,stok,srqms,sa1,sa2,sdat)) = STsAss(cait);
TokenSet(RS(tkass,rtyp,sn,sdat)) = tkass;
TokenSet(RA(tkass,sn,sdat)) = tkass;
InvTokenSet(CN(spr,cait,srqms,sa1,sa2,sdat)) = STsAss(Second(STsAss(cait)),First(STsAss(cait)),{});
InvTokenSet(AC(spr,cait,stok,srqms,sa1,sa2,sdat)) = STsAss(Second(STsAss(cait)),First(STsAss(cait)),{});
InvTokenSet(RS(tkass,rtyp,sn,sdat)) = STsAss(Second(tkass),First(tkass),{});
InvTokenSet(RA(tkass,sn,sdat)) = STsAss(Second(tkass),First(tkass),{});
ofsort STokens
Tokens(AC(spr,cait,stok,srqms,sa1,sa2,sdat)) = stok;     Tokens(GT(stok)) = stok;
Tokens(PT(stok,sdat)) = stok;
```

```
ofsort Bool
ProtOptions(CN(spr,cait,srqms,sa1,sa2,sdat)) = Options(cait);
ProtOptions(AC(spr,cait,stok,srqms,sa1,sa2,sdat)) = Options(cait);
ofsort PrepType
PrepType(PR(prt) = prt;
ofsort EncIt
EncItem(DT(encit,sdat)) = encit;        EncItem(TD(encit,sdat)) = encit;
ofsort SData
UserInf(CN(spr,cait,srqms,sa1,sa2,sdat)) = sdat;
UserInf(AC(spr,cait,stok,srqms,sa1,sa2,sdat)) = sdat;
UserInf(RF(spr,tdpar,srqms,vs,str)) = sdat;
UserInf(FN(tdpar,sdat)) = sdat;         UserInf(DN(sdat)) = sdat;
UserInf(NF(sdat)) = sdat;    UserInf(AB(tdpar,reas,rpv,sdat)) = sdat;
UserInf(DT(encit,sdat)) = sdat;         UserInf(TD(encit,sdat)) = sdat;
UserInf(CD(sdat)) = sdat;    UserInf(EX(sdat)) = sdat;
UserInf(CDA(sdat)) = sdat;   UserInf(PT(stok,sdat)) = sdat;
UserInf(AR(srf,actid1,sn,actid2,sdat)) = sdat;
UserInf(RS(tkass,rtyp,sn,sdat)) = sdat;
UserInf(RA(tkass,sn,sdat)) = sdat;
UserInf(MIP(styp,sn,sdat)) = sdat;
UserInf(MIA(sn,sdat)) = sdat;           UserInf(MAP(sn,sdat)) = sdat;
UserInf(MAA(sn,sdat)) = sdat;           UserInf(AEA(sn,sdat)) = sdat;
UserInf(AS(actid1,sdat)) = sdat;        UserInf(AE(sn,sdat)) = sdat;
UserInf(ED(reas,sdat)) = sdat;
endtype
(*------------------------------------------------------------------------------------------------
```

9.1.1.4 Equality of SPDUs

ASPDUEquality enriches the ASPDU construction with boolean equality on SPDUs.

```
------------------------------------------------------------------------------------------------*)
type ASPDUEquality is ASPDUParameterSelectors
opns _eq_, _ne_: ASPDU,ASPDU -> Bool
eqns forall pd1,pd2:ASPDU, spr1,spr2:SPReference, sreq1,sreq2:SFUs, cait1,cait2:CAItem,
        cg1,cd1,cg2,cd2:SAddress, d1,d2:SData, tk1,tk2:STokens, tdpar1,tdpar2:TDISPar,
        vs1,vs2,reas1,reas2:Nat, rcode1,rcode2,rpv1,rpv2,err1,err2:DatOctStr, sn1,sn2:SSPSN,
        a1,a2:STsAss, et1,et2:EncIt, sntp1,sntp2:SSyncType, rt1,rt2:SResynType, pt1,pt2:PrepType,
        aid1,aid2,aid3,aid4:SActId, scr1,scr2:SCRef
ofsort Bool
pd1 ne pd2 = not(pd1 eq pd2);
k(pd1) ne k(pd2) => pd1 eq pd2 = false;
CN(spr1,cait1,sreq1,cg1,cd1,d1) eq CN(spr2,cait2,sreq2,cg2,cd2,d2) = (spr1 eq spr2) and
        (cait1 eq cait2) and (sreq1 eq sreq2) and (cg1 eq cg2) and (cd1 eq cd2) and (d1 eq d2);
AC(spr1,cait1,tk1,sreq1,cg1,cd1,d1) eq AC(spr2,cait2,tk2,sreq2,cg2,cd2,d2) = (spr1 eq spr2) and
        (cait1 eq cait2) and (tk1 eq tk2) and (sreq1 eq sreq2) and (cg1 eq cg2) and (cd1 eq cd2) and
        (d1 eq d2);
RF(spr1,tdpar1,sreq1,vs1,rcode1) eq RF(spr2,tdpar2,sreq2,vs2,rcode2) = (spr1 eq spr2) and
        (tdpar1 eq tdpar2) and (sreq1 eq sreq2) and (vs1 eq vs2) and (rcode1 eq rcode2) ;
FN(tdpar1,d1) eq FN(tdpar2,d2) = (tdpar1 eq tdpar2) and (d1 eq d2);
DN(d1) eq DN(d2) = d1 eq d2;
NF(d1) eq NF(d2) = d1 eq d2;
AB(tdpar1,reas1,rpv1,d1) eq AB(tdpar2,reas2,rpv2,d2) = (tdpar1 eq tdpar2) and
        (reas1 eq reas2) and (rpv1 eq rpv2) and (d1 eq d2);
(k(pd1) eq k(pd2) and (IsGTC(pd1) or IsGTA(pd1) or IsAA(pd1) or IsAIA(pd1) or IsADA(pd1))) =>
        pd1 eq pd2 = true;
DT(et1,d1) eq DT(et2,d2) = (et1 eq et2) and (d1 eq d2);
TD(et1,d1) eq TD(et2,d2) = (et1 eq et2) and (d1 eq d2);
CD(d1) eq CD(d2) = d1 eq d2;
```

```
GT(tk1) eq GT(tk2) = tk1 eq tk2;
PT(tk1,d1) eq PT(tk2,d2) = (tk1 eq tk2) and (d1 eq d2);
MIP(sntp1,sn1,d1) eq MIP(sntp2,sn2,d2) = (sntp1 eq sntp2) and (sn1 eq sn2) and (d1 eq d2);
MIA(sn1,d1) eq MIA(sn2,d2) = (sn1 eq sn2) and (d1 eq d2);
MAP(sn1,d1) eq MAP(sn2,d2) = (sn1 eq sn2) and (d1 eq d2);
AE(sn1,d1) eq AE(sn2,d2) = (sn1 eq sn2) and (d1 eq d2);
MAA(sn1,d1) eq MAA(sn2,d2) = (sn1 eq sn2) and (d1 eq d2);
AEA(sn1,d1) eq AEA(sn2,d2) = (sn1 eq sn2) and (d1 eq d2);
RA(a1,sn1,d1) eq RA(a2,sn2,d2) = (a1 eq a2) and (sn1 eq sn2) and (d1 eq d2);
RS(a1,rt1,sn1,d1) eq RS(a2,rt2,sn2,d2) = (a1 eq a2) and (rt1 eq rt2) and (sn1 eq sn2) and (d1 eq d2);
PR(pt1) eq PR(pt2) = pt1 eq pt2;
AS(aid1,d1) eq AS(aid2,d2)= (aid1 eq aid2) and (d1 eq d2);
AR(scr1,aid1,sn1,aid2,d1) eq AR(scr2,aid3,sn2,aid4,d2) = (scr1 eq scr2) and
     (aid1 eq aid3) and (sn1 eq sn2) and (aid2 eq aid4) and (d1 eq d2);
AI(reas1) eq AI(reas2) = reas1 eq reas2;
AD(reas1) eq AD(reas2) = reas1 eq reas2;
DUM(err1) eq DUM(err2) = err1 eq err2;
endtype
(*------------------------------------------------------------------------------------------------
```

9.1.1.5 Other functions on abstract SPDUs

Type *ASPDU* enriches the previous definition with functions to determine whether certain conditions are satified by a given SPDU, e.g. concerning the SPDU parameter values. One auxiliary function, *PREP_Needed*, does not have a SPDU as argument, but a SSP; it determines the need to send a PR SPDU.

```
------------------------------------------------------------------------------------------------*)
type ASPDU is ASPDUEquality,TFlow
opns
PREP_Needed: SSP -> Bool                    _PREP_: ASPDU,SSP -> Bool
DequeuingPDUs: ASPDU -> Bool                CorrectCN: ASPDU -> Bool
Matches: ASPDU,ASPDU -> Bool                ValidSegmenting: Nat,ASPDU -> Bool
CL,C,CF: ASPDU -> Bool                      Err_Comb: ASPDU,TFlow,Bool -> Bool
DummyPTorGT: ASPDU -> Bool
eqns forall ssp:SSP, pd:ASPDU, tex:Bool, fl:TFlow, encit:EncIt, mt:Nat, spr,spr1,spr2:SPReference,
     cait,cait1,cait2:CAItem, fus,fus1,fus2:SFUs, sa1,sa2,sa3,sa4:SAddress, sdat,sdat1,sdat2:SData,
     stok:STokens
ofsort Bool
PREP_Needed(ssp) = (IsSRSYNreq(ssp) or IsSRSYNrsp(ssp) or IsSACTIreq(ssp) or IsSACTIrsp(ssp) or
     IsSACTDreq(ssp) or IsSACTDrsp(ssp) or IsSACTErsp(ssp) or IsSSYNMarsp(ssp));
not (PREP_Needed(ssp)) => pd PREP ssp = false;
(IsSRSYNreq(ssp) or IsSACTIreq(ssp) or IsSACTDreq(ssp)) => pd PREP ssp =  pd eq  PR(res) ;
(IsSRSYNrsp(ssp) or IsSACTIrsp(ssp) or IsSACTDrsp(ssp)) => pd PREP ssp =  pd eq  PR(rack);
(IsSACTErsp(ssp) or IsSSYNMarsp(ssp)) => pd PREP ssp = pd eq  PR(mack);
DequeuingPDUs(pd) = (IsAC(pd) or IsAEA(pd) or IsMAA(pd) or IsRS(pd) or IsRA(pd) or IsAIA(pd) or
     IsADA(pd) or IsAR(pd) or IsAS(pd) or IsGTA(pd));
DummyPTorGT(pd) = (IsPT(pd) or IsGT(pd)) and ({} eq Tokens(pd));
not (IsDT(pd) or IsTD(pd)) => CL(pd) = false;
CL(DT(encit,sdat)) = (encit eq absent) or (encit eq end);
CL(TD(encit,sdat)) = (encit eq absent) or (encit eq end);
(* CL: complete SSDU or last segment of a segmented SSDU. *)
not (IsDT(pd) or IsTD(pd)) => CF(pd) = false;
not (IsDT(pd) or IsTD(pd)) => C(pd) = false;
C(DT(encit,sdat)) = (encit eq absent);
C(TD(encit,sdat)) = (encit eq absent);
(* CF: complete SSDU or first segment of a segmented SSDU; C: complete SSDU. *)
not (IsDT(pd) or IsTD(pd)) => ValidSegmenting(mt,pd) = false;
( (pd eq DT(encit,sdat)) or (pd eq TD(encit,sdat)) ) => ValidSegmenting(mt,pd) =
     ( ((encit ne absent) iff (mt ne 0)) and ((encit ne end) implies (sdat ne (<> of SData))) );
```

```
(* Err_Comb: SPDU received on the wrong transport flow. *)
Err_Comb(pd,fl,tex) = ((IsEX(pd) or IsPR(pd)) and (fl eq normal)) or
        ( (IsAA(pd) or IsAB(pd)) and (tex implies (fl eq normal)) )
(* CorrectCN checks whether an incoming CN SPDU satisfies clause 7.1.1. of ISO 8327. *)
not(IsCN(pd)) => CorrectCN(pd) = false;
CorrectCN(CN(spr,cait,fus,sa1,sa2,sdat)) =
        ( ((not(ACT IsIn fus) and ((MA IsIn fus) or (SY IsIn fus) or (RESYN IsIn fus))) implies
        (SSPSNSSynNumber(cait) ne absent) ) or
        ( ((ACT IsIn fus) and (SSPSN(cait) ne absent)) implies
        ((MA IsIn fus) or (SY IsIn fus) or (RESYN IsIn fus))) );
(* Matches yields true if its arguments constitute a valid combination of a CN and an AC SPDU according to
clause 7.2.1 of ISO 8327. *)
not(IsCN(pd1) and IsAC(pd2)) => Matches(pd1,pd2) = false;
Matches(CN(spr1,cait1,fus1,sa1,sa2,sdat1),AC(spr2,cait2,stok,fus2,sa3,sa4,sdat2)) =
        ((sa3 ne absent) implies (sa3 eq sa1)) and ((sa4 ne absent) implies (sa4 eq sa2)) and
        (       (SSPSN(cait2) ne absent) iff
                (not(ACT IsIn Ints(fus1,fus2)) and ((MA IsIn Ints(fus1,fus2)) or (SY IsIn Ints(fus1,fus2)) or
                (RESYN IsIn Ints(fus1,fus2))) )
        ) and
        (stok eq Ints(stok,RqrTokens(STsAss(cait1)))) and
        not((FD IsIn fus2) and (HD IsIn fus2)) and
        ((FD IsIn fus2) implies (FD IsIn fus1)) and ((HD IsIn fus2) implies (HD IsIn fus1)) and
        (ChoiceTokens(STsAss(cait2)) eq {}) and
        (       ((mi IsIn RqrTokens(STsAss(cait2))) or (mi IsIn AcrTokens(STsAss(cait2))))
                implies (SY IsIn Ints(fus1,fus2))     ) and
        (       ((ma IsIn RqrTokens(STsAss(cait2))) or (ma IsIn AcrTokens(STsAss(cait2))))
                implies ((MA IsIn Ints(fus1,fus2)) or (ACT IsIn Ints(fus1,fus2)))     ) and
        (       ((dk IsIn RqrTokens(STsAss(cait2))) or (dk IsIn AcrTokens(STsAss(cait2))))
                implies (HD IsIn Ints(fus1,fus2))     ) and
        (       ((nr IsIn RqrTokens(STsAss(cait2))) or (nr IsIn AcrTokens(STsAss(cait2))))
                implies (NR IsIn Ints(fus1,fus2))     ) and
        (       (tk IsIn RqrTokens(STsAss(cait2))) implies
                ( (tk IsIn RqrTokens(tkass1)) or (tk IsIn ChoiceTokens(STsAss(cait1))) )     ) and
        (       (tk IsIn AcrTokens(STsAss(cait2))) implies
                ( (tk IsIn AcrTokens(tkass1)) or (tk IsIn ChoiceTokens(tkass1)) )     );
endtype
(*-----------------------------------------------------------------------------------------------------------
```

9.1.2 Concatenation of SPDUs

The basic construction of concatenated SPDUs is defined by type *BasicConcASPDUs* that consists of an actualization of the standard type *String*. The empty string of concatenated SPDUs is represented by constant *NULL*.

```
-----------------------------------------------------------------------------------------------------------*)
type BasicConcASPDUs is String actualizedby NaturalNumber,ASPDU,Boolean using
sortnames   ASPDU for Element          Bool for FBool            ASPDUs for String
opnnames    ASPDUs for String          NULL for <>               Number for Length
endtype
(*-----------------------------------------------------------------------------------------------------------
```

Type *ConcASPDUs* enriches this definition with functions:

- *Empty*: yields true if the argument string is equal to *NULL*;
- *PRAinQ*: yields true if the argument string contains a PR (RESYNCHRONIZE ACK) SPDU;
- *First,..,Fourth*: yield the first,..,fourth SPDU in the given string;
- *IsXConcat*: yields true if the argument string is an extended concatenated sequence of SPDUs;
- *IsValidConc*: yields true if the argument string is a valid concatenated sequence of SPDUs;

- *IsValConc*: yields true if the argument string could be part of a valid concatenated sequence of SPDUs;
- *M_S_w_GT*: yields true if only a dummy GT SPDU can be concatenated to the argument string in order to form a valid concatenated sequence of SPDUs;
- *CAT1,CAT0*: yield true if their argument string belong in certain categories;
- *Concatenate*: appends a SPDU to a string of SPDUs; and
- *ConcatPDUs*: adds a new SPDU to a concatenated sequence in such a way that the result still forms a valid concatenation according to clause 6 of ISO 8327.

```
------------------------------------------------------------------------------------------------------------*)
type ConcASPDUs is BasicConcASPDUs
opns
Empty,PRAinQ: ASPDUs -> Bool                        Rest: ASPDUs -> ASPDUs
First,Second,Third,Fourth: ASPDUs -> ASPDU
IsXConcat,IsValidConc,IsValConc,M_S_w_GT: ASPDUs -> Bool
CAT1,CAT0: ASPDU -> Bool
ConcatPDUs,Concatenate: ASPDU,ASPDUs -> ASPDUs
eqns forall ps:ASPDUs, pd,pd1,pd2,pd3:ASPDU
ofsort ASPDUs
Rest(NULL) = NULL;                                  Rest(Concatenate(pd,ps)) = ps;

ofsort ASPDU
First(NULL) = DUM(empty) ;                          Second(NULL) = DUM(empty);
Third(NULL) = DUM(empty);                           Fourth(NULL) = DUM(empty);
First(Concatenate(pd,ps)) = pd;                     Second(ps) = First(Rest(ps));
Third(ps) = Second(Rest(ps));                       Fourth(ps) = Third(Rest(ps));
ofsort Bool
Empty(ps) = (ps eq NULL);                           PRAinQ(NULL) = false;
PRAinQ(Concatenate(pd,ps)) = ( (pd eq PREP(rack)) or PRAinQ(ps) );

M_S_w_GT(ASPDUs(pd)) = ( (IsDT(pd) and not(CL(pd))) or IsRS(pd) or IsAD(pd) or IsAI(pd) or IsCD(pd) );
IsValConc(pd1+ASPDUs(pd2)) => M_S_w_GT(pd1+ASPDUs(pd2)) =
    ( ( (IsAS(pd1) or IsAR(pd1)) and (IsAE(pd2) or IsMAP(pd2)) ) or
    ( (IsAS(pd1) or IsAR(pd1)) and (begin eq EncItem(pd2)) ) );
( IsValConc(ps) and (Number(ps) eq NatNum(Dec(3))) ) => M_S_w_GT(ps) = (not(IsMIP(Second(ps))));
(* M_S_w_GT yields true if its argument can only be concatenated to a GT SPDU not containing a Token
Item parameter (i.e., a dummy GT SPDU). *)

(not(IsValConc(ps))) => M_S_w_GT(ps) = false;
(not(IsValidConc(ps))) => IsXConcat(ps) = false;
(IsValidConc(ps) and (Number(ps) gt NatNum(Dec(2)))) => IsXConcat(ps) = true;
(IsValidConc(ps) and (Number(ps) lt NatNum(Dec(2)))) => IsXConcat(ps) = false;
IsValidConc(pd1+ASPDUs(pd2)) => IsXConcat(pd1+ASPDUs(pd2)) =
    (IsGT(pd1) and (IsMIA(pd2) or IsMAA(pd2) or IsAEA(pd2)));
IsValConc(ASPDUs(pd1)) = true;
IsValConc(NULL) = false;
(Number(ps) gt NatNum(Dec(3))) => IsValConc(ps) = false;
IsValConc(pd1+ASPDUs(pd2)) =
    ( ( (IsAS(pd1) or IsAR(pd1)) and (IsMIP(pd2) or IsAE(pd2) or IsMAP(pd2) or CF(pd2)) ) or
    ( (IsMIP(pd1) or IsMIA(pd1) or IsMAP(pd1) or IsMAA(pd1) or IsAE(pd) or IsAEA(pd)) and CL(pd2) ) );
IsValConc(pd1+(pd2+ASPDUs(pd3))) =
    ( (IsAS(pd1) or IsAR(pd1)) and (IsMIP(pd2) or IsMAP(pd2) or IsAE(pd2)) and C(pd3) );
IsValidConc(ASPDUs(pd)) = CAT1(pd);

CAT1(pd) = ( IsCN(pd) or IsAC(pd) or IsRF(pd) or IsFN(pd) or IsDN(pd) or IsNF(pd) or IsAB(pd) or
    IsAA(pd) or IsGTC(pd) or IsGTA(pd) or IsTD(pd) or IsEX(pd) or IsPR(pd) );
CAT0(pd) = (IsGT(pd) or IsPT(pd));
```

```
IsValidConc(pd1+ASPDUs(pd2)) = ( not(CAT1(pd2) or CAT0(pd2)) and (((IsDT(pd2) and not(CL(pd2))) or
        IsRS(pd2) or IsAD(pd2) or IsAI(pd2) or IsCD(pd2)) implies (IsGT(pd1) and DummyPTorGT(pd1)) ) and
        ( (IsRA(pd2) or IsADA(pd2) or IsAIA(pd2) or IsCDA(pd2)) implies IsPT(pd1) ) and ( (IsMIA(pd2) or
        IsMAA(pd2) or IsAEA(pd2)) implies (IsPT(pd1) or IsGT(pd1)) ) and ( (CL(pd2) or IsMIP(pd2) or
        IsMAP(pd2) or IsAS(pd2) or IsAR(pd2) or IsAE(pd2)) implies IsGT(pd1) ) );
(Number(ps) gt NatNum(Dec(2))) => IsValidConc(ps) = IsGT(First(ps)) and IsValConc(Rest(ps)) and
        ( M_S_w_GT(Rest(ps)) implies DummyPTorGT(First(ps)) );

ofsort ASPDUs
(IsAS(pd1) or IsAR(pd1)) => ConcatPDUs(pd,ASPDUs(pd1)) = Reverse(pd+ASPDUs(pd1));
( IsDT(pd1) and (IsMIP(pd) or IsMIA(pd) or IsMAP(pd) or IsMAA(pd) or IsAE(pd) or IsAEA(pd)) ) =>
        ConcatPDUs(pd,ASPDUs(pd1)) = pd+ASPDUs(pd1);
( IsDT(pd1) and not (IsMIP(pd) or IsMIA(pd) or IsMAP(pd) or IsMAA(pd) or IsAE(pd) or IsAEA(pd)) ) =>
        ConcatPDUs(pd,ASPDUs(pd1)) = NULL;
ConcatPDUs(pd,NULL) =  ASPDUs(pd);
(IsMIP(pd1) or IsMIA(pd1) or IsMAP(pd1) or IsMAA(pd1) or IsAE(pd1) or IsAEA(pd1)) =>
        ConcatPDUs(pd1,ASPDUs(pd1)) = NULL;
( (IsAR(pd1) or IsAS(pd1)) and IsDT(pd2) and (IsMIP(pd) or IsMIA(pd) or IsMAP(pd) or IsMAA(pd) or
        IsAE(pd) or IsAEA(pd)) ) => ConcatPDUs(pd,pd1+ASPDUs(pd2)) = pd1+(pd+ASPDUs(pd2));
( not ( (IsAR(pd1) or IsAS(pd1)) and IsDT(pd2) and
        (IsMIP(pd) or IsMIA(pd) or IsMAP(pd) or IsMAA(pd) or IsAE(pd) or IsAEA(pd)) ) ) =>
        ConcatPDUs(pd,pd1+ASPDUs(pd2)) = NULL
endtype
(*---------------------------------------------------------------------------------------------------------
```

9.1.3 SPDU parameters

9.1.3.1 Connection Identifier parameter group

The Connection Identifier parameter group, contained in CN, AC and RF SPDUs, is defined by type *SPReference*. It consists of an actualization of the auxiliary type *Threetuple* (see 9.4.6).

```
---------------------------------------------------------------------------------------------------------*)
type SPReference is Threetuple actualizedby DatOctStr1,Boolean using
sortnames                      Bool for FBool                 DatOctStr for Element1
DatOctStr for Element2         DatOctStr for Element3         SPReference for Threetuple
opnnames                       SPReference for Threetuple     CgUserReference for First
CommonReference for Second     AdditionalRef for Third
endtype
(*---------------------------------------------------------------------------------------------------------
```

9.1.3.2 Connect/Accept Item parameter group

The Connect/Accept Item parameter group, contained in CN an AC SPDUs, is defined by type *CAItem*. It consists of an actualization of the auxiliary type *FiveTuple* (see 9.4.6). *SSynchronizationNumber* and *STokensAssignment* are imported from ISO/IEC/TR 9571.

```
---------------------------------------------------------------------------------------------------------*)
type CAItem is Fivetuple actualizedby TSDUSize,SSynchronizationNumber,STokensAssignment using
sortnames           Bool for FBool         Bool for Element1        TSDUSize for Element2
Nat for Element3    SSPSN  for Element4    STsAss for Element5      CAItem for Fivetuple
opnnames            Options for First      CAItem for Fivetuple     TSDUSize for Second
Version for Third   SSPSN for Fourth       STsAss for Fifth
endtype
(*---------------------------------------------------------------------------------------------------------
```

9.1.3.3 TSDU Maximum Size parameter

The TSDU Maximum Size parameter, contained in CN and AC SPDUs, is defined by type *TSDUSize*, an enrichment of type *TSDUSize0*. The latter type consists of an actualization of auxiliary type *TwoTuple* (see 9.4.6). Function *inresp* is used to determine the TSDU Maximum Size value for the direction from initiator to responder, *respin* is used to determine the value for the opposite direction. Value *0* indicates that segmenting is not allowed for the direction of concern. The constant *absent*, introduced by type *TSDUSize*, indicates that segmenting is not allowed for either direction.

```
------------------------------------------------------------------------------------------*)
type TSDUSize is TSDUSize0
opns                absent: -> TSDUSize       _eq_: TSDUSize,TSDUSize -> Bool
eqns ofsort Nat     inresp(absent) = 0;       respin(absent) = 0
endtype
type TSDUSize0 is Twotuple actualizedby Boolean,NaturalNumber using
sortnames
Bool for FBool      Nat for Element1          Nat for Element2      TSDUSize for Twotuple
opnnames            TSDUSize for Twotuple     inresp for First      respin for Second
endtype
(*------------------------------------------------------------------------------------------
```

9.1.3.4 Transport Disconnect parameter

The Transport Disconnect parameter, contained in RF, FN and AB SPDUs, is defined by type *TDISPar*. It consists of a renaming of auxiliary type *Doublet*, imported from ISO/IEC/TR 9571. Constant *nr* indicates that the sender of the SPDU requests termination of the TC; constant *r* indicates that re-use of the TC is requested.

```
------------------------------------------------------------------------------------------*)
type TDISPar is Doublet renamedby
sortnames   TDISPar for Doublet
opnnames    nr for constant1         r for constant2
endtype
(*------------------------------------------------------------------------------------------
```

9.1.3.5 Prepare Type parameter

The Prepare Type parameter, contained in PR SPDUs, is defined by type *Prepare*. It consists of a renaming of type *Triplet*, imported from ISO/IEC/TR 9571. The three constants *res*, *rack* and *mack*, respectively, correspond with the values "RESYNCHRONIZE", "RESYNCHRONIZE ACK" and "MAJOR SYNC ACK" defined in table 10 of ISO 8327.

```
------------------------------------------------------------------------------------------*)
type Prepare is Triplet renamedby
sortnames   PrepType for Triplet
opnnames    res for constant1        rack for constant2        mack for constant3
endtype
(*------------------------------------------------------------------------------------------
```

9.1.3.6 Enclosure Item parameter

The Enclosure Item parameter, contained in DT and TD SPDUs, is defined by type *EncItem*. It consists of a renaming of type *Quartet*, imported from ISO/IEC/TR 9571. Constant *begin* is used to indicate that the SPDU is the first segment of a segmented SSDU, *middle* that the SPDU is neither the first nor the last segment, *end* that the SPDU is the last segment of a segmented SSDU, and *absent* that segmenting is not allowed.

```
--------------------------------------------------------------------------------------------------------------*)
type EncItem is Quartet renamedby
sortnames   EncIt for Quartet
opnnames
begin for constant1        middle for constant2        end for constant3        absent for constant4
endtype
(*--------------------------------------------------------------------------------------------------------------
```

9.1.4 Encoded SPDU

9.1.4.1 SI field

Valid values of the SI field are defined by type *PDUSIs*. It imports type *PDUSIs0* that actualizes the standard type *Set*. Constant *ValidSIs* represents the set containing all valid SI values.

```
--------------------------------------------------------------------------------------------------------------*)
type PDUSIs is PDUSIs0
opns ValidSIs: -> SISet
eqns
ValidSIs = Insert(Dec(0), Insert(Dec(1), Insert(Dec(2), Insert(Dec(5), Insert(Dec(7), Insert(Dec(8),
      Insert(Dec(9), Insert(1+Dec(0), Insert(1+Dec(2), Insert(1+Dec(3), Insert(1+Dec(4), Insert(2+Dec(1),
      Insert(2+Dec(2), Insert(2+Dec(5), Insert(2+Dec(6), Insert(2+Dec(9), Insert(3+Dec(3), Insert(3+Dec(4),
      Insert(4+Dec(1), Insert(4+Dec(2), Insert(4+Dec(5), Insert(4+Dec(8), Insert(4+Dec(9), Insert(5+Dec(0),
      Insert(5+Dec(3), Insert(5+Dec(7), Insert(5+Dec(8), Insert(6+Dec(1),
      Insert(6+Dec(2),{}))))))))))))))))))))))))))))
endtype
type PDUSIs0 is Set actualizedby DecNatRepr,NaturalNumber using
sortnames   DecString for Element        Bool for FBool        SISet for Set
endtype
(*--------------------------------------------------------------------------------------------------------------
```

9.1.4.2 Auxiliary functions for SPDU valid encoding checks

Type *Err3Tuple* defines values to represent intermediate tests on the valid encoding of SPDUs. It imports type *Err3Tuple0*, an actualization of the auxiliary type *ThreeTuple* (see 9.4.6). The values are 3-tuples, where the second and third component are octet strings representing, respectively, the part of an encoding not yet tested and the part that is already tested, and the first component is a boolean indicating the result of the test so far. The function *eq* tests equality of two such 3-tuples (all three components must be equal).

```
--------------------------------------------------------------------------------------------------------------*)
type Err3Tuple is Err3Tuple0
opns _eq_ : Err3Tuple,Err3Tuple -> Bool
eqns forall s1,s2,s3,s4:DatOctStr, b1,b2:Bool ofsort Bool
Err3Tuple(b1,s1,s2) eq Err3Tuple(b2,s3,s4) = ( (b1 eq b2) and (s1 eq s3) and (s2 eq s4) );
endtype
type Err3Tuple0 is Threetuple actualizedby DatOctStr1 using
sortnames                Bool for FBool         Bool for Element1        Err3Tuple for ThreeTuple
DatOctStr for Element2   DatOctStr for Element3
opnnames                 Err3Tuple for ThreeTuple
endtype
(*--------------------------------------------------------------------------------------------------------------
```

9.1.4.3 Valid encoding checks

Type *ErrCoding* defines functions that check whether a given octet string constitutes a valid encoding of a SPDU. *ErrCodStr* yields an empty octet string if the argument octet string represents a correctly encoded SPDU, otherwise the erroneous part of the argument string is returned. The first octet of a valid encoding should represent a valid SI value (an "unknown" value is returned by *ErrCodStr*). If a valid SI value is present, the remaining part of the octet string should be an encoding of the SPDU class identified by the SI value; this is checked with *ErrCodXX* (*XX* being the denotation of the SPDU class, see 9.1.1.2). The valid encodings of the parameters (parameter groups) from which the SPDU encoding is constructed are again tested by separate functions, one for each parameter (group). The definition of the latter functions is based on a limited number of auxiliary functions.

```
---------------------------------------------------------------------------------------------------------*)
type ErrCoding is DatOctStr2,PDUSIs,Compare,Err3Tuple,BitString
opns
ErrCodStr: DatOctStr -> DatOctStr
ErrCodCN,ErrCodRF,ErrCodFN,ErrCodDNorNF,ErrCodABorAI,ErrCodEX,ErrCodTD,ErrCodGT,
        ErrCodCDorCDA,ErrCodPT,ErrCodMIA,ErrCodMAAorAEA,ErrCodRS,ErrCodRA,ErrCodPR,
        ErrCodAS,ErrCodAR,ErrCodAD,ErrCodAEorMAPorMIP,ErrCodAC,
        ErrCodNF: DatOctStr,DatOctStr -> DatOctStr
CodCN1,CodCN2,CodCN3: DatOctStr,DatOctStr,DatOctStr -> DatOctStr
CodCNLen: DatOctStr,DatOctStr -> DatOctStr
CorrConId: DatOctStr,Octet -> Err3Tuple
CorrConId1: DatOctStr,DatOctStr,DatOctStr,Octet -> Err3Tuple
CorrConId2,CorrConId3 : DatOctStr,DatOctStr,DatOctStr -> Err3Tuple
CorrCAlt,CorrRest: DatOctStr -> Err3Tuple
CorrCAlt1,CorrCAlt2,CorrCAlt3,CorrCAlt4,CorrCAlt5: DatOctStr,DatOctStr,DatOctStr -> Err3Tuple
CorrRest2,CorrRest3,CorrRest4: DatOctStr,DatOctStr -> Err3Tuple
CorrParmL: DecString,DecString,DatOctStr,DatOctStr -> Err3Tuple
CorrPar,CorrParm,MCorrPar,MCorrParm: DecString,DecString,DatOctStr -> Err3Tuple
CheckBits: DecString,DatOctStr -> DatOctStr
No11: BitString -> Bool
SPSNOctets: DatOctStr -> DatOctStr
SPSNOct2: DatOctStr,DatOctStr -> DatOctStr
eqns forall o,o1,o2,o3:Octet, ds,pgi,pi,len,li:DecString, b:Bool,
        st,s1,s2,s3,s4,s5:DatOctStr, bstr,bstr1,bstr2:BitString, b1,b2:Bit, dv1,dv2,dv3,dv4:DecString
ofsort DatOctStr
( (o1 eq ds) and not (ds IsIn ValidSIs) ) => ErrCodStr(o1+s1) = Octet(o1);
(* The SI code is not in the list of known SI codes so only this octet is returned as being the erroneous part
of the coded ASPDU. *)

(o1 eq (1+Dec(3))) => ErrCodStr(o1+s1) = ErrCodCN(s1,Octet(o1));
(o1 eq (1+Dec(4))) => ErrCodStr(o1+s1) = ErrCodAC(s1,Octet(o1));
(o1 eq (2+Dec(5))) => ErrCodStr(o1+s1) = ErrCodABorAI(s1,Octet(o1));
( (o1 eq (4+Dec(1))) or (o1 eq (4+Dec(9))) ) => ErrCodStr(o1+s1) = ErrCodAEorMAPorMIP(s1,Octet(o1));
(o1 eq (1+Dec(2))) => ErrCodStr(o1+s1) = ErrCodRF(s1,Octet(o1));
(o1 eq Dec(9)) => ErrCodStr(o1+s1) = ErrCodFN(s1,Octet(o1));
(o1 eq (1+Dec(0)) or (o1 eq Dec(8))) => ErrCodStr(o1+s1) = ErrCodDNorNF(s1,Octet(o1));
(o1 eq Dec(8)) => ErrCodStr(o1+s1) = ErrCodNF(s1,Octet(o1));
(o1 eq (3+Dec(3))) => ErrCodStr(o1+s1) = ErrCodTD(s1,Octet(o1));
(o1 eq Dec(1)) => ErrCodStr(o1+s1) = ErrCodGT(s1,Octet(o1));
(o1 eq Dec(5)) => ErrCodStr(o1+s1) = ErrCodEX(s1,Octet(o1));
( (o1 eq (6+Dec(1))) or (o1 eq (6+Dec(2))) ) => ErrCodStr(o1+s1) = ErrCodCDorCDA(s1,Octet(o1));
(o1 eq Dec(2)) => ErrCodStr(o1+s1) = ErrCodPT(s1,Octet(o1));
(o1 eq (5+Dec(0))) => ErrCodStr(o1+s1) = ErrCodMIA(s1,Octet(o1));
(o1 eq (4+Dec(2))) => ErrCodStr(o1+s1) = ErrCodMAAorAEA(s1,Octet(o1));
(o1 eq (5+Dec(3))) => ErrCodStr(o1+s1) = ErrCodRS(s1,Octet(o1));
(o1 eq (3+Dec(4))) => ErrCodStr(o1+s1) = ErrCodRA(s1,Octet(o1));
(o1 eq Dec(7)) => ErrCodStr(o1+s1) = ErrCodPR(s1,Octet(o1));
```

```
(o1 eq (4+Dec(5))) => ErrCodStr(o1+s1) = ErrCodAS(s1,Octet(o1));
(o1 eq (2+Dec(9))) => ErrCodStr(o1+s1) = ErrCodAR(s1,Octet(o1));
(o1 eq (5+Dec(7))) => ErrCodStr(o1+s1) = ErrCodAD(s1,Octet(o1));
( (o1 eq (2+Dec(6))) or (o1 eq (2+Dec(1))) or (o1 eq (2+Dec(2))) or (o1 eq (5+Dec(8))) and (o2 eq Dec(0)) )
     => ErrCodStr(o1+(o2+s1)) = ErrCodStr(s1);
( (o1 eq (2+Dec(6))) or (o1 eq (2+Dec(1))) or (o1 eq (2+Dec(2))) or (o1 eq (5+Dec(8))) and (o2 ne Dec(0)) )
     => ErrCodStr(o1+(o2+s1)) = o1+Octet(o2);
```

(CN SPDU valid encoding checks: Each parameter (group) is analysed separately to determine whether the encoding is according to the rules given in clause 8 of ISO 8327. The following is checked: whether the LI (length indicator) fields have the correct values, whether the parameters (parameter groups) are in the correct order, whether maximum length restrictions have not been violated and whether the individual parameter values have been correctly encoded. *)*

```
Empty(s1) => ErrCodCN(s1,s2) = s2;
(* The length indicator is absent. The first octet is returned as the erroneous string. *)
(o1 eq Dec(0)) => ErrCodCN(o1+s3,s2) = ErrCodStr(s3);
(* The length indicator contains the value zero, so there are no parameters present. *)

( (o1 lt (2+(5+Dec(5)))) and (Length(s3) lt o1) ) => ErrCodCN(o1+s3,s2) = s2++s1;
(* The length of the remaining string is smaller than indicated by the LI value; the whole octet string is
returned as being erroneous. *)

( (o1 lt (2+(5+Dec(5)))) and (Length(s3) eq o1) ) => ErrCodCN(o1+s3++s4,s2) = CodCN1(s3,s2--o1,s4);
(o1 eq (2+(5+Dec(5)))) => ErrCodCN(o1+s3,s2) = CodCNLen(s3,s2--o1);
```

(Two cases are distinguished in the following: 1) the LI value indicates a length smaller than 255 octets, in which case the LI field is encoded using one octet, and 2) the first octet represents the value 255, in which case the encoding of the LI field should consist of three octets. The actual LI value is then encoded in the remaining two octets and it should be greater than or equal to 255, otherwise the encoding of LI is considered to be erroneous. Furthermore, the length of any correctly encoded CN SPDU is implicitly constrained by a maximum value equal to 719. This is also checked. *)*

```
(Length(s1) lt NatNum(Dec(2))) => CodCNLen(s1,s2) = s2++s1;
( (NatNum(BStr(o1)++BStr(o2)) gt (7+(1+Dec(9)))) or (NatNum(BStr(o1)++BStr(o2)) lt (2+(5+Dec(5)))) ) =>
     CodCNLen(o1+(o2+s3),s2) = s2++(o1+Octet(o2));
( (Length(s3) lt NatNum(BStr(o1)++BStr(o2))) and (NatNum(BStr(o1)++BStr(o2)) le (7+(1+Dec(9)))) and
     (NatNum(BStr(o1)++BStr(o2)) ge (2+(5+Dec(5)))) ) =>
     CodCNLen(o1+(o2+s3),s2) = s2++s1;
(* The LI field is correctly encoded but the remainder of the string is too short. *)

( (Length(s3) eq NatNum(BStr(o1)++BStr(o2))) and (NatNum(BStr(o1)++BStr(o2)) le (7+(1+Dec(9)))) and
     (NatNum(BStr(o1)++BStr(o2)) ge (2+(5+Dec(5)))) ) =>
     CodCNLen(o1+(o2+s3++s4),s2) = CodCN1(s3,s2++(o1+Octet(o2)),s4);
(* LI and length restrictions are satisfied: next, the parameter groups are checked, starting with the
Connection Identifier parameter group. *)

First(CorrConId(s1,First(s2))) => CodCN1(s1,s2,s3) =
     CodCN2(Third(CorrConId(s1,First(s2))),s2++Second(CorrConId(s1,First(s2))),s3);
(* CorrConId returns a boolean value and two octet strings. The boolean is true if no error was "detected" in
the encoding. In that case, the second string is the part found to be correct so far and the third string is the
part still to be checked. *)

not(First(CorrConId(s1,First(s2)))) => CodCN1(s1,s2,s3) = s2++Second(CorrConId(s1,First(s2)));
First(CorrCAlt(s1)) => CodCN2(s1,s2,s3) = CodCN3(Third(CorrCAlt(s1)),s2++Second(CorrCAlt(s1)),s3);
(* The Connect/Accept Item parameter group is not mandatory. If a parameter (group) is not mandatory, an
empty octet string is considered to be a valid encoding. *)

not(First(CorrCAlt(s1))) => CodCN2(s1,s2,s3) = s2++Second(CorrCAlt(s1));
First(CorrRest(s1)) => CodCN3(s1,s2,s3) = ErrCodStr(s3);
```

not(First(CorrRest(s1))) => CodCN3(s1,s2,s3) = s2++Second(CorrRest(s1));
(CorrRest takes care of the remaining parameters of the CN SPDU.*)*

***ofsort** Err3Tuple*
Empty(s1) => CorrConId(s1,o) = Err3Tuple(true,s1,s1);
(No parameters (parameter groups) are present if s1 is empty. Only the octet representing the SI value is present. This octet is denoted by o. *)*

(o1 ne Dec(1)) => CorrConId(o1+s2,o) = Err3Tuple(true,empty,o1+s2);
(The encoding of the Connection Identifier parameter group is absent. Note that this parameter group is not mandatory. *)*

```
( (o1 eq Dec(1)) and Empty(s2) ) => CorrConId(o1+s2,o) = Err3Tuple(false,Octet(o1),s2);
( (o1 eq Dec(1)) and (o2 gt (1+(3+Dec(2)))) ) =>
      CorrConId(o1+(o2+s2),o) = Err3Tuple(false,o1+Octet(o2),s2);
(* The value contained in the LI field is too large. *)

( (o1 eq Dec(1)) and (o2 le (1+(3+Dec(2)))) and (Length(s2) lt o2) ) =>
      CorrConId(o1+(o2+s2),o) = Err3Tuple(false,o1+(o2+s2),empty);
(* The LI field contains a valid value but the remaining string is too short. *)

( (o1 eq Dec(1)) and (o2 le (1+(3+Dec(2)))) and (Length(s2) eq o2) ) =>
      CorrConId(o1+(o2+s2++s3),o) = CorrConId1(s2,o1+Octet(o2),s3,o);
```

(Individual parameters are checked below. This check is performed by the function CorrParm whose arguments represent the PI field, the maximum length of the parameter value, and the parameter encoding to be checked. For example, the PI value 10 identifies the Calling SS-user Reference parameter of the CN SPDU, and PI value 9 identifies the Called SS-user Reference parameter of an AC or RF SPDU. The maximum length of these parameters is 64 octets. CorrParm is used to check the correctness of variable, bounded, lengths of non-mandatory parameters. *)*

```
( First(CorrParm(1+Dec(0),6+Dec(4),s1)) and (o eq (1+Dec(3))) ) => CorrConId1(s1,s2,s3,o) =
      CorrConId2 (Third(CorrParm(1+Dec(0),6+Dec(4),s1)),
            s2++Second(CorrParm(1+Dec(0),6+Dec(4),s1)), s3);
( First(CorrParm(Dec(9),6+Dec(4),s1)) and (o ne (1+Dec(3))) ) => CorrConId1(s1,s2,s3,o) =
      CorrConId2 (Third(CorrParm(Dec(9),6+Dec(4),s1)), s2++Second(CorrParm(Dec(9),6+Dec(4),s1)), s3);

( not(First(CorrParm(1+Dec(0),6+Dec(4),s1))) and (o eq (1+Dec(3))) ) =>
      CorrConId1(s1,s2,s3,o) = Err3Tuple(false,s2++Second(CorrParm(1+Dec(0),6+Dec(4),s1)),s3);
( not(First(CorrParm(Dec(9),6+Dec(4),s1))) and (o ne (1+Dec(3))) ) =>
      CorrConId1(s1,s2,s3,o) = Err3Tuple(false,s2++Second(CorrParm(Dec(9),6+Dec(4),s1)),s3);

First(CorrParm(1+Dec(1),6+Dec(4),s1)) => CorrConId2(s1,s2,s3) =
      CorrConId3 (Third(CorrParm(1+Dec(1),6+Dec(4),s1)),
            s2++Second(CorrParm(1+Dec(1),6+Dec(4),s1)), s3);
not(First(CorrParm(1+Dec(1),6+Dec(4),s1))) => CorrConId2(s1,s2,s3) =
      Err3Tuple(false,s2++Second(CorrParm(1+Dec(1),6+Dec(4),s1)),s3);
( First(CorrParm(1+Dec(2),Dec(4),s1)) and Empty(Third(CorrParm(1+Dec(2),Dec(4),s1))) ) =>
      CorrConId3(s1,s2,s3) = Err3Tuple(true,s2++Second(CorrParm(1+Dec(2),Dec(4),s1)),s3);
```

(The Connection Identifier parameter group is found to be correctly encoded if there are no octets left for checking and the length indicated in the LI field corresponds to the length of the octet string representing the encoding. *)*

```
( First(CorrParm(1+Dec(2),Dec(4),s1)) and not(Empty(Third(CorrParm(1+Dec(2),Dec(4),s1)))) ) =>
      CorrConId3(s1,s2,s3) = Err3Tuple (false, s2++Second(CorrParm(1+Dec(2),Dec(4),s1))++
            Octet(First(Third(CorrParm(1+Dec(2),Dec(4),s1)))), s3);
not(First(CorrParm(1+Dec(2),Dec(4),s1))) =>
      CorrConId3(s1,s2,s3) = Err3Tuple(false,s2++Second(CorrParm(1+Dec(2),Dec(4),s1)),s3);
```

(CorrCAIt, below, checks the encoding of the Connect/Accept Item parameter group. This check is analogous to that presented above. Additional auxiliary functions are defined similar to CorrParm. These are: MCorrParm (mandatory, bounded-variable length), MCorrPar (mandatory, fixed-length) and CorrPar (non-mandatory, fixed-length). Note that in certain parameter value encodings some bits are "reserved" or set to certain specific values. This implies that additional checks are involved which are taken care of by function CheckBits. *)*

```
Empty(s1) => CorrCAIt(s1) = Err3Tuple(true,s1,s1);
(o1 ne Dec(5)) => CorrCAIt(o1+s2) = Err3Tuple(true,empty,o1+s2);
( (o1 eq Dec(5)) and Empty(s2) ) => CorrCAIt(o1+s2) = Err3Tuple(false,Octet(o1),s2);
( (o1 eq Dec(5)) and (o2 gt (1+Dec(3))) ) =>
        CorrCAIt(o1+(o2+s2)) = Err3Tuple(false,o1+Octet(o2),s2);
( (o1 eq Dec(5)) and (o2 le (1+Dec(3))) and (Length(s2) lt o2) ) =>
        CorrCAIt(o1+(o2+s2)) = Err3Tuple(false,o1+(o2+s2),empty);
( (o1 eq Dec(5)) and (o2 le (1+Dec(3))) and (Length(s2) eq o2) ) =>
        CorrCAIt(o1+(o2+s2++s3)) = CorrCAIt1(s2,o1+Octet(o2),s3);
First(MCorrPar(1+Dec(9),Dec(1),s1)) => CorrCAIt1(s1,s2,s3) =
        CorrCAIt2(Third(MCorrPar(1+Dec(9),Dec(1),s1)),s2++Second(MCorrPar(1+Dec(9),Dec(1),s1)),s3);
not(First(MCorrPar(1+Dec(9),Dec(1),s1))) =>
        CorrCAIt1(s1,s2,s3) = Err3Tuple(false,s2++Second(MCorrPar(1+Dec(9),Dec(1),s1)),s3);
First(CorrPar(2+Dec(1),Dec(4),s1)) => CorrCAIt2(s1,s2,s3) =
        CorrCAIt3(Third(CorrPar(2+Dec(1),Dec(4),s1)),s2++Second(CorrPar(2+Dec(1),Dec(4),s1)),s3);
not(First(CorrPar(2+Dec(1),Dec(4),s1))) =>
        CorrCAIt2(s1,s2,s3) = Err3Tuple(false,s2++Second(CorrPar(2+Dec(1),Dec(4),s1)),s3);
First(MCorrPar(2+Dec(2),Dec(1),s1)) => CorrCAIt3(s1,s2,s3) =
        CorrCAIt4(Third(MCorrPar(2+Dec(2),Dec(1),s1)),s2++Second(MCorrPar(2+Dec(2),Dec(1),s1)),s3);
not(First(MCorrPar(2+Dec(2),Dec(1),s1))) =>
        CorrCAIt3(s1,s2,s3) = Err3Tuple(false,s2++Second(MCorrPar(2+Dec(2),Dec(1),s1)),s3);
First(CorrParm(2+Dec(3),Dec(6),s1)) => CorrCAIt4(s1,s2,s3) =
        CorrCAIt5(Third(CorrParm(2+Dec(3),Dec(6),s1)),s2++Second(CorrParm(2+Dec(3),Dec(6),s1)),s3);
not(First(CorrParm(2+Dec(3),Dec(6),s1))) =>
        CorrCAIt4(s1,s2,s3) = Err3Tuple(false,s2++Second(CorrParm(2+Dec(3),Dec(6),s1)),s3);
( First(CorrParm(2+Dec(6),Dec(1),s1)) and Empty(Third(CorrParm(2+Dec(6),Dec(1),s1))) ) =>
         CorrCAIt5(s1,s2,s3) = Err3Tuple(true,s2++Second(CorrParm(2+Dec(6),Dec(1),s1)),s3);
( First(CorrParm(2+Dec(6),Dec(1),s1)) and not(Empty(Third(CorrParm(2+Dec(6),Dec(1),s1)))) ) =>
        CorrCAIt5(s1,s2,s3) = Err3Tuple(false,s2++Second(CorrParm(2+Dec(6),Dec(1),s1))++
                Octet(First(Third(CorrParm(2+Dec(6),Dec(1),s1)))),s3);
not(First(CorrParm(2+Dec(6),Dec(1),s1))) =>
        CorrCAIt5(s1,s2,s3) = Err3Tuple(false,s2++Second(CorrParm(2+Dec(6),Dec(1),s1)),s3);
```

(CorrRest checks whether the Session Requirements parameter, the Session Address parameters and the SS-user Data parameter have been correctly encoded. *)*

```
First(CorrPar(2+Dec(0),Dec(2),s1)) => CorrRest(s1) =
        CorrRest2(Third(CorrPar(2+Dec(0),Dec(2),s1)),Second(CorrPar(2+Dec(0),Dec(2),s1)));
not(First(CorrPar(2+Dec(0),Dec(2),s1))) => CorrRest(s1) =
        Err3Tuple(false,Second(CorrPar(2+Dec(0),Dec(2),s1)),Third(CorrPar(2+Dec(0),Dec(2),s1)));
First(CorrParm(5+Dec(1),1+Dec(6),s1)) => CorrRest2(s1,s4) =
        CorrRest3(Third(CorrParm(5+Dec(1),1+Dec(6),s1)),Second(CorrParm(5+Dec(1),1+Dec(6),s1))++s4);
not(First(CorrParm(5+Dec(1),1+Dec(6),s1))) => CorrRest2(s1,s4) =
        Err3Tuple (false, Second(CorrParm(5+Dec(1),1+Dec(6),s1))++s4,
                Third(CorrParm(5+Dec(1),1+Dec(6),s1)));
First(CorrParm(5+Dec(2),1+Dec(6),s1)) =>
        CorrRest3(s1,s4) = CorrRest4(s3,Second(CorrParm(5+Dec(2),1+Dec(6),s1))++s4);
not(First(CorrParm(5+Dec(2),1+Dec(6),s1))) => CorrRest3(s1,s4) =
        Err3Tuple (false, Second(CorrParm(5+Dec(2),1+Dec(6),s1))++s4,
                Third(CorrParm(5+Dec(2),1+Dec(6),s1)));
( First(CorrParm(1+(9+Dec(3)),5+(1+Dec(2)),s1)) and
        Empty(Third(CorrParm(1+(9+Dec(3)),5+(1+Dec(2)),s1))) ) => CorrRest4(s1,s4) =
                Err3Tuple(true,Second(CorrParm(1+(9+Dec(3)),5+(1+Dec(2)),s1))++s4,empty);
```

```
( First(CorrParm(1+(9+Dec(3)),5+(1+Dec(2)),s1)) and
        not(Empty(Third(CorrParm(1+(9+Dec(3)),5+(1+Dec(2)),s1)))) ) => CorrRest4(s1,s4) =
                Err3Tuple (false, s2++Octet(First(Third(CorrParm(1+(9+Dec(3)),5+(1+Dec(2)),s1))))++s4,
                        Third(CorrParm(1+(9+Dec(3)),5+(1+Dec(2)),s1)));
not(First(CorrParm(1+(9+Dec(3)),5+(1+Dec(2)),s1))) => CorrRest4(s1,s4) =
        Err3Tuple(false,Second(CorrParm(1+(9+Dec(3)),5+(1+Dec(2)),s1))++s4,
                Third(CorrParm(1+(9+Dec(3)),5+(1+Dec(2)),s1)));

(* AC SPDU valid encoding checks: *)
ofsort DatOctStr
Empty(s1) => ErrCodAC(s1,s2) = s2;
(o1 eq Dec(0)) => ErrCodAC(o1+s3,s2) = ErrCodStr(s3);
( (o1 lt (2+(5+Dec(5)))) and (Length(s3) lt o1) ) => ErrCodAC(o1+s3,s2) = s2++s1;
( (o1 lt (2+(5+Dec(5)))) and (Length(s3) eq o1) ) => ErrCodAC(o1+s3++s4,s2) = CodAC1(s3,s2--o1,s4);
(o1 eq (2+(5+Dec(5)))) => ErrCodAC(o1+s3,s2) = CodACLen(s3,s2--o1);
(Length(s1) lt NatNum(Dec(2))) => CodACLen(s1,s2) = s2++s1;
( (NatNum(BStr(o1)++BStr(o2)) gt (7+(2+Dec(2)))) or (NatNum(BStr(o1)++BStr(o2)) lt (2+(5+Dec(5)))) ) =>
        CodACLen(o1+(o2+s3),s2) = s2++(o1+Octet(o2));
( (Length(s3) lt NatNum(BStr(o1)++BStr(o2))) and (NatNum(BStr(o1)++BStr(o2)) le (7+(2+Dec(2)))) and
        (NatNum(BStr(o1)++BStr(o2)) ge (2+(5+Dec(5)))) ) =>
        CodACLen(o1+(o2+s3),s2) = s2++s1;
( (Length(s3) eq NatNum(BStr(o1)++BStr(o2))) and (NatNum(BStr(o1)++BStr(o2)) le (7+(2+Dec(2)))) and
        (NatNum(BStr(o1)++BStr(o2)) ge (2+(5+Dec(5)))) ) =>
        CodACLen(o1+(o2+s3++s4),s2) = CodAC1(s3,s2++(o1+Octet(o2)),s4);
First(CorrConId(s1,First(s2))) => CodAC1(s1,s2,s3) =
        CodAC2(Third(CorrConId(s1,First(s2))),s2++Second(CorrConId(s1,First(s2))),s3);
not(First(CorrConId(s1,First(s2)))) => CodAC1(s1,s2,s3) = s2++Second(CorrConId(s1,First(s2)));
First(CorrCAlt(s1)) => CodAC2(s1,s2,s3) = CodAC3(Third(CorrCAlt(s1)),s2++Second(CorrCAlt(s1)),s3);
not(First(CorrCAlt(s1))) => CodAC2(s1,s2,s3) = s2++Second(CorrCAlt(s1));
First(CorrTokenItem(s1)) =>
        CodAC3(s1,s2,s3) = CodAC4(Third(CorrTokenItem(s1)),s2++Second(CorrTokenItem(s1)),s3);
not(First(CorrTokenItem(s1))) => CodAC3(s1,s2,s3) = s2++Second(CorrTokenItem(s1));
First(CorrRest(s1)) => CodAC4(s1,s2,s3) = ErrCodStr(s3);
not(First(CorrRest(s1))) => CodAC4(s1,s2,s3) = s2++Second(CorrRest(s1));

ofsort Err3Tuple
CorrTokenItem(s1) =
Err3Tuple( First(CorrPar(1+Dec(6),Dec(1),s1)), Second(CorrPar(1+Dec(6),Dec(1),s1)),
Third(CorrPar(1+Dec(6),Dec(1),s1)) );

(* RF SPDU valid encoding checks: *)
ofsort DatOctStr
Empty(s1) => ErrCodRF(s1,s2) = s2;
(o1 eq Dec(0)) => ErrCodRF(o1+s3,s2) = ErrCodStr(s3);
( (o1 lt (2+(5+Dec(5)))) and (Length(s3) lt o1) ) => ErrCodRF(o1+s3,s2) = s2++s1;
( (o1 lt (2+(5+Dec(5)))) and (Length(s3) eq o1) ) => ErrCodRF(o1+s3++s4,s2) = CodRF1(s3,s2--o1,s4);
(o1 eq (2+(5+Dec(5)))) => ErrCodRF(o1+s3,s2) = CodRFLen(s3,s2--o1);
(Length(s1) lt NatNum(Dec(2))) => CodRFLen(s1,s2) = s2++s1;
( (NatNum(BStr(o1)++BStr(o2)) gt (6+(6+Dec(5)))) or (NatNum(BStr(o1)++BStr(o2)) lt (2+(5+Dec(5)))) ) =>
        CodRFLen(o1+(o2+s3),s2) = s2++(o1+Octet(o2));
( (Length(s3) lt NatNum(BStr(o1)++BStr(o2))) and (NatNum(BStr(o1)++BStr(o2)) le (6+(6+Dec(5)))) and
        (NatNum(BStr(o1)++BStr(o2)) ge (2+(5+Dec(5)))) ) =>
        CodRFLen(o1+(o2+s3),s2) = s2++s1;
( (Length(s3) eq NatNum(BStr(o1)++BStr(o2))) and (NatNum(BStr(o1)++BStr(o2)) le (6+(6+Dec(5)))) and
        (NatNum(BStr(o1)++BStr(o2)) ge (2+(5+Dec(5)))) ) =>
        CodRFLen(o1+(o2+s3++s4),s2) = CodRF1(s3,s2++(o1+Octet(o2)),s4);
First(CorrConId(s1,First(s2))) => CodRF1(s1,s2,s3) =
        CodRF2(Third(CorrConId(s1,First(s2))),s2++Second(CorrConId(s1,First(s2))),s3);
not(First(CorrConId(s1,First(s2)))) => CodRF1(s1,s2,s3) = s2++Second(CorrConId(s1,First(s2)));
First(CorrTransDisc(s1)) => CodRF2(s1,s2,s3) =
        CodRF3(Third(CorrTransDisc(s1)),s2++Second(CorrTransDisc(s1)),s3);
```

```
not(First(CorrTransDisc(s1))) => CodRF2(s1,s2,s3) = s2++Second(CorrTransDisc(s1));
First(CorrSesReq(s1)) =>
      CodRF3(s1,s2,s3) = CodRF4(Third(CorrSesReqs1)),s2++Second(CorrSesReq(s1)),s3);
not(First(CorrSesReq(s1))) => CodRF3(s1,s2,s3) = s2++Second(CorrSesReq(s1));
First(CorrVersNum(s1)) =>
      CodRF4(s1,s2,s3) = CodRF5(Third(CorrVersNum(s1)),s2++Second(CorrVersNum(s1)),s3);
not(First(CorrVersNum(s1))) => CodRF4(s1,s2,s3) = s2++Second(CorrVersNum(s1));
First(CorrReasCode(s1)) => CodRF5(s1,s2,s3) = ErrCodStr(s3);
not(First(CorrReasCode(s1))) => CodRF5(s1,s2,s3) = s2++Second(CorrReasCode(s1));

ofsort Err3Tuple
CorrTransDisc(s1) = CorrPar(1+Dec(7),Dec(1),s1);
CorrSesReq(s1) = CorrPar(2+Dec(0),Dec(2),s1);
CorrVersNum(s1) = CorrPar(2+Dec(2),Dec(1),s1);
CorrReasCode(s1) = CorrParm(5+Dec(0),5+(1+Dec(3)),s1);

(* FN SPDU valid encoding checks: *)
ofsort DatOctStr
Empty(s1) => ErrCodFN(s1,s2) = s2;
(o1 eq Dec(0)) => ErrCodRF(o1+s3,s2) = ErrCodStr(s3);
( (o1 lt (2+(5+Dec(5)))) and (Length(s3) lt o1) ) => ErrCodFN(o1+s3,s2) = s2++s1;
( (o1 lt (2+(5+Dec(5)))) and (Length(s3) eq o1) ) => ErrCodFN(o1+s3++s4,s2) = CodFN1(s3,s2--o1,s4);
(o1 eq (2+(5+Dec(5)))) => ErrCodFN(o1+s3,s2) = CodFNLen(s3,s2--o1);
(Length(s1) lt NatNum(Dec(2))) => CodFNLen(s1,s2) = s2++s1;
( (NatNum(BStr(o1)++BStr(o2)) gt (5+(2+Dec(3)))) or (NatNum(BStr(o1)++BStr(o2)) lt (2+(5+Dec(5)))) ) =>
      CodFNLen(o1+(o2+s3),s2) = s2++(o1+Octet(o2));
( (Length(s3) lt NatNum(BStr(o1)++BStr(o2))) and (NatNum(BStr(o1)++BStr(o2)) le (5+(2+Dec(3)))) and
      (NatNum(BStr(o1)++BStr(o2)) ge (2+(5+Dec(5)))) ) =>
      CodFNLen(o1+(o2+s3),s2) = s2++s1;
( (Length(s3) eq NatNum(BStr(o1)++BStr(o2))) and (NatNum(BStr(o1)++BStr(o2)) le (5+(2+Dec(3)))) and
      (NatNum(BStr(o1)++BStr(o2)) ge (2+(5+Dec(5)))) ) =>
CodFNLen(o1+(o2+s3++s4),s2) = CodFN1(s3,s2++(o1+Octet(o2)),s4);
First(CorrTransDisc(s1,First(s2))) => CodFN1(s1,s2,s3) =
      CodFN2(Third(CorrTransDisc(s1,First(s2))),s2++Second(CorrTransDisc(s1,First(s2))),s3);
not(First(CorrTransDisc(s1,First(s2)))) => CodFN1(s1,s2,s3) = s2++Second(CorrTransDisc(s1,First(s2)));
First(CorrParm(1+(9+Dec(3)),5+(1+Dec(3)),s1)) => CodFN2(s1,s2,s3) = ErrCodStr(s3);
not(First(CorrParm(1+(9+Dec(3)),5+(1+Dec(3)),s1))) =>
      CodFN2(s1,s2,s3) = s2++Second(CorrParm(1+(9+Dec(3)),5+(1+Dec(3)),s1));

(* DN and NF SPDU valid encoding checks: *)
Empty(s1) => ErrCodDNorNF(s1,s2) = s2;
(o1 eq Dec(0)) => ErrCodDNorNF(o1+s3,s2) = ErrCodStr(s3);
( (o1 lt (2+(5+Dec(5)))) and (Length(s3) lt o1) ) => ErrCodDNorNF(o1+s3,s2) = s2++s1;
( (o1 lt (2+(5+Dec(5)))) and (Length(s3) eq o1) ) =>
      ErrCodDNorNF(o1+s3++s4,s2) = CodDNorNF1(s3,s2--o1,s4);
(o1 eq (2+(5+Dec(5)))) => ErrCodDNorNF(o1+s3,s2) = CodDNorNFLen(s3,s2--o1);
(Length(s1) lt NatNum(Dec(2))) => CodDNorNFLen(s1,s2) = s2++s1;
( (NatNum(BStr(o1)++BStr(o2)) gt (5+(1+Dec(6)))) or (NatNum(BStr(o1)++BStr(o2)) lt (2+(5+Dec(5)))) ) =>
      CodDNorNFLen(o1+(o2+s3),s2) = s2++(o1+Octet(o2));
( (Length(s3) lt NatNum(BStr(o1)++BStr(o2))) and (NatNum(BStr(o1)++BStr(o2)) le (5+(1+Dec(6)))) and
      (NatNum(BStr(o1)++BStr(o2)) ge (2+(5+Dec(5)))) ) =>
      CodDNorNFLen(o1+(o2+s3),s2) = s2++s1;
( (Length(s3) eq NatNum(BStr(o1)++BStr(o2))) and (NatNum(BStr(o1)++BStr(o2)) le (5+(1+Dec(6)))) and
      (NatNum(BStr(o1)++BStr(o2)) ge (2+(5+Dec(5)))) ) =>
      CodDNorNFLen(o1+(o2+s3++s4),s2) = CodDNorNF1(s3,s2++(o1+Octet(o2)),s4);
First(CorrParm(1+(9+Dec(3)),5+(1+Dec(3)),s1)) => CodDNorNF1(s1,s2,s3) = ErrCodStr(s3);
not(First(CorrParm(1+(9+Dec(3)),5+(1+Dec(3)),s1))) =>
      CodDNorNF1(s1,s2,s3) = s2++Second(CorrParm(1+(9+Dec(3)),5+(1+Dec(3)),s1));
```

```
(* AB and AI SPDU valid encoding checks: *)
Empty(s1) => ErrCodABorAI(s1,s2) = s2;
(o1 eq Dec(0)) => ErrCodABorAI(o1+s3,s2) = ErrCodStr(s3);
( (o1 le (2+Dec(7))) and (Length(s3) lt o1) ) => ErrCodABorAI(o1+s3,s2) = s2++s1;
( (o1 le (2+Dec(7))) and (Length(s3) eq o1) ) =>
        ErrCodABorAI(o1+s3++s4,s2) = CodABorAI(s3,s2--o1,s4);
(o1 gt (2+Dec(7))) => ErrCodABorAI(o1+s3,s2) = s2++Octet(o1);
(o1 eq (1+Dec(7))) => CodABorAI(o1+s3,s2,s4) = CodAB1(o1+s3,s2,s4);
(o1 ne (1+Dec(7))) => CodABorAI(o1+s3,s2,s4) = CodAI(o1+s3,s2,s4);
First(CorrTransDisc(s1)) =>
        CodAB1(s1,s2,s4) = CodAB2(Third(CorrTransDisc(s1)),s2++Second(CorrTransDisc(s1)),s4);
not(CorrTransDisc(s1)) => CodAB1(s1,s2,s4) = s2++Second(CorrTransDisc(s1));
First(CorrParm(4+Dec(9),Dec(9),s1)) => CodAB2(s1,s2,s4) =
        CodAB3(Third(CorrParm(4+Dec(9),Dec(9),s1)),s2++Second(CorrParm(4+Dec(9),Dec(9),s1)),s4);
not(First(CorrParm(4+Dec(9),Dec(9),s1))) =>
        CodAB2(s1,s2,s4) = s2++Second(CorrParm(4+Dec(9),Dec(9),s1));
First(CorrParm(1+(9+Dec(3)),Dec(9),s1)),Empty(Third(CorrParm(1+(9+Dec(3)),Dec(9),s1))) =>
        CodAB3(s1,s2,s4) = ErrCodStr(s4);
First(CorrParm(1+(9+Dec(3)),Dec(9),s1)),not(Empty(Third(CorrParm(1+(9+Dec(3)),Dec(9),s1)))) =>
        CodAB3(s1,s2,s4) = s2++s1;
not(First(CorrParm(1+(9+Dec(3)),Dec(9),s1))) =>
        CodAB3(s1,s2,s4) = s2++Second(CorrParm(1+(9+Dec(3)),Dec(9),s1));
First(CorrPar(5+Dec(0),Dec(1),s1)),Empty(Third(CorrPar(5+Dec(0),Dec(1),s1))) =>
        CodAI(s1,s2,s4) = ErrCodStr(s4);
First(CorrPar(5+Dec(0),Dec(1),s1)),not(Empty(Third(CorrPar(5+Dec(0),Dec(1),s1)))) =>
        CodAI(s1,s2,s4) = s2++s1;
not(First(CorrPar(5+Dec(0),Dec(1),s1))) =>
        CodAI(s1,s2,s4) = s2++Second(CorrPar(5+Dec(0),Dec(1),s1));

(* EX SPDU valid encoding checks: *)
Empty(s1) => ErrCodEX(s1,s2) = s2;
(o1 eq Dec(0) and (Length(s3) le (1+Dec(4))) => ErrCodEX(o1+s3,s2) = empty;
(o1 eq Dec(0) and (Length(s3) gt (1+Dec(4))) => ErrCodEX(o1+s3,s2) = o1+s3;
(o1 ne Dec(0)) => ErrCodEX(o1+s3,s2) = s2++Octet(o1);

(* TD SPDU valid encoding checks: *)
Empty(s1) => ErrCodTD(s1,s2) = s2;
First(CorrPar(2+Dec(5),Dec(1),s1)) => ErrCodTD(s1,s2) = empty;
not(First(CorrPar(2+Dec(5),Dec(1),s1))) =>
        ErrCodTD(s1,s2) = s2++Second(CorrPar(2+Dec(5),Dec(1),s1));

(* GT SPDU valid encoding checks: *)
Empty(s1) => ErrCodGT(s1,s2) = s2;
(o1 eq Dec(0)) => ErrCodGT(o1+s3,s2) = ErrCodStr(s3);
(o1 eq Dec(3)) => ErrCodGT(o1+s3,s2) = CodGT(s3,s2+Octet(o1));
(o1 ne Dec(0) and (o1 ne Dec(3))) => ErrCodGT(o1+s3,s2) = s2++o1;
First(CorrPar(1+Dec(6),Dec(1),s1)) => CodGT(s1,s2) = ErrCodStr(Third(CorrPar(1+Dec(6),Dec(1),s1)));
not(First(CorrPar(1+Dec(6),Dec(1),s1))) =>
        CodGT(s1,s2) = s2++Second(CorrPar(1+Dec(6),Dec(1),s1));

(* CD and CDA SPDU valid encoding checks: *)
Empty(s1) => ErrCodCDorCDA(s1,s2) = s2;
(o1 eq Dec(0)) => ErrCodCDorCDA(o1+s3,s2) = ErrCodStr(s3);
( (o1 lt (2+(5+Dec(5)))) and (Length(s3) lt o1) ) => ErrCodCDorCDA(o1+s3,s2) = s2++s1;
( (o1 lt (2+(5+Dec(5)))) and (Length(s3) eq o1) ) =>
        ErrCodCDorCDA(o1+s3++s4,s2) = CodCDorCDA1(s3,s2--o1,s4);
(o1 eq (2+(5+Dec(5)))) => ErrCodCDorCDA(o1+s3,s2) = CodCDorCDALen(s3,s2--o1);
(Length(s1) lt NatNum(Dec(2))) => CodCDorCDALen(s1,s2) = s2++s1;
( (NatNum(BStr(o1)++BStr(o2)) gt (5+(1+Dec(6)))) or (NatNum(BStr(o1)++BStr(o2)) lt (2+(5+Dec(5)))) ) =>
        CodCDorCDALen(o1+(o2+s3),s2) = s2++(o1+Octet(o2));
```

```
( (Length(s3) lt NatNum(BStr(o1)++BStr(o2))) and (NatNum(BStr(o1)++BStr(o2)) le (5+(1+Dec(6)))) and
      (NatNum(BStr(o1)++BStr(o2)) ge (2+(5+Dec(5)))) ) =>
      CodCDorCDALen(o1+(o2+s3),s2) = s2++s1;
( (Length(s3) eq NatNum(BStr(o1)++BStr(o2))) and (NatNum(BStr(o1)++BStr(o2)) le (5+(1+Dec(6)))) and
      (NatNum(BStr(o1)++BStr(o2)) ge (2+(5+Dec(5)))) ) =>
      CodCDorCDALen(o1+(o2+s3++s4),s2) = CodCDorCDA1(s3,s2++(o1+Octet(o2)),s4);
First(CorrParm(1+(9+Dec(3)),5+(1+Dec(2)),s1)),Empty(Third(CorrParm(1+(9+Dec(3)),5+(1+Dec(2)),s1)))
      => CodCDorCDA1(s1,s2,s3) = ErrCodStr(s3);
not(First(CorrParm(1+(9+Dec(3)),5+(1+Dec(2)),s1))) =>
      CodCDorCDA1(s1,s2,s3) = s2++Second(CorrParm(1+(9+Dec(3)),5+(1+Dec(2)),s1));

(* PT SPDU valid encoding checks: *)
Empty(s1) => ErrCodPT(s1,s2) = s2;
(o1 eq Dec(0)) => ErrCodPT(o1+s3,s2) = ErrCodStr(s3);
( (o1 lt (2+(5+Dec(5)))) and (Length(s3) lt o1) ) => ErrCodPT(o1+s3,s2) = s2++s1;
( (o1 lt (2+(5+Dec(5)))) and (Length(s3) eq o1) ) => ErrCodPT(o1+s3++s4,s2) = CodPT1(s3,s2--o1,s4);
(o1 eq (2+(5+Dec(5)))) => ErrCodPT(o1+s3,s2) = CodPTLen(s3,s2--o1);
(Length(s1) lt NatNum(Dec(2))) => CodPTLen(s1,s2) = s2++s1;
( (NatNum(BStr(o1)++BStr(o2)) gt (5+(2+Dec(3)))) or (NatNum(BStr(o1)++BStr(o2)) lt (2+(5+Dec(5)))) ) =>
      CodPTLen(o1+(o2+s3),s2) = s2++(o1+Octet(o2));
( (Length(s3) lt NatNum(BStr(o1)++BStr(o2))) and (NatNum(BStr(o1)++BStr(o2)) le (5+(2+Dec(3)))) and
      (NatNum(BStr(o1)++BStr(o2)) ge (2+(5+Dec(5)))) ) =>
      CodPTLen(o1+(o2+s3),s2) = s2++s1;
( (Length(s3) eq NatNum(BStr(o1)++BStr(o2))) and (NatNum(BStr(o1)++BStr(o2)) le (5+(2+Dec(3)))) and
      (NatNum(BStr(o1)++BStr(o2)) ge (2+(5+Dec(5)))) ) =>
      CodPTLen(o1+(o2+s3++s4),s2) = CodPT1(s3,s2++(o1+Octet(o2)),s4);
First(CorrTokenItem(s1,First(s2))) => CodPT1(s1,s2,s3) =
      CodPT2(Third(CorrTokenItem(s1,First(s2))),s2++Second(CorrTokenItem(s1,First(s2))),s3);
not(First(CorrTokenItem(s1,First(s2)))) => CodPT1(s1,s2,s3) = s2++Second(CorrTokenItem(s1,First(s2)));
First(CorrParm(1+(9+Dec(3)),5+(1+Dec(2)),s1)) => CodPT2(s1,s2,s3) = ErrCodStr(s3);
not(First(CorrParm(1+(9+Dec(3)),5+(1+Dec(2)),s1))) =>
      CodPT2(s1,s2,s3) = s2++Second(CorrParm(1+(9+Dec(3)),5+(1+Dec(3)),s1));


(* Below the auxiliary functions CorrPar, CorrParm, MCorrPar and MCorrParm are defined. *)

ofsort Err3Tuple
Empty(s1) => CorrPar(pi,len,s1) = Err3Tuple(true,s1,s1);
(o1 ne pi) => CorrPar(pi,len,o1+s2) = Err3Tuple(true,empty,o1+s2);
(* The first octet represents a PI value unequal to pi, meaning that the parameter is absent in the encoding.
As this parameter is not mandatory, the boolean value true and the complete string are returned. *)

( (o1 eq pi) and Empty(s2) ) => CorrPar(pi,len,o1+s2) = Err3Tuple(false,o1+s2,s2);
(* Only the PI field is present. This constitutes an invalid encoding, so the erroneous part is returned
together with the boolean value false. *)

( (o1 eq pi) and (o2 ne len) ) =>
      CorrPar(pi,len,o1+(o2+s2)) = Err3Tuple(false,o1+Octet(o2),s2);
(* If the parameter is present, the length of the encoding of its value should be equal to len. This is not the
case: a boolean value false is returned, together with the first two octets (pi and li fields). *)

( (o1 eq pi) and (o2 eq len) and (Length(s2) lt NatNum(len)) ) =>
      CorrPar(pi,len,o1+(o2+s2)) = Err3Tuple(false,o1+(o2+s2),empty);
(* PI and LI fields are correct, but the encoding of the parameter value is too short. *)

((o1 eq pi) and (o2 eq len) and (Length(s2) eq NatNum(len)) and Empty(CheckBits(pi,s2))) =>
      CorrPar(pi,len,o1+(o2+s2++s3)) = Err3Tuple(true,o1+(o2+s2),s3);
(* PI and LI fields are correct and the value encoding is correct. *)
```

```
( (o1 eq pi) and (o2 eq len) and (Length(s2) eq NatNum(len)) and
    not(Empty(CheckBits(pi,s2))) ) =>
        CorrPar(pi,len,o1+(o2+s2++s3)) = Err3Tuple(false,o1+(o2+CheckBits(pi,s2)),s3);
(* PI and LI fields are correct but the value encoding is not according to additional, parameter-type specific,
encoding rules. *)

Empty(s1) => CorrParm(pi,len,s1) = Err3Tuple(true,s1,s1);
(o1 ne pi) => CorrParm(pi,len,o1+s2) = Err3Tuple(true,empty,o1+s2);
( (o1 eq pi) and Empty(s2) ) => CorrParm(pi,len,o1+s2) = Err3Tuple(false,o1+s2,s2);
( (o1 eq pi) and (o2 ge len) ) =>
    CorrParm(pi,len,o1+(o2+s2)) = Err3Tuple(false,o1+Octet(o2),s2);
( (o1 eq pi) and (o2 lt (2+(5+Dec(5)))) and (o2 le len) and (Length(s2) lt NatNum(BStr(o2))) ) =>
    CorrParm(pi,len,o1+(o2+s2)) = Err3Tuple(false,o1+(o2+s2),empty);
( (o1 eq pi) and (o2 lt (2+(5+Dec(5)))) and (o2 le len) and
    (Length(s2) eq NatNum(BStr(o2))) and Empty(CheckBits(pi,s2)) ) =>
        CorrParm(pi,len,o1+(o2+s2++s3)) = Err3Tuple(true,o1+(o2+s2),s3);
( (o1 eq pi) and (o2 lt (2+(5+Dec(5)))) and (o2 le len) and
    (Length(s2) eq NatNum(BStr(o2))) and not(Empty(CheckBits(pi,s2))) ) =>
        CorrParm(pi,len,o1+(o2+s2++s3) = Err3Tuple(false,o1+(o2+CheckBits(pi,s2)),s3);
( (o1 eq pi) and (len ge (2+(5+Dec(5)))) and
    (o2 eq (2+(5+Dec(5)))) and (Length(s2) lt NatNum(Dec(2))) ) =>
        CorrParm(pi,len,o1+(o2+s2)) = Err3Tuple(false,o1+(o2+s2),empty);
( (o1 eq pi) and (len ge (2+(5+Dec(5)))) and
    (o2 eq (2+(5+Dec(5)))) and (Length(s2) ge NatNum(Dec(2))) ) =>
        CorrParm(pi,len,o1+(o2+s2)) = CorrParmL(pi,len,s2,o1+Octet(o2));

(* CorrParmL is analogous to CorrParm, except that the former deals with LI fields representing values
greater than 255 octets. These values have to be represented using three octets. *)

( (NatNum(BStr(o1)++BStr(o2)) lt NatNum(2+(5+Dec(5)))) or
    (NatNum(len) lt NatNum(BStr(o1)++BStr(o2))) ) =>
        CorrParmL(pi,len,o1+(o2+s3),s2) = Err3Tuple(false,s2++(o1+Octet(o2)),s3);
( (NatNum(li) eq NatNum(BStr(o1)++BStr(o2))) and
    (NatNum(BStr(o1)++BStr(o2)) ge NatNum(2+(5+Dec(5)))) and
    (NatNum(len) ge NatNum(BStr(o1)++BStr(o2))) and
    (Length(s3) lt NatNum(BStr(o1)++BStr(o2))) ) =>
        CorrParmL(pi,len,o1+(o2+s3),s2) =  Err3Tuple(false,s2++o1+(o2+s3),empty);
( (NatNum(BStr(o1)++BStr(o2)) ge NatNum(2+(5+Dec(5)))) and
    (NatNum(len) ge NatNum(BStr(o1)++BStr(o2))) and
    (Length(s3) eq NatNum(BStr(o1)++BStr(o2))) and Empty(CheckBits(pi,s3)) ) =>
        CorrParmL(pi,len,o1+(o2+s3++s4),s2) = Err3Tuple(true,s2++(o1+(o2+s3)),s4);
( (NatNum(li) eq NatNum(BStr(o1)++BStr(o2))) and
    (NatNum(BStr(o1)++BStr(o2)) ge NatNum(2+(5+Dec(5)))) and
    (NatNum(len) ge NatNum(BStr(o1)++BStr(o2))) and
    (Length(s3) eq NatNum(BStr(o1)++BStr(o2))) and not(Empty(CheckBits(pi,s3))) ) =>
        CorrParmL(pi,len,o1+(o2+s3++s4),s2) = Err3Tuple(false,s2++(o1+(o2+CheckBits(pi,s3))),s4);

Empty(s1) => MCorrPar(pi,len,s1) = Err3Tuple(false,s1,s1);
(o1 ne pi) => MCorrPar(pi,len,o1+s2) = Err3Tuple(false,Octet(o1),s2);
(o1 eq pi) => MCorrPar(pi,len,Octet(o1)) = Err3Tuple(false,Octet(o1),empty);
( (o1 eq pi) and (o2 ne len) ) =>
    MCorrPar(pi,len,o1+(o2+s2)) = Err3Tuple(false,o1+Octet(o2),s2);
( (o1 eq pi) and (o2 eq len) and (Length(s2) lt NatNum(len)) ) =>
    MCorrPar(pi,len,o1+(o2+s2)) = Err3Tuple(false,o1+(o2+s2),empty);
( (o1 eq pi) and (o2 eq len) and (Length(s2) eq NatNum(len)) and Empty(CheckBits(pi,s2)) ) =>
    MCorrPar(pi,len,o1+(o2+s2++s3)) = Err3Tuple(true,o1+(o2+s2),s3);
( (o1 eq pi) and (o2 eq len) and (Length(s2) eq NatNum(len)) and
    not(Empty(CheckBits(pi,s2))) ) =>
        MCorrPar(pi,len,o1+(o2+s2++s3)) = Err3Tuple(false,o1+(o2+CheckBits(pi,s2)),s3);
```

```
(* In the definition of MCorrParm below, it is checked that a parameter is present and its length, in octets,
may not exceed the length indicated by the argument len. *)

Empty(s1) => MCorrParm(pi,len,s1) = Err3Tuple(false,s1,s1);
(o1 ne pi) => MCorrParm(pi,len,o1+s2) = Err3Tuple(false,Octet(o1),s2);
(o1 eq pi) => MCorrParm(pi,len,Octet(o1)) = Err3Tuple(false,Octet(o1),empty);
( (o1 eq pi) and ((o2 ge len) or (o2 eq Dec(0))) ) =>
      MCorrParm(pi,len,o1+(o2+s2)) = Err3Tuple(false,o1+Octet(o2),s2);
( (o1 eq pi) and ((o2 gt Dec(0)) and (o2 le len)) and (Length(s2) lt o2) ) =>
      MCorrParm(pi,len,o1+(o2+s2)) = Err3Tuple(false,o1+(o2+s2),empty);
( (o1 eq pi) and ((o2 gt Dec(0)) and (o2 le len)) and (Length(s2) eq o2)
      and Empty(CheckBits(pi,s2)) ) =>
            MCorrParm(pi,len,o1+(o2+s2++s3)) = Err3Tuple(true,o1+(o2+s2),s3);
( (o1 eq pi) and ((o2 gt Dec(0)) and (o2 le len)) and (Length(s2) eq o2) and
      not(Empty(CheckBits(pi,s2))) ) =>
            MCorrParm(pi,len,o1+(o2+s2++s3)) = Err3Tuple(false,o1+(o2+CheckBits(pi,s2)),s3);

ofsort DatOctStr
( (pi eq (1+Dec(9))) and ((o1 eq Dec(1)) or (o1 eq Dec(0))) ) => CheckBits(pi,Octet(o1)) = empty;
( (pi eq (1+Dec(9))) and (o1 ne Dec(1)) and (o1 ne Dec(0)) ) => CheckBits(pi,Octet(o1)) = s1;
( (pi eq (2+Dec(2))) and (o1 eq Dec(1)) ) => CheckBits(pi,Octet(o1)) = empty;
( (pi eq (2+Dec(2))) and (o1 ne Dec(1)) ) => CheckBits(pi,Octet(o1)) = Octet(o1);
( (pi eq (2+Dec(6))) and No11(Bstr(o1)) ) => CheckBits(pi,Octet(o1)) = empty;
( (pi eq (2+Dec(6))) and not(No11(Bstr(o1))) ) => CheckBits(pi,Octet(o1)) = Octet(o1);
( (pi eq (2+Dec(0))) and (o1 le Dec(7)) ) => CheckBits(pi,o1+Octet(o2)) = empty;
(pi eq (2+Dec(3))) => CheckBits(pi,s1) = SPSNOctets(s1);
( (pi eq (2+Dec(0))) and (o1 gt Dec(7)) ) => CheckBits(pi,o1+Octet(o2)) = Octet(o1);

ofsort Bool
No11(b1+(b2+bstr2)) = ((b1 ne 1) or (b2 ne 1)) and No11(bstr2);

ofsort DatOctStr
( (Length(bstr1++bstr2) eq NatNum(Dec(8))) and (Length(bstr1) eq NatNum(Dec(4))) and
      (bstr1 ne (0+(0+(1+Bit(1))))) ) =>
            SPSNOctets(OStr(bstr1++bstr2)+s2) = Octet(OStr(bstr1++bstr2));
( (Length(bstr1++bstr2) eq NatNum(Dec(8))) and (Length(bstr1) eq NatNum(Dec(4))) and
      (bstr1 eq (0+(0+(1+Bit(1))))) ) =>
            SPSNOctets(OStr(bstr1++bstr2)+s2) = SPSNOct2(s2,Octet(OStr(bstr1++bstr2)));
SPSNOct2(empty,s1) = empty;
( (Length(bstr1++bstr2) eq NatNum(Dec(8))) and (Length(bstr1) eq NatNum(Dec(4))) and
      (bstr1 ne (0+(0+(1+Bit(1))))) ) =>
            SPSNOct2(OStr(bstr1++bstr2)+s3,s2) = s2++Octet(OStr(bstr1++bstr2));
( (Length(bstr1++bstr2) eq NatNum(Dec(8))) and (Length(bstr1) eq NatNum(Dec(4))) and
      (bstr1 eq (0+(0+(1+Bit(1))))) ) =>
            SPSNOct2(OStr(bstr1++bstr2)+s3,s2) = SPSNOct2(s3,s2++Octet(OStr(bstr1++bstr2)));
endtype
(*-----------------------------------------------------------------------------------------------------------
```

9.1.4.4 Transformations between SPDUs, TSPs and SSPs

Type *Relations* defines some functions related to the mappings between SSPs and SPDUs, and between SPDUs and TSPs:

- *IsSSPOf* tests whether a given SSP can be derived from - i.e., is an indication or a confirm corresponding to - a given SPDU;
- *IsPDUOf* tests whether a given SPDU can be derived from a given - request or response - SSP;
- *IsEncOf* tests whether a given octet string (TSDU) is a valid encoding of a given SPDU;
- *Decode* yields the SPDU(s) that correspond(s) to a given octet string representing a TSDU. If a given octet string corresponds to both an AIA and AA SPDU (these two SPDUs have the same

encoding) then *Decode* yields the AA SPDU; another function, *Decode2*, is defined to be equivalent to *Decode*, except that it yields the AIA SPDU in the latter case.

```
-----------------------------------------------------------------------------------------------------------*)
type Relations is ASPDU,ASPDUs2,Encodings,ErrCoding
opns
_IsSSPOf_: SSP,ASPDU -> Bool                    _IsPDUOf_: ASPDU,SSP -> Bool
_IsEncOf_: DatOctStr,ASPDU -> Bool              Decode2,Decode: DatOctStr -> ASPDUs
eqns forall pd:ASPDU, len:Nat, str,str1,str2,str3,str4,str5,str6,str7:DatOctStr, ssp:SSP, spr:SPReference,
       cait:CAItem, sreq,fus:SFUs, sa1,sa2,sa3,sa4:SAddress, sdat1,sdat2,sdat:SData, sqos:SQOS,
       srdq:SFUs, srf,srf1,srf2:SCRef, sn,sn1,sn2:SSPSN, tkass,tkass1,tkass2:STsAss,
       stok,stok1,stok2:STokens, tdpar:TDISPar, vs,reas:Nat, scres:SConResult, encit:EncIt, rpv:DatOctStr,
       abreas:SPAbReason, o:Octet, pds:ASPDUs, actid1,actid2,actid3,actid4:SActId,
       stype,stype1,stype2:SSyncType, rtype1,rtype2:SResyncType, srel:SRELResult,
       uereas:SUExcReason, abreas:SPABReason, exreas:SPExcReason
ofsort Bool
(* Each parameter (group) is encoded separately. The resulting strings are then concatenated and the
length of the resulting string is determined and its value encoded. *)

IsCN(pd) => (Repr(1+Dec(3))+(Repr(len)+str1++str2++str3++str4++str5++str6)) IsEncOf pd =
       (str1 RepConId pd) and (str2 RepCAItem pd) and (str3 RepReq pd) and
       (str4 RepSAP1 pd) and (str5 RepSAP2 pd) and (str6 RepData pd) and
       (len eq (Length(str1)+Length(str2)+Length(str3)+Length(str4)+Length(str5)+Length(str6)));
( IsCN(pd) and (o ne (1+Dec(3))) ) => Octet(o) IsEncOf pd = false;
IsCN(pd) => empty IsEncOf pd = false;

(* Below are the equations defining the Decode function. ErrCodStr returns an empty string if its argument
is a correct encoding of a (sequence of) SPDU(s). Otherwise the badly encoded part of the SPDU
encoding, up to and including the first erroneous octet, is returned. *)

( Empty(ErrCodStr(str1++str2)) and not(IsAIA(pd)) ) =>
       Decode(str1++str2) eq (pd+pds) = (str1 IsEncOf pd) and (pds eq Decode2(str2));
Decode(empty) eq NULL = true;
not(Empty(ErrCodStr(str))) => Decode(str) eq ASPDUs( DUM(ErrCodStr(str)) ) = true;

(* Function Decode is defined in terms of the IsEncOf relation. This is possible because an encoding should
be unambiguous, i.e. it should be possible to determine a unique SPDU from a given encoding. If the octet
string is an invalid encoding, Decode returns a dummy SPDU carrying the erroneous (part of the) string.
Two "decode" functions, Decode and Decode2, have been defined because the AIA and AA SPDUs have
exactly the same encoding. They are distinguished by the fact that the AA SPDU is never concatenated
with other SPDUs whereas the AIA SPDU is always concatenated with a PT SPDU. *)

( Empty(ErrCodStr(str1++str2)) and not(IsAA(pd)) ) =>
       Decode2(str1++str2) eq (pd+pds) = (str1 IsEncOf pd) and (pds eq Decode2(str2));
Decode2(empty) eq NULL = true;
not(Empty(ErrCodStr(str))) => Decode2(str) eq ASPDUs( DUM(str1) ) = true;

( IsCN(pd) xor IsSCONreq(ssp) ) => pd IsPDUOf ssp = false;
CN(spr,cait,fus,sa1,sa2,sdat1) IsPDUOf SCONreq(srf,sa3,sa4,sqos,sreq,sn,tkass,sdat2) =
       ((CgUserReference(spr) eq CgUserRef(srf)) and (CommonReference(spr) eq CommonRef(srf)) and
       (AdditionalRef(spr) eq AdditionalRef(srf)) and (sa1 eq sa3) and (sa2 eq sa4) and (fus eq sreq) and
       (SSPSN(cait) eq sn) and (STsAss(cait) eq tkass) and (sdat1 eq sdat2));
( (IsAC(pd) or IsRF(pd)) xor IsSCONrsp(ssp) ) => pd IsPDUOf ssp = false;
AC(spr,cait,stok,fus,sa1,sa2,sdat1) IsPDUOf SCONrsp(srf,sa3,scres,sqos,sreq,sn,tkass,sdat2) =
       ( IsAccept(scres) and (CgUserReference(spr) eq CdUserRef(srf)) and
       (CommonReference(spr) eq CommonRef(srf)) and (AdditionalRef(spr) eq AdditionalRef(srf)) and
       (fus eq sreq) and (sdat1 eq sdat2) and (STsAss(cait) eq tkass) and (SSPSN(cait) eq sn) );
```

```
RF(spr,tdpar,fus,vs,reas,sdat1) IsPDUOf SCONrsp(srf,sa3,scres,sqos,sreq,sn,tkass,sdat2) =
        ( not(IsAccept(scres)) and
        (CdUserReference(spr) eq CdUserReference(srf)) and
        (CommonReference(spr) eq CommonReference(srf))
        (AdditionalRef(spr) eq AdditionalRef(srf)) and
        ( (reas ne Succ(Succ(0))) implies ((fus eq {}) and Empty(sdat1)) ) and
        ( (reas eq Succ(0)) implies (scres eq ureject2) )
        ( (reas eq Succ(Succ(0))) implies ((scres eq ureject3) and (fus eq sreq) and (sdat1 eq sdat2)) ) );
(IsFN(pd) xor IsSRELreq(ssp)) => pd IsPDUOf ssp = false;
FN(tdpar,sdat1) IsPDUOf SRELreq(sdat2) = (sdat1 eq sdat2);
( (IsDN(pd) or IsNF(pd)) xor IsSRELrsp(ssp) ) => pd IsPDUOf ssp = false;
DN(sdat1) IsPDUOf SRELrsp(srel,sdat2) = ( IsAffirm(srel) and (sdat1 eq sdat2) );
NF(sdat1) IsPDUOf SRELrsp(srel,sdat2) = ( IsNegative(srel) and (sdat1 eq sdat2) );
(IsAB(pd) xor IsSUABreq(ssp)) => pd IsPDUOf ssp = false;

AB(tdpar,reas,rpv,sdat1) IsPDUOf SUABreq(sdat2) =
        ( (reas eq Succ(0)) and (sdat1 eq sdat2) and Empty(rpv) );
(* The values 1, 2 and 3 of variable reas indicate respectively "user abort", "protocol error" and "no reason".
*)

(IsEX(pd) xor IsSEXreq(ssp)) => pd IsPDUOf ssp = false;
EX(sdat1) IsPDUOf SEXreq(sdat2) = (sdat1 eq sdat2);
(IsCD(pd) xor IsSCDreq(ssp)) => pd IsPDUOf ssp = false;
CD(sdat1) IsPDUOf SCDreq(sdat2) = (sdat1 eq sdat2);
(IsCDA(pd) xor IsSCDrsp(ssp)) => pd IsPDUOf ssp = false;
CDA(sdat1) IsPDUOf SCDrsp(sdat2) = (sdat1 eq sdat2);
(IsGT(pd) xor IsSGTreq(ssp)) => pd IsPDUOf ssp = false;
GT(stok1) IsPDUOf SGTreq(stok2) = (stok1 eq stok2);
(IsPT(pd) xor IsSPTreq(ssp)) => pd IsPDUOf ssp = false;
PT(stok1,sdat1) IsPDUOf SPTreq(stok2,sdat2) = ( (stok1 eq stok2) and (sdat1 eq sdat2) );
(IsGTC(pd) xor IsSCGreq(ssp)) => pd IsPDUOf ssp = false;
GTC IsPDUOf SCGreq = true;
(IsMIP(pd) xor IsSSYMNreq(ssp)) => pd IsPDUOf ssp = false;
MIP(stype1,sn1,sdat1) IsPDUOf SSYMNreq(stype2,sn2,sdat2) =
 ( (stype1 eq stype2) and (sn1 eq sn2) and (sdat1 eq sdat2) );
(IsMIA(pd) xor IsSSYMNrsp(ssp)) => pd IsPDUOf ssp = false;
MIA(sn1,sdat1) IsPDUOf SSYMNrsp(sn2,sdat2) = ( (sn1 eq sn2) and (sdat1 eq sdat2) );
(IsMAP(pd) xor IsSSYMJreq(ssp)) => pd IsPDUOf ssp = false;
MAP(sn1,sdat1) IsPDUOf SSYMJreq(sn2,sdat2) = ( (sn1 eq sn2) and (sdat1 eq sdat2) );
(IsMAA(pd) xor IsSSYMJrsp(ssp)) => pd IsPDUOf ssp = false;
MAA(sn,sdat1) IsPDUOf SSYMJrsp(sdat2) = (sdat1 = sdat2);
(IsRS(pd) xor IsSRSYNreq(ssp)) => pd IsPDUOf ssp = false;
RS(tkass1,rtype1,sn1,sdat1) IsPDUOf SRSYNreq(rtype2,sn2,tkass2,sdat2) =
        ( (tkass1 eq tkass2) and (rtype1 eq rtype2) and
        ( (rtype2 ne a) implies (sn1 eq sn2) ) and (sdat1 eq sdat2) );
(IsRA(pd) xor IsSRSYNrsp(ssp)) => pd IsPDUOf ssp = false;
RA(tkass1,sn1,sdat1) IsPDUOf SRSYNrsp(sn2,tkass2,sdat2) =
        ( (sn1 eq sn2) and (tkass1 eq tkass2) and (sdat1 eq sdat2) );
(IsED(pd) xor IsSUERreq(ssp)) => pd IsPDUOf ssp = false;
ED(reas,sdat1) IsPDUOf SUERreq(uereas,sdat2) =
        ( (sdat1 eq sdat2) and ( ((reas eq 0) and (uereas eq absent)) or
        ((reas eq NatNum(1)) and (uereas eq receiptProblem)) or
        ((reas eq NatNum(2)) and (uereas eq localError)) or
        ((reas eq NatNum(3)) and (uereas eq sequenceError)) or
        ((reas eq NatNum(4)) and (uereas eq demandDk)) or
        ((reas eq NatNum(5)) and (uereas eq unrecoverableError)) or
        ((reas eq NatNum(6)) and (uereas eq nonSpecificError)) ) );
(IsAS(pd) xor IsSACTSreq(ssp)) => pd IsPDUOf ssp = false;
AS(actid1,sdat1) IsPDUOf SACTSreq(actid2,sdat2) = ( (actid1 eq actid2) and (sdat1 eq sdat2) );
(IsAR(pd) xor IsSACTRreq(ssp)) => pd IsPDUOf ssp = false;
```

```
AR(srf1,actid1,sn1,actid2,sdat1) IsPDUOf SACTRreq(actid3,actid4,sn2,srf2,sdat2) =
        ( (srf1 eq srf2) and (actid1 eq actid3) and
        (actid2 eq actid4) and (sn1 eq sn2) and (sdat1 eq sdat2) );
(IsAI(pd) xor IsSACTIreq(ssp)) => pd IsPDUOf ssp = false;
AI(reas) IsPDUOf SACTIreq(uereas) =
        ( ((reas eq 0) and (uereas eq absent)) or
        ((reas eq Succ(0)) and (uereas eq receiptProblem)) or
        ((reas eq Succ(Succ(0))) and (uereas eq localError)) or
        ((reas eq NatNum(Dec(3))) and (uereas eq sequenceError)) or
        ((reas eq NatNum(Dec(4))) and (uereas eq demandDk)) or
        ((reas eq NatNum(Dec(5))) and (uereas eq unrecoverableError)) or
        ((reas eq NatNum(Dec(6))) and (uereas eq nonSpecificError)) );
(IsAD(pd) xor IsSACTDreq(ssp)) => pd IsPDUOf ssp = false;
AD(reas) IsPDUOf SACTDreq(uereas) =
        ( ((reas eq 0) and (uereas eq absent)) and
        ((reas eq Succ(0)) and (uereas eq receiptProblem)) and
        ((reas eq Succ(Succ(0))) and (uereas eq localError)) and
        ((reas eq NatNum(Dec(3))) and (uereas eq sequenceError)) and
        ((reas eq NatNum(Dec(4))) and (uereas eq demandDk)) and
        ((reas eq NatNum(Dec(5))) and (uereas eq unrecoverableError)) and
        ((reas eq NatNum(Dec(6))) and (uereas eq nonSpecificError)) );
(IsAIA(pd) xor IsSACTIrsp(ssp)) => pd IsPDUOf ssp = false;
AIA IsPDUOf SACTIrsp = true;
(IsADA(pd) xor IsSACTDrsp(ssp)) => pd IsPDUOf ssp = false;
ADA IsPDUOf SACTDrsp = true;
(IsAE(pd) xor IsSACTEreq(ssp)) => pd IsPDUOf ssp = false;
AE(sn1,sdat1) IsPDUOf SACTEreq(sn2,sdat2) = ( (sn1 eq sn2) and (sdat1 eq sdat2) );
(IsAEA(pd) xor IsSACTErsp(ssp)) => pd IsPDUOf ssp = false;
AEA(sn,sdat1) IsPDUOf SACTErsp(sdat2) = (sdat1 eq sdat2);

IsSCONind(ssp) xor IsCN(pd) => ssp IsSSPOf pd = false;
SCONind(srf,sa1,sa2,sqos,sreq,sn,tkass,sdat1) IsSSPOf CN(spr,cait,fus,sa3,sa4,sdat2) =
        ( (CgUserRef(srf) eq CgUserReference(spr)) and (CommonRef(srf) eq CommonReference(spr)) and
        (AdditionalRef(srf) eq AdditionalRef(spr)) and (sreq eq fus) and (sdat1 eq sdat2) and
        (sn eq SSPSN(cait)) and (tkass eq STsAss(cait)) and ((sa3 ne absent) implies (sa1 eq sa3)) and
        ((sa4 ne absent) implies (sa2 eq sa4)) );

IsSCONcnf(ssp) xor (IsAC(pd) or IsRF(pd)) => ssp IsSSPOf pd = false;
SCONcnf(srf,sa1,scres,sqos,sreq,sn,tkass,sdat1) IsSSPOf AC(spr,cait,stok,fus,sa2,sa3,sdat2) =
        ( (CdUserRef(srf) eq CgUserReference(spr)) and (CommonRef(srf) eq CommonReference(spr)) and
        (AdditionalRef(srf) eq AdditionalRef(spr)) and (sreq eq fus) and (sdat1 eq sdat2) and
        (sn eq SSPSN(cait)) and (tkass eq STsAss(cait)) and ((sa3 ne absent) implies (sa1 eq sa3)) and
        IsAccept(scres) );
SCONcnf(srf,sa1,scres,sqos,sreq,sn,tkass,sdat1) IsSSPOf RF(spr,tdpar,fus,vs,reas,sdat2) =
        ( (CdUserRef(srf) eq CgUserReference(spr)) and (CommonRef(srf) eq CommonReference(spr)) and
        (AdditionalRef(srf) eq AdditionalRef(spr)) and (sreq eq fus) and (sdat1 eq sdat2) and
        (sn eq SSPSN(cait)) and (tkass eq STsAss(cait)) and ((sa3 ne absent) implies (sa1 eq sa3)) and
        (h(scres) eq reas) );

(IsSUABind(ssp) or IsSPABind(ssp)) xor IsAB(pd) => ssp IsSSPOf pd = false;
SUABind(sdat1) IsSSPOf  AB(tdpar,reas,rpv,sdat2) = ( (reas eq Succ(0)) and (sdat1 eq sdat2) );
SPABind(abreas) IsSSPOf  AB(tdpar,reas,rpv,sdat2) =
        ( ((reas eq NatNum(Dec(3))) and (abreas eq undefined)) or
        ((reas eq NatNum(Dec(2))) and (abreas eq protocolError)) );
(IsSRELind(ssp) xor IsFN(pd)) => ssp IsSSPOf pd = false;
SRELind(sdat1) IsSSPOf FN(tdpar,sdat2) = (sdat1 eq sdat2);
(IsSRELcnf(ssp) xor (IsDN(pd) or IsNF(pd))) => ssp IsSSPOf pd = false;
SRELcnf(srel,sdat1) IsSSPOf DN(sdat2) = (IsAffirm(srel) and (sdat1 eq sdat2) );
SRELcnf(srel,sdat1) IsSSPOf NF(sdat2) = ( IsNegative(srel) and (sdat1 eq sdat2) );
(IsSEXind(ssp) xor IsEX(pd)) => ssp IsSSPOf pd = false;
SEXind(sdat1) IsSSPOf EX(sdat2) = (sdat1 eq sdat2);
```

```
(IsSCDind(ssp) xor IsCD(pd)) => ssp IsSSPOf pd = false;
SCDind(sdat1) IsSSPOf CD(sdat2) = (sdat1 eq sdat2);
(IsSCDcnf(ssp) xor IsCDA(pd)) => ssp IsSSPOf pd = false;
SCDcnf(sdat1) IsSSPOf CDA(sdat2) = (sdat1 eq sdat2);
(IsSGTind(ssp) xor IsGT(pd)) => ssp IsSSPOf pd = false;
SGTind(stok1) IsSSPOf GT(stok2) = (stok1 eq stok2);
(IsSPTind(ssp) xor IsPT(pd)) => ssp IsSSPOf pd = false;
SPTind(stok1,sdat1) IsSSPOf PT(stok2,sdat2) = ( (stok1 eq stok2) and (sdat1 eq sdat2) );
(IsCGind(ssp) xor IsGTC(pd)) => ssp IsSSPOf pd = false;
CGind IsSSPOf GTC = true;
(IsSSYMNind(ssp) xor IsMIP(pd)) => ssp IsSSPOf pd = false;
SSYMNind(stype1,sn1,sdat1) IsSSPOf MIP(stype2,sn2,sdat2) =
        ( (stype1 eq stype2) and (sn1 eq sn2) and (sdat1 eq sdat2) );
(IsSSYMNcnf(ssp) xor IsMIA(pd)) => ssp IsSSPOf pd = false;
SSYMNcnf(sn1,sdat1) IsSSPOf MIA(sn2,sdat2) = ( (sn1 eq sn2) and (sdat1 eq sdat2) );
(IsSSYMJind(ssp) xor IsMAP(pd)) => ssp IsSSPOf pd = false;
SSYMJind(sn1,sdat1) IsSSPOf MAP(sn2,sdat2) = ( (sn1 eq sn2) and (sdat1 eq sdat2) );
(IsSSYMJcnf(ssp) xor (IsMAA(pd) or IsAEA(pd))) => ssp IsSSPOf pd = false;
SSYMJcnf(sdat1) IsSSPOf MAA(sn,sdat2) = (sdat1 eq sdat2);
SSYMJcnf(sdat1) IsSSPOf AEA(sn,sdat2) = (sdat1 eq sdat2);
(IsSRSYNind(ssp) xor IsRS(pd)) => ssp IsSSPOf pd = false;
SRSYNind(rtype1,sn1,tkass1,sdat1) IsSSPOf RS(tkass2,rtype2,sn2,sdat2) = (rtype1 eq rtype2) and
        (tkass1 eq tkass2) and (sdat1 eq sdat2) and ((rtype2 ne a) implies (sn1 eq sn2));
(IsSRSYNcnf(ssp) xor IsRA(pd)) => ssp IsSSPOf pd = false;
SRSYNcnf(sn1,tkass1,sdat1) IsSSPOf RA(tkass2,sn2,sdat2) =
        ( (sn1 eq sn2) and (tkass1 eq tkass2) and (sdat1 eq sdat2) );
(IsSPERind(ssp) xor IsER(pd)) => ssp IsSSPOf pd = false;
SUERind(uereas,sdat1) IsSSPOf ED(reas,sdat2) =
        ( (sdat1 eq sdat2) and ( ((reas eq 0) and (uereas eq absent)) or
        ((reas eq Succ(0)) and (uereas eq receiptProblem)) or
        ((reas eq Succ(Succ(0))) and (uereas eq localError)) or
        ((reas eq NatNum(Dec(3))) and (uereas eq sequenceError)) or
        ((reas eq NatNum(Dec(4))) and (uereas eq demandDk)) or
        ((reas eq NatNum(Dec(5))) and (uereas eq unrecoverableError)) or
        ((reas eq NatNum(Dec(6))) and (uereas eq nonSpecificError)) ));
(IsSPERind(ssp) xor IsER(pd)) => ssp IsSSPOf pd = false;
SPERind(exreas) IsSSPOf ER(rpv) = (exreas eq protocolError);
(* It is assumed that the value "non-specific error" is never passed to the SS-user. *)

(IsSACTSind(ssp) xor IsAS(pd)) => ssp IsSSPOf pd = false;
SACTSind(actid1,sdat1) IsSSPOf AS(actid2,sdat2) = ( (actid1 eq actid2) and (sdat1 eq sdat2) );
(IsSACTRind(ssp) xor IsAR(pd)) => ssp IsSSPOf pd = false;
SACTRind(actid1,actid2,sn1,srf1,sdat1) IsSSPOf AR(srf2,actid3,sn2,actid4,sdat2) =
        ( (actid1 eq actid3) and (actid2 eq actid4) and
        (sn1 eq sn2) and (srf1 eq srf2) and (sdat1 eq sdat2) );
(IsSACTIind(ssp) xor IsAI(pd)) => ssp IsSSPOf pd = false;
SACTIind(uereas) IsSSPOf AI(reas) = (Conv(uereas) eq reas);
(IsSACTDind(ssp) xor IsAD(pd)) => ssp IsSSPOf pd = false;
SACTDind(uereas) IsSSPOf AD(reas) = (Conv(uereas) eq reas);
(IsSACTIcnf(ssp) xor IsAIA(pd)) => ssp IsSSPOf pd = false;
SACTIcnf IsSSPOf ASPDU(aia,AIA) = true;
SACTDcnf IsSSPOf ASPDU(ada,ADA) = true;
(IsSACTEind(ssp) xor IsAE(pd)) => ssp IsSSPOf pd = false;
SACTEind(sn1,sdat1) IsSSPOf AE(sn2,sdat2) = ( (sn1 eq sn2) and (sdat1 eq sdat2) );
(IsSACTEcnf(ssp) xor (IsAEA(pd) or IsMAA(pd))) =>
        SACTEcnf(sdat1) IsSSPOf AEA(sn,sdat2) = (sdat1 eq sdat2);
endtype
(*--------------------------------------------------------------------------------------------------------
```

9.1.4.5 SPDU encoding rules

The encoding rules for the SPDU parameters (parameter groups) are represented by functions introduced by type *Encodings.*

NOTE - It is assumed that, contrary to tables 11, 12 and 13 of clause 8 of ISO 8327, the Connection Identifier parameter group is mandatory in encoded CN, AC and RF SPDUs. This is done to enable determining the values of the Session Connection Identifier parameter in the corresponding indication and confirm SSPs.

```
------------------------------------------------------------------------------------------------------------*)
type Encodings is DatOctStr2,ASPDU,BitNatRepr,DecNatRepr,DatConv,OctConv
opns
_RepOpt_,_RepTSDU_,_RepVer_,_RepReq_,_RepSAP1_,_RepSAP2_, _RepData_,_RepSPSN_,
        _RepCAItem_,_RepConId_,_RepTokIt_: DatOctStr, ASPDU -> Bool
RepCadRef,RepCagRef,RepCoRef,RepAdRef: ASPDU -> DatOctStr
Repr2: Nat -> DatOctStr                    Repr: DecString -> Octet
Repr: Nat -> Octet                         ReprSn: SSPSN ->DatOctStr
Reprto: STsAss -> DatOctStr                Reprfu: SRqms -> DatOctStr
RepDec: DecString -> DatOctStr             RTok: SToken,STsAss -> BitString
Bit: Bool -> Bit
eqns forall str,str1,str2,str3,str4,str5, bstr:BitString, ds1:DecString, oct:Octet, pd:ASPDU, tokass:STsAss,
        tok:SToken, sn:SSPSN
ofsort Bool
IsCN(pd) => empty RepCAItem pd = (empty RepOpt pd) and (empty RepTSDU pd) and
        (empty RepVer pd) and (empty RepSPSN pd) and (empty RepTok pd);
(* So called default values may be associated with the Connect/Accept Item parameter, so that if the
Connect/Accept Item parameter group encoding is omitted, the peer SPM can assume these default
values. *)

IsCN(pd) => (Repr(Dec(5))+
        (Repr(Length(str1)+Length(str2)+Length(str3)+Length(str4)+Length(str5))+
        str1++str2++str3++str4++str5))  RepCAItem  pd =
                (str1 RepOpt pd) and (str2 RepTSDU pd) and (str3 RepVer pd) and (str4 RepSPSN pd) and
                (str5 RepTok pd);
(IsCN(pd) and (oct ne Dec(5))) => (oct+str1) RepCAItem pd = false;
not(IsCN(pd)) => str RepCAItem pd = false;

IsCN(pd) =>
        (Repr(Dec(1))+(Repr(Length(str1)+Length(str2)+Length(str3))+str1++str2++str3))  RepConId  pd =
                (str1 eq RepCagRef(pd)) and (str2 eq RepCoRef(pd)) and (str3 eq RepAdRef(pd));
IsCN(pd) => empty RepConId pd = false;
(IsCN(pd) and (oct ne Dec(1))) => (oct+str1) RepConId pd = false;
not (IsCN(pd)) => str RepConId pd = false;
(* see NOTE *)

(* The function Repr determines the octet encoding of a natural number value and of a natural number
value represented as a string of decimal digits. Repr2 determines the encoding of the value of the TSDU
Maximum Size parameter. Below, possible encodings of the following parameters are defined: Protocol
Options, TSDU maximum Size, Version Number, Initial Serial Number, and Token Setting Item. Some of
these parameters are not mandatory (viz. TSDU Maximum Size, Initial Serial Number and Token Setting
Item) in which case the corresponding encoding could be the empty string. The peer SPM, upon receiving
such an empty string, assumes the default values associated with the parameter. *)

IsCN(pd) => str RepConId pd =
        ( ProtOptions(pd) and (str eq (Repr(1+Dec(9))+(Repr(Dec(1))+(Repr(Dec(1))+empty)))) ) or
        ( not(ProtOptions(pd)) and (str eq (Repr(1+Dec(9))+(Repr(Dec(1))+(Repr(Dec(0))+empty)))) );
not (IsCN(pd)) => str RepOpt pd = false;
IsCN(pd) => str RepVer pd = (str eq (Repr(2+Dec(2))+(Repr(Dec(1))+(Repr(Version(pd))+empty))));
not (IsCN(pd)) => str RepVer pd = false;
```

```
IsCN(pd) => str RepTSDU pd = ( (MtsRecFlow(pd) eq 0) and (MtsSendFlow(pd) eq 0) and
        ((str eq empty) or (str eq (Repr(2+Dec(1))+(Repr(Dec(0))+empty)))) ) or
        ( str eq (Repr(2+Dec(1))+(Repr(Dec(4))+Repr2(MtsSendFlow(pd))++Repr2(MtsRecFlow(pd)))) );
not (IsCN(pd)) => str RepTSDU pd = false;

IsCN(pd) => str RepSPSN pd = ( (SPSN(pd) eq absent) and ((str eq empty) or
        (str eq (Repr(2+Dec(3))+(Repr(Dec(0))+empty)))) ) or ( (SPSN(pd) ne absent) and
        (str eq (Repr(2+Dec(3))+(Repr(Length(ReprSn(SPSN(pd))))+ReprSn(SPSN(pd))))) );
not (IsCN(pd)) => str RepSPSN pd = false;

IsCN(pd) => str RepTok pd = (AcrTokens(TokenSet(pd)) eq {}) and
        (ChoiceTokens(TokenSet(pd)) eq {}) and
        ( (str eq empty) or (str eq (Repr(2+Dec(6))+(Repr(Dec(0))+empty))) ) or
        ( ((AcrTokens(TokenSet(pd)) ne {}) or (ChoiceTokens(TokenSet(pd)) ne {})) and
        (str eq (Repr(2+Dec(6))+(Repr(Dec(1))+Reprto(TokenSet(pd))))) );
not (IsCN(pd)) => str RepTok pd = false;

(* RepReq defines possible encodings of the Session User Requirements parameter. *)

IsCN(pd) => str RepReq pd =
        ( ((Card(FUs(pd)) eq NatNum(Dec(5))) and (HD IsIn FUs(pd)) and (SY IsIn FUs(pd)) and
        (ACT IsIn FUs(pd)) and (CD IsIn FUs(pd)) and (EXCEP IsIn FUs(pd)) and (str eq empty)) or
        (str eq (Repr(2+Dec(0))+(Repr(Dec(2))+Reprfu(FUs(pd))))) );
not (IsCN(pd)) => str RepReq pd = false;

(* RepSAP1 and RepSAP2 define the possible encodings of the Calling and Called Session Selector
respectively. *)

IsCN(pd) => str RepSAP1 pd =
        ((str eq empty) or (str eq (Repr(5+Dec(1))+(Repr(Dec(0))+empty))) or
        (str eq (Repr(5+Dec(1))+(Repr(Length(SAddress(CgAddr(pd))))+SAddress(CgAddr(pd))))) );
IsCN(pd) => str RepSAP2 pd =
        ((str eq empty) or (str eq (Repr(5+Dec(2))+(Repr(Dec(0))+empty))) or
        (str eq (Repr(5+Dec(2))+(Repr(Length(SAddress(CdAddr(pd))))+SAddress(CdAddr(pd))))) );
not (IsCN(pd)) => str RepSAP1 pd = false;            not (IsCN(pd)) => str RepSAP2 pd = false;

(* The User Data encodings are presented below. *)

IsCN(pd) => str RepData pd = ( (DaDs(UserInf(pd)) eq empty) and (str eq empty)) or
        (str eq (Repr(1+(9+Dec(3)))+(Repr(Length(DaDs(UserInf(pd))))+DaDs(UserInf(pd))))) );
not (IsCN(pd)) => str RepData pd = false;
```

ofsort DatOctStr

```
(* Reprto defines the encoding of the value of the Token Setting Item. OStr converts a bit string consisting
of eight bits to a value of sort Octet, and function Octet converts a value of sort Octet to a value of sort
DatOctStr. *)

Reprto(tokass) = Octet(OStr(RTok(tr,tokass)++RTok(ma,tokass)++RTok(mi,tokass)++RTok(dk,tokass))) ;

(* ReprSn defines the encoding of the value of the Initial Serial Number parameter. The value of this
parameter should not exceed 999 999. DStr converts a bit string to its decimal digit equivalent. The value of
the serial number is represented by a string of decimal digits. Each digit is encoded as a separate octet
consisting of a binary encoding in four bits of its value appended to the bit string 0011. *)

(Length(ds1) le NatNum(Dec(6))) => ReprSn(s(NatNum(ds1))) = RepDec(ds1);
( h(sn) ge (NatNum(1+Dec(0))**NatNum(Dec(6))) ) => ReprSn(sn) = empty;
RepDec(d+ds1) = RepDec(d)+RepDec(ds1) ;
RepDec(0) = Octet(OStr(0+(0+(1+(1+(0+(0+(0+Bit(0)))))))) ;
RepDec(1) = Octet(OStr(0+(0+(1+(1+(0+(0+(0+Bit(1)))))))) ;
RepDec(2) = Octet(OStr(0+(0+(1+(1+(0+(0+(1+Bit(0)))))))) ;
RepDec(3) = Octet(OStr(0+(0+(1+(1+(0+(0+(1+Bit(1)))))))) ;
RepDec(4) = Octet(OStr(0+(0+(1+(1+(0+(1+(0+Bit(0)))))))) ;
RepDec(5) = Octet(OStr(0+(0+(1+(1+(0+(1+(0+Bit(1)))))))) ;
RepDec(6) = Octet(OStr(0+(0+(1+(1+(0+(1+(1+Bit(0)))))))) ;
RepDec(7) = Octet(OStr(0+(0+(1+(1+(0+(1+(1+Bit(1)))))))) ;
RepDec(8) = Octet(OStr(0+(0+(1+(1+(1+(0+(0+Bit(0)))))))) ;
RepDec(9) = Octet(OStr(0+(0+(1+(1+(1+(0+(0+Bit(1)))))))) ;

ofsort BitString
(* RTok delivers a two-bit encoding for each token depending on its assignment. *)

(tok IsIn RqrTokens(tokass)) => RTok(tok,tokass) = (0+Bit(0));
(tok IsIn AcrTokens(tokass)) => RTok(tok,tokass) = (0+Bit(1));
(tok IsIn ChoiceTokens(tokass))  => RTok(tok,tokass) = (1+Bit(0));
endtype
(*----------------------------------------------------------------------------------------------------------------
```

9.2 Session variables

9.2.1 Vrsp

Type *VVRSP* defines constants that represent the values of variable "Vrsp" defined in subclause A.5.4.4 of ISO 8327: *int* (activity interrupt), *dsc* (activity discard), *a* (resynchronize abandon), *s* (resynchronize set), *r* (resynchronize restart) and *no* (no resynchronization). The definition consists of a renaming of the auxiliary type *Sextet* (see 9.4.7).

```
----------------------------------------------------------------------------------------------------------------*)
type VVRSP is Sextet renamedby
sortnames VVRSP for Sextet
opnnames              int for constant1       dsc for constant2        a for constant3
s for constant4       r for constant5         no for constant6
endtype
(*----------------------------------------------------------------------------------------------------------------
```

9.2.2 Vact

Type *VACT* defines constants that represent the values of variable "Vact" defined in subclause A.5.4.2 of ISO 8327. One additional constant, *NotApplicable*, is introduced to indicate that the activity management functional unit is not selected. The definition consists of a renaming of the auxiliary type *Triplet*, imported from ISO/IEC/TR 9571.

```
----------------------------------------------------------------------------------------------------------------*)
type VACT is Triplet renamedby
sortnames VACT for Triplet
opnnames   NotApplicable for constant1     InProgress for constant2      NotInProgress for constant3
endtype
(*----------------------------------------------------------------------------------------------------------------
```

9.3 Additional functions on session and transport service primitives

Type *SesTransRelations* defines a boolean function, *TCON_SCON*, that tests whether a given T-CONNECT request is related to a given S-CONNECT request by way of fulfilling the conditions: 1) if extended control is requested in the SSP, then the transport expedited flow is requested in the TSP, and 2) the transport QOS should be better than or equal to the desired session QOS.

```
----------------------------------------------------------------------------------------------------------*)
type SesTransRelations is SessionServicePrimitive, TransportServicePrimitive
opns TCON_SCON: TSP,SSP -> Bool
eqns forall ssp:SSP, tsp:TSP, ta1,ta2:TAddress, tex:TEXOption, tq:TQOS, d:OctetString ofsort Bool
not( (IsSCONreq(ssp) or IsSCONind(ssp)) and IsTCONreq(tsp) ) => tsp TCON_SCON ssp = false;
(IsSCONreq(ssp) or IsSCONind(ssp)) => TCONreq(ta1,ta2,tex,tq,d) TCON_SCON ssp =
    ( (ExtControl(QOS(ssp)) eq desired) implies (tex eq UseTEX) );
endtype
(*----------------------------------------------------------------------------------------------------------
```

9.4 Auxiliary data types

9.4.1 Data octet string

Type *DatOctStr1* is a renaming and enrichment of the standard type *OctetString. DatOctStr2* enriches the latter type with functions:

- *Empty* that tests whether a given octet string is the empty string;
- *First* that yields the first (leftmost) octet of a given octet string (or an octet consisting of only "0" bits if the string is empty); and
- -- that appends an octet to the back of a given octet string.

```
----------------------------------------------------------------------------------------------------------*)
type DatOctStr1 is OctetString renamedby
sortnames   DatOctStr for OctetString        opnnames empty for <>
endtype

type DatOctStr2 is DatOctStr1
opns
Empty: DatOctStr -> Bool          First: DatOctStr -> Octet              _--_: DatOctStr,Octet -> DatOctStr
eqns forall str:DatOctStr, oc:Octet
ofsort Bool                 Empty(str) = (str eq empty);
ofsort DatOctStr            str--oc = Reverse(oc+Reverse(str));
ofsort Octet
First(<> of DatOctStr) = Octet(0,0,0,0,0,0,0,0);          First(oc+str) = oc;
endtype
(*----------------------------------------------------------------------------------------------------------
```

9.4.2 Notational shorthands for comparing natural number values in different representations

Type *Compare* introduces a number of functions that allow the comparison of two natural number values, where at least one of the values is not abstractly represented (defined by standard type *NaturalNumber*) but by way of a particular representation (octet, decimal string). The definition imports the standard types *Octet, DecNatRepr* and *BitNatRepr*, and the auxiliary type *OctConv* (9.4.4).

```
--------------------------------------------------------------------------------------------------------*)
type Compare is Octet,DecNatRepr,BitNatRepr,OctetToFromBitString
opns
_eq_, _le_, _lt_, _ge_, _gt_, _ne_: Octet,DecString -> Bool
_eq_, _le_, _lt_, _ge_, _gt_, _ne_: DecString,DecString -> Bool
_eq_, _le_, _lt_, _ge_, _gt_, _ne_: Nat,Octet -> Bool
ToNatNum : Octet -> Nat                        BitToNat : Bit -> Nat (* 0 to 0 and 1 to 1 *)
eqns  forall oc:Octet, n:Nat, ds1,ds2:DecString, b1,b2,b3,b4,b5,b6,b7,b8:Bit
ofsort Bool
oc eq ds1 = ToNatNum(oc) eq NatNum(ds1);       ds1 eq ds2 = NatNum(ds1) eq NatNum(ds2);
oc le ds1 = ToNatNum(oc) le NatNum(ds1);       ds1 le ds2 = NatNum(ds1) le NatNum(ds2);
oc ge ds1 = ToNatNum(oc) ge NatNum(ds1);       ds1 ge ds2 = NatNum(ds1) ge NatNum(ds2);
oc lt ds1 = ToNatNum(oc) lt NatNum(ds1);       ds1 lt ds2 = NatNum(ds1) lt NatNum(ds2);
oc gt ds1 = ToNatNum(oc) gt NatNum(ds1);       ds1 gt ds2 = NatNum(ds1) gt NatNum(ds2);
oc ne ds1 = ToNatNum(oc) ne NatNum(ds1);       ds1 ne ds2 = NatNum(ds1) ne NatNum(ds2);
n eq oc = n eq ToNatNum(oc);                   n le oc = n le ToNatNum(oc);
n ge oc = n ge ToNatNum(oc);                   n lt oc = n lt ToNatNum(oc);
n gt oc = n gt ToNatNum(oc);                   n ne oc = n ne ToNatNum(oc);
(* ToNatNum converts an octet to the natural number it represents. *)
ofsort Nat
BitToNat(0) = 0 ;                              BitToNat(1) = Succ(0);
ToNatNum(Octet(b0,b1,b2,b3,b4,b5,b6,b7)) = (BitToNat(b0)*NatNum(1+(2+Dec(8)))) +
      (BitToNat(b1)*NatNum(6+Dec(4))) + (BitToNat(b2)*NatNum(3+Dec(2))) +
      (BitToNat(b3)*NatNum(1+Dec(6))) + (BitToNat(b4)*NatNum(Dec(8))) +
      (BitToNat(b5)*NatNum(Dec(4))) + (BitToNat(b6)*NatNum(Dec(2))) + (BitToNat(b7)*Succ(0));
endtype
(*----------------------------------------------------------------------------------------------------------
```

9.4.3 Mapping between octet strings of different types

Type *DatConv* introduces functions that allow the mapping between octet strings of different types (one of which is the standard type *OctetString* and the other a renaming of *OctetString*).

```
--------------------------------------------------------------------------------------------------------*)
type DatConv is DatOctStr1,SData,OctetString
opns
DaDs: SData -> DatOctStr                       DaDs: OctetString -> DatOctStr
DsDa: DatOctStr -> OctetString
eqns forall sd:SData, oct:Octet, so:OctetString, dos:DatOctStr
ofsort DatOctStr
DaDs(<> of SData) = empty;                     DaDs(oct+sd) = oct+DaDs(sd);
DaDs(<> of OctetString) = empty;               DaDs(oct+so) = oct+DaDs(so);
ofsort OctetString
DsDa(empty) = <>;                              DsDa(oct+dos) = oct+DsDa(dos);
endtype
(*----------------------------------------------------------------------------------------------------------
```

9.4.4 Mapping between octets and bit strings

Type *OctConv* introduces functions that allow the conversion from an octet to a bitstring (such that they represent the same natural number when interpreted as a binary encoding) and reversely.

```
--------------------------------------------------------------------------------------------------------*)
type OctConv is BitString,Octet,NaturalNumber,DecDigit
opns
BStr: Octet -> BitString                       OStr: BitString -> Octet
```

```
eqns forall b0,b1,b2,b3,b4,b5,b6,b7:Bit,bstr:BitString
ofsort BitString
BStr(Octet(b0,b1,b2,b3,b4,b5,b6,b7)) = b0+(b1+(b2+(b3+(b4+(b5+(b6+Bit(b7))))))) ;
ofsort Octet
Length(bstr) ne NatNum(Dec(8)) => OStr(bstr) = Octet(0,0,0,0,0,0,0,0);
OStr(b0+(b1+(b2+(b3+(b4+(b5+(b6+Bit(b7)))))))) = Octet(b0,b1,b2,b3,b4,b5,b6,b7);
endtype
(*-----------------------------------------------------------------------------------------------
```

9.4.5 Additional functions on synchronization numbers and natural numbers

Type *MaxMinPred* adds the following functions to those already defined in the standard type *NaturalNumber* and type *SSynchronizationNumber*, imported from ISO/IEC/TR 9571:

- *Pred* determines the predecessor of a given a natural number. If the argument number is *0* then *Pred* yields *0*;
- *Max* and *Min* are defined twice, namely for two natural number arguments and for two synchronization number arguments; in both cases they determine the maximum and minimum, respectively, of their two arguments.

```
-----------------------------------------------------------------------------------------------*)
type MaxMinPred is SSynchronizationNumber
opns
Pred: Nat -> Nat                 Max,Min: SSPSN,SSPSN -> SSPSN     Max,Min: Nat,Nat -> Nat
eqns forall s1,s2:SSPSN, nn,nn1,nn2:Nat
ofsort SSPSN
Max(s1,s2) = s(Max(h(s1),h(s2)));              Min(s1,s2) = s(Min(h(s1),h(s2)))
ofsort Nat
Pred(0) = 0;                                   Pred(Succ(nn))= nn;
nn1 ge nn2 => Max(nn1,nn2) = nn1;              nn1 lt nn2 => Max(nn1,nn2) = nn2;
nn1 ge nn2 => Min(nn1,nn2) = nn2;              nn1 lt nn2 => Min(nn1,nn2) = nn1;
endtype
(*-----------------------------------------------------------------------------------------------
```

9.4.6 Two-tuple, three-tuple and five-tuple

Type *Twotuple* defines the formal requirements for building 2-component values and extracting the components of these values, where the components are renamings of the standard formal type *Element*. Similarly, type Threetuple and Fivetuple define the formal requirements for building 3-component and 5-component values, respectively, and extracting the components of these values.

```
-----------------------------------------------------------------------------------------------*)
type Twotuple is Element1,Element2 sorts Twotuple
opns
Twotuple: Element1,Element2 -> Twotuple
First: Twotuple -> Element1                    Second: Twotuple -> Element2
eqns forall e1:Element1, e2:Element2
ofsort Element1          First(Twotuple(e1,e2)) = e1;
ofsort Element2          Second(Twotuple(e1,e2)) = e2;
endtype
```

```
type Threetuple is Element1,Element2,Element3 sorts Threetuple
opns
Threetuple: Element1,Element2,Element3 -> Threetuple
First: Threetuple -> Element1                    Second: Threetuple -> Element2
Third: Threetuple -> Element3
eqns forall e1:Element1, e2:Element2, e3:Element3
ofsort Element1           First(Threetuple(e1,e2,e3)) = e1;
ofsort Element2           Second(Threetuple(e1,e2,e3)) = e2;
ofsort Element3           Third(Threetuple(e1,e2,e3)) = e3;
endtype

type Fivetuple is Element1,Element2,Element3,Element4,Element5 sorts Fivetuple
opns
Fivetuple: Element1,Element2,Element3,Element4,Element5 -> Fivetuple
First: Fivetuple -> Element1                     Second: Fivetuple -> Element2
Third: Fivetuple -> Element3                     Fourth: Fivetuple -> Element4
Fifth: Fivetuple -> Element5
eqns forall e1:Element1, e2:Element2, e3:Element3, e4:Element4, e5:Element5
ofsort Element1           First(Fivetuple(e1,e2,e3,e4,e5)) = e1;
ofsort Element2           Second(Fivetuple(e1,e2,e3,e4,e5)) = e2;
ofsort Element3           Third(Fivetuple(e1,e2,e3,e4,e5)) = e3;
ofsort Element4           Fourth(Fivetuple(e1,e2,e3,e4,e5)) = e4;
ofsort Element5           Fifth(Fivetuple(e1,e2,e3,e4,e5)) = e5;
endtype

type Element1 is Element renamedby sortnames Element1 for Element endtype
type Element2 is Element renamedby sortnames Element2 for Element endtype
type Element3 is Element renamedby sortnames Element3 for Element endtype
type Element4 is Element renamedby sortnames Element4 for Element endtype
type Element5 is Element renamedby sortnames Element5 for Element endtype
(*-----------------------------------------------------------------------------------------*)
```

9.4.7 Sextet

Sextet defines a set of six constants, endowed with boolean equality.

```
------------------------------------------------------------------------------------------*)
type Sextet is Boolean,NaturalNumber sorts Sextet
opns
constant1,constant2,constant3,constant4,constant5,constant6: -> Sextet
_eq_,_ne_: Sextet,Sextet -> Bool                 f: Sextet -> Nat
eqns forall x,y:Sextet ofsort Nat
f(constant1) = 0;                                f(constant2) = Succ(0);
f(constant3) = Succ(Succ(0));                    f(constant4) = Succ(f(constant3));
f(constant5) = Succ(f(constant4));               f(constant6) = Succ(f(constant5));
ofsort Bool
x ne y = not(x eq y);                            x eq y = f(x) eq f(y);
endtype
(*-----------------------------------------------------------------------------------------*)
```

9.5 Types for components of the internal event structure

In the following subclauses types are presented that are related to the representation of component values in interactions at the internal gate *p* (see 6 and 10.1.2).

9.5.1 Transport flow

Type *TFlow* defines two constants to represent the two types of flow control that can be offered by the TS-provider. Type *Doublet* is imported from ISO/IEC/TR 9571.

```
----------------------------------------------------------------------------------------------------*)
type TFlow is Doublet renamedby
sortnames TFlow for Doublet
opnnames   normal for constant1             exp for constant2
endtype
(*----------------------------------------------------------------------------------------------------
```

9.5.2 SPDU exchange direction

A SPDU can be characterized as either outgoing (it is sent) or incoming (it is received). These directions are represented by two constants, *send* and *rec*, defined by type *Dir*. As a result of the constraint-oriented specification style, it is possible that a SPDU, that is generated internally by process *SSPSPDU*, will have to be discarded by process *SPDUTSP*. This is represented by a third constant *no_send*.

```
----------------------------------------------------------------------------------------------------*)
type Dir is Triplet renamedby
sortnames Dir for Triplet
opnnames   send for constant1             rec for constant2             no_send for constant3
endtype
(*----------------------------------------------------------------------------------------------------
```

9.5.3 SPDU status

The SPDU status values that need to be distinguished for SPDU exchanges are represented by a "bit register". Type *Index* defines constants that are associated with the position of a bit in this bit register. The interpretation of each of the bits is discussed in 10.1.2.1. Type *VAL* introduces functions for the manipulation and inspection of the bits, namely:

- *Set* and *Reset*, that assign the value *1*, respectively *0*, to the indicated bit;
- *Valid*, that checks whether bits 0 through 11 are set to *1*;
- *IsTDr*, that checks whether any of the bits 12, 13 or 14 is set to *1*;
- *IsValNoDel*, that checks whether bits 0 through 11 and bit 16 are set to *1*;
- *ValidDel*, that checks whether bits 0 through 11 are set to *1* and bit 16 is set to *0*;
- *Winner*, that checks whether bit 19 is equal to *1*.

```
----------------------------------------------------------------------------------------------------*)
type Index is sorts Index
opns 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19: -> Index
endtype

type VAL is Bit,Index sorts Validity
opns
VAL: Bit,Bit,Bit,Bit,Bit,Bit,Bit,Bit,Bit,Bit,Bit,Bit,Bit,Bit,Bit,Bit,Bit,Bit,Bit,Bit -> Validity
Valid, ValidDel, IsTDr, IsValNoDel, Winner: Validity -> Bool
Set, Reset: Validity,Index -> Bool
eqns forall b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15,b16,b17,b18,b19:Bit, v:Validity,
       ind:Index ofsort Bool
Reset(v,ind) = not(Set(v,ind);
IsTDr(v) = Set(v,12) or Set(v,13) or Set(v,14);
```

```
Set(VAL(b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15,b16,b17,b18,b19),0) = (b0 eq 1);
Set(VAL(b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15,b16,b17,b18,b19),1) = (b1 eq 1);
Set(VAL(b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15,b16,b17,b18,b19),2) = (b2 eq 1);
Set(VAL(b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15,b16,b17,b18,b19),3) = (b3 eq 1);
Set(VAL(b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15,b16,b17,b18,b19),4) = (b4 eq 1);
Set(VAL(b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15,b16,b17,b18,b19),5) = (b5 eq 1);
Set(VAL(b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15,b16,b17,b18,b19),6) = (b6 eq 1);
Set(VAL(b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15,b16,b17,b18,b19),7) = (b7 eq 1);
Set(VAL(b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15,b16,b17,b18,b19),8) = (b8 eq 1);
Set(VAL(b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15,b16,b17,b18,b19),9) = (b9 eq 1);
Set(VAL(b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15,b16,b17,b18,b19),10) = (b10 eq 1);
Set(VAL(b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15,b16,b17,b18,b19),11) = (b11 eq 1);
Set(VAL(b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15,b16,b17,b18,b19),12) = (b12 eq 1);
Set(VAL(b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15,b16,b17,b18,b19),13) = (b13 eq 1);
Set(VAL(b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15,b16,b17,b18,b19),14) = (b14 eq 1);
Set(VAL(b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15,b16,b17,b18,b19),15) = (b15 eq 1);
Set(VAL(b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15,b16,b17,b18,b19),16) = (b16 eq 1);
Set(VAL(b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15,b16,b17,b18,b19),17) = (b17 eq 1);
Set(VAL(b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15,b16,b17,b18,b19),18) = (b18 eq 1);
Set(VAL(b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15,b16,b17,b18,b19),19) = (b19 eq 1);
Valid(v) = Set(v,0) and Set(v,1) and Set(v,2) and Set(v,3) and Set(v,4) and Set(v,5) and Set(v,6) and
     Set(v,7) and Set(v,8) and Set(v,9) and Set(v,10) and Set(v,11) ;
ValidDel(v) = Valid(v) and Reset(v,16);
ValidNoDel(v) = Valid(v) and Set(v,16);
Winner(v) = Set(v,18) or Set(v,19);
endtype
(*-----------------------------------------------------------------------------------------------
```

9.5.4 Transfer state

The transfer state values that need to be distinguished for SPDU exchanges, represented by interactions at gate *p*, are defined by type *TrPhase*. The interpretation of these values is discussed in 10.1.2.2. In addition, an auxiliary function, *tr*, is introduced for mapping the values on natural numbers, and a number of functions for comparing two transfer state values.

```
-----------------------------------------------------------------------------------------------*)
type TrPhase is NaturalNumber sorts TrPhase
opns
tr: TrPhase -> Nat                              0,1,2,3,4,5,6: -> TrPhase
_eq_, _ne_, _gt_, _le_, _lt_, _ge_: TrPhase,TrPhase -> Bool
eqns forall p1,p2:TrPhase
ofsort Nat
tr(0) = 0;             tr(1) = Succ(0);         tr(2) = Succ(tr(1));       tr(3) = Succ(tr(2));
tr(4) = Succ(tr(3));   tr(5) = Succ(tr(4));     tr(6) = Succ(tr(5));
ofsort Bool
p1 eq p2 = tr(p1) eq tr(p2);     p1 ne p2 = tr(p1) ne tr(p2);     p1 le p2 = tr(p1) le tr(p2);
p1 ge p2 = tr(p1) ge tr(p2);     p1 lt p2 = tr(p1) lt tr(p2);     p1 gt p2 = tr(p1) gt tr(p2)
endtype
(*-----------------------------------------------------------------------------------------------
```

10 Processes for the SPM session to transport service boundary relation

The behaviour of a single SPM, deprived from constraints that are local to either the SCEP or the TCEP, is described by the composition of a number of processes as explained in clause 6. This decomposition of the session to transport boundary relation in terms of component processes is presented in 10.1, together with a further explanation of some of the components of interactions that are internal to the SPM (i.e., at gate *p*). The definition of the component processes is subsequently presented in 10.2 through 10.5.

10.1 First decomposition of the session to transport boundary relation

10.1.1 Service primitive mappings and SPDU exchanges

Mappings between SSPs and TSPs fall into two categories, namely direct and indirect (via SPDUs) mappings. The direct mappings are defined by *SSPTSP*; the indirect mappings are defined by *SSPSPDU* (representing the relationship between SSPs and SPDUs) and *SPDUTSP* (representing the relationship between SPDUs and TSPs). The latter two processes synchronize at the hidden gate *p*, the protocol gate, to model the exchange of SPDUs, together with a third process *SPDUConstraints* that defines the local protocol constraints on the exchange of SPDUs consistent with the state tables description in annex A of ISO 8327.

The disabling ***exit*** is used to synchronize with the termination of *TCEPs* (see clause 8).

```
---------------------------------------------------------------------------------------------------------*)
process STPM[s,t](spei:SPEImplementation): exit :=
( SSPTSP[s,t] || ( hide p in ( SSPSPDU[s,p] |[p]| SPDUTSP[p,t] |[p]| SPDUConstraints[p](spei) ) )
) [> exit
endproc
(*---------------------------------------------------------------------------------------------------------
```

10.1.2 Some comments on the interactions at the protocol gate

Two components of the internal event structure, i.e. of interactions at the protocol gate *p*, need some further explanation, viz. the SPDU status (see 9.5.3) and the transfer state (see 9.5.4).

10.1.2.1 Status of incoming SPDUs

The status (validity) of an incoming SPDU is not determined in one process but simultaneously in several processes synchronizing at the *p* gate; each process then imposes its partial constraints. The sum of these partial constraints constitutes the total constraint which has to be satisfied.

In order to model this summation, the SPDU status component is interpreted as a "bit register". Each bit "belongs" to a particular process and is set or reset by that process depending on whether this SPDU satisfies the partial constraints imposed by the process manipulating that bit.

The "bit register" consists of 20 bits, numbered from 0 through 19, with the following meaning:
0: Set to *0* if any of the following conditions hold (set to *1* otherwise):

- receipt of an extended concatenated sequence of SPDUs when no extended concatenation is allowed on the receiving flow;
- receipt of a DT or TD SPDU of a segmented SSDU when no segmenting was selected on the receiving flow;
- a DT SPDU or TD SPDU of a segmented SSDU which does not contain user data and is not the last SPDU of the segmented SSDU;
- receipt of an invalid concatenation of SPDUs.

1: Set to *0* if a badly encoded SPDU is received (otherwise *1*).
2: Set to *0* if a SPDU was received on the wrong transport flow (otherwise *1*).
3: Set to *0* if the constraints for valid SPDUs related to the availability of the transport expedited flow are not satisfied (otherwise *1*).
4: Idem for constraints related to "Vtca".
5: Idem for the SPDU parameter values constraints.
6: Idem for constraints related to functional units.

7: Idem for constraints related to token settings.
8: Idem for constraints related to "Vact".
9: Idem for constraints related to "V(M)", "V(A)" and "Vsc".
10: Idem for constraints related to "Vrsp" and "V(R)".
11: Idem for the SPDU temporal ordering constraints
12, 13, 14: Set to *1* if a T-DISCONNECT request is to be issued (otherwise *0*). (These bits are (re)set by processes *VtcaCon*, *TexCon* and *SPDUOrd* (see 10.4.2 and 10.5.5), respectively.)
15: Set to *0* if an incoming (AEA or MAA) SPDU should lead to the execution of a S-SYNCHRONIZE-MAJOR confirm. It is set to *1* if an incoming SPDU should prompt the execution of an S-ACTIVITY-END confirm (no change otherwise).
16: Set to *1* if an incoming SPDU does not lead to the execution of the corresponding SSP (otherwise *0*)
17: Set to *1* to signal any of the following events (otherwise *0*):
- queuing of a PR_RS SPDU while in state tables state STA713;
- queuing of an EX SPDU while in either of the state tables states STA15C or STA02A;
- too many PR SPDUs in the queue.

18, 19: Set to *1* to signal that the "SPMwinner" condition is true (otherwise *0*).

10.1.2.2 SPDU transfer states

A particular value of the transfer state component indicates one of the states of the state tables in annex A of ISO 8327, or a collection of such states. The transfer state is kept track of by one of the subprocesses of *SPDUConstraints*, namely *SPDUOrd* (see 10.5.5) which handles the pure ordering constraints of SPDUs.

Usually, the transfer state component is relevant for incoming SPDUs only. In one instance, however, it is used for an outgoing SPDU as well (viz. a DN SPDU). In this case another subprocess of *SPDUConstraints* (viz. *VtrrCon*, see 10.5.6.2, which imposes constraints on SPDU exchanges related to "Vtrr") uses the transfer state component to inform *SPDUOrd* about the transfer state that is valid for the next incoming SPDU. Below, the correspondence between the seven possible transfer state values and the protocol state tables states is listed for each relevant incoming SPDU (for some incoming SPDUs there is no need to know, hence to constrain, the transfer state value).

	transfer state			
	0	*1*	*2*	*3* (*4*, *5*, *6* see below)
SPDU				
DN	16	1C		
AB	1A,6,15,16	*		
AD,AI	16,1	*	6,15B,15C	
FN,DN	16,1	3	*	
ADA,AIA	1,16	5B,5C	*	
RA	1,16	5A	*	
MAP,MIP,AE	1,16	713	*	
MIA	1,16	3,713	4,15A	*
RS	1,16	3	*	713,15B
MAA,AEA	1,16	4A,4B	15A	*
DT	1,16	*	18,21	
EX,TD	1,16	*		
ER,ED	1,16	3	*	713
**	1,16	*		

* stands for all other states not explicitly mentioned in the corresponding row of the table.
** denotes the following SPDUs: AC, RF, PR, CN, AR, CD, AS, GT, PT, NF, and GTC.

The transfer state values *4* and *5* are only relevant for incoming RS SPDUs, and ER and ED SPDUs; *4* corresponds to state tables states 5B, 5C or 6 (for a RS SPDU), and 713 (for an ER or ED SPDU), and *5* corresponds to state tables state 20 (for a RS SPDU), and 4A or 4B (for an ER or ED SPDU). Transfer state value *6* is exclusively used for ER and ED SPDUs to indicate state table state 4A or 4B.

10.2 Direct mapping between SSPs and TSPs

When a SC is requested and no suitable underlying TC is available, a new TC has to be established. The relation between the parameters of the resulting T-CONNECT request and the previous S-CONNECT request is tested with function *TCON_SCON* (see 9.3) in process *SSPTSP*.

Once a TC is established, process *SSPTSP1* is invoked with the following parameters: 1) the QOS of the TC, 2) the availability of the transport expedited flow and 3) a boolean indicating whether the SPM was the initiator or acceptor of the TC.

Establishment of a SC can be abruptly terminated either by the SS-user by means of a S-U-ABORT request, or by the TS-provider by means of a T-DISCONNECT indication.

```
-------------------------------------------------------------------------------------------------------*)
process SSPTSP[s,t]: exit :=
        s ?sa:SAddress ?si:SCEI ?ssp:SSP; (*SCONreq*)
        t ?ta:TAddress ?ti:TCEI ?tsp:TSP [tsp TCON_SCON ssp]; SSPTSP[s,t]
[]      t ?ta:TAddress ?ti:TCEI ?tcn:TSP [IsTCONconf(tcn) or IsTCONresp(tcn)];
        (       choice tq:TQOS,tex:TEXOption [] [(tex IsTEXOptionOf tcn) and (tq IsTQOSOf tcn)] ->
                SSPTSP1[s,t](tq,IsTCONresp(tcn),tex)    )
[]      s ?sa:SAddress ?si:SCEI ?ssp:SSP [IsSUABreq(ssp)];
        t ?ta:TAddress ?ti:TCEI ?tsp:TSP [IsTDISreq(tsp)]; exit
[]      t ?ta:TAddress ?ti:TCEI ?td:TSP [IsTDISind(td)];
        s ?sa:SAddress ?si:SCEI !SPABind(transportDisconnect); exit
[]      t ?ta:TAddress ?ti:TCEI ?tsp:TSP [IsTCONind(tsp)]; SSPTSP[s,t]
endproc
(*-----------------------------------------------------------------------------------------------------
```

Re-use of an existing TC is only possible if the (implicit) requirements derived from the S-CONNECT service primitive are "upwards compatible" with the characteristics of the TC. This notion of upward compatibility has not been formalized.

A T-DISCONNECT indication will cause a S-P-ABORT indication with the Reason parameter equal to "transport disconnect" to be executed at the SS boundary.

```
-------------------------------------------------------------------------------------------------------*)
process SSPTSP1[s,t](tq:TQOS,Vtca:Bool,tex:TEXOption): exit :=
        (       choice ssp:SSP, ta1,ta2:TAddress, d:OctetString []
                [ ( (IsSCONind(ssp) or IsSCONreq(ssp)) implies (TCONreq(ta1,ta2,tex,tq,d) TCON_SCON ssp) )
                and (Vtca implies not(IsSCONreq(ssp))) and (ssp ne SPABind(transportDisconnect)) ] ->
                s ?sa:SAddress ?si:SCEI !ssp; SSPTSP1[s,t](tq,Vtca,tex)    )
[]      t ?ta:TAddress ?ti:TCEI ?tdo:TSP [not(IsTDIS(tdo))]; SSPTSP1[s,t](tq,Vtca,tex)
[]      t ?ta:TAddress ?ti:TCEI ?td:TSP [IsTDISind(td)];
        s ?sa:SAddress ?si:SCEI !SPABind(transportDisconnect); exit
[]      t ?ta:TAddress ?ti:TCEI ?tdr:TSP [IsTDISreq(tdr)]; exit
endproc
(*-----------------------------------------------------------------------------------------------------
```

10.3 Relation between SSPs and SPDUs

In the description of the relationship between SSPs and SPDUs, two concerns are separated. Process *SSPSPDUCon* specifies the constraints on the issuance of certain SSPs (viz. S-RESYNCHRONIZE (abandon) indications and S-CONNECT requests) related to the SPM state, and process *SSPSPDUTran* specifies the constraints on the transformation of SSPs to SPDUs and vice versa.

```
-------------------------------------------------------------------------------------------------------*)
process SSPSPDU[s,p]: noexit := SSPSPDUCon[s,p] || SSPSPDUTran[s,p] endproc
(*-------------------------------------------------------------------------------------------------------
```

10.3.1 Relation between SPM state and session service constraints

SSPSPDUCon performs the following functions:

- it keeps track of the session (protocol) variable "V(M)" and ensures that the synchronization point serial number in a S-RESYNCHRONIZE (abandon) indication is equal to the maximum of "V(M)" and the serial number in the received RS (abandon) SPDU;
- it prevents the execution of a S-CONNECT request if, following some event, the SC ceases to exist, until it has been determined that the TC is to be kept for re-use.

```
-------------------------------------------------------------------------------------------------------*)
process SSPSPDUCon[s,p]: noexit := VMCon[s,p](s(Succ(0))) || SPCon[s,p] endproc

process VMCon[s,p](VM:SSPSN): noexit :=
     choice pd:ASPDU,fl:TFlow,d:Dir,sn:SSPSN,v:Validity,st:TrSTate []
     [( (d eq rec) implies ValidDel(v) ) and (sn eq SPSN(pd))] ->
     (    [IsAC(pd) or IsAS(pd) or IsRA(pd)] -> p !pd !fl !d !v !st; VMCon[s,p](sn)
     []   [IsAR(pd)] -> p !pd !fl !d !v !st; VMCon[s,p](s(Succ(h(sn))))
     []   [IsMAP(pd) or IsAE(pd)] -> p !pd !fl !d !v !st; VMCon[s,p](s(Succ(h(VM))))
     []   [a eq ResynType(pd)] -> p !pd !fl !d !v !st; VMCon[s,p](Max(VM,sn))
     []   [not(IsAC(pd) or IsAR(pd) or IsAS(pd) or IsRA(pd) or IsMAP(pd) or IsAE(pd) or
          (a eq ResynType(pd))) or ( (d eq rec) and not(ValidDel(v)))] ->
          p !pd !fl !d !v !st; VMCon[s,p](VM)
     )
[]   choice ssp:SSP [] [IsSRSYNind(ssp) and (a IsResynTypeOf ssp) implies (VM IsSPSNOf ssp)] ->
     s ?sa:SAddress ?si:SCEI !ssp; VMCon[s,p](VM)
endproc
(*-------------------------------------------------------------------------------------------------------
```

In order to constrain the execution of a new S-CONNECT request following termination of the SC, *SPCon* and *SPConAA* monitor the exchange of SPDUs which indicate the termination of the SC, the exchange of SPDUs which (indirectly) indicate termination of the TC, and SPDU exchanges which indicate that the idle TC state is entered thus permitting a new S-CONNECT request to be issued.

```
-------------------------------------------------------------------------------------------------------*)
process SPCon[s,p]: exit :=
     choice pd:ASPDU,fl:TFlow,v:Validity,st:TrState []
     (    [( (IsRF(pd) or IsAB(pd)) and (nr eq TDISPar(pd)) ) or (IsDN(pd) and (st eq 0))] ->
          p !pd !fl !send !v !st; WaitForDisableExit[p]
     []   [IsAB(pd) and (r eq TDISPar(pd)) ] -> p !pd !fl !send !v !st; SPConAA[s,p]
     []   [not ( IsAB(pd) or (IsDN(pd) and (st eq 0)) or (IsRF(pd) and (nr eq TDISPar(pd))) )] ->
          p !pd !fl !send !v !st; SPCon[s,p]
     []   p !pd !fl ?d:Dir !v !st [not(d eq send)]; SPCon[s,p]
     )
[]   s ?sa:SAddress ?si:SCEI ?ssp:SSP; SPCon[s,p]
endproc
```

```
process SPConAA[s,p]: exit :=
choice pd:ASPDU,fl:TFlow,v:Validity,st:TrState []
(     [Valid(v) and (IsAA(pd) or (IsAB(pd) and (r eq TDISPar(pd)))) ] -> p !pd !fl !rec !v !st; SPCon[s,p]
[]    [not( Valid(v) and (IsAA(pd) or ( IsAB(pd) and (r eq TDISPar(pd)) )) )] ->
      p !pd !fl !rec !v !st; SPConAA[s,p]
[]    p !pd !fl ?d:Dir !v !st [not(d eq rec)]; SPConAA[s,p]
)
endproc
process WaitForDisableExit[p]: noexit :=
p ?p:ASPDU ?f:TFlow ?d:Dir ?s:Validity ?st:TrState; WaitForDisableExit[p]
endproc
(*----------------------------------------------------------------------------------------------------
```

10.3.2 Constraints on transformations between SSPs and SPDUs

Besides transforming SSPs to SPDUs and vice versa, *SSPSPDU* takes care of the the following constraints:

- a valid incoming GTC SPDU will generate an outgoing GTA SPDU;
- an invalid incoming SPDU could generate an outgoing AB SPDU (indicating a protocol error) and will lead to the execution of an S-P-ABORT indication if the SC is still in progress;
- the Maximum TSDU Size parameter value for the sending flow is monitored in order to perform segmenting (if selected) correctly;
- a valid incoming MAA or AEA SPDU will lead to the execution of either a S-SYNCHRONIZE-MAJOR or a S-ACTIVITY-END confirm;
- certain incoming or outgoing SPDUs cause stored PR SPDUs to be dequeued. In certain cases, the number and/or prepare type of the stored SPDUs are invalid. In such a case, if the dequeueing event was an incoming SPDU, the SSP corresponding to the incoming SPDU is executed first. Subsequently, a S-P-ABORT indication could be issued and an AB SPDU transmitted.

DownFlow models the transformation of SSPs to SPDUs. *UpFlow* defines the relationship between received SPDUs and the corresponding SSPs. Certain incoming SPDUs lead to the generation of an outgoing SPDU. This is modelled in process *TwoWay*. *OctetFlow* deals with data octets of S-DATA and S-TYPED-DATA primitives, DT and TD SPDUs, and segmenting.

```
----------------------------------------------------------------------------------------------------*)
process SSPSPDUTran[s,p]: noexit :=
( DownFlow[s,p] || UpFlow[s,p](NULL) || OctetFlow[s,p] ) |[p]| TwoWay[p]
endproc
(*----------------------------------------------------------------------------------------------------
```

The Session Protocol does not view SSPs that may carry an unbounded amount of user data (i.e. S-DATA and S-TYPED-DATA primitives) as atomic, but as consisting of data octets. This allows for the convenient description of segmenting and the possibility of starting the transmission of DT or TD SPDUs even if the complete SSDU has not yet been received. Furthermore, the receipt or delivery of SSDUs can then be interrupted by a disruptive SSP or SPDU (e.g a S-ABORT request or AD SPDU). The data type below introduces the functions that enrich the SSP type definition in order to represent this refined view. *SDATO* and *STDO* stand for session normal data octet and session typed data octet respectively. *SDATE* and *STDE* indicate the end of a sequence of data octets corresponding to a single SSDU.

```
--------------------------------------------------------------------------------------------------------*)
type SessionDataOctets is SessionServicePrimitive,Octet
opns
IsSDATO,IsSTDO,IsSDATE: ssp -> Bool            SDATEreq,STDEreq,SDATEind,STDEind: -> ssp
SDATOreq,STDOreq,SDATOind,STDOind: Octet -> ssp
endtype
(*--------------------------------------------------------------------------------------------------------
```

The synchronized composition of *DownFlow* with other processes implies that it must synchronize with all events (at the synchronization gates, *s* and *p*) that are specified in the other processes, even though *DownFlow* itself may not always add additional constraints to these events. In such cases the events are included to prevent the specification of unintended deadlock. This is a characteristic of the constrained-oriented style of specification. The same remark applies to the other subprocesses of *SSPSPDUTran*.

```
--------------------------------------------------------------------------------------------------------*)
process DownFlow[s,p]: noexit :=
     choice ssp:SSP [] [not(IsSDATO(ssp) or IsSDATE(ssp)) and IsReq(ssp)] ->
     s ?sa:SAddress ?si:SCEI !ssp;
     (     [PREP_Needed(ssp)] ->
           p ?pd:ASPDU ?f:TFlow ?d:Dir ?v:Validity ?st:TrState [not(d eq rec) and (pd PREP ssp)]; exit
     []    [not( PREP_Needed(ssp) )] -> exit
     ) >>
     (     p ?p:ASPDU ?f:TFlow ?d:Dir ?v:Validity ?st:TrState
           [not (d eq rec) and (p IsPDUOf ssp)]; DownFlow[s,p]
     )
[]   p ?p:ASPDU ?f:TFlow ?d:Dir ?v:Validity ?st:TrState
           [(d eq rec) or IsPAB(p) or IsAA(p) or IsGTA(p) or IsDT(p) or IsTD(p)]; DownFlow[s,p]
[]   (     choice ssp:SSP [] [IsInd(ssp) or IsSDATO(ssp) or IsSDATE(ssp)] ->
           s ?sa:SAddress ?si:SCEI !ssp; DownFlow[s,p]   )
endproc
(*--------------------------------------------------------------------------------------------------------
```

SPDUs which are meant to be ignored do not cause a SSP to be executed. Invalid incoming SPDUs could prompt the execution of a S-P-ABORT indication unless the SC has already ceased to exist. Absence of a SC is indicated by transfer state value *0*. When a valid incoming RS SPDU is received, and there are too many queued PR SPDUs, then bit 17 of the SPDU status parameter is reset. The S-RESYNCHRONIZE indication is subsequently executed, possibly followed by the transmission of a provider generated AB (specified in *TwoWay*) and the execution of a S-P-ABORT indication.

```
--------------------------------------------------------------------------------------------------------*)
process UpFlow[s,p](expds:ASPDUs): noexit :=
     choice pd:ASPDU,fl:TFlow,v:Validity,st:TrState []
     (     p !pd !fl ?d:Dir !v !st [not(d eq rec)];
           (     [IsPAB(pd)] -> s ?sa:SAddress ?si:SCEI ?sp:SSP [sp IsSSPOf pd]; UpFlow[s,p](NULL)
           []    [not(IsPAB(pd))] -> UpFlow[s,p](expds)    )
     []    [(IsMAA(pd) or IsAEA(pd)) and ValidDel(v)] -> p !pd !fl !rec !v !st;
           (     choice ssp:SSP [] [(Set(v,15) iff IsSACTEcnf(ssp)) and (ssp IsSSPOf pd)] ->
                 s ?sa:SAddress ?si:SCEI !ssp; UpFlow[s,p](expds)    )
     []    [ValidDel(v) and not(IsMAA(pd) or IsAEA(pd) or IsDT(pd) or IsEX(pd))] ->
           p !pd !fl !rec !v !st; s ?sa:SAddress ?si:SCEI ?ssp:SSP [ssp IsSSPOf pd];
           (     [DequeuingPDUs(pd)] -> ( DeliverEX[s,p](expds) >> UpFlow[s,p](NULL) )
           []    [not(DequeuingPDUs(pd))] -> UpFlow[s,p](expds)    )
     []    [ValidDel(v) and IsEX(pd)] -> p !pd !fl !rec !v !st;
           (     [Set(v,17)] -> UpFlow[s,p](pd+expds)
           []    [Reset(v,17)] -> s ?sa:SAddress ?si:SCEI ?ssp:SSP [ssp IsSSPOf pd]; UpFlow[s,p](expds)
           )
     []    [not(ValidDel(v)) or IsDT(pd)] -> p !pd !fl !rec !v !st; UpFlow[s,p](expds)
     )
```

```
[]      choice sp:SSP [] [IsReq(sp) or IsSDATO(sp) or IsSDATE(sp)] ->
        s ?sa:SAddress ?si:SCEI !sp; UpFlow[s,p](expds)
where
process DeliverEX[s,p](expds: ASPDUs): exit :=
        [expds eq NULL] -> exit
[]      [expds ne NULL] ->
        (       s ?sa:SAddress ?si:SCEI ?ssp:SSP [ssp IsSSPOf First(expds)]; DeliverEX[s,p](Rest(expds))
        []      p ?p:ASPDU ?f:TFlow ?d:Dir ?v:Validity ?st:TrState [not(d eq rec)]; DeliverEX[s,p](expds)   )
endproc
endproc (* UpFlow *)

process TwoWay[p]: noexit :=
choice pd:ASPDU,fl:TFlow,v:Validity,st:TrState []
(       [IsGTC(pd) and ValidDel(v)] -> p !pd !fl !rec !v !st;
        p ?p:ASPDU ?fl1:TFlow !send ?v1:Validity ?st1:TrState [IsGTA(p)]; TwoWay[p]
[]      [IsAB(pd) and Valid(v)] -> p !pd !fl !rec !v !st;
        p ?p: ASPDU ?fl1:TFlow !send ?v1:Validity ?st1:TrState [IsAA(p)]; TwoWay[p]
[]      [Valid(v) implies (DequeuingPDUs(pd) and Set(v,17))] -> p !pd !fl !rec !v !st;
        (       choice p:ASPDU,fl2:TFlow,v2:Validity,st2:TrState []
                [IsPAB(p)] -> p !p !fl2 !send !v2 !st2; TwoWay[p] [] i (* ignore erroneous SPDU *); TwoWay[p]   )
[]      [not( (Valid(v) and IsAB(pd)) or (IsGTC(pd) and ValidDel(v)) or
        (Valid(v) and DequeuingPDUs(pd) and Set(v,17)) )] ->
        p !pd !fl !rec !v !st; TwoWay[p]
[]      [not(IsPAB(pd) or IsGTA(pd) or IsAA(pd))] -> p !pd !fl ?d:Dir !v !st [not (d eq rec)]; TwoWay[p]
)
endproc
(*------------------------------------------------------------------------------------------------------------
```

In *OctetFlow* the Maximum TSDU Size value for the sending flow in support of the present SC is determined. *MtsRecFlow* and *MtsSendFlow* yield the Maximum TSDU Size value for the receiving and sending flow, respectively, of the SPM which had sent the CN or AC SPDU. Process *OctFlo*, parameterized with this value, then describes the subsequent behaviour.

```
------------------------------------------------------------------------------------------------------------*)
process OctetFlow[s,p]: noexit :=
choice pd:ASPDU,fl:TFlow,v:Validity,st:TrState []
(       [IsCN(pd) and ValidDel(v)] ->
        p !pd !fl !rec !v !st; ( choice mt:Nat [] [mt eq MtsRecFlow(pd)] -> OctFlo[s,p](mt) )
[]      [IsCN(pd)] -> p !pd !fl !send !v !st; ( choice mt:Nat [] [mt eq MtsSendFlow(pd)] -> OctFlo[s,p](mt) )
[]      [not(IsCN(pd))] -> p !pd !fl ?d:Dir !v !st [(d eq rec) implies ValidDel(v)]; OctetFlow[s,p] )
)
endproc
```

```
process OctFlo[s,p](mOld:Nat): noexit :=
choice pd:ASPDU,fl:TFlow,v:Validity,st:TrState,ssp:SSP,oct:Octet,mt:Nat []
(    [IsCN(pd) and ValidDel(v) and (mt eq MtsRecFlow(pd))] -> p !pd !fl !rec !v !st; OctFlo[s,p](mt)
[]   [IsCN(pd) and (mt eq MtsSendFlow(pd))] -> p !pd !fl !send !v !st;  OctFlo[s,p](mt)
[]   [IsAC(pd) and ValidDel(v) and (mt eq MtsRecFlow(pd))] -> p !pd!fl!rec!v!st; OctFlo[s,p](Min(mOld,mt))
[]   [IsAC(pd) and (mt eq MtsSendFlow(pd))] -> p !pd !fl !send !v !st; OctFlo[s,p]( Min(mOld,mt) )
[]   s ?sa:SAddress ?si:SCEI !SDATOreq(oct); SendSDO[s,p](Octet(oct) of DatOctStr,mOld,true)
[]   s ?sa:SAddress ?si:SCEI !STDOreq(oct); SendSTDO[s,p](Octet(oct) of DatOctStr,mOld,true)
[]   [IsDT(pd) and ValidDel(v)] -> p !pd !fl !rec !v !st;
     (    choice dt:DatOctStr,encit:EncIt [] [(dt eq DaDs(UserInf(pd))) and (encit eq EncItem(pd))] ->
          DeliverDT[s,p](dt,encit) >> OctFlo[s,p](mOld)   )
[]   [IsTD(pd) and ValidDel(v)] -> p !pd !fl !rec !v !st;
     (    choice dt:DatOctStr,encit:EncIt [] [(dt eq DaDs(UserInf(pd))) and (encit eq EncItem(pd))] ->
          DeliverTD[s,p](dt,encit) >> OctFlo[s,p](mOld)   )
[]   [not(IsSDATO(ssp) or IsSTDO(ssp))] -> s ?sa:SAddress ?si:SCEI !ssp; OctFlo[s,p](mOld)
[]   [not(IsAC(pd) or IsCN(pd) or IsDT(pd) or IsTD(pd))] ->
     p !pd !fl ?d:Dir !v !st [not (d eq rec)]; OctFlo[s,p](mOld)
[]   [not(ValidDel(v) and (IsDT(pd) or IsCN(pd) or IsAC(pd) or IsTD(pd)))] ->
     p !pd !fl !rec !v !st; OctFlo[s,p](mOld)
)
endproc
(*-----------------------------------------------------------------------------------------------------------
```

SendSDO describes the processing of normal data octets until a complete SSDU is transmitted, either as a single SPDU, or as a sequence of SPDUs if segmenting is allowed. *SendSTDO* does the same for typed data octets. The way segmenting is carried out (i.e. the way in which data octets are combined into several DT or TD SPDUs) depends on the value of the maximum TSDU size, *mt*, but apart from that is non-deterministic. Processing of octet requests can be interrupted, either by a disruptive SSP (e.g. a S-U-ABORT request) or by a disruptive incoming SPDU (e.g. an AI SPDU).

NOTE - The total length of the encoded DT SPDU is the sum of the number of data octets and the number of octets for encoding the SI and LI fields of the SPDU (2 octets) and the PI, LI and PV fields of the Enclosure Item parameter (3 octets). This total length should not exceed the TSDU Maximum Size value for the sending flow.

DeliverDT describes the execution of a S-DATA indication as the delivery of a sequence of normal data octets. It has two parameters: one representing a sequence of S-DATA-OCTET indications and the other the value of the Enclosure Item parameter. Analogous to *SendSDO*, *DeliverDT* can be interrupted by a disruptive event caused either by the SS-user or by an incoming SPDU. *DeliverTD* describes the execution of a S-TYPED-DATA indication in a similar way.

```
-----------------------------------------------------------------------------------------------------------*)
process SendSDO[s,p](str:DatOctStr,mt:Nat,first:Bool): noexit :=
     s ?sa:SAddress ?si:SCEI !SDATEreq;
     (    choice et:EncIt,sdat:SData []
          [(mt eq 0) and (et eq absent) or ((mt ne 0) and (et eq end)) and (str eq DaDs(sdat))] ->
          p !DT(et,sdat) !normal ?d:Dir ?v:Validity ?st:TrState [not(d eq rec)]; OctFlo[s,p](mt)
     )
[]   choice oct:Octet,ei:EncIt,sdat:SData []
     [first and (ei eq begin) and (not(first) and (ei eq middle)) and (str eq DaDs(sdat))] ->
     s ?sa:SAddress ?si:SCEI !SDATOreq(oct);
     (    let size:Nat = (Length(oct+str) + NatNum(Dec(5))) in (* NOTE *)
          (    [size le mt or (mt eq 0)] -> SendSDO[s,p](oct+str,mt,first)
          []   [mt ne 0] -> i; p !DT(ei,sdat) !normal ?d:Dir ?v:Validity ?st:TrState [not (d eq rec)];
               SendSDO[s,p](Octet(oct) of DatOctStr,mt,false)
     )    )
[]   ( Interrupt_OFlo[s,p] >> OctFlo[s,p](mt) )
endproc
```

```
process SendSTDO[s,p] (str:DatOctStr,mt:Nat,first:Bool): noexit :=
     s ?sa:SAddress ?si:SCEI !STDTEreq;
     (    choice et:EncIt [] [(mt eq 0) and (et eq absent) or ((mt ne 0) and (et eq end))] ->
          p !ASPDU(dt,TD(et,str)) !normal ?d:Dir ?v:Validity ?st:TrState [not (d eq rec)];
          OctFlo[s,p](mt)
     )
[]   (    choice oct:Octet,ei:EncIt [] [first and (ei eq begin) and (not(first) and (ei eq middle))] ->
          s ?sa:SAddress ?si:SCEI !STDTOreq(oct);
          (    let str2:DatOctStr = (oct+str) in
               (    let size:Nat = (Length(str2) + NatNum(Dec(5))) in
                    (    [size le mt or (mt eq 0)] -> SendSTDO[s,p](str2,mt,first)
                    []   [mt ne 0] -> i;
                         p !ASPDU(td,TD(ei,str)) !normal ?d:Dir v:Validity ?st:TrState
                              [not (d eq rec)];
                         SendSTDO[s,p](String(oct),mt,false)
     )    )    )    )
[]   Interrupt_OFlo[s,p] >> OctFlo[s,p](mt)
endproc

process DeliverDT[s,p](str:DatOctStr,et:EncIt): exit :=
     choice sa:SAddress,si:SCEI,ssp:SSP,oct:Octet,s2:DatOctStr []
     (    [str eq empty and (et eq absent or (et eq end))] -> s !sa !si !SDATEind; exit
     []   [str ne empty and (str eq (oct+s2))] -> s !sa !si !SDATOind(oct); DeliverDT[s,p](s2,et)
     )
[]   Interrupt_OFlo[s,p]
endproc

process DeliverTD[s,p](str:DatOctStr,et:EncIt): exit :=
     choice sa:SAddress,si:SCEI,ssp:SSP,oct:Octet,s2:DatOctStr []
     [(str eq empty) and (et eq absent or (et eq end))] -> s !sa !si !STDEind; exit
[]   [(str ne empty) and (str eq (oct+s2))] -> s !sa !si !STDOind(oct); DeliverTD[s,p](s2,et)
[]   Interrupt_OFlo[s,p]
endproc

process Interrupt_OFlo[s,p]: exit :=
choice pd:ASPDU,ssp:SSP []
(    [IsSUABreq(ssp) or (ssp eq SPABind(transportdisconnect)) or IsSRSYNreq(ssp) or
     IsSACTDreq(ssp) or IsSACTIreq(ssp)] -> s ?sa:SAddress ?si:SCEI !ssp; exit
[]   [IsAB(pd) or IsRS(pd) or IsAD(pd) or IsAI(pd) or IsER(pd) or IsED(pd) or (res eq PrepType(pd))] ->
     p !pd ?fl:TFlow !rec ?v:Validity ?st:TrState [ValidDel(v)]; exit
[]   [IsER(pd) or IsAB(pd)] -> p !pd ?fl:TFlow !send ?v:Validity ?st:TrState; exit
)
endproc
(*-----------------------------------------------------------------------------------------------------
```

10.4 Relation between SPDUs and TSPs

The relationship between SPDUs and TSPs is described by the composition of three processes that can be disabled by the occurrence of a T-DISCONNECT indication. *SPDUTSP1* models the mappings between SPDUs and TSPs. Constraints related to "Vtca" and the availability of the transport expedited flow are imposed on the receipt and/or transmission of certain SPDUs by *TCONVar* (see 10.4.4). Process *Timer* (see 10.4.5) models the timer employed in the state tables of ISO 8327.

```
-----------------------------------------------------------------------------------------------------*)
process SPDUTSP[p,t]: exit :=
( SPDUTSP1[p,t] |[p]| TCONVar[p,t] |[p]| Timer[p,t] )
[> t ?ta:TAddress ?ti:TCEI ?tdis:TSP [IsTDISind(tdis)]; exit
endproc
(*-----------------------------------------------------------------------------------------------------
```

SPDUTSP1 consists of three synchronizing processes:

- *SPDUtoTSP* (see 10.4.1) deals with the transformation of SPDUs to TSPs. The concatenation of SPDUs is taken care of in this process;
- *TSPtoSPDU* (see 10.4.2) deals with the transformation of TSPs to SPDUs. Separation is handled in this process. It also checks if a received (sequence of) SPDU(s) satisfies the maximum TSDU size requirements and ensures that an incoming extended concatenated sequence of SPDUs is considered to be valid only if the SPM supports extended concatenation; and
- *Int_BP* (see 10.4.3) models internal backpressure.

```
---------------------------------------------------------------------------------------------------------*)
process SPDUTSP1[p,t]: noexit :=
TSPtoSPDU[p,t] |[p]| SPDUtoTSP[p,t](false,0 of Nat,NULL) |[p]| Int_BP[p]
endproc
(*---------------------------------------------------------------------------------------------------------
```

10.4.1 Constraints on transformations from TSPs to SPDUs

The constraints on transformations from TSPs to SPDUs are described in two separate processes, viz. *TDTtoSPDU* for the T-DATA indications, and *TEXtoSPDU* for the T-EXPEDITED-DATA indications. *RPVal* is composed in conjunction with the latter two processes in order to be able to constrain the value of the Reflect Parameter Values parameter value of an ER SPDU that is transmitted following receipt of an invalid SPDU.

```
---------------------------------------------------------------------------------------------------------*)
process TSPtoSPDU[p,t]: noexit := ( TDTtoSPDU[p,t](0 of Nat,false) ||| TEXtoSPDU[p,t] ) || RPVal[p,t]
endproc
(*---------------------------------------------------------------------------------------------------------
```

TDTtoSPDU checks whether all constraints which are related to valid concatenation, segmenting, maximum TSDU size as well as encoding are satisfied in T-DATA indications. *TEXtoSPDU* checks whether the incoming SPDUs have been correctly encoded in T-EXPEDITED-DATA indications.

If an extended concatenated sequence of SPDUs is received while no extended concatenation is allowed, if invalid segmenting is detected, or if the maximum TSDU size is exceeded, this will be signalled by resetting certain bits in the SPDU status component (see 10.2.1). The function *ErrCodStr* is used to determine the validity of encodings (see 9.1.4.3).

Even if the TC is not to be re-used, the SPM may decide, as a local implementation matter, either to send an AA SPDU upon receipt of an AB SPDU, or to execute a T-DISCONNECT request (described in *ExecTDr* below). This is not included in the state tables of ISO 8327 but is mentioned in the accompanying text (clause 7.8.2). Following transmission of the AA SPDU, a T-DISCONNECT request is executed (see also 10.4.2).

ValSegm checks whether DT and TD SPDUs satisfy certain requirements related to segmenting, e.g. a DT SPDU is invalid if it carries an Enclosure Item parameter when no segmenting has been agreed on.

ReceivePDUs takes care of separation and executes a T-DISCONNECT request if necessary. If the received octet string constitutes a badly encoded SPDU, this is indicated by resetting bit 1 of the SPDU status component. Otherwise, *NextPDU* determines the next SPDU to be delivered. If the received sequence of SPDUs satisfies all constraints w.r.t. concatenation and segmenting (bit 0 of the SPDU status component is set to *1*), the remaining SPDUs are delivered, otherwise process *ReceivePDUs* terminates successfully.

```
-----------------------------------------------------------------------------------------------------*)
process TDTtoSPDU [p,t](MTRF:Nat,XConc:Bool): noexit :=
choice str:OctetString [] t ?ta:TAddress ?ti:TCEI !TDTind(str);
(     choice pds:ASPDUs [] [Decode(DaDs(str)) eq pds] ->
      (ValSegm[p](MTRF) || ReceivePDUs[p,t](pds) || CONCandTSDUSize[p](pds,DaDs(str),MTRF,XConc))
      >> accept mt:Nat,ExtConc:Bool in TDTtoSPDU[p,t](mt,ExtConc)
)
endproc

process TEXtoSPDU[p,t]: noexit :=
choice dstr:OctetString [] t ?ta:TAddress ?ti:TCEI !TEXind(dstr);
(     choice pds: ASPDUs,v:Validity,st:TrState,num:Nat [] [Empty(ErrCodStr(DaDs(dstr))) iff Set(v,1)] ->
      (     [Reset(v,1)] -> p ?pd:ASPDU !exp !rec !v !st [IsDUM(pd)]; TEXtoSPDU[p,t]
      []    [(Decode(DaDs(dstr)) eq pds) and (num eq Number(pds)) and
            ( (num eq Succ(0)) iff Set(v,0) ) and Set(v,1)] ->
            (     [Set(v,0)] -> p ?pd:ASPDU !exp !rec !v !st [ASPDUs(pd) eq pds];
                  (     [IsTDr(v) and IsAB(pd)] -> OptionAA[p,t]
                  []    [IsTDr(v) and not(IsAB(pd))] -> ExecTDr[t]
                  []    [not(IsTDr(v))] -> exit
                  ) >> TEXtoSPDU[p,t]
            []    [Reset(v,0)] -> p ?pd:ASPDU !exp !rec !v !st [IsDUM(pd)]; TEXtoSPDU[p,t]
)     )     )
where
process OptionAA[p,t]: exit := i; p !AA ?f:TFlow !send ?v:Validity ?st:TrState; stop  []  ExecTDr[t] endproc
endproc (* TEXtoSPDU *)

process ValSegm[p](mtrf:Nat): exit(Nat,Bool) :=
choice p:ASPDU,fl:TFlow,d:Dir,v:Validity,st:TrState []
(     [(d eq rec) implies not(IsTD(p) or IsDT(p))] -> p !p !fl !d !v !st; ValSegm[p](mtrf)
[]    [IsTD(p) or IsDT(p) and (ValidSegmenting(mtrf,p) iff Set(v,0))] -> p !p !fl !rec !v !st; ValSegm[p](mtrf)
[]    exit(any Nat,any Bool)
)
endproc

process ReceivePDUs[p,t](pds: ASPDUs): exit(Nat,Bool) :=
choice pd:ASPDU,v:Validity,st:TrState []
(     [(ASPDUs(pd) eq pds) and IsDUM(pd) and Reset(v,1)] -> p!pd!normal!rec!v!st; exit(any Nat,any Bool)
[]    [Number(pds) ne Succ(0) or ( ASPDUs(pd) eq pds and not(IsDUM(pd)) )] ->
      (     NextPDU(pds) >> accept pdu:ASPDU, pdus:ASPDUs in
            (     choice v:Validity,st:TrState [] [Set(v,1)] ->
                  (     [Reset(v,0)] -> p ?pd:ASPDU !normal !rec !v !st; exit(any Nat,any Bool)
                  []    [Set(v,0)] -> p !pdu !normal !rec !v !st;
                        (     [not(IsTDr(v)) and (pdus ne NULL)] -> ReceivePDUs[p,t](pdus)
                        []    [not(IsTDr(v)) and (pdus eq NULL)] -> exit(any Nat,any Bool)
                        []    [IsTDr(v)] -> ( ExecTDr[t] >> exit(any Nat,any Bool) )
)     )     )     )     )
[]    p ?pdu:ASPDU ?fl:TFlow ?d:Dir ?v:Validity ?st:TrState [not(d eq rec)]; ReceivePDUs[p,t](pds)
endproc
(*----------------------------------------------------------------------------------------------------
```

NextPDU determines the processing order of SPDUs according to subclause 6.3.7.1 of ISO 8327. If the first SPDU of a concatenated sequence is a GT or PT SPDU with no Token Item parameter, this "dummy" is removed before the "next" SPDU is determined.

```
----------------------------------------------------------------------------------------------------------*)
process NextPDU(pds1:ASPDUs): exit(ASPDU,ASPDUs) :=
let strip:Bool = DummyPTorGT(First(pds1)) in
(       choice pds:ASPDUs [] [strip and (pds eq Rest(pds1) or (not(strip) and (pds eq pds1)) )] ->
        (       let n:Nat = Number(pds), p1:ASPDU = First(pds), p2:ASPDU = Second(pds),
                p3:ASPDU = Third(pds), p4:ASPDU = Fourth(pds) in
                (       choice pdus:ASPDUs []
                        (       [n eq NatNum(Dec(1))] -> exit(p1,NULL)
                        []      [n eq NatNum(Dec(2))] -> exit(p2,ASPDUs(p1))
                        []      [n eq NatNum(Dec(3))] ->
                                (       [(IsAR(p2) or IsAS(p2)) and (pdus eq Concatenate(p1,ASPDUs(p3)))] ->
                                        exit(p2,pdus)
                                []      [not(IsAR(p2) or IsAS(p2)) and (pdus eq Concatenate(p1,ASPDUs(p2)))] ->
                                        exit(p3,pdus)
                                )
                        []      [(n eq NatNum(Dec(4))) and
                                (pdus eq Concatenate(p1,Concatenate(p3,ASPDUs(p4))))] -> exit(Second(pds),pdus)
)       )       )       )
endproc
(*----------------------------------------------------------------------------------------------------------
```

Process *CONCandTSDUSize* checks whether the length of the received octet string exceeds the negotiated maximum and whether the concatenated sequence is valid. The latter depends on the value of the Protocol Options parameter in the CN (initiator side) or AC (responder side) SPDU.

```
----------------------------------------------------------------------------------------------------------*)
process CONCandTSDUSize[p](pds:ASPDUs,str:DatOctStr,mtrf:Nat,XConc:Bool): exit(Nat,Bool) :=
choice pdu:ASPDU,fl:TFlow,v:Validity,st:TrState []
(       [Reset(v,1)] -> p !pdu !fl !rec !v !st; exit(mtrf,XConc)
[]      [Set(v,1) and
        ((Length(str) le mtrf) and IsValidConc(pds) and (IsXConcat(pds) implies XConc) iff Set(v,0))] ->
        (       [Set(v,0)] -> SPDUExchange[p](mtrf,XConc)
        []      [Reset(v,0)] -> p ?pd:ASPDU !fl !rec !v !st [IsDUM(pd)]; exit(mtrf,XConc)
        )
[]      p !pdu !fl ?d:Dir !v !st [not(d eq rec)]; CONCandTSDUSize[p](pds,str,mtrf,XConc)
)
endproc
(*----------------------------------------------------------------------------------------------------------
```

In *SPDUExchange*, new values for the maximum TSDU size and the extended concatenation option are determined from the CN and AC SPDUs which are exchanged. For the TD and DT, bit 0 is set or reset in process *ValSegm*. Thus, this bit is not constrained in *SPDUExchange*.

```
-------------------------------------------------------------------------------------------------------*)
process SPDUExchange[p](MTRF:Nat,XConc:Bool): exit(Nat,Bool) :=
    choice pd:ASPDU,fl:TFlow,v:Validity,st:TrState,d:Dir []
    (   [not(IsTD(pd) or IsDT(pd)) and Set(v,0)] ->
        (   [d eq rec and ValidDel(v) implies not(IsCN(pd) or IsAC(pd))] ->
            p !pd !fl !d !v !st; SPDUExchange[p](MTRF,XConc)
        []  [IsCN(pd)] -> p !pd !fl !send !v !st; SPDUExchange[p](MtsRecFlow(pd),ProtOptions(pd))
        []  [IsAC(pd)] -> p !pd !fl !send !v !st;
            ( let Mts:Nat = Min(MtsRecFlow(pd),MTRF) in SPDUExchange[p](Mts,ProtOptions(pd)) )
        []  [IsCN(pd) and ValidDel(v)] ->
            p !pd !fl !rec !v !st; SPDUExchange[p](MtsSendFlow(pd),XConc)
        []  [IsAC(pd) and ValidDel(v)] -> p !pd !fl !rec !v !st;
            ( let Mts:Nat = Min(MtsSendFlow(pd),MTRF) in SPDUExchange[p](Mts,XConc) )
        )
    []  [IsTD(pd) or IsDT(pd)] -> p !pd !fl !d !v !st; SPDUExchange[p](MTRF,XConc)
    )
[]  exit(MTRF,XConc)
endproc

process ExecTDr[t]: exit := t ?ta:TAddress ?ti:TCEI ?tdr:TSP [IsTDISreq(tdr)]; exit endproc
(*-------------------------------------------------------------------------------------------------------
```

RPVal determines the value of the Reflect Parameter Values parameter of an ER SPDU to be transmitted. If a received octet string is an invalid encoding, the erroneous string is transmitted up to and including the invalid octet. If the received string represents an invalid concatenation or if the string is too long, the whole string is transmitted. If a received SPDU is determined to be invalid by *ValSegm* or by any process outside *TSPtoSPDU*, the complete encoded SPDU is transmitted.

```
-------------------------------------------------------------------------------------------------------*)
process RPVal[p,t]: noexit :=
( RPVN[p,t](empty) |[p]| RPVE[p,t](empty) || RPVD[p] )
[> t ?ta:TADDR ?ti:TCEI ?tsp:TSP [IsTDISreq(tsp)]; RPVal[p,t]
endproc
(*-------------------------------------------------------------------------------------------------------
```

If an incoming SPDU is found to be correctly encoded, as part of a correctly concatenated SPDU sequence, it is offered as a non-dummy SPDU at gate *p*. It is still possible that the SPDU is regarded invalid by one or more of the other synchronizing processes; in such a case the SPM may decide to transmit an ER SPDU containing the octetstring that is the encoding of the invalid SPDU. *RPVN* ensures that the Reflect Parameter Values parameter of the transmitted ER SPDU contains this octetstring. In case the next action, following receipt on the transport normal flow of an invalid non-dummy SPDU, is not the transmission of an ER SPDU, it is assumed that the SPM has decided not to transmit an ER PDU. *RPVE* is analogous to *RPVN*, except that it handles SPDUs received on the transport expedited flow.

```
-------------------------------------------------------------------------------------------------------*)
process RPVN[p,t](nstr:DatOctStr): noexit :=
        ( choice oct:Octet [] t ?ta:TADDR ?ti:TCEI !TDTOInd(oct); RPVN[p,t](nstr--oct) )
[]      t ?ta:TADDR ?ti:TCEI !TDTEind; RPVN[p,t](nstr)
[]      (       choice pd:ASPDU,fl:TFlow,d:Dir,v:Validity,st:TrState,str1,str2:DatOctStr []
                (       [(d eq rec) implies (fl eq exp)] -> p !pd !fl !d !v !st; RPVN[p,t](nstr)
                []      [IsDUM(pd)] -> p !pd !normal !rec !v !st; RPVN[p,t](empty)
                []      [not(IsDUM(pd)) and (str1 IsEncOf pd) and (nstr eq (str1++str2))] ->
                        p !pd !normal !rec !v !st; RPVN1[p](str1,str2) >> RPVN[p,t](empty)
        )       )
endproc

process RPVN1[p](OldStr,NewStr:DatOctStr): exit :=
choice pd:ASPDU,f:TFlow,d:Dir,v:Validity,st:TrState []
(       p !ASPDU(er,ER(OldStr)) !f !send !v !st; exit
[]      [(d ne rec) and not(IsER(pd))] -> p !pd !f !d !v !st;
        (       [not(Empty(NewStr))] -> RPVN1[p](OldStr,NewStr)
        []      [Empty(NewStr)] -> exit    )
[]      (       choice st1,st2:DatOctStr [] [(st1 IsEncOf pd) and (NewStr eq (st1++st2))] ->
                p !pd !normal !d !v !st;  RPVN1[p](st1,st2)
        []      p !pd !exp !d !v !st; RPVN1[p](OldStr,NewStr)
)       )
endproc

process RPVE[p,t](estr:DatOctStr): noexit :=
        ( choice dstr:DatOctStr [] t ?ta:TADDR ?ti:TCEI !TEXind(dstr); RPVE[p,t](dstr) )
[]      (       choice pd:ASPDU,fl:TFlow,d:Dir,v:Validity,st:TrState []
                (       [(d eq rec) implies (fl eq normal)] -> p !pd !fl !d !v !st; RPVE[p,t](estr)
                []      [IsDUM(pd)] -> p !pd !exp !rec !v !st; RPVE[p,t](empty)
                []      [not IsDUM(pd)] -> p !pd !exp !rec !v !st; RPVE1[p,t](estr,empty)
        )       )
endproc

process RPVE1[p,t](OldStr,NewStr:DatOctStr): noexit :=
        (       choice pd:ASPDU,fl:TFlow,d:Dir,v:Validity,st:TrState []
                [(IsER(pd) and (d eq send)) implies (OldStr RPV pd)] -> p !pd !fl !d !v !st; RPVE[p,t](NewStr)    )
[]      ( choice dstr:DatOctStr [] t ?ta:TADDR ?ti:TCEI !TEXind(dstr); RPVE1[p,t](OldStr,dstr) )
endproc
(*-----------------------------------------------------------------------------------------------------
```

In case one of the subprocesses of *TSPtoSPDU* can decide that a SPDU is invalid, a dummy SPDU is offered at gate *p* containing the erroneous octetstring. The SPM may decide, upon receipt of such a SPDU, to transmit an ER SPDU with the Reflect Parameter Values parameter containing the invalid octetstring. This is ensured by *RPVD*. It is assumed that if the next action is not the transmission of an ER SPDU, the SPM has decided not to transmit the ER SPDU.

```
-------------------------------------------------------------------------------------------------------*)
process RPVD[p]: noexit :=
        choice pd:ASPDU,fl:TFlow,d:Dir,v:Validity,st:TrState,estr:DatOctStr [] [not(IsDUM(pd))] ->
        p !pd !fl !d !v !st; RPVD[p]
[]      p !ASPDU(dum,DUM(estr)) !fl !d !v !st;
        p ?p:ASPDU ?f:TFlow ?d:Dir ?v:Validity ?st:TrState
                [IsER(p) and (d eq send) implies (estr RPV p)]; RPVD[p]
endproc
(*-----------------------------------------------------------------------------------------------------
```

NOTE - In general, the following error cases can be distinguished:

(a) the size of the incoming user data is larger than the TSDU Maximum Size value for the receiving flow;

(b) the incoming user data constitutes a valid extended concatenation of SPDUs but the receiving SPM indicated its unwillingness to receive extended concatenated SPDUs;
(c) the incoming user data constitutes an invalid concatenation of SPDUs;
(d) a segmented SSDU is received while segmenting has not been agreed on;
(e) a "middle" SPDU of a segmented SSDU is received without SS-user data;
(f1) an SPDU is received on an incorrect transport flow (e.g., an AA SPDU that is received on the normal flow when the transport expedited flow was selected);
(f2) a concatenated sequence of SPDUs is received on the expedited flow;
(g) the encoding of an incoming SPDU is incorrect;
(h) the Enclosure Item parameter in a DT SPDU of a segmented SSDU is absent, or this parameter is present while segmenting was not agreed on for use on the connection.

These errors are detected in the different synchronizing processes. This illustrates the principle of the separation of concerns by which several processes handle separate aspects of the correctness of incoming SPDUs. Together they ensure that SPDUs which are passed "upwards" are correct as far as the above criteria are concerned.

10.4.2 Constraints on transformations from SPDUs to TSPs

The constraints on transformations from SPDUs to TSPs impose that the most recent outgoing SPDU is either concatenated to a sequence of stored SPDUs (represented by *ConcPDUs*) or transmitted separately after the transmission of the present concatenated sequence. Which alternative is chosen depends on the TSDU maximum size and the value of the extended concatenation option but is otherwise non-deterministic. Each class of outgoing SPDUs is handled in a separate subprocess of *SPDUtoTSP* (*SendPREP*, *SendABA*, etc.). These subprocesses may invoke process *SendConcPDUs* which takes care of appending a PT or GT SPDU to a concatenated sequence of SPDUs. The concatenated sequence is subsequently encoded according to the rules in clause 8 of ISO 8327.

```
---------------------------------------------------------------------------------------------------*)
process SPDUtoTSP[p,t](SendXConc:Bool,MTSF:Nat,ConcPDUs:ASPDUs): noexit :=
    (   (   SendPREP[p,t](SendXConc,ConcPDUs)
        []  SendABA[p,t]
        []  SendCAT21[p,t](SendXConc,ConcPDUs)
        []  SendPT[p,t](SendXConc,ConcPDUs,MTSF)
        []  SendGT[p,t](SendXConc,ConcPDUs,MTSF)
        []  Rest_of_PDUs1[p,t](SendXConc,ConcPDUs,MTSF)
        []  Rest_of_PDUs2[p,t](SendXConc,ConcPDUs,MTSF)
        ) >> accept CncPDs: ASPDUs in SPDUtoTSP[p,t](SendXConc,MTSF,CncPDs)
    )
[]  p ?pd:ASPDU ?fl:TFlow !no_send ?v:Validity ?st:TrState;
    SPDUtoTSP[p,t](SendXConc,MTSF,ConcPDUs)
[]  p ?pd:ASPDU ?fl:TFlow !rec ?v:Validity ?st:TrState [not(ValidDel(v) and (IsAC(pd) or IsCN(pd)))];
    (   [not(IsAB(pd) and IsTDr(v) and Valid(v))] -> SPDUtoTSP[p,t](SendXConc,MTSF,ConcPDUs)
    []  [IsAB(pd) and IsTDr(v) and Valid(v)] -> SendAA[p,t]   )
[]  ( RecACorCN[p](MTSF) >> accept Mts:Nat,SndXCnc:Bool in SPDUtoTSP[p,t](SndXCnc,Mts,NULL) )
[]  ( SendACorCN[p,t](MTSF) >> accept Mts:Nat in SPDUtoTSP[p,t](SendXConc,Mts,NULL) )
endproc
(*---------------------------------------------------------------------------------------------------
```

Process *SendAA* takes care of the optional transmission of an AA SPDU. Following this transmission, a T-DISCONNECT request is executed. In accordance to the state tables of ISO 8327, it is described that no SPDU can be received between the receipt of the AB SPDU and transmission of the AA SPDU.

```
------------------------------------------------------------------------------------------------------------*)
process SendAA[p,t]: noexit :=
choice pd:ASPDU,fl:TFlow,v:Validity,st:TrState,epd:OctetString [] [DaDs(epd) IsEncOf pd] ->
p !pd !fl !send !v !st;
(     [fl eq exp] -> t ?ta:TAddress ?ti:TCEI !TEXreq(epd); exit
[]    [fl eq normal] -> t ?ta:TAddress ?ti:TCEI !TDTreq(epd); exit
) >> t ?ta:TAddress ?ti:TCEI ?tsp:TSP [IsTDISreq(tsp)]; stop
endproc
(*-----------------------------------------------------------------------------------------------------------
```

Processes *RecACorCN* and *SendACorCN* deal with the valid exchanges of CN and AC SPDUs and determine the resulting maximum TSDU size for the TC. Also it is established whether or not the peer SPM is willing to accept an extended concatenated sequence of SPDUs.

The maximum TSDU size parameter in the CN or AC SPDU consists of two values, each applicable to one direction of flow. *MtsRecFlow* determines the value corresponding to the receiving flow of the SPM which sent the SPDU (here, the peer SPM). The value which will be applicable for the lifetime of the SC, is the minimum value of the values proposed by the SPM and its peer SPM.

```
------------------------------------------------------------------------------------------------------------*)
process RecACorCN[p](mts:Nat): exit(Nat,Bool) :=
choice pd:ASPDU,fl:TFlow,v:Validity,st:TrState,mt:Nat,XC:Bool []
[(XC eq ProtOptions(pd)) and (mt eq MtsRecFlow(pd))] ->
(     [IsCN(pd) and ValidDel(v)] -> p !pd !fl !rec !v !st; exit(mt,XC)
[]    [IsAC(pd) and ValidDel(v)] -> p !pd !fl !rec !v !st; exit(Min(mt,mts),XC)
)
endproc

process SendACorCN[p,t](mts:Nat): exit(Nat) :=
choice pd:ASPDU,fl:TFlow,v:Validity,st:TrState,epd:OctetString,mt:Nat []
[DaDs(epd) IsEncOf pd and (mt eq MtsSendFlow(pd))] ->
(     [IsCN(pd) or IsAC(pd)] -> p !pd !fl !send !v !st;
      t ?ta:TAddress ?ti:TCEI !TDTreq(epd);
      ( [IsCN(pd)] -> exit(mt) [] [IsAC(pd)] -> exit(Min(mt,mts)) )
)
endproc
(*-----------------------------------------------------------------------------------------------------------
```

If a concatenated sequence of SPDUs can only be concatenated to a GT SPDU with no Token Item parameter, this sequence is directly transmitted with the dummy GT SPDU appended to it. Otherwise, *SendConcPDUs* is invoked and the concatenated sequence is passed on as an argument. *SendConcPDUs* describes the possible concatenation of the given sequence to a GT or to a PT SPDU of which the Token Item parameter is not necessarily absent.

Encode is a process which determines the data octet sequence that corresponds to a valid encoding of the sequence of SPDUs.

```
-----------------------------------------------------------------------------------------------------------------------------*)
process SendConcPDUs(CPDs:ASPDUs,SendXConc:Bool): exit(DatOctStr) :=
      [CPDs ne NULL] ->
      (     choice asp:ASPDUs,TokPd:ASPDU,n:Nat,pd:ASPDU [] [asp eq (TokPd+CPDs)] ->
            (     [SendXConc and (n eq Number(asp)) and DummyPTorGT(TokPd)] ->
                  (     [n gt Succ(0) and IsGT(TokPd)] -> exit
                  []    [CPDs eq ASPDUs(pd) and (IsMAA(pd) or IsMIA(pd) or IsAEA(pd))] -> i; exit
                  []    [CPDs eq ASPDUs(pd) and not(IsMAA(pd) or IsMIA(pd) or
                        IsAEA(pd)) and (IsER(pd) or IsED(pd) or IsACK(pd) iff IsPT(TokPd))] -> exit
                  )
            []    [not(SendXConc) and (CPDs eq  ASPDUs(pd)) and (IsER(pd) or IsED(pd) or IsACK(pd) iff
                  IsPT(TokPd))] -> exit
            ) >> Encode(asp)
      )
[]    [CPDs eq NULL] -> exit(Empty)
endproc

process Encode(asp:ASPDUs): exit(DatOctStr) := Encoder(asp,empty) where
process Encoder(asp:ASPDUs,str:DatOctStr): exit(DatOctStr) :=
      [asp eq NULL] -> exit(str)
[]    [asp ne NULL] -> ( choice epd:DatOctStr [] [epd IsEncOf First(asp)] -> Encoder(Rest(asp),str++epd) )
endproc
endproc (* Encode *)
(*-----------------------------------------------------------------------------------------------------------------------------
```

SendPREP deals with outgoing PR SPDUs. *SendABA* handles outgoing AB and AA SPDUs.

```
-----------------------------------------------------------------------------------------------------------------------------*)
process SendPREP[p,t](SendXConc:Bool,ConcPDUs:ASPDUs): exit(ASPDUs) :=
p ?pd:ASPDU ?fl:TFlow !send ?v:Validity ?st:TrState [IsPR(pd)];
(     SendConcPDUs(ConcPDUs,SendXConc) >> accept Nstr:DatOctStr in
      (     choice epd,Ostr:OctetString [] [DaDs(epd) IsEncOf pd and (Nstr eq DaDs(Ostr))] ->
            t ?ta:TAddress ?ti:TCEI !TEXreq(epd); t ?ta:TAddress ?ti:TCEI !TDTreq(Ostr); exit(NULL)
)     )
endproc

process SendABA[p,t]: exit(ASPDUs) :=
choice pd:ASPDU,epd:OctetString [] [DaDs(epd) IsEncOf pd and (IsAB(pd) or IsAA(pd))] ->
p !pd ?fl:TFlow !send ?v:Validity ?st:TrState;
(     [fl eq exp] -> t ?ta:TAddress ?ti:TCEI !TEXreq(epd); exit(NULL)
[]    [fl eq normal] -> t ?ta:TAddress ?ti:TCEI !TDTreq(epd); exit(NULL)
)
endproc
(*-----------------------------------------------------------------------------------------------------------------------------
```

SendCAT21 takes care of sending RS, RA, AD, ADA, AI, AIA, ER, ED, CD and CDA SPDUs. An acknowledgement SPDU in this category can be sent with a PT SPDU with or without a Token Item parameter. In the former case, the PT SPDU is the result of an S-PLEASE-TOKENS request. This choice, which is undeterministic as far as the protocol standard is concerned, is modelled by an internal event.

```
------------------------------------------------------------------------------------------------------------*)
process SendCAT21[p,t](SendXConc:Bool,ConcPDUs:ASPDUs): exit(ASPDUs) :=
choice pd:ASPDU [] [IsCAT21(pd)] -> p !pd ?fl:TFlow !send ?v:Validity ?st:TrState;
(       SendConcPDUs(ConcPDUs,SendXConc) >> accept Nstr:DatOctStr in
        (       choice apd:ASPDUs, GorPT:ASPDU []
                [DummyPTorGT(GorPT) and (apd eq (GorPT+ASPDUs(pd))) and (IsER(pd) or IsED(pd) or
                IsACK(pd) iff IsPT(GorPT))] ->
                Encode(apd) >> accept eapd:DatOctStr in
                (       t ?ta:TAddress ?ti:TCEI !TDTreq(DsDa(Nstr));
                        t ?ta:TAddress ?ti:TCEI !TDTreq(DsDa(eapd)); exit(NULL)
                []      [IsER(pd) or IsED(pd) or IsACK(pd)] ->
                        i; t ?ta:TAddress ?ti:TCEI !TDTreq(DsDa(Nstr)); exit(ASPDUs(pd))
)       )       )
endproc
(*------------------------------------------------------------------------------------------------------------
```

SendPT and *SendGT* deal with the transmission of PT and GT SPDUs respectively, which have been constructed as a result of an SS-user request. These SPDUs can be concatenated to a (sequence of) SPDU(s) or they have to be transmitted separately. In the case where both alternatives are possible, the choice is non-deterministic and is modelled by an internal event.

If the sequence of SPDUs does not consist of just one acknowledgement SPDU, it has to be sent separately. The process subsequently terminates, exporting value *NULL* to indicate that there are no SPDUs pending for concatenation.

In *SendGT*, the function *CL* is used to check whether a given SPDU contains a complete SSDU or the last segment of a segmented SSDU. A DT SPDU may contains a complete SSDU if: 1) the Enclosure Item parameter of the SPDU is absent, or 2) the parameter is present but its first bit is equal to zero and its second bit is equal to one.

```
------------------------------------------------------------------------------------------------------------*)
process SendPT[p,t](SendXConc:Bool,ConcPDUs:ASPDUs,MTSF:Nat): exit(ASPDUs) :=
choice pd:ASPDU,epd:DatOctStr [] [IsPT(pd) and (epd IsEncOf pd)] ->
p !pd ?fl:TFlow !send ?v:Validity ?st:TrState;
(       let apd: ASPDUs = (pd+ConcPDUs) in Encode(apd) >> accept eapd:DatOctStr in
        (       choice pp:ASPDU []
                [(ConcPDUs eq ASPDUs(pp)) and (IsER(pd) or IsED(pd) or IsACK(pp)) and
                ((Length(eapd) lt MTSF) or (MTSF eq 0))] ->
                t ?ta:TAddress ?ti:TCEI !TDTreq(DsDa(eapd)); exit(NULL)
        []      (       i; SendConcPDUs(ConcPDUs,SendXConc) >> accept Nstr:DatOctStr in
                        (       t ?ta:TAddress ?ti:TCEI !TDTreq(DsDa(Nstr));
                                t ?ta:TAddress ?ti:TCEI !TDTreq(DsDa(epd)); exit(NULL)
)       )       )       )
endproc
```

```
process SendGT[p,t](SendXConc:Bool,ConcPDUs:ASPDUs,MTSF:Nat): exit(ASPDUs) :=
choice pd:ASPDU,epd:DatOctStr [] [IsGT(pd) and (epd IsEncOf pd)] ->
p !pd ?fl:TFlow !send ?v:Validity ?st:TrState;
(       Encode(pd+ConcPDUs) >> accept eapd:DatOctStr in
        (       choice pp:ASPDU [] [(ConcPDUs eq ASPDUs(pp)) and
                ( (IsMAA(pp) or IsMIA(pp) or IsAEA(pp)) implies SendXConc ) and
                (IsDT(pp) implies CL(pp)) and ( (Length(eapd) lt MTSF) or (MTSF eq 0) )] ->
                t ?ta:TAddress ?ti:TCEI !TDTreq(DsDa(eapd)); exit(NULL)
        []      (       i; SendConcPDUs(ConcPDUs,SendXConc) >> accept Nstr:DatOctStr in
                        (       t ?ta:TAddress ?ti:TCEI !TDTreq(DsDa(Nstr));
                                t ?ta:TAddress ?ti:TCEI !TDTreq(DsDa(epd)); exit(NULL)
)       )       )       )
endproc
(*------------------------------------------------------------------------------------------------------------
```

The remaining SPDUs are handled by processes *Rest_of_PDUs1* and *Rest_of_PDUs2*. The SPDUs handled by *Rest_of_PDUs2* are never concatenated with other SPDUs. If any of these SPDUs is to be transmitted, the pending (concatenated) SPDUs have to be sent first.

In *Rest_of_PDUs1*, it is assumed that the minimal length of a dummy PT or GT SPDU is one octet, corresponding to the length of the SI field. If extended concatenation is not permitted (indicated by *SendXConc*), then DT SPDUs which contain neither the last segment of a segmented SSDU nor a complete SSDU are immediately transmitted and are not retained. Otherwise, a DT SPDU containing the first segment of a segmented SSDU may be stored for further concatenation. Function *IsValConc* is used to test whether its argument is a valid concatenation according to tables 7 and 8 of ISO 8327, not taking into account the PT or GT SPDU still to be concatenated to the sequence. *M_S_w_GT* tests whether its argument, a valid concatenation of SPDUs, can only be concatenated to a GT SPDU which does not contain a Token Item parameter. This concatenated sequence is subsequently transmitted.

```
------------------------------------------------------------------------------------------------------------*)
process Rest_of_PDUs1[p,t](SendXConc:Bool,ConcPDUs:ASPDUs,MTSF:Nat): exit(ASPDUs) :=
choice pd:ASPDU, pds,cp:ASPDUs []
[(cp eq ConcatPDUs(pd,ConcPDUs)) and (pds eq ASPDUs(pd)) and (IsAS(pd) or IsAR(pd) or IsMAP(pd) or
IsMIP(pd) or IsAE(pd) or IsDT(pd) or IsMAA(pd) or IsMIA(pd))] ->
p !pd ?fl:TFlow !send ?v:Validity ?st:TrState;
Encode(cp) >> accept ecp:DatOctStr in
(       let size:Nat = Succ(Length(ecp)) in
        (       [(size gt MTSF) and (MTSF ne 0) or not(SendXConc) or not(IsValConc(cp))] ->
                (       SendConcPDUs(ConcPDUs,SendXConc) >> accept Nstr1:DatOctStr in
                        (       i;SendConcPDUs(pds,SendXConc) >> accept Nstr2:DatOctStr in
                                (       t ?ta:TAddress ?ti:TCEI !TDTreq(DsDa(Nstr1));
                                        t ?ta:TAddress ?ti:TCEI !TDTreq(DsDa(Nstr2)); exit(NULL)
                        )       )
                []t ?ta:TAddress ?ti:TCEI !TDTreq(DsDa(Nstr1)); exit(pds)
                )
        []      [(size lt MTSF) or (MTSF eq 0) and SendXConc and IsValConc(cp)] ->
                (       (       SendConcPDUs(cp,SendXConc) >> accept Nstr:DatOctStr in
                                t ?ta:TAddress ?ti:TCEI !TDTreq(DsDa(Nstr)); exit(NULL)     )
                [] i;
                        (       [not(M_S_w_GT(cp))] -> i; exit(cp)
                        []      (       SendConcPDUs(ConcPDUs,SendXConc) >> accept Nstr1:DatOctStr in
                                        (       i;
                                                (       SendConcPDUs(pds,SendXConc) >> accept Nstr2:DatOctStr in
                                                        t ?ta:TAddress ?ti:TCEI !TDTreq(DsDa(Nstr1));
                                                        t ?ta:TAddress ?ti:TCEI !TDTreq(DsDa(Nstr2)); exit(NULL)     )
                                        []      t ?ta:TAddress ?ti:TCEI !TDTreq(DsDa(Nstr1)); exit(pds)
)       )       )       )       )       )
endproc
```

```
process Rest_of_PDUs2[p,t](SendXConc:Bool,ConcPDUs:ASPDUs,MTSF:Nat): exit(ASPDUs) :=
choice pd:ASPDU,epd:DatOctStr [] [IsRF(pd) or IsFN(pd) or IsDN(pd) or IsNF(pd) or IsGTC(pd) or
IsGTA(pd) or IsEX(pd) or IsTD(pd) and (epd IsEncOf pd)] ->
p !pd ?fl:TFlow !send ?v:Validity ?st:TrState;
(       SendConcPDUs(ConcPDUs,SendXConc) >> accept Nstr:DatOctStr in
        t ?ta:TAddress ?ti:TCEI !TDTreq(DsDa(Nstr)); t ?ta:TAddress ?ti:TCEI !TDTreq(DsDa(epd));
        exit(NULL)
)
endproc
(*--------------------------------------------------------------------------------------------------------
```

10.4.3 Internal backpressure

Process *Int_BP* models the internal backpressure which the SPM can apply w.r.t. SPDUs that are accepted for transmission.

```
------------------------------------------------------------------------------------------------------*)
process Int_BP[p]:noexit :=
choice bpr:Bool [] I;
(       choice pd:ASPDU,fl:TFlow,v:Validity,st:TrState []
        (       p !pd !fl ?d:Dir !v !st [not(d eq send)]; Int_BP[p]
        []      [not(bpr)] -> p !pd !fl !send !v !st; Int_BP[p]
        []      Int_BP[p]
)       )
endproc
(*--------------------------------------------------------------------------------------------------------
```

10.4.4 Transport connection constraints

The constraints on the relationship between SPDUs and TSPs related to the TC characteristics can be described as consisting of two synchronizing processes:

- *TexCon*, which imposes constraints related to the availability of the transport expedited option (represented by the value of *tex*);
- *VtcaCon*, which imposes constraints related to the status of the SPM as an initiator or as a responder of the supporting TC (this status is determined with *IsTCONresp*).

```
------------------------------------------------------------------------------------------------------*)
process TCONVar[p,t]: noexit :=
        t ?ta:TAddress ?ti:TCEI ?ts:TSP [not(IsTCONresp(ts) or IsTCONconf(ts))]; TCONVar[p,t]
[]      t ?ta:TAddress ?ti:TCEI ?ts:TSP [IsTCONresp(ts) or IsTCONconf(ts)];
        ( choice tex:TEXOption [] [tex IsTEXOptionOf ts] -> TCONVar1[p](IsTCONresp(ts),tex eq usetex) )
endproc

process TCONVar1[p](Vtca,tex:Bool): noexit := TexCon[p](tex) || VtcaCon[p](Vtca) endproc
(*--------------------------------------------------------------------------------------------------------
```

The parameter of *VtcaCon*, *Vtca*, is true if the SPM is the initiator of the supporting TC. The parameter of *TexCon*, *tex*, is true if the transport expedited flow is available.

If the SPM is the initiator of the TC and a CN SPDU is received, a T-DISCONNECT request is to be executed (bit 12 of the SPDU status component is set). The S-CONNECT indication corresponding to the CN SPDU is not issued (bit 16 is set).

If a RS SPDU is received and bit 18 of SPDU status component is set to *0*, then the "SPMwinner" condition is determined in *VtcaCon*, with function *Winner*, and bit 19 is set or reset.

The Transport Disconnect parameter of an outgoing FN SPDU may indicate re-use of the underlying TC only if the transport expedited option is selected. As a local choice, this parameter may be set to no reuse at all times.

If an incoming SPDU is received on the correct transport flow, bit 2 of the SPDU status component is set. An outgoing PR SPDU is actually transmitted only if the transport expedited flow is available.

```
------------------------------------------------------------------------------------------------------*)
process VtcaCon[p](Vtca:Bool): noexit :=
choice pd:ASPDU,fl:TFlow,v:Validity,st:TrState,tdr:Bool []
(     [IsCN(pd) and Valid(v) and (not(Vtca) and Set(v,12) and Set(v,16) or
      (Vtca and Reset(v,12) and Reset(v,16)))] -> p !pd !fl !rec !v !(0 of TrState); exit
[]    [Reset(v(v,12)] ->
      (     [IsFN(pd) and (tdr eq (r eq TDISPar(pd)))] ->
            ( [not(Vtca) and tdr] -> p !pd !fl !send !v !st; exit [] [not(tdr)] -> i; p !pd !fl !send !v !st; exit )
      []    [IsFN(pd) and (r eq TDISPar(pd)) or (IsDN(pd) and (st eq 2)) and (Vtca iff Set(v,4))] ->
            p !pd !fl !rec !v !st; exit
      []    [IsRS(pd) and Set(v,4) and (Set(v,18) or (Reset(v,18) and (Vtca iff Set(v,19))))] ->
            p !pd !fl !rec !v !st; exit
      )
[]    [not(IsFN(pd))] -> p !pd !fl ?d:Dir !v !st [not(d eq rec)]; exit
[]    [not(IsCN(pd) or IsRS(pd) or (IsFN(pd) and (r eq TDISPar(pd))) or (IsDN(pd) and (st eq 2))) and
      Set(v,4) and Reset(v,12)] -> p !pd !fl !rec !v !st; exit
) >> VtcaCon[p](Vtca)
endproc

process TexCon[p](tex:Bool): noexit :=
choice pd:ASPDU,fl:TFlow,v:Validity,st:TrState,tdr,invalid:Bool []
(     [IsRF(pd) or IsAB(pd) and (tdr eq (r eq TDISPar(pd)))] ->
      (     [not(tex) and tdr] -> p !pd !normal !send !v !st; exit
      []    [not(tdr) and (IsAB(pd) implies (tex iff (fl eq exp)))] -> i; p !pd !fl !send !v !st; exit    )
[]    [IsFN(pd) and (tdr eq (r eq TDISPar(pd)))] ->
      ( [not(tex) and tdr] -> p !pd !fl !send !v !st; exit [] [not(tdr)] -> i; p !pd !fl !send !v !st; exit )
[]    [IsPR(pd)] -> p !pd !fl ?d:Dir !v !st [d eq send implies tex]; exit
[]    [IsAA(pd) and (tex iff (fl eq exp))] -> p !pd !fl !send !v !st; exit
[]    [Err_Comb(pd,fl,tex) and Reset(v,2) and Reset(v,12)] -> p !pd !fl !rec !v !st; exit
[]    [not(Err_Comb(pd,fl,tex)) and Set(v,2) and (tdr eq (r eq TDISPar(pd)))] ->
      (     [IsRF(pd) or IsAB(pd) and tdr and Valid(v)] ->
            (     [not(tex) and Reset(v,12)] -> p !pd !fl !rec !v !st; exit
            []    [Set(v,12)] -> i; p !pd !fl !rec !v !st; exit   )
      []    [IsFN(pd) and tdr and Reset(v,12) and (tex iff Reset(v,3))] -> p !pd !fl !rec !v !st; exit
      []    [invalid iff (st eq 1 or (st eq 2) and tex) and (IsMAA(pd) or IsAEA(pd)) and Reset(v,12) and
            (invalid iff Reset(v,3)) ] -> p !pd !fl !rec !v !st; exit
      []    [invalid iff (st eq 1 or (st eq 3) and tex) and  Reset(v,12) and IsRS(pd) and
            (invalid iff Reset(v,3))] -> p !pd !fl !rec !v !st; exit
      []    [IsRA(pd) or IsAD(pd) or IsAI(pd) or IsADA(pd) or IsAIA(pd) and (tex iff Reset(v,3)) and
            Reset(v,12)] -> p !pd !fl !rec !v !(1 of TrState); exit
      []    (     let p1:Bool = (IsAB(pd) or IsFN(pd) or IsRF(pd) and tdr),
                  p2:Bool = (IsRS(pd) or IsMAA(pd) or IsAEA(pd) or IsRA(pd) or IsAD(pd) or IsAI(pd) or
                  IsADA(pd) or IsAIA(pd)) in
                  ( [not(p1 or p2) and Set(v,3) and Reset(v,12)] -> p !pd !fl !rec !v !st; exit )
      )     )
[]    [not(IsAB(pd) or IsRF(pd) or IsFN(pd) or IsPR(pd) or IsAA(pd))] -> p !pd !fl !send !v !st; exit
[]    [not(IsPR(pd))] -> p !pd !fl !no_send !v !st; exit
) >> TexCon[p](tex)
endproc
(*------------------------------------------------------------------------------------------------------
```

10.4.5 Timer constraints

Timer models the timer events in the state tables of ISO 8327. The timer behaviour, activated upon the exchange of certain SPDUs, is represented by *Tim*. In *Tim*, an internal event can take place followed by the execution of a T-DISCONNECT request. This internal event models a "time-out". If an AA SPDU is awaited (indicated by the value of boolean *awaa*) and a correct AA SPDU or a correct AB SPDU (the Transport Disconnect parameter must indicate the re-use of the TC) is received, the timer is canceled.

```
-----------------------------------------------------------------------------------------*)
process Timer[p,t]: exit :=
choice pd:ASPDU,fl:TFlow,d:Dir,v:Validity,st:TrState []
(     [(d eq rec) or not( IsAB(pd) or (IsRF(pd) and (nr eq TDISPar(pd))) or (IsDN(pd) and (st eq 0)) )] ->
      p !pd !fl !d !v !st; Timer[p,t]
[]    [IsAB(pd) or (IsRF(pd) and (nr eq TDISPar(pd))) or (IsDN(pd) and (st eq 0))] ->
      p !pd !fl !send !v !st; Tim[p,t](IsAB(pd) and (r eq TDISPar(pd)))
)
endproc

process Tim[p,t](awaa:Bool): exit :=
      p ?pd:ASPDU ?fl:TFlow !rec ?v:Validity ?st:TrState;
      (     [Valid(v) and awaa and ( IsAA(pd) or (IsAB(pd) and (r eq TDISPar(pd))) )] -> Timer[p,t]
      []    [not(Valid(v) and awaa and ( IsAA(pd) or (IsAB(pd) and (r eq TDISPar(pd))) ) )] -> Tim[p,t](awaa)
      )
[]    i; ExecTDr[t] (* Time out*)
endproc
(*-----------------------------------------------------------------------------------------
```

10.5 SPDU constraints

Constraints related to the state tables of ISO 8327 are described by *SPDUConstraints*. It consists of process *SPDUConstraints1*, which describes the same constraints without those concerning events that immediately terminate the SPM behaviour, and two processes, *SendAB* and *RecABnr*, that are only concerned with such terminating events. *SendAB* describes the transmission of an AB SPDU; *RecABnr* describes the receipt of an AB (no reuse) SPDU. An instance of either of these processes may at any time disable *SPDUConstraints1*.

NOTE - An incoming AB SPDU (with the Transport Disconnect parameter set to re-use) cannot be defined as being able to disable *SPDUConstraints1* because in state tables state STA16 of ISO 8327 there is a different predicate-action combination possible than in the remaining states. Also, when the SPDU is received in any of the STA15 states or in state STA06, it is in fact invalid in which case subclause A.4.1.2 is applicable. In other words, depending on the state in which the AB (reuse) SPDU is received, different event sequences are possible (described in process *SPDUOrd1*, see 10.5.5.2).

```
-----------------------------------------------------------------------------------------*)
process SPDUConstraints[p](spei:SPEImplementation): noexit :=
SPDUConstraints1[p](spei) [> ( SendAB[p](spei) [] RecABnr[p] )
endproc
(*-----------------------------------------------------------------------------------------
```

10.5.1 Abort processes

The first event of *RecABnr* is the receipt of an AB (no reuse) SPDU. Bit 14 of the SPDU status component is set, indicating that a T-DISCONNECT request is to be executed. If the SC is still in progress, either an S-U-ABORT indication or an S-P-ABORT indication is executed depending on the received AB SPDU. If the incoming AB SPDU is valid, it will lead to the execution of a T-DISCONNECT request, after which the instance of *STPM* (see 10.1.1) will successfully terminate. As a local implementation matter, an AA SPDU can be sent prior to the execution of the T-DISCONNECT request.

The first event of *SendAB* is the transmission of an AB SPDU, prompted by either a S-U-ABORT request or a protocol error. *TDISPar* extracts the Transport Disconnect parameter from the SPDU.

If the Transport Disconnect parameter indicates no re-use of the TC (represented by value *nr*), a T-DISCONNECT indication is awaited; this is described by process *AwaitTDIS*. Otherwise (value *r*), an AA SPDU is expected, as described by process *AwaitAA*. In both cases, a timer is started. If a time-out occurs before a T-DISCONNECT indication has occurred, or before an AA SPDU has been received, a T-DISCONNECT request is executed. This is modelled by process *Timer* (see 10.4.5).

A correctly received CN or AB (no reuse) SPDU, while awaiting receipt of a correct AA SPDU, will lead to execution of a T-DISCONNECT request. A correctly received AB (reuse) or AA SPDU will bring the SPM in the idle TC state. A correctly received SPDU which is not any of a CN, AB or AA SPDU, or an incorrect SPDU (e.g. badly encoded or received on the wrong transport flow) will be ignored.

A correctly received AA, AB or CN SPDU, while awaiting a T-DISCONNECT indication, will lead to the execution of a T-DISCONNECT request. Any other SPDU will be ignored.

```
------------------------------------------------------------------------------------------------------*)
process RecABnr[p]: noexit :=
p ?ab:ASPDU ?fl:TFlow !rec ?v:Validity ?st:TrState
     [IsAB(ab) and (nr eq TDISPar(ab)) and Set(v,14) and ValidDel(v)];
p ?aa:ASPDU ?fl:TFlow !send ?v:Validity ?st:TrState; stop
endproc

process SendAB[p](spei:SPEImplementation): noexit :=
choice ab:ASPDU,fl:TFlow,v:Validity,st:TrState [] [IsAB(ab)] ->
p !ab !fl !send !v !st;
( [r eq TDISPar(ab)] -> AwaitAA[p](spei) [] [nr eq TDISPar(ab)] -> AwaitTDIS[p] )
endproc

process AwaitAA[p](spei:SPEImplementation): noexit :=
choice pd:ASPDU,fl:TFlow,v:Validity []
(    [(IsCN(pd) or (IsAB(pd) and (nr eq TDISPar(pd)))) and IsValNoDel(v) and Set(v,14)] ->
     p !pd !fl !rec !v !(0 of TrState); AwaitAA[p](spei)
[]   [IsAA(pd) or (IsAB(pd) and (r eq TDISPar(pd))) and IsValNoDel(v) and Reset(v,14)] ->
     p !pd !fl !rec !v !(0 of TrState); SPDUConstraints[p](spei)
[]   [not(IsAA(pd) or IsAB(pd) or IsCN(pd) and IsValNoDel(v)) and Reset(v,14)] ->
     p !pd !fl !rec !v !(0 of TrState); AwaitAA[p](spei)
)
endproc

process AwaitTDIS[p]: noexit :=
choice pd:ASPDU,fl:TFlow,v:Validity,st:TrState []
(    [IsCN(pd) or IsAB(pd) or IsAA(pd) and IsValNoDel(v) and Set(v,14)] ->
     p !pd !fl !rec !v !(0 of TrState); AwaitTDIS[p]
[]   [not(IsAA(pd) or IsAB(pd) or IsCN(pd) and Valid(v)) and Reset(v,14)] ->
     p !pd !fl !rec !v !(0 of TrState); AwaitTDIS[p]
)
endproc
(*------------------------------------------------------------------------------------------------------
```

10.5.2 Non-abort SPDU constraints

Further constraints on SPDU exchanges can be considered as consisting of the conjunction of four component processes:

- *ImplCon* (see 10.5.3) imposes constraints related to the protocol implementation parameter;
- *TwoSPDUs* (see 10.5.4) checks whether certain pairs of received and transmitted SPDUs, and sometimes certain parameters or parts of individual SPDUs, are compatible; and
- *SPDUOrd* (see 10.5.5) deals with the temporal ordering of SPDUs;
- *VarCon* (see 10.5.6) imposes constraints related to the session variables ("V(A)", "V(R)", "Vsc", etc.).

```
-----------------------------------------------------------------------------------------------*)
process SPDUConstraints1[p](spei:SPEImplementation): noexit :=
ImplCon[p](spei) || TwoSPDUs[p] || SPDUOrd[p] || VarCon[p]
endproc
(*-----------------------------------------------------------------------------------------------
```

10.5.3 Implementation constraints

ImplCon takes care of setting the correct values for the Maximum TSDU size and Protocol Options parameters of an outgoing CN (or AC) SPDU. These values are extracted from the protocol implementation parameter (see clause 7).

```
-----------------------------------------------------------------------------------------------*)
process ImplCon[p](spei:SPEImplementation): noexit :=
choice pd:ASPDU,d:Dir [] [( (d eq send) and (IsCN(pd) or IsAC(pd)) ) implies
(MaxTSize(spei) eq MTSDUPar(pd) and (ExtConc(spei) iff ProtOptions(ps)))] ->
p !pd ?fl:TFlow !(d of Dir) ?v:Validity ?st:TrState; ImplCon[p](spei)
endproc
(*-----------------------------------------------------------------------------------------------
```

10.5.4 Dependencies between parameter values of SPDUs

The constraints that reflect the dependencies between parameter values of (pairs of) SPDUs are specified in different component processes, each dealing with a different pair of SPDUs. Process *CNPDU* checks whether the parameter values of an incoming CN SPDU satisfy the constraints mentioned in subclause 7.1.1 of ISO 8327. The function *CorrectCN* is used for this purpose. *CN_AC* ensures that a received AC SPDU is compatible with the CN SPDU sent by this SPM, as defined in subclause 7.2.1 of ISO 8327 (since a S-CONNECT response is constrained by process *SCEP*, an outgoing AC SPDU is assumed to be correct). The function *Matches* is employed here. *RS_RA* deals with the conditions related to RS and RA SPDUs, as mentioned in subclause 7.23.2 of ISO 8327. *TD_DT* imposes conditions on sequences of TD or DT SPDUs pertaining to a segmented SSDU. For example, the receipt of two incoming DT SPDUs, both of which have an Enclosure Item parameter indicating the beginning of the SPDU (*begin*), constitutes a protocol error.

In all cases, if an incoming SPDU does not satisfy the conditions, bit 5 of the SPDU status component is set to *0*.

```
-----------------------------------------------------------------------------------------------*)
process TwoSPDUs[p]: noexit := CNPDU[p] || CN_AC[p] || RS_RA[p] || TD_DT[p] endproc
process CNPDU[p]: noexit :=
choice pd:ASPDU,fl:TFlow,v:Validity,st:TrState []
(     [IsCN(pd) and (CorrectCN(pd) iff Set(v,5))] -> p !pd !fl !rec !v !st; CNPDU[p]
[]    [not (IsCN(pd) or IsAC(pd) or IsRF(pd) or IsTD(pd) or IsDT(pd) or IsRA(pd)) and Set(v,5)] ->
      p !pd !fl !rec !v !st; CNPDU[p]
[]    p !pd !fl ?d:Dir !v !st [not (d eq rec)]; CNPDU[p]
)
endproc

process CN_AC[p]: noexit :=
choice pd:ASPDU,fl:TFlow,v:Validity,st:TrState []
(     [IsCN(pd)] -> p !pd !fl !send !v !st; CN_AC1[p](pd)
[]    [not(IsCN(pd))] -> p !pd !fl !send !v !st; CN_AC[p]
[]    p !pd !fl ?d:Dir !v !st [not(d eq send)]; CN_AC[p]
)
where
process CN_AC1[p](cn:ASPDU): noexit :=
choice pd:ASPDU,fl:TFlow,v:Validity,st:TrState []
(     [IsAC(pd) and (Matches(cn,pd) iff Set(v,5))] -> p !pd !fl !rec !v !st; CN_AC1[p](cn)
[]    [IsCN(pd)] -> p !pd !fl !send !v !st; CN_AC1[p](pd)
[]    [not(IsAC(pd))] -> p !pd !fl !rec !v !st; CN_AC1[p](cn)
[]    [not(IsCN(pd))] -> p !pd !fl !send !v !st; CN_AC1[p](cn)
[]    p !pd !fl !no_send !v !st; CN_AC1[p](cn)
)
endproc
endproc (* CN_AC *)

process RS_RA[p]: noexit :=
choice pd:ASPDU,fl:TFlow,v:Validity,st:TrState []
(     [IsRS(pd)] -> p !pd !fl !send !v !st; RS_RA1[p](pd)
[]    [not(IsRS(pd))] -> p !pd !fl !send !v !st; RS_RA[p]
[]    p !pd !fl ?d:Dir !v !st [not (d eq send)]; RS_RA[p]
)
where
process RS_RA1[p](rspd: ASPDU): noexit :=
choice pd: ASPDU,fl:TFlow,v:Validity,st:TrState []
(     [IsRS(pd)] -> p !pd !fl !send !v !st; RS_RA1[p](pd)
[]    [not(IsRS(pd))] -> p !pd !fl !send !v !st; RS_RA1[p](rspd)
[]    (     choice tass1,tass2:STsAss,rq1,rq2,ac1,ac2,ch:STokens [] [(tass1 eq TokenSet(rspd)) and
            IsRA(pd) and (tass2 eq TokenSet(pd)) and (rq1 eq RqrTokens(tass1)) and
            (ac1 eq AcrTokens(tass1)) and (rq2 eq RqrTokens(tass2)) and (ac2 eq AcrTokens(tass2)) and
            (ch eq ChoiceTokens(tass1)) and ( ((rq1 IsSubsetOf rq2) and (ac1 IsSubsetOf ac2) and
            (ch eq ((rq2 Minus rq1) Union (ac2 Minus ac1)))) iff  Set(v,5))] ->
            p !pd !fl !rec !v !st; RS_RA1[p](rspd)
      )
[]    [not(IsRA(pd))] -> p !pd !fl !rec !v !st; RS_RA1[p](rspd)
[]    p !pd !fl !no_send !v !st; RS_RA1[p](rspd)
)
endproc
endproc (* RS_RA *)

process TD_DT[p]: noexit :=
choice pd:ASPDU,fl:TFlow,v:Validity,st:TrState,encit:EncIt []
(     [(IsDT(pd) or IsTD(pd)) and (encit eq EncItem(pd)) and
      ( (encit eq begin) or ((encit eq absent) iff Set(v,5)) )] -> p !pd !fl !rec !v !st; TD_DT1[p]
[]    [not(IsDT(pd) or IsTD(pd))] -> p !pd !fl !rec !v !st; TD_DT[p]
)
where
```

```
process TD_DT1[p]: noexit :=
choice pd:ASPDU,fl:TFlow,v:Validity,st:TrState,encit:EncIt []
(    [(IsDT(pd) or IsTD(pd)) and (encit eq EncItem(pd)) and (encit eq begin) and Reset(v,5)] ->
     p !pd !fl !rec !v !st; TD_DT[p]
[]   [(IsDT(pd) or IsTD(pd)) and (encit eq EncItem(pd)) and (encit eq middle) and Set(v,5)] ->
     p !pd !fl !rec !v !st; TD_DT1[p]
[]   [(IsDT(pd) or IsTD(pd)) and (encit eq EncItem(pd)) and ((encit eq end) or (encit eq absent)) and
     Set(v,5)] -> p !pd !fl !rec !v !st; TD_DT[p]
[]   [not(IsDT(pd) or IsTD(pd))] -> p !pd !fl !rec !v !st; TD_DT[p]
[]   p !pd !fl ?d:Dir !v !st [not(d eq rec)]; TD_DT1[p]
)
endproc
endproc (* TD_DT *)
(*---------------------------------------------------------------------------------------------------
```

10.5.5 SPDU ordering constraints

10.5.5.1 Connection phase ordering constraints

SPDUOrd determines the order in which SPDUs may be sent or received. Following receipt of an AB (reuse) SPDU, either a T-DISCONNECT request is executed or an AA SPDU is transmitted. A valid incoming SPDU which is neither an AB SPDU nor a CN SPDU will result in the execution of a T-DISCONNECT request. To indicate this, bit 14 of the SPDU status component is set to *1*. Incorrect SPDUs (e.g. badly encoded SPDUs or SPDUs received on the wrong transport flow) are either ignored or prompt the transmission of an AB SPDU. Which alternative is chosen is a local implementation matter.

After transmission of a CN SPDU, a AC SPDU is expected. The latter is described by process *Await_Rec_AC*. This process is parameterized with the number of "stored" PR-RS SPDUs, received while awaiting the AC SPDU (initially zero).

If a CN SPDU is received, an AC can be transmitted, as described by process *Await_Trans_AC*.

```
---------------------------------------------------------------------------------------------------*)
process SPDUOrd[p]: noexit :=
choice pd:ASPDU,fl:TFlow,v:Validity,st:TrState []
(    [IsCN(pd)] -> p !pd !fl !send !v !st; Await_Rec_AC[p](0 of Nat)
[]   [not(IsCN(pd))] -> p !pd !fl !send !v !st; SPDUOrd[p]
[]   [Set(v,11)] ->
     (    [IsCN(pd) and ValidDel(v) and Reset(v,14)] -> p!pd!fl!rec!v!(0 of TrState); Await_Trans_AC[p]
     []   [( (IsAB(pd) and (r eq TDISPar(pd)) and IsValNoDel(v)) or (IsCN(pd) and IsValNoDel(v)) or
          not(Valid(v)) ) and Reset(v,14)] -> p !pd !fl !rec !v !(0 of TrState); SPDUOrd[p]
     []   [not (IsAB(pd) or IsCN(pd)) and IsValNoDel(v) and Set(v,14)] ->
          p !pd !fl !rec !v !(0 of TrState); SPDUOrd[p]
)    )
endproc
(*---------------------------------------------------------------------------------------------------
```

Following receipt of the AC SPDU, the data transfer state is entered. The permissible SPDU sequences in the data transfer state are specified in process *SPDUOrd1*. If a RF SPDU is received, either a T-DISCONNECT request is executed or the idle TC state is entered.

PR-RS SPDUs which are received before the AC SPDU, are stored and are processed following receipt of the AC. Receipt of any other SPDU leads to the execution of the procedure outlined in subclause A.4.1.2 a) of ISO 8327 (incorrect SPDUs may be ignored).

If, after receipt of a valid AC SPDU, the number of received PR-RS SPDUs is greater than one, the procedure of subclause A.4.1.2 a) is followed. If a correct SPDU is received which is not a PR-RS, RF, AC, EX or AB SPDU, it is an invalid SPDU because it is not allowed in this state. This is indicated by setting bit 11 of the SPDU status component to *0*.

Resetting bit 14 of the SPDU status component indicates that *Await_Rec_AC* may not impose the execution of a T-DISCONNECT. Other processes may still determine that the TC is to be terminated (refer to the definition of processes *TexCon* and *VtcaCon* in 10.4.4).

When awaiting an AC SPDU, PR SPDUs can be received. These are stored until an AC SPDU is received, and then "dequeued". If the number of SPDUs is greater than one, an invalid number of PR SPDUs was received. This invalid condition is indicated by setting bit 17 of the SPDU status component. When in *SSPSPDU* this bit is checked and found set, the S-CONNECT confirm corresponding to the received AC SPDU will first be executed. Subsequently, any of the valid actions corresponding to the detection of a protocol error will be imposed.

Process *Await_Trans_AC* prescribes that any incoming SPDU except an AB is invalid. This is indicated by setting bit 11 to *0* of the SPDU status component. Following receipt of an AB SPDU, either:

- an AA SPDU is sent and the TC is maintained;
- optionally an AA SPDU is sent and the TC is released.

```
-------------------------------------------------------------------------------------------*)
process Await_Rec_AC[p](pr:Nat): noexit :=
choice pd:ASPDU,fl:TFlow,v:Validity,st:TrState []
(    let ac:Bool = IsAC(pd), rf:Bool = IsRF(pd), prep:Bool = (res eq Preptype(pd)), ab:Bool = IsAB(pd),
     ex:Bool = IsEX(pd), tdr:Bool = (r eq TDISPar(pd)) in
     (    [Set(v,11)] ->
          (    [ac and ValidDel(v) and Reset(v,14) and (pr gt Succ(0) iff Set(v,17))] ->
               p !pd !fl !rec !v !(1 of TrState);
               ( [pr ne Succ(0)] -> SPDUOrd1[p] [] [pr eq Succ(0)] -> After_PRRS[p] )
          []   [ex and Reset(v,14) and Set(v,17)] -> p!pd!fl!rec!v!(1 of TrState); Await_Rec_AC[p](pr)
          []   [rf and ValidDel(v) and (tdr iff Reset(v,14))] -> p!pd!fl!rec!v!(1 of TrState); SPDUOrd[p]
          []   [prep and IsValNoDel(v) and Reset(v,14)] ->
               p !pd !fl !rec !v !(1 of TrState); Await_Rec_AC[p](Succ(pr))
          []   [ab and tdr and ValidDel(v) and Reset(v,14)] ->
               p !pd !fl !rec !v !(1 of TrState); SPDUOrd[p]
          []   [not(Valid(v)) and Reset(v,14)] -> p!pd!fl!rec!v!(1 of TrState) ; Await_Rec_AC[p](pr) )
          []   [Reset(v,11) and Reset(v,14) and not(prep or ac or rf or ab or ex)] ->
               p !pd !fl !rec !v !(1 of TrState); SPDUOrd[p]
)    )    )
endproc

process Await_Trans_AC[p]: noexit :=
choice pd:ASPDU,fl:TFlow,v:Validity [] [Reset(v,14)] ->
(    let ab:Bool = IsAB(pd), ac:Bool = IsAC(pd), rf:Bool = IsRF(pd), tr:Bool = (r eq TDISPar(pd)) in
     (    [not(ab) and Reset(v,11)] -> p !pd !fl !rec !v !(1 of TrState); SPDUOrd[p]
     []   [ab and tr and ValidDel(v)] -> p !pd !fl !rec !v !(1 of TrState); SPDUOrd[p]
     []   [ac] -> p !pd !fl !send !v ?st:TrState; SPDUOrd1[p]
     []   [rf] -> p !pd !fl !send !v ?st:TrState;
          ( [tr] -> SPDUOrd[p] [] [not(tr)] -> AwaitTDIS[p] )
)    )
endproc
(*-----------------------------------------------------------------------------------------
```

10.5.5.2 Transfer phase ordering constraints

SPDUOrd1 models the permissible sequences of SPDUs once the SC is established. *SPDUOrd2* is the immediate constituent of *SPDUOrd1*, but parameterized with the number of PR-RS SPDUs that are stored by the SPM (initially zero). It is described as the choice of several behaviours, each dealing with sequences of specific SPDUs, viz. for connection release (*REL*), for synchronization (*SYNMm*), for resynchronization (*RSYN*), for token management (*Tok*), for activity management (*Act*), for capability data (*CapDT*) and for exceptions (*Exception*). Correctly received SPDUs which are not allowed in the state tables state STA713 of ISO 8327 may be treated as described in subclause A.4.1.2a) of ISO 8327.

If the activity functional unit has been selected but no activity is in progress, any correctly received PR-RS SPDUs are stored until either an AR or AS SPDU is received, after which the stored SPDUs are dequeued. Whether an activity is in progress can be derived from bit 17 of the SPDU status component; this bit is set or reset in process *VactCon* (see 10.5.6.2).

```
--------------------------------------------------------------------------------------------*)
process SPDUOrd1[p]: noexit := SPDUOrd2[p](0 of Nat) endproc

process SPDUOrd2[p](pr:Nat): noexit :=
choice pd:ASPDU,fl:TFlow,v:Validity [] [Reset(v,14)] ->
(    [IsTD(pd) or IsEX(pd) or IsDT(pd) and Set(v,11) and Reset(v,16)] ->
     p !pd !fl ?d:Dir !v !(1 of TrState); SPDUOrd1[p]
[]   [res eq Preptype(pd) and Set(v,16) and Set(v,11) and Set(v,17)] ->
     p !pd !fl !rec !v !(1 of TrState); SPDUOrd2[p](Succ(pr))
[]   [IsAB(pd) and (r eq TDISPar(pd)) and Set(v,11) and Reset(v,16)] ->
     p !pd !fl !rec !v !(1 of TrState); SPDUOrd[p]
[]   REL[p] [] SYNMm[p] [] RSYN[p] [] Tok[p] [] Act[p](pr) [] CapDT[p] [] Exception[p]
[]   [not( IsFN(pd) or IsAB(pd) or IsDT(pd) or IsMAP(pd) or (res eq Preptype(pd)) or IsRS(pd) or
     IsGT(pd) or IsGTC(pd) or IsPT(pd) or IsAD(pd) or IsAI(pd) or IsAR(pd) or IsAE(pd) or IsAS(pd) or
     IsCD(pd) or IsMIP(pd) or IsMIA(pd) or IsER(pd) or IsED(pd)) and Reset(v,11) and Set(v,16)] ->
     p !pd !fl !rec !v !(1 of TrState); SPDUOrd2[p](pr)
)
endproc
(*--------------------------------------------------------------------------------------------
```

In process *REL*, an instance of *AwSRELrsp* or *AwDN* is invoked to describe respectively the states in which an S-RELEASE response and a DN SPDU is awaited. Bit 11 of the SPDU status component is set, to indicate that the received SPDU is in sequence. Bit 16 is reset, meaning that a corresponding S-RELEASE indication should be executed after receipt of a FN SPDU. This value is determined in *SSPSPDU* (see 10.3) only if all the bits 0 through 10 are set.

```
--------------------------------------------------------------------------------------------*)
process REL[p]: noexit :=
choice pd:ASPDU,fl:TFlow,v:Validity [] [IsFN(pd)] ->
(    [Set(v,11) and Reset(v,14) and Reset(v,16)] -> p !pd !fl !rec !v !(2 of TrState); AwSRELrsp[p]
[]   p !pd !fl !send !v ?st:TrState; AwDN[p]
)
endproc
(*--------------------------------------------------------------------------------------------
```