Summary

This Recommendation defines a protocol operating through the hierarchy of a multipont communication domain. It specifies the format of protocol messages and procedures governing their exchange over a set of transport connections. The purpose of the protocol is to implement the Multipoint Communication Service defined by ITU-T Rec. T.122.

30 October 1993
DRAFT Rec. T.125 (1994)

DRAFT RECOMMENDATION

MULTIPOINT COMMUNICATION SERVICE PROTOCOL SPECIFICATION

1 Scope

This Recommendation specifies:

a. procedures for a single protocol for the transfer of data and control information from one MCS provider to a peer MCS provider;

b. the structure and encoding of the MCS protocol data units used for the transfer of data and control information.

The procedures are defined in terms of:

a. the interactions between peer MCS providers through the exchange of MCS protocol data units;

b. the interactions between an MCS provider and MCS users through the exchange of MCS primitives;

c. the interactions between an MCS provider and a transport service provider through the exchange of transport service primitives;

These procedures are applicable to instances of multipeer communication among systems that support MCS and wish to interconnect in an open systems environment.

2 Normative references

The following Recommendations and International Standards contain provisions which, through reference in this text, constitute provisions of this Recommendation. At the time of publication, the editions indicated were valid. All Recommendations and Standards are subject to revision, and parties to agreements based on this Recommendation are encouraged to investigate the possibility of applying the most recent edition of the Recommendations and Standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards. The Telecommunications Standardization Bureau of the ITU maintains a list of currently valid CCITT Recommendations.

– ITU-T Recommendation T.122 (1993), *Multipoint Communication Service for audio-graphics and audio visual conferencing – Sservice definition*.

– ITU-T Recommendation T.123 (1993), *Protocol stacks for audiographic and audiovisual teleconference applications*.

– CCITT Recommendation X.200 (1988), *Reference model of Open Systems Interconnection for CCITT applications*.

– CCITT Recommendation X.214 (1988), *Transport service definition for Open Systems Interconnection for CCITT applications*.

– CCITT Recommendation X.208 (1988), *Specification of Abstract Syntax Notation One (ASN.1)*.

– CCITT Recommendation X.209 (1988), *Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*.

~~ISO/IEC 8825: 1990, *Information processing systems – Open Systems Interconnection – Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1).*~~

– ISO/IEC DIS~~CD~~ 8825-2.~~2~~: (JTC1/SC21 document out for ballot~~N7302, 1992-08-14~~), *Information processing systems – Open Systems Interconnection – Specification of Encoding Rules for ASN.1 – Part 2: Packed Encoding Rules*

3 Definitions

NOTE – These definitions make use of the abbreviations defined in clause 4.

This specification is based on the concepts developed in CCITT Rec. X.200 and makes use of the following terms defined therein:

a. flow control
b. reassembling
c. recombining
d. segmenting
e. sequencing
f. splitting
g. transfer syntax
h. transport connection
i. transport connection endpoint identifier
j. transport service
k. transport service access point
l. transport service access point address
m. transport service data unit

This specification is also based on the concepts developed in ITU-T Rec. T.122 and makes use of the following terms defined therein:

a. Control MCSAP
b. MCS attachment
c. MCS channel
d. MCS connection
e. MCS domain
f. MCS domain selector
g. MCS private channel
h. MCS private channel manager
i. MCS provider
j. MCS service access point
k. MCS user
l. MCS user id
m. Top MCS provider

For the purposes of this specification, the following definitions apply.

3.1 MCS service data unit: An amount of MCS user data whose identity is preserved during the transfer from transmitter to receivers. Specifically, the content of one MCS-SEND-DATA request or one MCS-UNIFORM-SEND-DATA request.

3.2 MCS interface data unit: The unit of information transferred across an MCSAP between an MCS user and an MCS provider in a single interaction. Each MCS interface data unit contains interface control information and may also contain all or part of an MCS service data unit.

3.3 MCS protocol data unit: A unit of information exchanged in the MCS protocol, consisting of control information transferred between MCS providers to coordinate their joint operation and possibly data transferred on behalf of MCS users to whom they are providing service.

3.4 MCS data transfer priority: One of four levels: top, high, medium, low. The value is communicated unchanged from transmitter to receivers. Depending on an MCS domain parameter for the number of distinct data transfer priorities implemented, two or more lowest priorities may receive the same quality of service.

3.5 valid MCSPDU: An MCSPDU whose structure and encoding complies with this specification.

3.6 invalid MCSPDU: An MCSPDU that is not a valid MCSPDU.

3.7 protocol error: Use of an MCSPDU in a manner inconsistent with the procedures of this specification.

3.8 connect MCSPDU: Any one of **`Connect-Initial`**, **`Connect-Response`**, **`Connect-Additional`**, **`Connect-Result`**.

3.9 domain MCSPDU: Any MCSPDU that is not a connect MCSPDU.

3.10 data MCSPDU: Any one of **`SDrq`**, **`SDin`**, **`USrq`**, **`USin`**.

3.11 control MCSPDU: Any domain MCSPDU that is not a data MCSPDU.

3.12 initial TC: The first transport connection of an MCS connection, used to exchange control MCSPDUs and data MCSPDUs of top priority.

3.13 additional TC: A subsequent transport connection belonging to an MCS connection, used to exchange data MCSPDUs of lesser priority.

3.14 subtree of an MCS provider: In the context of an MCS domain, this consists of the MCS provider itself and its MCS attachments plus all MCS providers hierarchically subordinate to it and their MCS attachments.

3.15 height of an MCS provider: In the context of an MCS domain, this is one more than the maximum height of all hierarchically subordinate MCS providers. An MCS provider without subordinates has height one.

4 Abbreviations

MCS	Multipoint Communication Service
MCSAP	MCS service access point
MCSPDU	MCS protocol data unit
TC	Transport connection
TS	Transport service
TSAP	Transport service access point
TSDU	Transport service data unit

5 Overview of the MCS protocol

5.1 Model of the MCS layer

An MCS provider communicates with MCS users through an MCSAP by means of the MCS primitives defined in ITU-T Rec. T.122. These primitives can be the cause or result of MCSPDU exchanges between peer MCS providers using an MCS connection, or they can be the cause or result of actions taken within a single MCS provider. MCSPDU exchanges occur between MCS providers that host the same MCS domain.

An MCS provider can have multiple peers, each reached directly by a different MCS connection or indirectly through a peer MCS provider. An MCS connection is composed of one or more transport connections, depending on the number of data transfer priorities implemented in an MCS domain. Protocol exchanges are effected using the services of the transport layer through a pair of TSAPs.

This model of the MCS layer is illustrated in Figure 5-1

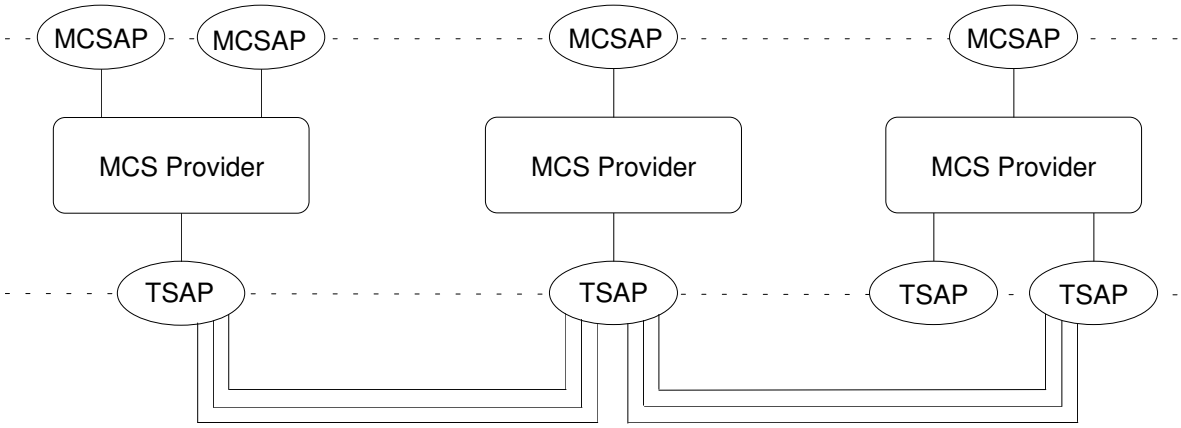


Figure 5-1 –Model of the MCS layer

5.2 Services provided by the MCS layer

The MCS protocol supports the services defined in ITU-T Rec. T.122. Information is transferred to and from an MCS user using the MCS primitives listed in Table 5-1.

Table 5-1 – MCS primitives

Functional Unit	Primitives	Associated MCSPDUs
Domain Management	MCS-CONNECT-PROVIDER request	**Connect-Initial**
	MCS-CONNECT-PROVIDER indication	**Connect-Initial**
	MCS-CONNECT-PROVIDER response	**Connect-Response**
	MCS-CONNECT-PROVIDER confirm	**Connect-Response**
	(side effects)	**Connect-Additional** **Connect-Result** **PDin EDrq** **MCrq MCcf PCin** **MTrq MTcf PTin**
	MCS-DISCONNECT-PROVIDER request	**DPum**
	MCS-DISCONNECT-PROVIDER indication	**DPum RJum**
	MCS-ATTACH-USER request	**AUrq**
	MCS-ATTACH-USER confirm	**AUcf**
	MCS-DETACH-USER request	**DUrq**
	MCS-DETACH-USER indication	**DUin** **MCcf PCin** **MTcf PTin**
Channel Management	MCS-CHANNEL-JOIN request	**CJrq**
	MCS-CHANNEL-JOIN confirm	**CJcf**
	MCS-CHANNEL-LEAVE request	**CLrq**
	MCS-CHANNEL-LEAVE indication	**MCcf PCin**
	MCS-CHANNEL-CONVENE request	**CCrq**
	MCS-CHANNEL-CONVENE confirm	**CCcf**
	MCS-CHANNEL-DISBAND request	**CDrq**
	MCS-CHANNEL-DISBAND indication	**MCcf PCin**
	MCS-CHANNEL-ADMIT request	**CArq**
	MCS-CHANNEL-ADMIT indication	**CAin**
	MCS-CHANNEL-EXPEL request	**CErq**
	MCS-CHANNEL-EXPEL indication	**CEin CDin** **MCcf PCin**

Table 5-1 (concluded)

Functional Unit	Primitives	Associated MCSPDUs
Data Transfer	MCS-SEND-DATA request MCS-SEND-DATA indication	**SDrq** **SDin**
	MCS-UNIFORM-SEND-DATA request MCS-UNIFORM-SEND-DATA indication	**USrq** **USin**
Token Management	MCS-TOKEN-GRAB request MCS-TOKEN-GRAB confirm	**TGrq** **TGcf**
	MCS-TOKEN-INHIBIT request MCS-TOKEN-INHIBIT confirm	**TIrq** **TIcf**
	MCS-TOKEN-GIVE request MCS-TOKEN-GIVE indication MCS-TOKEN-GIVE response MCS-TOKEN-GIVE confirm	**TVrq** **TVin** **TVrs** **TVcf**
	MCS-TOKEN-PLEASE request MCS-TOKEN-PLEASE indication	**TPrq** **TPin**
	MCS-TOKEN-RELEASE request MCS-TOKEN-RELEASE confirm	**TRrq** **TRcf**
	MCS-TOKEN-TEST request MCS-TOKEN-TEST confirm	**TTrq** **TTcf**

5.3 Services assumed from the transport layer

The MCS protocol assumes the use of a subset of the connection-oriented transport service defined in CCITT Rec. X.214. Information is transferred to and from a TS provider by the primitives listed in Table 5-2.

Table 5-2 –Transport service primitives

Primitives	Use	Parameters	Use
T-CONNECT request T-CONNECT indication	X X	Called address Calling address Expedited data option Quality of service TS user-data	X X - X -
T-CONNECT response T-CONNECT confirm	X X	Responding address Expedited data option Quality of service TS user-data	- - X -
T-DATA request T-DATA indication	X X	TS user-data	X
T-EXPEDITED-DATA request T-EXPEDITED-DATA indication	- -	TS user-data	-
T-DISCONNECT request	X	TS user-data	-
T-DISCONNECT indication	X	Reason TS user-data	- -

X: The MCS protocol assumes that this feature is always available.
-: The MCS protocol does not use this feature.

5.4 Functions of the MCS layer

Table 5-1 identifies the functional units of MCS and the MCSPDUs associated with each MCS primitive. MCSPDUs are defined in clause 7. The relationship between primitives and MCSPDUs can be as simple as cause and effect, in either direction. For example, MCS-ATTACH-USER request generates **AUrq**, and **AUcf** generates MCS-ATTACH-USER confirm. Other cases may be more complicated. The completion of MCS-CONNECT-PROVIDER, for example, requires the exchange of additional MCSPDUs as side effects of the four-phase primitive. And any one of five MCSPDUs can cause an MCS-DETACH-USER indication, any of four an MCS-CHANNEL-EXPEL indication.

5.4.1 Domain management

The MCS layer maintains the integrity of the MCS connections comprising an MCS domain. An MCS connection is oriented, with one end hierarchically superior to the other. There is a single MCS provider at the top of each domain.

Establishing an MCS connection merges two domains into one. The MCS layer ensures that one top provider remains. It resolves any conflicts of unique identity or exclusive ownership that may arise.

Disconnecting an MCS connection splits a domain into two portions. The portion containing the top provider survives. The bottom portion eradicates itself.

The MCS layer uniquely identifies users attached to a domain. Users may become aware of each other through their interactions via MCS primitives. The MCS layer notifies all users of a domain when one of them detaches. The MCS layer recovers any resources of a detached user.

5.4.2 Channel management

The MCS layer records which parts of an MCS domain contain one or more users joined to a given channel, so that it can optimize the transfer of data to destinations that wish to receive it.

The MCS layer treats user ids as single-member channels, which only the designated user is allowed to join. On request, it can create private channels to which only admitted users are allowed access or assign public channels to which no other users are currently joined.

5.4.3 Data transfer

The MCS layer maintains a sequenced flow of data to the users who have joined a channel. A channel becomes, in effect, a multicast distribution list, with a range somewhere between zero destinations and a complete broadcast.

By default, the MCS layer routes data to each receiver over the shortest path of MCS connections. Optionally, it routes specified MCS service data units through the top MCS provider, thereby guaranteeing their uniform receipt at all receivers, which may include the transmitter too.

The MCS layer recognizes one or more priorities of data transfer and extends them preferential processing. Through segmentation, it allows MCS service data units of unlimited size.

The MCS layer regulates the global flow of data within a domain. The inability of a receiver to accept data at the rate it is offered eventually creates back-pressure that causes transmitters to be blocked. A user may be detached involuntarily if it fails to maintain a minimum receiving rate.

The MCS layer guarantees error-free receipt of transmitted data, as long as the source and destination users remains attached and the destination user remains joined to the channel. However, higher priority data takes precedence, and a surfeit of it may indefinitely delay the delivery of lower priority data.

5.4.4 Token management

The MCS layer implements token operations at the top MCS provider, thereby ensuring consistency and exclusion.

5.5 Hierarchical processing

Hierarchical processing in an MCS domain is illustrated in Figure 5-2.

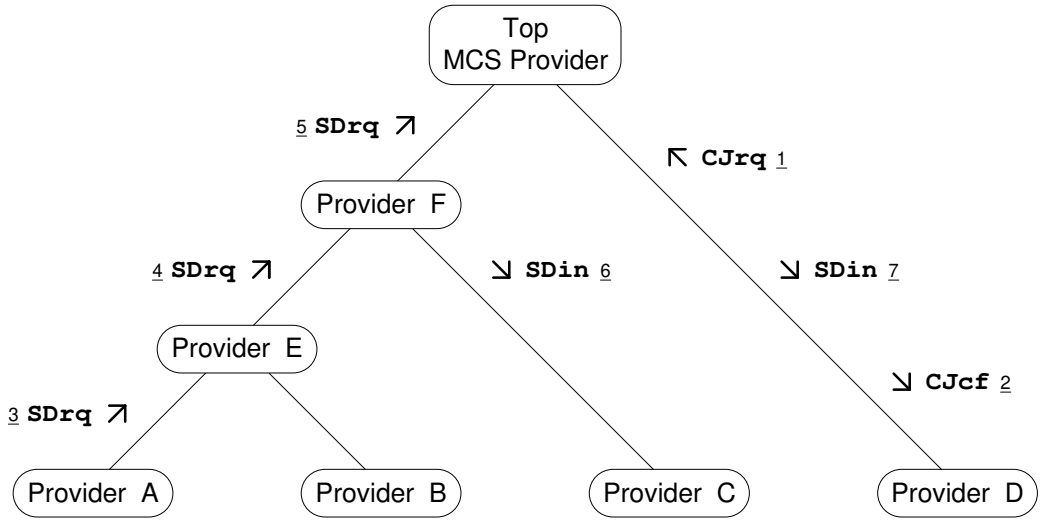


Figure 5-2 – Hierarchical processing in an MCS domain

The nodes in the figure represent MCS providers, and the labeled arrows represent MCSPDUs. This example focuses on a period of time after the domain has been established through connections between MCS providers and the use of data transfer is beginning to expand. At step 1 provider D requests on behalf of a user to join a channel over which data will be distributed, and at step 2 the request is confirmed as successful. At step 3 a user attached to provider A sends data, and the corresponding **SDrq** begins to flow upward. Assuming that only attachments at providers A, C, and D are joined to the channel over which the data is being sent, the request MCSPDU is reflected downward at steps 6 and 7 as **SDin**. Provider E, aware that no other subordinate needs to receive the data, simply forwards **SDrq** upward at step 4. Provider F forwards **SDrq** upward at step 5 but also reflects it downward at step 6, knowing that provider C has expressed interest in the channel.

MCS providers are not especially concerned with their height in the hierarchy, except for their role in maintaining an overall limit on the height of the domain and in the broad sense that they are either the top provider or they are not . The top MCS provider has no upward connection. All others have exactly one.

An MCS provider records information about channels and tokens used in its subtree of an MCS domain.

An MCS provider records channels that are joined by users within the subtree and, for each such channel, where the joins originate, that is, from which attachments and from which downward MCS connections. It records user ids that are assigned in the subtree and where they originate. It records private channels that have either a manager or an admitted user in the subtree, and it records the associated user ids.

An MCS provider records tokens that are grabbed or inhibited by users within the subtree, and it records the associated user ids.

An MCS provider inspects requests arising from its subtree to verify that the initiating user id is legitimately assigned to the originating attachment or downward MCS connection. This creates rings of protection around the top MCS provider and limits how much disruption a malicious participant can cause in an otherwise cooperative domain.

In general terms, to which there are several exceptions, the operation of the MCS layer can be described as follows.

a. An MCS primitive request invoked at an MCS attachment generates an MCSPDU at the corresponding MCS provider and dispatches it upward towards the top MCS provider. There, where full information about the MCS domain is held, the MCSPDU is acted on.

b. A confirm MCSPDU may be generated at the top MCS provider to return results to the requesting attachment. MCS providers that pass it along update their records according to the impact of the operation on their subtree. A confirm is routed to the initiating user id by consulting local records at each successive downward hop.

c. An indication MCSPDU may be generated instead to inform other attachments about the action taken. Indications may be replicated and sent downward on several connections that lead to affected users. MCS providers may also update their records with the impact of an operation as part of processing an indication.

The preceding description is an overview intended to create a conceptual framework. Full details are specified in later clauses concerning the information recorded in an MCS provider and how specific MCSPDUs are processed.

Among exceptions are the following: Some requests, notably **CJrq** and **CLrq**, may stop rising at a level short of the top provider, and some indications, notably **SDin**, may be generated at a level below the top provider. One MCSPDU, **TVrs**, belongs to the response category. And some MCSPDUs may be generated by an MCS provider as a continuation of processing unlike MCSPDUs, such as **CLrq** following **DUin**.

5.6 Domain parameters

The MCS providers hosting a single MCS domain allocate resources and execute procedures according to the following parameters. The values of these parameters are identical throughout a domain.

a. Maximum number of MCS channels that may be in use simultaneously. This includes channels that are joined by any user, user ids that have been assigned, and private channels that have been created.

b. Maximum number of user ids that may be assigned simultaneously. This is a sublimit within the constraint of the preceding parameter.

c. Maximum number of token ids that may be grabbed or inhibited simultaneously.

d. Number of data transfer priorities implemented. This equals the number of TCs in an MCS connection. An MCS user may still send and receive data with priorities outside the limit. However, such priorities may be treated the same as the lowest priority that is implemented.

e. Enforced throughput. Although global flow control limits data transfer within a domain to the rate of the slowest receiver, receivers must not be allowed to run arbitrarily slowly. Otherwise, one party in a conference may obstruct all others. This parameter instructs MCS providers to enforce a minimum receiving rate at each MCS attachment and over each downward MCS connection. Violators run the risk of being involuntarily detached or disconnected, respectively.

f. Maximum height. This constrains the height of all MCS providers, in particular the top provider.

g. Maximum size of domain MCSPDUs. Global flow control is based on buffering domain MCSPDUs within an MCS provider (but not connect MCSPDUs). For simplicity, fixed-size buffers are assumed. An MCS provider shall not generate larger MCSPDUs. This constrains the amount of information that can be

packed into a single control MCSPDU and suggests where unlimited user data should be segmented in data MCSPDUs.

h. Protocol version. This takes one of two values defining different encodings for domain MCSPDUs.

NOTE – A given instance of an MCS provider may operate with local resource constraints that are also parameterized. These may include the amount of memory available for buffering MCSPDUs awaiting transport, the maximum number of MCS attachments, and the maximum number of MCS connections to other providers. Such parameters are local matters and are not communicated across an MCS domain.

6 Use of the transport service

6.1 Model of the transport service

This description paraphrases relevant parts of CCITT Rec. X.214 assuming that no use is made of expedited data.

The transport service offers these features to a TS user:

a. Means to establish a TC with another TS user for the purpose of exchanging TSDUs. More than one TC may exist between the same pair of TS users.

b. Associated with each TC at its time of establishment, the opportunity to request, negotiate, and have agreed by the TS provider a certain Quality of Service as specified by parameters representing characteristics such as throughput, transit delay, residual error rate, and priority.

c. Means of transferring TSDUs on a TC. The transfer of TSDUs, which consist of an integral number of octets, is transparent, in that the boundaries of TSDUs and the contents of TSDUs are preserved unchanged by the TS provider.

d. Means by which a receiving TS user may control the rate at which the sending TS user may send data.

e. The unconditional and therefore possibly destructive release of a TC.

The operation of a TC is modeled in an abstract way by a pair of queues linking two TSAPs. There is one queue for each direction of information flow. Each TC is modeled by a separate pair of queues.

The queue model is used to express the flow control feature. A queue has a limited capacity, but this capacity is not necessarily either fixed or determinable. Connect, TSDU, and disconnect objects are entered and removed from a queue as the result of interactions at the two TSAPs. The ability of a TS user to add objects to a queue is determined by the behavior of the TS user removing objects from that queue and the state of the queue. The only objects that can be placed in a queue by the TS provider are disconnect objects. Objects are added to a queue subject to control by the TS provider. Objects are normally removed from the queue subject to control by the receiving TS user. Objects are normally removed in the same order that they were added. The only exception to normal removal is that an object may be deleted by the TS provider if, and only if, the following one is a disconnect object.

A TC endpoint identification mechanism must be provided locally if the TS user and the TS provider need to distinguish between several TCs at a TSAP. All primitives must then make use of this identification mechanism to identify the TC to which they apply. This implicit identification is not shown as a parameter of the TS primitives and must not be confused with the address parameters of T-CONNECT.

6.2 Use of multiple connections

An MCS connection consists of one or more TCs between the same pair of MCS providers. The first TC established is called the initial TC; those established subsequently are called additional TCs. All the TCs that belong to one MCS connection are established by the same MCS provider, in reaction to an MCS-CONNECT-PROVIDER request. This request contains address parameters that are the calling and called TSAP addresses. These are used unmodified in the T-CONNECT requests that result.

The number of TCs per MCS connection is uniform throughout an MCS domain. This domain parameter equals the number of data transfer priority levels implemented. Separate TCs are required because each is a vehicle for flow control. Blockages in lower priority data should not result in back pressure against higher priority data. To be fully implemented, lower and higher priority data must be carried on different TCs.

The quality of service requested for a TC may vary depending on the data priority for which it is established. These quality of service targets need not be uniform across an MCS domain.

Aspects of quality of service that are of interest include maximum or average throughput and transit delay. High priority data may favor low transit delay for real-time response but may not require high throughput. Low priority data, on the other hand, may favor high throughput for bulk transfers but may not require low transit delay.

TC priority is another aspect of quality of service, although it does not align precisely with the concept of MCS data transfer priority. It specifies the relative order in which TCs are to have their quality of service degraded, if necessary. High TC priority may be requested along with other characteristics, like low transit delay, to ensure that MCS priority data receives the preferential treatment it deserves.

Connect MCSPDUs occur only as the first TSDU carried in either direction of a TC. **Connect-Initial** and **Connect-Response** traverse the initial TC of an MCS connection. **Connect-Additional** and **Connect-Result** traverse additional TCs, if any.

A calling MCS provider, issuing T-CONNECT requests, controls through its own actions which TCs are part of the same MCS connection and what data transfer priorities they represent.

A called MCS provider, receiving T-CONNECT indications, must in general accept a TC and read its first TSDU before learning its significance. **Connect-Initial** identifies an incoming TC as the beginning of a new MCS connection. **Connect-Additional** identifies an incoming TC as part of an MCS connection in progress.

Connect-Additional contains a value assigned by the called MCS provider and conveyed to the calling MCS provider in a **Connect-Response** over the initial TC that designates the additional TC as belonging to the same MCS connection. **Connect-Additional** also states explicitly the data priority that the TC represents.

Connect MCSPDUs are exchanged immediately following TC establishment. Once an MCS connection has been confirmed, it becomes part of a hierarchical MCS domain. Thereafter the MCS connection conveys domain MCSPDUs

With the exception of data MCSPDUs, domain MCSPDUs traverse the initial TC of an MCS connection. Data MCSPDUs traverse the TC that corresponds to their data priority. If the priority specified is beyond the number implemented in an MCS domain, a data MCSPDU traverses the TC of the lowest priority that is implemented.

If only one priority is implemented in an MCS domain, its MCS connections each consist of a single TC, no use is made of **Connect-Additional** or **Connect-Result**, and all MCSPDUs travel in sequence between providers.

6.3 Transport connection release

A TS provider adds reliability to end-to-end connections by executing enough protocol to compensate for any weakness in underlying network services. An MCS provider does not duplicate this functionality. It does not attempt further automatic recovery in the event of a transport failure.

Unrecoverable errors are announced through a T-DISCONNECT indication. If any of the TCs belonging to an MCS connection is disconnected, the others are immediately disconnected too. Unless this was requested by the user, an MCS-DISCONNECT-PROVIDER indication is generated, and the reason is given as provider-initiated.

An MCS-DISCONNECT-PROVIDER request, on the other hand, should appear as an indication at the other side with the reason given as user-requested. Despite assurances in X.214, the simplest class of transport protocol does not allow passing user data in T-DISCONNECT. Hence, the disconnect reason code is transferred in an explicit MCSPDU. This MCSPDU compels an MCS provider, upon receipt, to disconnect the MCS connection that conveyed it.

7 Structure of MCSPDUs

The structure of MCSPDUs is specified using the notation ASN.1 of CCITT Rec. X.208. The use and significance of these MCSPDUs is further described in clauses 9 and 10.

```
MCS-PROTOCOL DEFINITIONS ::=
BEGIN

-- Part 1: Fundamental MCS types

ChannelId        ::= INTEGER (0..65535)          -- range is 16 bits

StaticChannelId  ::= ChannelId (1..1000)         -- those known permanently

DynamicChannelId::= ChannelId (1001..65535)      -- those created and deleted
```

```
UserId              ::= DynamicChannelId            -- created by Attach-User
                                                    -- deleted by Detach-User

PrivateChannelId::= DynamicChannelId                -- created by Channel-Convene
                                                    -- deleted by Channel-Disband

AssignedChannelId::=DynamicChannelId                -- created by Channel-Join zero
                                                    -- deleted by last Channel-Leave

TokenId             ::= INTEGER (1..65535)          -- all are known permanently

TokenStatus         ::= ENUMERATED
{
                            notInUse                (0),
                            selfGrabbed             (1),
                            otherGrabbed            (2),
                            selfInhibited           (3),
                            otherInhibited          (4),
                            selfRecipient           (5),
                            selfGiving              (6),
                            otherGiving             (7)
}

DataPriority        ::= ENUMERATED
{
                            top                     (0),
                            high                    (1),
                            medium                  (2),
                            low                     (3)
}

Segmentation        ::= BIT STRING
{
                            begin                   (0),
                            end                     (1)
} (SIZE (2))

DomainParameters::= SEQUENCE
{
        maxChannelIds   INTEGER (0..MAX),
                        -- a limit on channel ids in use,
                        -- static + user id + private + assigned
        maxUserIds      INTEGER (0..MAX),
                        -- a sublimit on user id channels alone
        maxTokenIds     INTEGER (0..MAX),
                        -- a limit on token ids in use
                        -- grabbed + inhibited + giving + ungivable + given
        numPriorities   INTEGER (0..MAX),
                        -- the number of TCs in an MCS connection
        minThroughput   INTEGER (0..MAX),
                        -- the enforced number of octets per second
        maxHeight       INTEGER (0..MAX),
                        -- a limit on the height of a provider
        maxMCSPDUsize   INTEGER (0..MAX),
                        -- an octet limit on domain MCSPDUs
        protocolVersion INTEGER (0..MAX)
}

-- Part 2:  Connect provider

Connect-Initial         ::= [APPLICATION 101] IMPLICIT SEQUENCE
{
        callingDomainSelector   OCTET STRING,
        calledDomainSelector    OCTET STRING,
        upwardFlag              BOOLEAN,
                                -- TRUE if called provider is higher
        targetParameters        DomainParameters,
```

```
        minimumParameters       DomainParameters,
        maximumParameters       DomainParameters,
        userData                OCTET STRING
}

Connect-Response        ::= [APPLICATION 102] IMPLICIT SEQUENCE
{
        result                  Result,
        calledConnectId         INTEGER (0..MAX),
                                -- assigned by the called provider
                                -- to identify additional TCs of
                                -- the same MCS connection
        domainParameters        DomainParameters,
        userData                OCTET STRING
}

Connect-Additional      ::= [APPLICATION 103] IMPLICIT SEQUENCE
{
        calledConnectId         INTEGER (0..MAX),
        dataPriority            DataPriority
}

Connect-Result          ::= [APPLICATION 104] IMPLICIT SEQUENCE
{
        result                  Result
}

-- Part 3:  Merge domain

PDin    ::= [APPLICATION 0] IMPLICIT SEQUENCE     -- plumb domain indication
{
        heightLimit             INTEGER (0..MAX)
                                -- a restriction on the MCSPDU receiver
}

EDrq    ::= [APPLICATION 1] IMPLICIT SEQUENCE     -- erect domain request
{
        subHeight               INTEGER (0..MAX),
                                -- height in domain of the MCSPDU transmitter
        subInterval             INTEGER (0..MAX)
                                -- its throughput enforcement interval in milliseconds
}

ChannelAttributes ::= CHOICE
{
        static                  [0] IMPLICIT SEQUENCE
        {
                channelId               StaticChannelId,
                                        -- joined is implicitly TRUE
        },
        userId                  [1] IMPLICIT SEQUENCE
        {
                userId                  UserId,
                joined                  BOOLEAN,
                                        -- TRUE if user is joined to its user id
                userId                  UserId
        },
        private                 [2] IMPLICIT SEQUENCE
        {
                joined                  BOOLEAN,
                                        -- TRUE if channel id is joined below
                channelId               PrivateChannelId,
                manager                 UserId,
                joined                  BOOLEAN,
                                        -- TRUE if channel id is joined below
                admitted                SET OF UserId
                                        -- may span multiple MCrq
```

```
        },
        assigned                [3] IMPLICIT SEQUENCE
        {
                channelId               AssignedChannelId
                                        -- joined is implicitly TRUE
        }
}

MCrq    ::= [APPLICATION 2] IMPLICIT SEQUENCE     -- merge channels request
{
        mergeChannels           SET OF ChannelAttributes,
        purgeChannelIds         SET OF ChannelId
}

MCcf    ::= [APPLICATION 3] IMPLICIT SEQUENCE     -- merge channels confirm
{
        mergeChannels           SET OF ChannelAttributes,
        purgeChannelIds         SET OF ChannelId
}

PCin    ::= [APPLICATION 4] IMPLICIT SEQUENCE     -- purge channels indication
{
        detachUserChannelIds        SET OF UserChannelId,
                                -- purge user id channels
        leaveChannelIds         SET OF ChannelId,
                                -- purge static or assigned channels
        purgedisbandChannelIds      SET OF ChannelId
                                -- purge otherprivate channels
}

TokenAttributes ::= CHOICE
{
        grabbed                 [0] IMPLICIT SEQUENCE
        {
                tokenId                 TokenId,
                grabber                 UserId
        },
        inhibited               [1] IMPLICIT SEQUENCE
        {
                tokenId                 TokenId,
                inhibitors              SET OF UserId
                                        -- may span multiple MTrq
        },
        giving                  [2] IMPLICIT SEQUENCE
        {
                tokenId                 TokenId,
                grabber                 UserId,
                recipient               UserId
        },
        ungivable               [3] IMPLICIT SEQUENCE
        {
                tokenId                 TokenId,
                grabber                 UserId
                -- recipient has since detached
        },
        given                   [4] IMPLICIT SEQUENCE
        {
                tokenId                 TokenId,
                recipient               UserId
                -- grabber released or detached
        }
}

MTrq    ::= [APPLICATION 5] IMPLICIT SEQUENCE     -- merge tokens request
{
        mergeTokens             SET OF TokenAttributes,
```

```
        purgeTokenIds           SET OF TokenId
}

MTcf    ::= [APPLICATION 6] IMPLICIT SEQUENCE     -- merge tokens indication
{
        mergeTokens             SET OF TokenAttributes,
        purgeTokenIds           SET OF TokenId
}

PTin    ::= [APPLICATION 7] IMPLICIT SEQUENCE     -- purge tokens indication
{
        purgeTokenIds           SET OF TokenId
}
```

-- Part 4: Disconnect provider

```
DPum    ::= [APPLICATION 8] IMPLICIT SEQUENCE     -- disconnect provider ultimatum
{
        reason                  Reason
}

RJum    ::= [APPLICATION 9] IMPLICIT SEQUENCE     -- reject MCSPDU ultimatum
{
        diagnostic              Diagnostic,
        initialOctets           OCTET STRING
}
```

-- Part 5: Attach/Detach user

```
AUrq    ::= [APPLICATION 10] IMPLICIT SEQUENCE    -- attach user request
{
}

AUcf    ::= [APPLICATION 11] IMPLICIT SEQUENCE    -- attach user confirm
{
        result                  Result,
        initiator               UserId OPTIONAL
}

DUrq    ::= [APPLICATION 12] IMPLICIT SEQUENCE    -- detach user request
{
        reason                  Reason,
        userIds                 SET OF UserId
}
DUin    ::= [APPLICATION 13] IMPLICIT SEQUENCE    -- detach user indication
{
        reason                  Reason,
        userIds                 SET OF UserId
}
```

-- Part 6: Channel management

```
CJrq    ::= [APPLICATION 14] IMPLICIT SEQUENCE    -- channel join request
{
        initiator               UserId,
        channelId               ChannelId
                                -- may be zero
}

CJcf    ::= [APPLICATION 15] IMPLICIT SEQUENCE    -- channel join confirm
{
        result                  Result,
        initiator               UserId,
        requested               ChannelId,
                                -- may be zero
        channelId               ChannelId OPTIONAL
                                -- same as in CJrq, except zero may be
```

```
                                        -- replaced by an assigned channel id
}

CLrq    ::= [APPLICATION 16] IMPLICIT SEQUENCE    -- channel leave request
{
        channelIds              SET OF ChannelId
}

CCrq    ::= [APPLICATION 17] IMPLICIT SEQUENCE    -- channel convene request
{
        initiator               UserId
}

CCcf    ::= [APPLICATION 18] IMPLICIT SEQUENCE    -- channel convene confirm
{
        result                  Result,
        initiator               UserId,
        channelId               PrivateChannelId OPTIONAL
}

CDrq    ::= [APPLICATION 19] IMPLICIT SEQUENCE    -- channel disband request
{
        initiator               UserId,
        channelId               PrivateChannelId
}

CDin    ::= [APPLICATION 20] IMPLICIT SEQUENCE    -- channel disband indication
{
        channelId               PrivateChannelId
}

CArq    ::= [APPLICATION 21] IMPLICIT SEQUENCE    -- channel admit request
{
        initiator               UserId,
        channelId               PrivateChannelId,
        userIds                 SET OF UserId
}

CAin    ::= [APPLICATION 22] IMPLICIT SEQUENCE    -- channel admit indication
{
        initiator               UserId,
        channelId               PrivateChannelId,
        userIds                 SET OF UserId
}

CErq    ::= [APPLICATION 23] IMPLICIT SEQUENCE    -- channel expel request
{
        initiator               UserId,
        channelId               PrivateChannelId,
        userIds                 SET OF UserId
}

CEin    ::= [APPLICATION 24] IMPLICIT SEQUENCE    -- channel expel indication
{
        channelId               PrivateChannelId,
        userIds                 SET OF UserId
}

-- Part 7:  Data transfer

SDrq    ::= [APPLICATION 25] IMPLICIT SEQUENCE    -- send data request
{
        initiator               UserId,
        channelId               ChannelId,
        dataPriority            DataPriority,
        segmentation            Segmentation,
        userData                OCTET STRING
}
```

```
SDin    ::= [APPLICATION 26] IMPLICIT SEQUENCE    -- send data indication
{
        initiator               UserId,
        channelId               ChannelId,
        dataPriority            DataPriority,
        segmentation            Segmentation,
        userData                OCTET STRING
}

USrq    ::= [APPLICATION 27] IMPLICIT SEQUENCE    -- uniform send data request
{
        initiator               UserId,
        channelId               ChannelId,
        dataPriority            DataPriority,
        segmentation            Segmentation,
        userData                OCTET STRING
}

USin    ::= [APPLICATION 28] IMPLICIT SEQUENCE    -- uniform send data indication
{
        initiator               UserId,
        channelId               ChannelId,
        dataPriority            DataPriority,
        segmentation            Segmentation,
        userData                OCTET STRING
}

-- Part 8:  Token management

TGrq    ::= [APPLICATION 29] IMPLICIT SEQUENCE    -- token grab request
{
        initiator               UserId,
        tokenId                 TokenId
}

TGcf    ::= [APPLICATION 30] IMPLICIT SEQUENCE    -- token grab confirm
{
        result                  Result,
        initiator               UserId,
        tokenId                 TokenId,
        tokenStatus             TokenStatus
}

TIrq    ::= [APPLICATION 31] IMPLICIT SEQUENCE    -- token inhibit request
{
        initiator               UserId,
        tokenId                 TokenId
}

TIcf    ::= [APPLICATION 32] IMPLICIT SEQUENCE    -- token inhibit confirm
{
        result                  Result,
        initiator               UserId,
        tokenId                 TokenId,
        tokenStatus             TokenStatus
}

TVrq    ::= [APPLICATION 33] IMPLICIT SEQUENCE    -- token give request
{
        initiator               UserId,
        tokenId                 TokenId,
        recipient               UserId
}

TVin    ::= [APPLICATION 34] IMPLICIT SEQUENCE    -- token give indication
{
        initiator               UserId,
```

```
        tokenId                 TokenId,
        recipient               UserId
}

TVrs    ::= [APPLICATION 35] IMPLICIT SEQUENCE    -- token give response
{
        result                  Result,
        recipient               UserId,
        tokenId                 TokenId
}

TVcf    ::= [APPLICATION 36] IMPLICIT SEQUENCE    -- token give confirm
{
        result                  Result,
        initiator               UserId,
        tokenId                 TokenId,
        tokenStatus             TokenStatus
}

TPrq    ::= [APPLICATION 37] IMPLICIT SEQUENCE    -- token please request
{
        initiator               UserId,
        tokenId                 TokenId
}

TPin    ::= [APPLICATION 38] IMPLICIT SEQUENCE    -- token please indication
{
        initiator               UserId,
        tokenId                 TokenId
}

TRrq    ::= [APPLICATION 39] IMPLICIT SEQUENCE    -- token release request
{
        initiator               UserId,
        tokenId                 TokenId
}

TRcf    ::= [APPLICATION 40] IMPLICIT SEQUENCE    -- token release confirm
{
        result                  Result,
        initiator               UserId,
        tokenId                 TokenId,
        tokenStatus             TokenStatus
}

TTrq    ::= [APPLICATION 41] IMPLICIT SEQUENCE    -- token test request
{
        initiator               UserId,
        tokenId                 TokenId
}

TTcf    ::= [APPLICATION 42] IMPLICIT SEQUENCE    -- token test confirm
{
        initiator               UserId,
        tokenId                 TokenId,
        tokenStatus             TokenStatus
}

-- Part 9:  Status codes

Reason          ::= ENUMERATED          -- in DPum, DUrq, DUin
{
                rn-domain-disconnected          (0),
                rn-provider-initiated           (1),
                rn-token-purged                 (2),
                rn-user-requested               (3),
                rn-channel-purged               (4)
}
```

```
Result              ::= ENUMERATED          -- in Connect, response, confirm
{
                    rt-successful                       (0),
                    rt-domain-merging                   (1),
                    rt-domain-not-hierarchical          (2),
                    rt-no-such-channel                  (3),
                    rt-no-such-domain                   (4),
                    rt-no-such-user                     (5),
                    rt-not-admitted                     (6),
                    rt-other-user-id                    (7),
                    rt-parameters-unacceptable          (8),
                    rt-token-not-available              (9),
                    rt-token-not-possessed             (10),
                    rt-too-many-channels               (11),
                    rt-too-many-tokens                 (12),
                    rt-too-many-users                  (13),
                    rt-unspecified-failure             (14),
                    rt-user-rejected                   (15)
}

Diagnostic          ::= ENUMERATED          -- in RJum
{
                    dc-inconsistent-merge               (0),
                    dc-forbidden-PDU-downward           (1),
                    dc-forbidden-PDU-upward             (2),
                    dc-invalid-BER-encoding             (3),
                    dc-invalid-PER-encoding             (4),
                    dc-misrouted-user                   (5),
                    dc-unrequested-confirm              (6),
                    dc-wrong-transport-priority         (7),
                    dc-channel-id-conflict              (8),
                    dc-token-id-conflict                (9),
                    dc-not-user-id-channel             (10),
                    dc-too-many-channels               (11),
                    dc-too-many-tokens                 (12),
                    dc-too-many-users                  (13)
}
```

-- Part 10: MCSPDU repertoire

```
ConnectMCSPDU       ::= CHOICE
{
        connect-initial         Connect-Initial,
        connect-response        Connect-Response,
        connect-additional      Connect-Additional,
        connect-result          Connect-Result
}

DomainMCSPDU        ::= CHOICE
{
        pdin                    PDin,
        edrq                    EDrq,
        mcrq                    MCrq,
        mccf                    MCcf,
        pcin                    PCin,
        mtrq                    MTrq,
        mtcf                    MTcf,
        ptin                    PTin,
        dpum                    DPum,
        rjum                    RJum,
        aurq                    AUrq,
        aucf                    AUcf,
        durq                    DUrq,
        duin                    DUin,
        cjrq                    CJrq,
        cjcf                    CJcf,
```

```
        clrq                    CLrq,
        ccrq                    CCrq,
        cccf                    CCcf,
        cdrq                    CDrq,
        cdin                    CDin,
        carq                    CArq,
        cain                    CAin,
        cerq                    CErq,
        cein                    CEin,
        sdrq                    SDrq,
        sdin                    SDin,
        usrq                    USrq,
        usin                    USin,
        tgrq                    TGrq,
        tgcf                    TGcf,
        tirq                    TIrq,
        ticf                    TIcf,
        tvrq                    TVrq,
        tvin                    TVin,
        tvrs                    TVrs,
        tvcf                    TVcf,
        tprq                    TPrq,
        tpin                    TPin,
        trrq                    TRrq,
        trcf                    TRcf,
        ttrq                    TTrq,
        ttcf                    TTcf
}

END
```

8 Encoding of MCSPDUs

Each MCSPDU is transported as one TSDU across a TC belonging to an MCS connection. Connect MCSPDUs are unlimited in size. Domain MCSPDUs are limited in size by a parameter of the MCS domain.

A standard ASN.1 data value encoding is used to transfer MCSPDUs between peer MCS providers. Two versions of this protocol are defined, differing only in the specification of encoding rules:

Version 1 uses the Basic Encoding Rules of CCITT Rec. X.209 for all MCSPDUs.

Version 2 uses the Basic Encoding Rules for connect MCSPDUs and the Packed Encoding Rules of ISO/IEC 8825-2 for all subsequent domain MCSPDUs. Specifically, the ALIGNED variant of BASIC-PER shall be applied to the ASN.1 type **DomainMCSPDU**. The bit string produced shall be conveyed as an integral number of octets. The leading bit of this string shall coincide with the most significant bit of the first octet.

Appendix I provides an example of a data transfer MCSPDU under the alternative encodings of Version 1 and 2.

Negotiation of the protocol version involves the exchange of a **Connect-Initial** and a **Connect-Response** MCSPDU over the initial TC. These two MCSPDUs are always encoded by the Basic Encoding Rules. So are any **Connect-Additional** and **Connect-Result** MCSPDUs that follow, prior to the onset of domain MCSPDUs. Domain MCSPDUs begin with the second TSDU transmitted over a TC.

Version 2 of this protocol shall not be used until the Packed Encoding Rules have been adopted as part of a CCITT Recommendation or ISO/IEC International Standard.

NOTES

1 The Packed Encoding Rules yield more compact MCSPDU headers~~, but they have not yet been finalized. A revision of CCITT Rec. X.209 is expected to include them as part of common text with the forthcoming ISO/IEC 8825~~.

2 Both BER and PER are self-delimiting, in the sense that they contain enough information to locate the end of each encoded MCSPDU. It might be argued that this makes the use of TSDUs unnecessary and that this protocol could be implemented over non-standard transport services that convey octet streams without preserving TSDU boundaries. However, such an approach is more vulnerable to implementation errors. If the boundary between MCSPDUs were ever lost, recovery would be difficult.

9 Routing of MCSPDUs

9.1 Connect MCSPDUs

Figure 9-1 specifies the exchange of connect MCSPDUs.

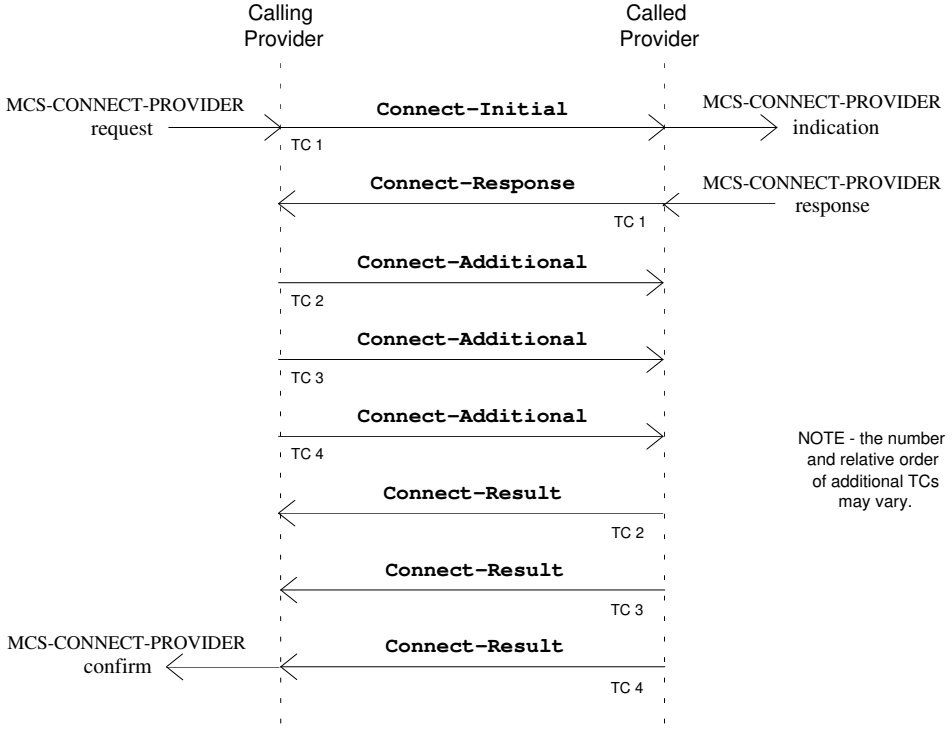


Figure 9-1 – Message flow of connect MCSPDUs

With the receipt of a **Connect-Response**, the calling MCS provider learns the negotiated value for the number of data transfer priorities implemented in the domain. For illustration, connect MCSPDUs on additional TCs are shown obeying the strict sequence 2, 3, 4 at the called MCS provider. In reality, transport connections may not be established in the same order that they are requested. Indeed, a **Connect-Additional** may arrive in a different order than it was sent, causing a **Connect-Result** to be returned out of sequence too. Or the latter, even if sent in sequence, may still be reordered in transit. A calling MCS provider need not wait for all additional TCs to be established before sending the first **Connect-Additional**. The called MCS provider need not wait for a full set of **Connect-Additional** MCSPDUs to arrive before returning the first **Connect-Result**. Receipt of a full set of successful results at the calling MCS provider, in whatever order, generates a successful MCS-CONNECT-PROVIDER confirm.

An unsuccessful **Connect-Response** or **Connect-Result** or a T-DISCONNECT indication at some intermediate point shall cause all TCs belonging to the MCS connection so far to be disconnected and shall generate an unsuccessful MCS-CONNECT-PROVIDER confirm.

An MCS-CONNECT-PROVIDER request specifies which of two MCS providers is higher than the other. This hierarchical relationship determines the subsequent routing of domain MCSPDUs, and the distinction between calling and called MCS provider is thereafter irrelevant. For example, a next step is for the MCS layer to merge the resources of two previously independent domains. **MCrq** and **MTrq** are generated by the lower MCS provider and are transmitted across the new MCS connection to the higher MCS provider. The direction in which these MCSPDUs are sent could be either calling-to-called or called-to-calling, depending on how the upward flag was set.

9.2 Domain MCSPDUs

Table 9-1 specifies the routing of domain MCSPDUs.

Table 9-1 – Routing of domain MCSPDUs

Category	MCSPDUs	TC	Direction
Request	**EDrq** **MCrq** **MTrq** **AUrq** **DUrq** **CJrq** **CLrq** **CCrq** **CDrq** **CArq** **CErq** **TGrq** **TIrq** **TVrq** **TPrq** **TRrq** **TTrq**	I	Up
	SDrq **USrq**	A	
Indication	**PDin** **PCin** **PTin** **DUin** **CDin** **CAin** **CEin** **TVin** **TPin**	I	Down
	SDin **USin**	A	
Response	**TVrs**	I	Up
Confirm	**MCcf** **MTcf** **AUcf** **CJcf** **CCcf** **TGcf** **TIcf** **TVcf** **TRcf** **TTcf**	I	Down
Ultimatum	**DPum** **RJum**	I	Up or Down

I: The MCSPDU traverses the initial TC.
A: The MCSPDU may traverse an additional TC, depending on its data transfer priority.

Up: The MCSPDU travels toward the top MCS provider.
Down: The MCSPDU travels away from the top MCS provider.

If an MCS provider generates or forwards an MCSPDU of category *request*, it travels over the unique MCS connection upward. **EDrq**, **CJrq**, and **CLrq** may be consumed at some intermediate MCS provider. Other requests rise to be acted on by the top MCS provider, unless the content of a request is determined to be invalid, in which case its MCSPDU may be ignored, without confirmation.

If an MCS provider generates or forwards an MCSPDU of category *indication*, copies of it, possibly amended in content, travel over zero or more MCS connections downward, according to the following rules:

a. **PDin** is sent on all MCS connections downward. The height limit it contains is decremented by one. An MCS provider receiving this MCSPDU with a height limit of zero shall disconnect.

b. **PCin** is sent on all MCS connections downward. The sets of channel ids forwarded are unchanged, except at a former top provider that is still merging into an upper domain, at which point they are restricted to channel ids whose acceptance into the upper domain has been confirmed.

c. **PTin** is sent on all MCS connections downward. The set of token ids forwarded is unchanged, except at a former top provider that is still merging into an upper domain, at which point they are restricted to token ids whose acceptance into the upper domain has been confirmed.

d. **DUin** is sent on all MCS connections downward. The set of user ids forwarded is unchanged, except at a former top provider that is still merging into an upper domain, at which point they are restricted to user ids whose acceptance into the upper domain has been confirmed, so that all detached users will be announced to those that remain.

e. **CDin** is sent on all MCS connections downward that contain in their subtree the manager of the private channel or any admitted user.

f. **CAin** and **CEin** are sent on all MCS connections downward that contain one or more of the affected users in their subtree. The set of user ids forwarded may be restricted to those residing in a subtree.

g. **TVin** is sent on a single MCS connection downward that contains the designated recipient in its subtree.

h. **TPin** is sent on all MCS connections downward that contain in their subtree a user who has grabbed, inhibited, or is being given the token.

i. **SDin** and **USin** are sent on all MCS connections downward by which the specified channel is joined, except that when **SDin** is generated, it is not sent back on the connection by which **SDrq** arrived.

Indications **PCin**, **PTin**, **DUin**, **CAin**, and **CEin** need not be forwarded if the sets of ids they contain are empty.

If an MCS provider generates or forwards an MCSPDU of category *response*, it travels over the unique MCS connection upward. It rises to be acted on by the top MCS provider, unless its content is determined to be invalid.

If an MCS provider generates or forwards an MCSPDU of category *confirm*, it travels over a single MCS connection downward, according to the following rules.

a. **MCcf** retraces, in the opposite direction, the path of the earliest **MCrq** that has not yet been answered by a confirm. This requires each MCS provider to maintain a first-in first-out queue of pending requests.

b. **MTcf** retraces, in the opposite direction, the path of the earliest **MTrq** that has not yet been answered by a confirm. This requires each MCS provider to maintain a first-in first-out queue of pending requests.

c. **AUcf** retraces, in the opposite direction, the path of an earlier **AUrq** that has not yet been answered by a confirm. It is not critical which **AUrq**, if more than one is pending, but to be fair each MCS provider should maintain a first-in first-out queue. Upon sending **AUcf**, an MCS provider shall record to which subtree the user id it contains is thereby being assigned.

d. Other MCSPDUs of category confirm contain an initiator user id that was previously assigned through the action of **AUcf** as just explained. These MCSPDUs are sent on the MCS connection downward that leads to the subtree where the user id was assigned. Continuing in this way, they eventually return to the provider that hosts the requesting MCS attachment.

Confirms are generated in the course of processing like requests. All but **CJcf** are generated by the top MCS provider.

If an MCS provider generates an MCSPDU of category *ultimatum*, it travels over a single MCS connection, either upward or downward. **DPum** commands the receiving MCS provider to disconnect the MCS connection that conveys it. **RJum** rejects an erroneous MCSPDU with a diagnostic code and invites the MCS provider that transmitted it to disconnect. Ultimatums are not forwarded.

10 Meaning of MCSPDUs

10.1 Connect-Initial

Connect-Initial is generated by an MCS-CONNECT-PROVIDER request. It is sent as the first TSDU over the initial TC of a new MCS connection. At the receiver it generates an MCS-CONNECT-PROVIDER indication.

Table 10-1 – `Connect-Initial` MCSPDU

Contents	Source	Sink
Calling Domain Selector	request	indication
Called Domain Selector	request	indication
Upward Flag	request	indication
Target Domain Parameters	request	indication
Minimum Domain Parameters	request	indication
Maximum Domain Parameters	request	indication
User Data	request	indication

Calling transport address and called transport address are additional parameters of the MCS-CONNECT-PROVIDER request and indication. They become parameters of T-CONNECT and are not passed explicitly in any MCSPDU. The same pair of transport addresses shall be used to request all TCs belonging to the same MCS connection.

Transport quality of service is an additional parameter of the MCS-CONNECT-PROVIDER request but not of the indication. Quality of service can vary from one TC to another, and the quality available is only disclosed in the process of establishing a given TC. Since the number of additional TCs needed is not known until MCS-CONNECT-PROVIDER has negotiated the domain parameter for number of data priorities implemented, this primitive cannot at the same time fully negotiate their transport quality of service. A called MCS provider shall therefore accept incoming TCs automatically at the offered quality of service, so long as this meets any minimum optionally specified by the calling MCS provider and indicated through each individual T-CONNECT.

The interpretation of domain selector values is a local matter for each MCS provider. These are octet strings that have the characteristics of an address. Acceptable values may be determined through the process of configuring an MCS provider. More than one value may select the same domain. An unspecified domain selector is an octet string of length zero. This may be resolved, through local convention, to some explicit value.

The upward flag specifies~~d~~ the direction of a new MCS connection: true if the called provider is to be considered higher than the calling provider and false otherwise. An MCS provider plays a role in the hierarchy of a domain based on the direction of MCS connections in which it participates. No provider shall allow~~accept~~ two ~~upward~~ connections to higher providers. A provider without a~~n upward~~ connection to a higher provider shall act as top MCS provider.

The target domain parameters of **`Connect-Initial`** individually shall lie between the minimum and maximum values specified. An MCS provider shall revise the requested domain parameters to reflect limits of its implementation or to impose values already agreed among the existing members of a domain. It may increase the minimums and decrease the maximums. It shall change targets only to keep them within the interval. An MCS provider should be prepared to honor any response that falls within the range of values it proposes.

User data is an arbitrary octet string. It may have length zero.

An MCS provider automatically accepts each incoming TC to the limit of its capacity. User data in T-CONNECT is unused. The first TSDU received in data transfer, being either a **`Connect-Initial`** or a **`Connect-Additional`** MCSPDU, determines the nature of the TC. If the content is unacceptable, a called MCS provider may disconnect the TC immediately. The preferred reaction is to return a **`Connect-Response`** or **`Connect-Result`**, as the case may be, explaining why the MCS connection failed. The calling MCS provider shall then disconnect.

10.2 `Connect-Response`

`Connect-Response` is generated by an MCS-CONNECT-PROVIDER response. It is the first TSDU sent in reverse over the initial TC of a new MCS connection. It conveys the acceptance of an MCS connection to the calling MCS provider, which then proceeds to establish any additional TCs required.

Table 10-2 – `Connect-Response` MCSPDU

Contents	Source	Sink
Result	response	confirm
Domain Parameters	response	confirm
Called Connect Id	called provider	calling provider
User Data	response	confirm

If the result is successful, this MCSPDU fixes the domain parameters in effect. Among these is the number of MCS data transfer priorities implemented, equal to the number of TCs in an MCS connection. If this exceeds one, additional TCs shall be created and bound to the MCS connection through the exchange of **`Connect-Additional`** and **`Connect-Result`** MCSPDUs.

The called connect id serves as the means for associating additional incoming TCs at the called MCS provider with this initial TC. Its value is chosen for this purpose alone. It shall uniquely identify one MCS connection in progress at the called provider. This id has no lasting significance following the completion of MCS-CONNECT-PROVIDER.

Most of the parameters of MCS-CONNECT-PROVIDER confirm are conveyed in **`Connect-Response`**. If the result is unsuccessful or no additional TCs are needed, confirm is generated immediately. Otherwise, it is deferred until the results are known for binding additional TCs to the MCS connection.

10.3 `Connect-Additional`

`Connect-Additional` is generated following receipt of **`Connect-Response`**. It is sent as the first TSDU over an additional TC of a new MCS connection.

Table 10-3 – `Connect-Additional` MCSPDU

Contents	Source	Sink
Called Connect Id	calling provider	called provider
Data Priority	calling provider	called provider

Data priority takes on the values *high*, *medium*, and *low* in sequence, up to the number of additional TCs required.

10.4 `Connect-Result`

`Connect-Result` is generated following receipt of **`Connect-Additional`**. It is the first TSDU sent in reverse over an additional TC of a new MCS connection.

Table 10-4 – `Connect-Result` MCSPDU

Contents	Source	Sink
Result	called provider	confirm

If any result is unsuccessful, MCS-CONNECT-PROVIDER confirm shall be generated immediately. All TCs associated with the MCS connection shall be disconnected and any MCSPDUs they convey shall be ignored.

Otherwise, successful results shall be awaited for each additional TC. These may return out of sequence. When all have been collected, a successful MCS-CONNECT-PROVIDER confirm shall be generated.

After MCS-CONNECT-PROVIDER confirm, domain MCSPDUs may flow across an MCS connection. Each MCS connection belongs to a single domain. In configurations where an MCS provider hosts more than one domain, the MCS connection that carries MCSPDUs determines which domain they apply to. The descriptions of domain MCSPDUs that occupy the remainder of this clause are set in the context of a single domain.

10.5 PDin

PDin is generated following the successful completion of MCS-CONNECT-PROVIDER. It plumbs the hierarchy of MCS providers below a new MCS connection to ensure that no cycle has been created. **PDin** is also generated by the top MCS provider to enforce the maximum height of a domain.

Table 10-5 – PDin MCSPDU

Contents	Source	Sink
Height Limit	former top or top	subordinates

PDin is generated at the lower end of a new MCS connection, by the provider that has ceased to be top of a domain. Its content is initialized to the domain parameter for the maximum height of the domain. **PDin** is transmitted over all MCS connections downward.

As needed, **PDin** is generated in the same way at the top MCS provider.

Wherever **PDin** is received, the height limit it contains is inspected. If greater than zero, the limit is decremented by one and **PDin** is forwarded over all MCS connections downward. A value of zero, on the other hand, means that the receiver lies too far from the top provider. It shall react by disconnecting the MCS connection upward. This deletes an entire subtree and helps to repair the height of the domain.

In the presence of a cycle, the height limit of **PDin** must decrease until it reaches zero. The provider that detects this will break the cycle, at the expense of deleting all providers in the cycle and their subordinates from the domain.

NOTE – An MCS provider, with purely local knowledge of MCS connections, cannot prevent the creation of cycles. It can ensure that there is at most one connection upward at all times, but it cannot ensure that the upward connection does not loop back to some provider below. When a cycle is created, the immediate cause is a faulty upward connection from the top MCS provider. Controller applications, which specify the MCS connections to be created, must strive to avoid such errors.

10.6 EDrq

EDrq is generated following the successful completion of MCS-CONNECT-PROVIDER. It communicates upward changes in the height of providers and their throughput enforcement intervals. **EDrq** is generated by an MCS provider whenever its height or interval changes.

Table 10-6 – EDrq MCSPDU

Contents	Source	Sink
Height in Domain	subordinate	higher provider
Throughput Enforcement Interval	subordinate	higher provider

The height of an MCS provider may change when an MCS connection is added or dropped or when a subordinate provider reports a change through **EDrq**. Its monitoring interval to enforce the minimum throughput specified as a domain parameter may change by adapting to the intervals reported by subordinates or for other reasons. If either value changes, an MCS provider shall transmit **EDrq** to its immediate superior.

10.7 MCrq

MCrq is generated following the successful completion of MCS-CONNECT-PROVIDER. It communicates upward the attributes of channels held by a former top provider so that they may be incorporated into the merged domain.

Table 10-7 – MCrq MCSPDU

Contents	Source	Sink
Merge Channels	former top	top provider
Purge Channel Ids	intermediates	top provider

MCrq may be filled with the attributes of multiple channels, up to the domain limit on MCSPDU size. As detailed in the ASN.1 definitions of clause 7, each of the four kinds of channel in use (static, user id, private, assigned) has its relevant set of attributes. These are held in the information base of the top MCS provider and are partially replicated into the subtrees where a channel is used. When two domains are merged, through the action of MCS-CONNECT-PROVIDER, channels that are in use in the lower domain must either be incorporated into the information base of the upper domain or be purged from the lower domain. This decision rests with the top provider of the merged domain.

Each channel shall be considered individually. If the domain limits on channels in use allow it and the channel id has not already been put to some conflicting use, the upper domain shall expand to include it. The use of a static channel id is never a conflict. The use of a private channel id in the lower domain is not a conflict if it is also used as a private channel id in the upper domain and has the same user id as manager. All other combinations of simultaneous use are disallowed, requiring the channel id to be purged from the lower domain.

If a private channel has a large set of admitted users, its attributes may not fit into a single **MCrq** and shall be sent upward in multiple MCSPDUs. However, the second and succeeding requests to merge the same private channel shall be delayed until an **MCcf** has been received in reply to the first. Only then is it known if domain limits have allowed the channel to be put into use in the upper domain. If the first request fails, it shall not be repeated with a remaining subset of admitted users.

Each **MCrq** elicits an **MCcf** reply from the top MCS provider, in the same sequence. An **MCcf** contains nothing that explicitly identifies the preceding **MCrq**. Replies shall be routed solely by the order in which these MCSPDUs are received. MCS providers above the former top shall make a record of each unanswered **MCrq** and whence it arrived, so that the corresponding **MCcf** may be returned via the same MCS connection.

Intermediate MCS providers shall validate the user ids proclaimed in private channel attributes, to ensure that they are legitimately assigned to the subtree where the **MCrq** originates. Invalid user ids shall be deleted. If the manager of a private channel is deleted, all of the channel attributes shall be deleted from the merge request and the channel id alone shall be included in the set to be purged. Except for this validation of user ids, intermediate MCS providers shall not modify the contents of an **MCrq**.

A former top provider shall await individual confirmation that all user ids and all token ids have been incorporated into a merged domain or purged before it begins to submit static, assigned, or private channel attributes for merger.

10.8 MCcf

MCcf replies to a preceding **MCrq**. It reflects the same set of channel ids and a subset of the attributes. Channel attributes not incorporated into the merged domain are reported as channel ids to be purged.

Table 10-8 – MCcf MCSPDU

Contents	Source	Sink
Merge Channels	top provider	intermediates
Purge Channel Ids	top provider	former top

Accepted channel ids are reflected with the attributes that were entered into the information base of the top MCS provider. Intermediate providers shall update their information base to conform.

Channels to be purged from the lower domain are listed by id only. If the same channel ids are used in the upper domain, they are left undisturbed. Intermediate providers shall forward purged channel ids without acting on them.

MCS providers shall route **MCcf** to the source of the antecedent **MCrq**, using the knowledge that there is a one-to-one reply. **MCcf** returns to the former top provider that generated **MCrq**. There the merged channels may be ignored, as they have remained in the information base pending a reply. Purged channel ids shall be deleted as they are for **PCin**.

10.9 PCin

PCin is generated at a former top provider following receipt of **MCcf**. It is broadcast downward and purges the use of specified channel ids from subordinate providers.

Table 10-9 – PCin MCSPDU

Contents	Source	Sink
Detach User~~Channel~~ Ids	former top	subordinates
Purge~~Disband~~ Channel Ids	former top	subordinates

Depending on the current use of a channel id, the effect of purging it is: MCS-DETACH-USER indication to all users ~~o~~if a user id channel; MCS-CHANNEL-LEAVE indication to the joined users ~~o~~if a static or assigned channel id; MCS-CHANNEL-DISBAND indication to the manager and MCS-CHANNEL-EXPEL indication to the admitted users ~~o~~if a private channel id.

A former top provider, knowing the use of all channels in its lower domain, can generate the proper indications from channel ids alone. Its subordinates, however, may have only partial knowledge. They must be told which channel ids represent detached users, for which an indication is always generated, and which represent other kinds of channels, for which an indication is generated only if the channel is in use at the subordinate provider. Therefore, ~~the~~ **PCin** divides~~that they receive breaks down~~ the channel ids to be purged into these two categories.~~according to the indication that is appropriate.~~

The purging of a user id through **PCin** shall have the same consequences as deletion through **DUin**, except that, since the receiver of **PCin** is no longer top provider, it need not generate **CDin** or **TVcf** as a side effect.

NOTE – A provider may receive in PCin static and assigned channel ids to which it is not joined or private channel ids for which its attachments are neither managers nor admitted users. Records of use maintained in the information base allow a provider to suppress primitive indications for such channel ids.

10.10 MTrq

MTrq is generated following the successful completion of MCS-CONNECT-PROVIDER. It communicates upward the attributes of tokens held by a former top provider so that they may be incorporated into the merged domain.

Table 10-10 – `MTrq` MCSPDU

Contents	Source	Sink
Merge Tokens	former top	top provider
Purge Token Ids	intermediates	top provider

`MTrq` may be filled with the attributes of multiple tokens, up to the domain limit on MCSPDU size. As detailed in the ASN.1 definitions of clause 7, each state of a token in use (grabbed, inhibited, giving, ungivable, given) has its relevant set of attributes. These are held in the information base of the top MCS provider and are partially replicated into the subtrees where a token is used. When two domains are merged, through the action of MCS-CONNECT-PROVIDER, tokens that are in use in the lower domain must either be incorporated into the information base of the upper domain or be purged from the lower domain. This decision rests with the top provider of the merged domain.

Each token shall be considered individually. If the domain limits on tokens in use allow it and the token id has not already been put to some conflicting use, the upper domain shall expand to include it. The inhibiting of a token id in the lower domain is not a conflict if it is also inhibited in the upper domain. All other combinations of simultaneous use are disallowed, requiring the token id to be purged from the lower domain.

If a token has a large set of inhibiting users, its attributes may not fit into a single **`MTrq`** and shall be sent upward in multiple MCSPDUs. However, the second and succeeding requests for the same inhibited token shall be delayed until an **`MTcf`** has been received in reply to the first. Only then is it known if domain limits have allowed the token to be put into use in the upper domain. If the first request fails, it shall not be repeated with a remaining subset of inhibitors.

Each **`MTrq`** elicits an **`MTcf`** reply from the top MCS provider, in the same sequence. An **`MTcf`** contains nothing that explicitly identifies the preceding **`MTrq`**. Replies shall be routed solely by the order in which these MCSPDUs are received. MCS providers above the former top shall make a record of each unanswered **`MTrq`** and whence it arrived, so that the corresponding **`MTcf`** may be returned via the same MCS connection.

Intermediate MCS providers shall validate the user ids proclaimed in token attributes, to ensure that they are legitimately assigned to the subtree where the **`MTrq`** originates. Invalid user ids shall be deleted. A token being given shall remain grabbed if either the grabber or the recipient, but not both, is deleted. ~~An inhibited token shall remain inhibited unless all inhibitors are deleted.~~ If a token becomes released through this deletion of user ids, all of its attributes shall be deleted from the merge request and the token id alone shall be included in the set to be purged. An inhibited token shall remain inhibited in **`MTrq`** even if all inhibitors are deleted, leaving an empty set in the attributes, because inhibitors may survive from other MCSPDUs. Except for this validation of user ids, intermediate MCS providers shall not modify the contents of an **`MTrq`**.

A former top provider shall await individual confirmation that all user ids have been incorporated into a merged domain or purged before it begins to submit token attributes for merger.

10.11 `MTcf`

`MTcf` replies to a preceding **`MTrq`**. It reflects the same set of token ids and a subset of the attributes. Token attributes not incorporated into the merged domain are reported as token ids to be purged.

Table 10-11 – `MTcf` MCSPDU

Contents	Source	Sink
Merge Tokens	top provider	intermediates
Purge Token Ids	top provider	former top

Accepted token ids are reflected with the attributes that have been entered into the information base at the top MCS provider. Intermediate providers shall update their information base to conform.

Tokens to be purged from the lower domain are listed by id only. If the same token ids are used in the upper domain, they are left undisturbed. Intermediate providers shall forward purged token ids without acting on them.

MCS providers shall route **MTcf** to the source of the antecedent **MTrq**, using the knowledge that there is a one-to-one reply. **MTcf** returns to the former top provider that generated **MTrq**. There the merged tokens may be ignored, as they have remained in the information base pending a reply. Purged token ids shall be deleted as they are for **PTin**.

10.12 PTin

PTin is generated at a former top provider following receipt of **MTcf**. It is broadcast downward and purges the use of specified token ids from subordinate providers.

Table 10-12 – PTin MCSPDU

Contents	Source	Sink
Purge Token Ids	former top	subordinates

The effect of purging a token is severe: MCS-DETACH-USER indication to any user who has grabbed, inhibited, or is being given one of the token ids. A provider shall implement this by generating **DUrq** on behalf of the affected users with reason *token purged*.

NOTE – It is anticipated that T.122 may be revised in the future to allow MCS-TOKEN-RELEASE indication in this situation. This would allow the affected user to remain attached even though its right to use the token is withdrawn.

10.13 DPum

DPum is generated by an MCS-DISCONNECT-PROVIDER request. In turn, it generates an MCS-DISCONNECT-PROVIDER indication at the other end of an MCS connection. **DPum** compels the receiver to disconnect the MCS connection that conveyed it.

Table 10-13 – DPum MCSPDU

Contents	Source	Sink
Reason	requesting provider	indication

DPum may also be generated by an MCS provider when it detects an error condition like the existence of a cycle in the domain hierarchy. In such cases, the reason is other than *user-requested*.

10.14 RJum

RJum is generated when an MCS provider receives an invalid MCSPDU or detects an MCS protocol error. It invites the peer provider at the other end of an MCS connection to disconnect, since recovery is uncertain from a situation that should not occur.

Table 10-14 – RJum MCSPDU

Contents	Source	Sink
Diagnostic	rejecting provider	rejected provider
Initial Octets	rejecting provider	rejected provider

RJum diagnoses the error and returns an initial portion of the offending TSDU, typically as many octets as will fit in the maximum size MCSPDU. The receiving provider has the option to disconnect or to persevere.

10.15 AUrq

AUrq is generated by an MCS-ATTACH-USER request. It rises to the top MCS provider, which returns an **AUcf** reply. If the domain limit on number of user ids allows, a new user id is generated.

Table 10-15 – AUrq MCSPDU

Contents	Source	Sink
(none)	–	–

AUrq contains no information other than its MCSPDU type. The domain to which the user attaches is determined by the MCS connection conveying the MCSPDU. The only initial characteristic of the user id generated is its uniqueness.

An MCS provider shall make a record of each unanswered **AUrq** received and by which MCS connection it arrived, so that a replying **AUcf** can be routed back to the same source. To distribute replies fairly, each provider should maintain a first-in, first-out queue for this purpose.

10.16 AUcf

AUcf is generated at the top MCS provider upon receipt of **AUrq**. Routed back to the requesting provider, it generates an MCS-ATTACH-USER confirm.

Table 10-16 – AUcf MCSPDU

Contents	Source	Sink
Result	top provider	confirm
Initiator (optional)	top provider	confirm

AUcf contain a user id if and only if the result is successful. Providers that receive a successful **Aucf** shall enter the user id into their information base.

MCS providers shall route **AUcf** to the source of an antecedent **AUrq**, using the knowledge that there is a one-to-one reply. A provider that transmits **AUcf** shall note to which downward MCS connection the new user id is thereby assigned, so that it may validate the user id when it arises later in other requests.

10.17 DUrq

DUrq is generated by an MCS-DETACH-USER request. If valid, it rises to the top MCS provider, which deletes the user from its information base and broadcasts **DUin** to advise other providers of the change.

Table 10-17 – DUrq MCSPDU

Contents	Source	Sink
Reason	requesting provider	top provider
User Ids	requesting provider	top provider

An MCS-DETACH-USER request generates a **DUrq** containing reason *user-requested* and a single user id.

DUrq shall also be generated by an MCS provider when a downward MCS connection is disconnected. At that point, all users in the affected subtree are lost and shall be reported as detached with reason *domain disconnected*. If user id assignments are pending, a later reply, either **MCcf** or **AUcf**, may then be left with no route back to its source in the disconnected subtree. A provider confronted with this shall also generate **DUrq** to delete the unassignable user ids.

Providers that receive **DUrq** shall validate the user ids it contains to ensure that they are legitimately assigned to the subtree of origin. Invalid user ids shall be deleted. If no user ids remain, a **DUrq** shall be ignored.

The user ids contained in **DUrq** shall not be deleted from the information base until a provider receives **DUin**. This maintains consistency with the top MCS provider.

NOTE – If more than one data priority is implemented in an MCS domain, **DUin** may arrive at a given provider before data sent earlier but at a low priority by the same user. This protocol does not prevent data being delivered to an attachment even after it was reported through **DUin** that the sender had detached.

10.18 DUin

DUin is generated at the top MCS provider upon receipt of **DUrq**. It is broadcast downward to all other providers and generates MCS-DETACH-USER indications at all attachments.

Table 10-18 – DUin MCSPDU

Contents	Source	Sink
Reason	top provider	indication
User Ids	top provider	indication

At a surviving attachment, **DUin** generates one MCS-DETACH-USER indication for each user id it contains. It does not matter whether the notified user was previously aware of the existence of a detached user.

Upon receipt of a **DUin** containing its own user id, an MCS attachment ceases to exist. Any channels than become unjoined as a result of a user detaching, except for the user id channel itself, shall be left via **CLrq**.

Providers that receive **DUin** shall delete the specified user ids from their information base. Private channels managed by a detached user shall be deleted if there are no other admitted users. If other users remain, deletion of the manager shall cause the top provider to multicast a **CDin** towards them. Any tokens grabbed, being given to, or inhibited by a detached user shall have their state adjusted accordingly. The deletion of an intended token recipient shall cause the top provider to generate an unsuccessful **TVcf** towards the donor, unless it has released the token or itself detached.

10.19 CJrq

CJrq is generated by an MCS-CHANNEL-JOIN request. If valid, it rises until it reaches an MCS provider with enough information to generate a **CJcf** reply. This may be the top MCS provider.

Table 10-19 – CJrq MCSPDU

Contents	Source	Sink
Initiator	requesting provider	higher provider
Channel Id	request	higher provider

The user id of the initiating MCS attachment is supplied by the MCS provider that receives the primitive request. Providers that receive **CJrq** subsequently shall validate the user id to ensure that it is legitimately assigned to the subtree of origin. If the user id is invalid, the MCSPDU shall be ignored.

NOTE – This allows for the possibility that **CJrq** may be racing upward against a purge of the initiating user id flowing down. A provider that receives **PCin** first might receive a **CJrq** soon thereafter that contains an invalid user id. This is a normal occurrence and is not cause for rejecting the MCSPDU.

CJrq may rise to an MCS provider that has the requested channel id in its information base. Any such provider, being consistent with the top MCS provider, will agree whether the request should succeed. If the request should fail, the provider shall generate an unsuccessful **CJcf**. If it should succeed and the provider is already joined to the same channel, the provider shall generate a successful **CJcf**. In these two cases, MCS-CHANNEL-JOIN completes without necessarily visiting the top MCS provider. Otherwise, if the request should succeed but the channel is not yet joined, a provider shall forward **CJrq** upward.

If **CJrq** rises to the top MCS provider, the channel id requested may be zero, which is in no information base because it is an invalid id. If the domain limit on the number of channels in use allows, a new assigned channel id shall be generated and returned in a successful **CJcf**. If the channel id requested is in the static range and the domain limit on the number of channels in use allows, the channel id shall be entered into the information base and shall likewise be returned in a successful **CJcf**.

Otherwise, the request will succeed only if the channel id is already in the information base of the top MCS provider. A user id channel can only be joined by the same user. A private channel id can be joined only by users previously admitted by its manager. An assigned channel id can be joined by any user.

10.20 CJcf

CJcf is generated at a higher MCS provider upon receipt of **CJrq**. Routed back to the requesting provider, it generates an MCS-CHANNEL-JOIN confirm.

Table 10-20 – CJcf MCSPDU

Contents	Source	Sink
Result	higher provider	confirm
Initiator	higher provider	MCSPDU routing
Requested	higher provider	confirm
Channel Id (optional)	higher provider	confirm

CJcf contains a joined channel id if and only if the result is successful.

The channel id requesturned is the same as in **CJrq**. This helps the initiating attachment relate MCS-CHANNEL-JOIN confirm to an antecedent request~~, unless the request was for zero and an assigned channel id was successfully generated~~. Since **CJrq** need not rise to the top provider, confirms may occur out of order.

If the result is successful, **CJcf** joins the receiving MCS provider to the specified channel. Thereafter, higher providers shall route to it any data that users send over the channel. A provider shall remain joined to a channel as long as any of its attachments or subordinate providers does. To leave the channel, a provider shall generate **CLrq**.

Providers that receive a successful **CJcf** shall enter the channel id into their information base. If not already there, the channel id shall be given type static or assigned, depending on its range.

CJcf shall be forwarded in the direction of the initiating user id. If the user id is unreachable because an MCS connection no longer exists, the provider shall decide whether it has reason to remain joined to the channel. If not, it shall generate **CLrq**.

10.21 CLrq

CLrq is generated by an MCS provider to remove itself from a set of channels. The motivation may be an MCS-CHANNEL-LEAVE request from the last attachment joined to a channel. **CLrq** continues to rise if higher providers, as a consequence, also lose their reason for being joined.

Table 10-21 – CLrq MCSPDU

Contents	Source	Sink
Channel Ids	requesting provider	higher provider

Providers that receive **CLrq** shall stop routing to the MCS connection that conveyed it any data that users send over the specified channels. When the last attachment or subordinate provider leaves a channel, an MCS provider shall generate a corresponding **CLrq**.

10.22 CCrq

CCrq is generated by an MCS-CHANNEL-CONVENE request. If valid, it rises to the top MCS provider, which returns a **CCcf** reply. If the domain limit on number of channel ids allows, a new private channel id is generated.

Table 10-22 – CCrq MCSPDU

Contents	Source	Sink
Initiator	requesting provider	top provider

CCrq contains the initiating user id, which shall be validated as explained for **CJrq**.

The requester becomes manager of the private channel. Initially the channel is unjoined and its manager is the only admitted user.

10.23 CCcf

CCcf is generated at the top MCS provider upon receipt of **CCrq**. Routed back to the requesting provider, it generates an MCS-CHANNEL-CONVENE confirm.

Table 10-23 – CCcf MCSPDU

Contents	Source	Sink
Result	top provider	confirm
Initiator	top provider	MCSPDU routing
Channel Id (optional)	top provider	confirm

CCcf contains a private channel id if and only if the result is successful.

Providers that receive a successful **CCcf** shall enter the channel id into their information base as a private channel with the initiating user id as its manager.

CCcf shall be forwarded in the direction of the initiating user id. If the user id is unreachable because an MCS connection no longer exists, no special actions need be taken, as a **DUin** must arrive later to report that the initiator has detached. Since the initiator is its manager, this will delete the channel id from the information base.

10.24 CDrq

CDrq is generated by an MCS-CHANNEL-DISBAND request. If valid, it rises to the top MCS provider, which deletes the private channel id and generates **CDin**.

Table 10-24 – CDrq MCSPDU

Contents	Source	Sink
Initiator	requesting provider	top provider
Channel Id	request	top provider

CDrq contains the initiating user id, which shall be validated to ensure that it is legitimately assigned to the subtree of origin. If the initiator does not equal the manager of the private channel, as recorded in the information base, the MCSPDU shall be ignored.

10.25 CDin

CDin is generated at the top MCS provider upon receipt of **CDrq**. It is multicast downward to providers that contain the manager or an admitted user in their subtree. It generates MCS-CHANNEL-EXPEL indications with reason *channel disbanded*.

Table 10-25 – CDin MCSPDU

Contents	Source	Sink
Channel Id	top provider	indication

CDin shall also be generated by the top MCS provider when the manager of a private channel is detached.

Providers that receive **CDin** shall delete the channel from their information base.

10.26 CArq

CArq is generated by an MCS-CHANNEL-ADMIT request. If valid, it rises to the top MCS provider, which admits the specified users to the private channel and multicasts **CAin** to advise providers in whose subtree they reside.

Table 10-26 – CArq MCSPDU

Contents	Source	Sink
Initiator	requesting provider	top provider
Channel Id	request	top provider
User Ids	request	top provider

CArq contains the initiating user id, which shall be validated as explained for **CDrq**.

The other user ids of **CArq**, representing users to be admitted, shall be validated at the top MCS provider, which alone knows the entire user population. Those that are invalid shall be omitted from the resulting **CAin**.

The user ids contained in **CArq** shall not be admitted to the private channel until a provider receives **CAin**. This maintains consistency with the top MCS provider.

10.27 CAin

CAin is generated at the top MCS provider upon receipt of **CArq**. It is multicast downward to providers that contain a newly admitted user in their subtree. It generates MCS-CHANNEL-ADMIT indications at the affected attachments.

Table 10-27 – CAin MCSPDU

Contents	Source	Sink
Initiator	top provider	indication
Channel Id	top provider	indication
User Ids	top provider	MCSPDU routing

Providers that receive **CAin** shall ordinarily update the channel in their information base, admitting the specified users that reside in their subtree. However, if a provider is the former top of a lower domain that is being merged as a result of MCS-CONNECT-PROVIDER, it may refuse the admission by generating **DUrq** for the affected user ids with reason *channel purged.*

10.28 CErq

CErq is generated by an MCS-CHANNEL-EXPEL request. If valid, it rises to the top MCS provider, which expels the specified users from the private channel and multicasts **CEin** to advise providers in whose subtree they reside.

Table 10-28 – CErq MCSPDU

Contents	Source	Sink
Initiator	requesting provider	top provider
Channel Id	request	top provider
User Ids	request	top provider

CErq contains the initiating user id, which shall be validated as explained for **CDrq**.

The other user ids of **CErq**, representing users to be expelled, shall be validated at the top MCS provider, which alone knows the entire set of admitted users. Those that were not admitted shall be omitted from the resulting **CEin**.

The user ids contained in **CErq** shall not be expelled from the private channel until a provider receives **CEin**. This maintains consistency with the top MCS provider.

10.29 CEin

CEin is generated at the top MCS provider upon receipt of **CErq**. It is multicast downward to providers that contain an expelled user in their subtree. It generates MCS-CHANNEL-EXPEL indications at the affected attachments with reason *user~~manager~~-requested.*

Table 10-29 – `CEin` MCSPDU

Contents	Source	Sink
Channel Id	top provider	indication
User Ids	top provider	MCSPDU routing

Providers that receive **`CEin`** shall update the channel in their information base, deleting the specified users from the set of users admitted to the channel. If the set of users admitted to a private channel becomes empty and the manager does not reside in the subtree, the channel id shall be deleted from the information base. Otherwise, if the channel becomes unjoined as a result of expulsions, a provider shall generate a corresponding **`CLrq`**.

A provider forwarding **`CEin`** shall compute, for each destination subtree, whether it afterwards contains any attachments admitted to the private channel. If none, a provider shall conclude that the corresponding subordinate provider is no longer joined to the private channel and shall update its information base to this effect immediately, without waiting for **`CLrq`**.

10.30 `SDrq`

`SDrq` is generated by an MCS-SEND-DATA request. If valid, it rises toward the top MCS provider. Along the way, providers may generate from it an **`SDin`** with identical contents and multicast this downward.

Table 10-30 – `SDrq` MCSPDU

Contents	Source	Sink
Initiator	requesting provider	higher provider
Channel Id	request	higher provider
Data Priority	request	higher provider
Segmentation	requesting provider	higher provider
User Data	request	higher provider

`SDrq` contains the initiating user id, which shall be validated as explained for **`CJrq`**.

If the channel id is listed in the information base of the receiving MCS provider as a private channel and the initiator of **`SDrq`** is not an admitted user, the MCSPDU shall be ignored.

The initial or additional TC that conveys **`SDrq`** shall match its data priority, taking into account the number of priorities implemented in the domain. MCSPDUs arriving over an MCS connection by the wrong TC shall be rejected.

The segmentation flags *begin* and *end* shall be set by a provider to show the relationship of user data in the **`SDrq`** to the boundaries of an MCS service data unit. Providers have freedom to fragment and reassemble MCSPDUs that are part of the same MCS service data unit, so long as this does not disturb the integrity of user data. However, there should be little advantage in such manipulation, as the maximum size of an MCSPDU is constant throughout a domain.

A provider shall generate from **`SDrq`** an **`SDin`** with the same content and shall transmit it to all providers that are joined to the specified channel, excepting the subordinate provider that transmitted **`SDrq`** upward. Unless the channel is listed in the provider's information base as a user id residing in its subtree, it shall also forward **`SDrq`** upward.

10.31 SDin

SDin is generated at a higher MCS provider upon receipt of **SDrq**. It is multicast downward and generates MCS-SEND-DATA indications at all attachments that are joined to the channel.

Table 10-31 – SDin MCSPDU

Contents	Source	Sink
Initiator	higher provider	indication
Channel Id	higher provider	indication
Data Priority	higher provider	indication
Segmentation	higher provider	indicating provider
User Data	higher provider	indication

The initial or additional TC that conveys **SDin** shall match its data priority, taking into account the number of priorities implemented in a domain. MCSPDUs arriving over an MCS connection by the wrong TC shall be rejected.

The segmentation flags *begin* and *end* permit user data to be reassembled into a complete MCS service data unit. These flags shall be interpreted in the context of **SDin** MCSPDUs arriving from the same user over the same channel and at the same priority. A stream of fragments to be reassembled may be interleaved with other MCSPDUs and data from other users over other channels at other priorities.

It is a matter of local implementation how service data units are indicated to attached MCS users. One possibility is to deliver each MCSPDU as a separate interface data unit, with segmentation flags included. Alternative approaches that may seek to reassemble within the receiving provider should make some provision for large service data units and should reflect to the user the relative order in which service data units begin to arrive.

Providers that receive **SDin** shall forward it to all subordinates that are joined to the channel.

10.32 USrq

USrq is generated by an MCS-UNIFORM-SEND-DATA request. If valid, it rises to the top MCS provider, which generates from it an **USDin** with identical contents and multicasts this downward.

Table 10-32 – USrq MCSPDU

Contents	Source	Sink
Initiator	requesting provider	top provider
Channel Id	request	top provider
Data Priority	request	top provider
Segmentation	requesting provider	top provider
User Data	request	top provider

USrq contains the initiating user id, which shall be validated as explained for **CJrq**.

If the channel id is listed in the information base of the receiving MCS provider as a private channel and the initiator of **USrq** is not an admitted user, the MCSPDU shall be ignored.

The initial or additional TC that conveys **USrq** shall match its data priority, taking into account the number of priorities implemented in the domain. MCSPDUs arriving over an MCS connection by the wrong TC shall be rejected.

The segmentation flags *begin* and *end* shall be set by a provider to show the relationship of user data in the **USrq** to the boundaries of an MCS service data unit. Providers have freedom to fragment and reassemble MCSPDUs that are part of the same MCS service data unit, so long as this does not disturb the integrity of user data. However, there should be little advantage in such manipulation, as the maximum size of an MCSPDU is constant throughout a domain.

The top MCS provider shall generate from **USrq** a **USin** with the same content.

10.33 USin

USin is generated at the top MCS provider upon receipt of **USrq**. It is multicast downward and generates MCS-UNIFORM-SEND-DATA indications at all attachments that are joined to the channel.

Table 10-33 – USin MCSPDU

Contents	Source	Sink
Initiator	top provider	indication
Channel Id	top provider	indication
Data Priority	top provider	indication
Segmentation	top provider	indicating provider
User Data	top provider	indication

The initial or additional TC that conveys **USin** shall match its data priority, taking into account the number of priorities implemented in the domain. MCSPDUs arriving over an MCS connection by the wrong TC shall be rejected.

The segmentation flags *begin* and *end* permit user data to be reassembled into a complete MCS service data unit. These flags shall be interpreted in the context of **USin** MCSPDUs arriving from the same user over the same channel and at the same priority. A stream of fragments to be reassembled may be interleaved with other MCSPDUs and data from other users over other channels at other priorities.

It is a matter of local implementation how service data units are indicated to attached MCS users.

Providers that receive **USin** shall forward it to all subordinates that are joined to the channel.

10.34 TGrq

TGrq is generated by an MCS-TOKEN-GRAB request. If valid, it rises to the top MCS provider, which returns a **TGcf** reply.

Table 10-34 – TGrq MCSPDU

Contents	Source	Sink
Initiator	requesting provider	top provider
Token Id	request	top provider

TGrq contains the initiating user id, which shall be validated as explained for **CJrq**.

If the token is free and the domain limit on the number of tokens in use allows, it shall become grabbed. If the token is inhibited by the requesting user only, it shall become grabbed. Otherwise, the state of the token shall not change.

10.35 TGcf

TGcf is generated at the top MCS provider upon receipt of **TGrq**. Routed back to the requesting provider, it generates an MCS-TOKEN-GRAB confirm.

Table 10-35 – TGcf MCSPDU

Contents	Source	Sink
Result	top provider	confirm
Initiator	top provider	MCSPDU routing
Token Id	top provider	confirm
Token Status	top provider	confirm

The result shall be *successful* if the token was previously free or if the token was converted from inhibited to grabbed by the same user. Other results are *too many tokens* and *token not available*. The latter applies to a token already grabbed by the requester; this may be discerned by examining the token status.

Providers that receive **TGcf** shall update the token state in their information base to agree with the status returned.

TGcf shall be forwarded in the direction of the initiating user id. If the user id is unreachable because an MCS connection no longer exists, no special actions need be taken, as a **DUin** must arrive later to report that the initiator has detached. This will release its hold on the token id in the information base.

10.36 TIrq

TIrq is generated by an MCS-TOKEN-INHIBIT request. If valid, it rises to the top MCS provider, which returns a **TIcf** reply.

Table 10-36 – TIrq MCSPDU

Contents	Source	Sink
Initiator	requesting provider	top provider
Token Id	request	top provider

TIrq contains the initiating user id, which shall be validated as explained for **CJrq**.

If the token is free and the domain limit on the number of tokens in use allows, it shall become inhibited. If the token is grabbed by the requesting user, it shall be become inhibited. If the token is already inhibited, the requester shall be added to the set of inhibitors. Otherwise, the state of the token shall not change.

10.37 TIcf

TIcf is generated at the top MCS provider upon receipt of **TIrq**. Routed back to the requesting provider, it generates an MCS-TOKEN-INHIBIT confirm.

Table 10-37 – `TIcf` MCSPDU

Contents	Source	Sink
Result	top provider	confirm
Initiator	top provider	MCSPDU routing
Token Id	top provider	confirm
Token Status	top provider	confirm

The result shall be *successful* if the token was previously free or inhibited or if the token was converted from grabbed to inhibited by the same user. Other results are *too many tokens* and *token not available*.

Providers that receive **`TIcf`** shall update the token state in their information base to agree with the status returned.

This MCSPDU is routed in the same way as **`TGcf`**.

10.38 `TVrq`

`TVrq` is generated by an MCS-TOKEN-GIVE request. If valid, it rises to the top MCS provider, which generates either **`TVin`** or an unsuccessful **`TVcf`**.

Table 10-38 – `TVrq` MCSPDU

Contents	Source	Sink
Initiator	requesting provider	top provider
Token Id	request	top provider
Recipient	request	top provider

`TVrq` contains the initiating user id, which shall be validated as explained for **`CJrq`**.

If the token is grabbed by the requester and the intended recipient exists, **`TVin`** shall be transmitted toward the recipient. Otherwise, the request shall fail, the state of the token shall be unchanged, and **`TVcf`** shall be transmitted toward the requester with the result *token not possessed* or *no such user*.

10.39 `TVin`

`TVin` is generated at the top MCS provider upon receipt of **`TVrq`**. Routed to the intended recipient, it generates an MCS-TOKEN-GIVE indication.

Table 10-39 – `TVin` MCSPDU

Contents	Source	Sink
Initiator	top provider	indication
Token Id	top provider	indication
Recipient	top provider	MCSPDU routing

Providers that receive **`TVin`** shall ordinarily update the token id in their information base to the state of being given from initiator to recipient. However, if a provider is the former top of a lower domain that is being merged as a result of MCS-CONNECT-PROVIDER, it may refuse~~ject~~ the offered token by generating **`TVrs`** with reason *domain merging*.

`TVin` shall be forwarded in the direction of the recipient user id. If the user id is unreachable because an MCS connection no longer exists, no special actions need be taken, as a **`DUin`** must arrive later to report that the recipient has detached. This will release its hold on the token id in the information base.

10.40 `TVrs`

`TVrs` is generated by an MCS-TOKEN-GIVE response. If valid, it rises to the top MCS provider, which generates **`TVcf`** to inform the token's donor of the outcome.

Table 10-40 – `TVrs` MCSPDU

Contents	Source	Sink
Result	response	top provider
Recipient	responding provider	top provider
Token Id	responding provider	top provider

A *successful* result shall signify the recipient's acceptance of the offered token.

The user id of the responding MCS attachment is supplied by the MCS provider that receives the primitive response. Providers that receive **`TVrs`** subsequently shall validate the user id to ensure that it is legitimately assigned to the subtree of origin. If the user id is invalid, the MCSPDU shall be ignored.

If the token id is not listed in the provider's information base as being given to the recipient, the MCSPDU shall be ignored. If the token id is still grabbed by the donor, its state shall be updated to grabbed by the recipient if the result is successful; otherwise it shall revert to grabbed by the donor or shall be deleted from the information base, depending on whether the donor resides in the subtree of the provider. If the token id has since been released by the donor and the result is not successful, the token shall be deleted from the provider's information base.

If the MCSPDU is not invalid and ignored, it shall be forwarded upward. The top MCS provider shall act on **`TVrs`** as specified above. In addition, if the donor has not already released the token, the top provider shall generate **`TVcf`** containing the same result as **`TVrs`**.

10.41 `TVcf`

`TVcf` is generated at the top MCS provider upon receipt of **`TVrs`**. Routed back to the requesting provider, it generates a MCS-TOKEN-GIVE confirm.

Table 10-41 – `TVcf` MCSPDU

Contents	Source	Sink
Result	top provider	confirm
Initiator	top provider	MCSPDU routing
Token Id	top provider	confirm
Token Status	top provider	confirm

`TVcf` shall also be generated by the top MCS provider upon receipt of **`TVrq`** if a token cannot be offered to the intended recipient. This takes the place of generating **`TVin`**. **`TVcf`** shall also be generated with result *no such user* if the recipient is detached before **`TVrs`** is received.

Providers that receive **`TVcf`** shall update the token state in their information base to agree with the status returned.

This MCSPDU is routed in the same way as **`TGcf`**.

10.42 `TPrq`

`TPrq` is generated by an MCS-TOKEN-PLEASE request. If valid, it rises to the top MCS provider, which multicasts **`TPin`** to alert current users of the token.

Table 10-42 – `TPrq` MCSPDU

Contents	Source	Sink
Initiator	requesting provider	top provider
Token Id	request	top provider

`TPrq` contains the initiating user id, which shall be validated as explained for **`CJrq`**.

10.43 `TPin`

`TPin` is generated at the top MCS provider upon receipt of **`TPrq`**. It is multicast downward and generates MCS-TOKEN-PLEASE indications.

Table 10-43 – `TPin` MCSPDU

Contents	Source	Sink
Initiator	top provider	indication
Token Id	top provider	indication

Providers that receive **`TPin`** shall forward it to all subordinates that contain in their subtree a user who has grabbed, inhibited, or is being given the specified token.

10.44 TRrq

TRrq is generated by an MCS-TOKEN-RELEASE request. If valid, it rises to the top MCS provider, which returns a **TRcf** reply.

Table 10-44 – TRrq MCSPDU

Contents	Source	Sink
Initiator	requesting provider	top provider
Token Id	request	top provider

TRrq contains the initiating user id, which shall be validated as explained for **CJrq**.

If the token is grabbed by the requester, it shall become free. If it is inhibited, the requester shall be removed from the set of inhibitors; if this set becomes empty, the token shall become free. If the token is in the process of being given away by the requester, it shall enter a distinct intermediate state of given to the intended recipient, pending receipt of **TVrs**. Otherwise the state of the token shall not change.

10.45 TRcf

TRcf is generated at the top MCS provider upon receipt of **TRrq**. Routed back to the requesting provider, it generates an MCS-TOKEN-RELEASE confirm.

Table 10-45 – TRcf MCSPDU

Contents	Source	Sink
Result	top provider	confirm
Initiator	top provider	MCSPDU routing
Token Id	top provider	confirm
Token Status	top provider	confirm

The result shall be *successful* if the token was grabbed or inhibited by the requester or if the requester was in the process of giving it away. The other possible result is *token not possessed*.

Providers that receive **TRcf** shall update the token state in their information base to agree with the status returned.

This MCSPDU is routed in the same way as **TGcf**.

10.46 TTrq

TTrq is generated by an MCS-TOKEN-TEST request. If valid, it rises to the top MCS provider, which returns a **TTcf** reply.

Table 10-46 – `TTrq` MCSPDU

Contents	Source	Sink
Initiator	requesting provider	top provider
Token Id	request	top provider

`TTrq` contains the initiating user id, which shall be validated as explained for **`CJrq`**.

10.47 `TTcf`

`TTcf` is generated at the top MCS provider upon receipt of **`TTrq`**. Routed back to the requesting provider, it generates an MCS-TOKEN-TEST confirm.

Table 10-47 – `TTcf` MCSPDU

Contents	Source	Sink
Initiator	top provider	MCSPDU routing
Token Id	top provider	confirm
Token Status	top provider	confirm

Providers that receive **`TTcf`** should find that the token state in their information base agrees with the status returned.

This MCSPDU is routed in the same way as **`TGcf`**.

11 MCS provider information base

11.1 Hierarchical replication

Although an MCS provider may host multiple domains, it serves each one independently. It maintains logically separate information bases for each to record the state of channel and token resources in use. The description that follows is set in the context of a single domain.

The MCS resources that need to be managed in a domain are channel ids and token ids. User ids are a subset of channel ids. Domain parameters limit how many ids of each category can be in use simultaneously. This allows a provider to compute how much memory is needed for the information base in the worst case of a fully utilized domain.

In the hierarchy of a domain, the ids in use at any given MCS provider stabilize to a subset of those in use at its immediate superior. Information about an id is recorded where it can be used to support MCS services involving that id. Recording information more widely would entail extra costs in MCSPDU traffic to keep the information up to date. Since the information recorded at a provider is consistent with that recorded at superior providers, within the limit of MCSPDU propagation delay, it may be said that the provider information base is partially replicated through the domain hierarchy.

Domain parameters become fixed and unchangeable with the establishment of the first MCS connection of a domain. A provider that lacks capacity for the maximum number of ids specified in each category may negotiate to join a domain on false pretense. It may speculate that at its low position in a hierarchy it will not be called upon to retain more than a fraction of the total information base. Such a provider may not support attachments and subordinates with the full range of MCS services they expect. Nonetheless, until its capacity is actually exceeded, such a provider may appear to be an equal member of the domain. This strategy may appeal to terminal nodes with limited aspirations.

Ids are placed into use first at the top MCS provider. They are placed into use at subordinate providers by a selective downward flow of MCSPDUs. Most are deleted from use in the same top-down manner. There are necessarily intervals

during which a subordinate provider records as in use an id that its superiors do not, because the MCSPDU that deletes the id remains in transit. This is not, however, a situation that endures. Control MCSPDUs are received and processed in the order of their transmission. The consequences of processing an MCSPDU, including its creation or deletion of channel and token ids, take effect before attention shifts to the next input event.

An exception to the preceding paragraph is the deletion of static and assigned channel ids. Although placed into use by a downward flow of **CJcf**, these channel ids are deleted in the opposite order – from the bottom up. Specifically, they are deleted when an accumulation of MCS-CHANNEL-LEAVE requests from attachments and **CLrq** MCSPDUs from subordinate providers combine to leave a channel unjoined. Such transitions motivate the transmission of **CLrq** further upward. Thus, for these two cases, the channel ids recorded as in use are a strict subset of those in use at the superior provider. This is an accidental corollary of optimizations designed to speed channel management as a prelude to data transfer.

Channels ids are put into use by **MCcf**, **AUcf**, **CJcf**, **CCcf**, and **CAin**; they are deleted by **MCcf**, **PCin**, **DUin**, **CLrq**, **CDin**, and **CEin**. Token ids are put into use by **MTcf**, **TGcf**, **TIcf**, and **TVin**; they are deleted by **MTcf**, **PTin**, **TRcf**, **TVrs** and **TVcf**. When an id is put into use at a given provider, the MCSPDU that is the cause may be forwarded to zero, one, several, or all subordinate providers. The use of an id may grow or contract gradually as, for example, individual users are admitted to and expelled from a private channel. When an id is deleted from a given provider, the MCSPDU that effects this is forwarded to all subordinates who may still be recording the id as in use.

The use of an id is ultimately tied to actions on a channel or token by a user attached to the domain (although there may be some delay, as explained, in communicating changes through the transmission of MCSPDUs). The ids recorded stably as in use at a given MCS provider are those that are actively employed by some user in the subtree of the provider. It follows that they are a subset of those recorded stably at any superior provider.

Deleting a user id has the corollary effect of deleting channel ids and token ids of which it is the sole user in a subtree.

Criteria for considering channel ids and token ids to be in use are specified in the following paragraphs.

11.2 Channel information

The four kinds of channel have corresponding criteria to determine whether a given attachment is considered to be using the channel id, hence whether it is to be represented in a provider's information base:

a. A static channel id (range 1..1000) is in use if the user has joined the channel with a successful MCS-CHANNEL-JOIN confirm and not left by MCS-CHANNEL-LEAVE request or indication.

b. A user id channel is in use if it was assigned to the user by a successful MCS-ATTACH-USER confirm and the user has not detached by MCS-DETACH-USER request or indication.

c. A private channel id is in use if the user has created the channel with a successful MCS-CHANNEL-CONVENE confirm or been admitted to it with MCS-CHANNEL-ADMIT indication and not expelled by MCS-CHANNEL-EXPEL indication and the channel has not been disbanded by MCS-CHANNEL-DISBAND request or indication.

d. An assigned channel id is in use if the user has joined the channel with a successful MCS-CHANNEL-JOIN confirm and not left by MCS-CHANNEL-LEAVE request or indication.

This information shall be recorded for a channel id in use:

a. The kind of channel it represents (static, user id, private, or assigned).

b. By which MCS attachments and which MCS connections to subordinate providers the channel is joined.

c. If a user id channel, the direction to it, that is, either the local MCS attachment to which the user id is assigned or the downward MCS connection to a subordinate provider in whose subtree the user resides.

d If a private channel id, the user id of the manager who convened it (whether or not the manager itself is in the subtree of the provider) and the set of all user ids in the subtree of the provider who have been admitted to the channel.

The information recorded for channel ids is employed as explained in clause 10 to validate request MCSPDUs and to route indication and confirm MCSPDUs.

11.3 Token information

The state transitions of a token id are shown in Figure 11-1.

An individual token id can be *grabbed* by a single user or *inhibited* by one or more. The action of **TVin** converts the state to *giving* along the branch of a domain hierarchy leading from the top MCS provider toward the intended recipient. This state decays to *ungivable* if the recipient detaches before its provider responds with **TVrs**. It resolves to *given* if instead the donor releases the token explicitly or detaches. During the giving of a token, the branch of a domain hierarchy leading from the donor intersects the branch leading toward the recipient at least at the top MCS provider. The token state changes from *grabbed* to *giving*, and possibly thereafter to *ungivable* or *given*, only along this intersection.

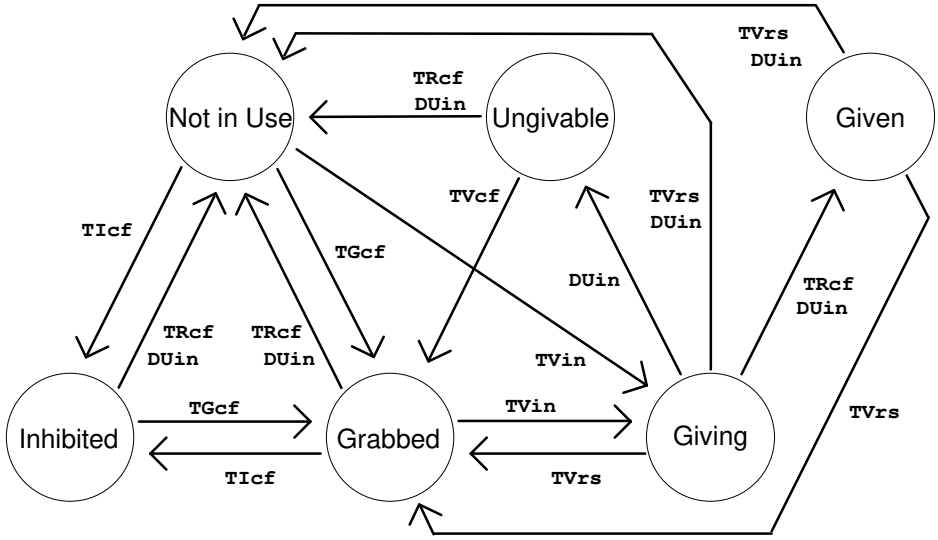


Figure 11-1 –State transitions of a token id

The user of a token stands in relationship to it as grabber, inhibitor, recipient, or both grabber and recipient (when giving a token to itself):

a. The user is a grabber if it has seized a token with a successful MCS-TOKEN-GRAB confirm and not released it by MCS-TOKEN-RELEASE request or successful MCS-TOKEN-GIVE confirm nor converted it with a successful MCS-TOKEN-INHIBIT confirm or if it has accepted an offered token with a successful MCS-TOKEN-GIVE response.

b. The user is an inhibitor if it has seized a token with a successful MCS-TOKEN-INHIBIT confirm and not released it with MCS-TOKEN-RELEASE request nor converted it with a successful MCS-TOKEN-GRAB confirm.

c. The user is a recipient if it has been offered a token by MCS-TOKEN-GIVE indication and not released it with an unsuccessful MCS-TOKEN-GIVE response.

This information shall be recorded for a token id in use:

a. The state of the token id at the MCS provider (not necessarily identical to that at the top provider).

b. If grabbed or ungivable, the user id of the grabber in the subtree of the provider.

c. If giving, the user id of the grabber (whether or not the grabber is in the subtree of the provider).

d. If giving or given, the user id of the recipient in the subtree of the provider.

e. If inhibited, the set of all user ids in the subtree of the provider who have inhibited the token.

The information recorded for token ids in use is employed as explained in clause 10 to validate response MCSPDUs and to route indication MCSPDUs.

The state of a token id at a subordinate provider need not be identical to that at the top MCS provider. This is due to the fact that a token donor does not in general process **TVin** or **TVrs** and that a recipient does not in general process a donor's **TRcf**. Figure 11-2 shows states that may arise in a complex token interaction.

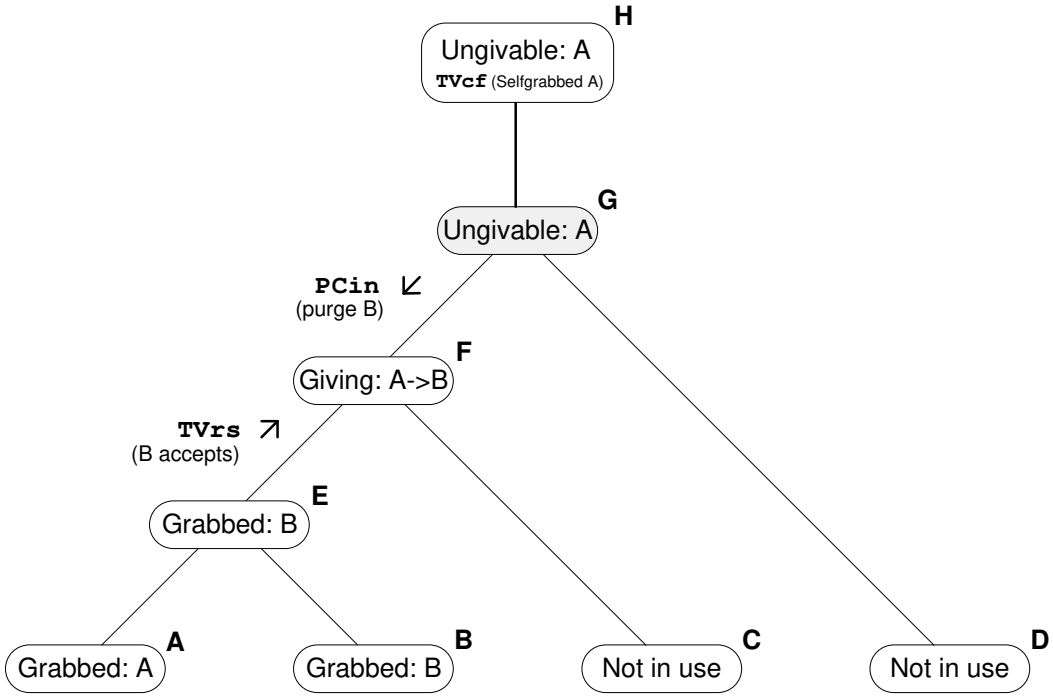


Figure 11-2–States of a token that may arise in a complex interaction

The figures focuses on one token id in the information base of providers A through H. A plausible history is that the token was given by user A, attached to provider A, to user B, attached to provider B. Before user B could respond, however, provider G connected the domain to a new top provider H and began a merger. The new domain, unable to accept user B because of a conflict in channel ids, initiated its purge through **PCin**. This MCSPDU is shown having completed part of its journey, through providers G and D. Provider G, as a result, has adjusted its token state from *giving* to *ungivable* and merged the token, with this state, into the new domain. The new top provider H, presented with a token in such a state, has queued for transmission an unsuccessful **TVcf** to return the token to user A. At the time shown, the token has also finally been accepted by user B, and a **TVrs** is making its way up the recipient's branch of the domain hierarchy, changing the token state at each provider from *giving* to *grabbed* by B. Two MCSPDUs now converge on provider F, and whichever arrives first determines its subsequent state there: either returned to *grabbed* by A or transiently *grabbed* by B and then *not in use* again after user B is detached. In either case, the state of the token will stabilize to *grabbed* by A when the new top provider transmits the **TVcf** it has pending.

12 Elements of procedure

12.1 MCSPDU sequencing

Control MCSPDUs remain in sequence between any pair of MCS providers because they travel over a single TC, the initial TC of an MCS connection. MCS providers shall process received MCSPDUs and shall transmit any resulting output MCSPDUs in the same order. This applies to MCSPDUs that are simply forwarded up or down the domain hierarchy as well as to MCSPDUs that are transformed, such as requests or responses into indications or confirms. The sequencing of MCSPDUs shall be maintained within an MCS provider if it is necessary to queue outputs for later transmission due to the back-pressure of flow control along a TC.

Data MCSPDUs of different priorities need not remain in sequence. On the contrary, the benefit of relative priorities is only achieved when higher priority data is advanced ahead of lower priority data. This means that it should be transmitted over separate TCs and queued separately within each MCS provider. If the number of data priorities

implemented in a domain is less than the maximum, fewer TCs are available, but providers that elect to do so may still maintain separate queues internally. This realizes some but not all the benefits of relative priority.

An MCS provider shall maintain the sequence of data MCSPDUs transmitted at a given priority. This is a tighter restriction than that imposed by ITU-T Rec. T.122, which guarantees only the sequencing of service data units transmitted at a given priority over the same destination channel.

Control MCSPDUs and data MCSPDUs of top priority are transmitted over the same initial TC and shall receive equal attention by an MCS provider. Data of lower priorities may lag behind. Control indications advancing ahead may arrive before lower priority data that was actually transmitted first.

12.2 Input flow control

An MCS provider has goals that sometimes conflict: to keep data moving briskly through a domain despite transient blockages towards some receivers, to give transmitters fair access to the available bandwidth, and to prevent any party from lagging far behind peers who are receiving the same multicast data. MCS is a reliable service that preserves the integrity of user data. Since a provider has limited capacity to store MCSPDUs when they cannot be transmitted immediately, it must have a defense of sometimes refusing further inputs. While details of the interface to transport services are a local matter, the abstract effect must be that incoming TSDUs are held in TC pipelines, intact and in sequence, to be received at some future time when flow control is lifted. As TC pipelines fill, remote MCS providers may find that they are blocked by back-pressure from transmitting further MCSPDUs and may need to invoke a similar defense.

Flow control is not signaled explicitly in the MCS protocol. This is a function of lower layers that would be wasteful to duplicate. As a result, it is difficult to sense, through the medium of an intervening TC, whether a remote provider is resisting further inputs. ~~There is no requirement for a conforming MCS provider to implement specific responses. However, t~~To meet conflicting goals as well as possible, the policies described next may be recommended.

An MCS provider may grant each incoming TC a fixed quota of buffers that it may fill with MCSPDUs before back pressure is applied. Each buffer is processed as specified in this protocol, then assigned for output to zero or more outgoing TCs. Output may occur immediately or may be delayed because a transport pipeline is full. Until it is output to the last TC, a buffer is charged against the input quota of the TC by which it arrived. After it is output to all requisite TCs, it is recycled as an increment to that quota. When a quota is exhausted, further inputs over the corresponding TC are halted. Input quotas may be set taking into account whether a TC is part of an upward or downward MCS connection and what data priority it represents.

Buffering can mitigate a disparity of rates between transmitters and receivers. A quota on inputs can prevent any one TC from monopolizing resources. It can also bound how far out of step two receivers of the same multicast data can get. The recommended scheme is not clever enough, however, to anticipate all patterns of use and may sometimes slow the rate of data transfer through a domain when there are acceptable alternatives. The invention of better flow control policies (implemented locally and requiring no additional communication of MCSPDUs) may be a means of product differentiation.

12.3 Throughput enforcement

In contrast to input flow control, some support for throughput enforcement is explicit in the MCS protocol. Firstly, the rate that is enforced is a domain parameter negotiated through MCS-CONNECT-PROVIDER. Secondly, the time interval over which throughput is monitored before taking adverse action is communicated by each MCS provider to its superior through **EDrq**. To describe the throughput enforcement interval requires that providers share some common principles of behavior. Still, enforcement remains a heuristic technique with room for invention.

Enforcing a minimum input rate at each receiver is an option available to controller~~ing~~ applications, which establish the MCS connections of a domain. This option is selected through the domain parameter for enforced throughput, which is stated in octets per second. While it seems intuitive that a party should not be allowed to run arbitrarily slowly and thereby obstruct data transfer among others, there is danger in seeking to enforce throughput too strictly.

Complex patterns of multiple transmitters are a problem. The kinds of back-pressure to which an enforcement policy reacts may not result from a single abnormally slow receiver. In the first place, MCSPDUs heading downward must compete for attention with MCSPDUs that rise from below but are reflected back, notably **SDin**. The downward flow experienced through a peer provider may therefore attain only a fraction of the nominal bandwidth and may vary dynamically with the number of other connections and attachments to the peer. Secondly, only top priority MCSPDUs can be expected to flow continuously over an MCS connection. Lesser priorities can properly be blocked by a peer provider for long periods of time due to the intensity of traffic received from other sources at higher priorities. Finally, there can be great variation in instantaneous throughput if it is measured by how long a blocked MCSPDU must wait

before being accepted onto an MCS connection. Among other things, this can depend on how many and in what order MCSPDUs from other sources are queued inside the peer provider.

Nonetheless, throughput enforcement is a valuable option in practical situations where patterns of data transfer are known to be more uniform. It shall be interpreted as requiring a minimal output of MCSPDUs to each direct MCS attachment and to each MCS connection downward over a time interval to be specified by the enforcing MCS provider. Each MCSPDU output, both control and data, shall count towards throughput as though it were the maximum size allowed by~~the~~ domain parameters. The output of an MCSPDU to an MCS attachment shall mean delivery of the associated primitive indication or confirm. Output to a downward MCS connection shall mean the absence of back-pressure at the transport service interface and acceptance into the corresponding TC pipeline.

Output shall be monitored as long as one or more MCSPDUs are queued, regardless of data priority, towards a given attachment or downward connection. Whenever the queues empty, monitoring shall cease, with no enforcement action taken. At the same time, no credit for good behavior shall be recorded to offset future slowdowns. Monitoring shall resume as soon as back-pressure prevents any MCSPDU from being output and requires it to be queued instead. While one or more MCSPDUs remain queued, the number actually output shall be counted over a set interval of time.

A throughput enforcement interval shall be chosen by each MCS provider. The interval shall be long enough to allow at least one MCSPDU of maximum size to be output at the minimum throughput. An MCS provider shall advise its superior of the interval chosen and of any subsequent changes to it by transmitting **`EDrq`** upward. A provider shall act against the offending MCS attachment or downward MCS connection at the end of any interval during which the monitored throughput fails to meet expectations. It shall detach the user or disconnect the connection.

Superior providers may set their throughput enforcement interval to be longer than that of any subordinate plus some margin of reaction time. The aim is to encourage enforcement action first at the lowest provider in a position to detect a problem. When a violator is removed at the end of some lower interval, superior providers should have enough time to sense the restoration of throughput to adequate levels. If they act too quickly, they may penalize a larger subtree than necessary and disrupt innocents in the domain.

The controller applications that set domain parameters should be conservative in their demands, expecting that throughput may occasionally dip during periods of intense application stimuli. If their concern is simply to defend against receivers that stop accepting anything at all, they may set the minimum throughput very low. Knowing the maximum MCSPDU size and enforced throughput rate, controllers can calculate the minimum interval that must elapse before any blockage is detected and overcome.

12.4 Domain configuration

ITU-T Rec. T.122 provides no mechanism for configuring the set of domains supported by an MCS provider. This must be considered a local matter whose standardization may be the subject of further study. This protocol assumes that an MCS provider will recognize some domain selectors as valid and others as invalid. It provides for domain selectors to be communicated as part of establishing an MCS connection.

An MCS provider participates implicitly in the negotiation of domain parameters. Whether the calling or the called side, it constrains the range of allowed parameter values according to limits for which it is configured. An MCS provider shall freeze the negotiability of domain parameters once any user has attached to a domain or once the first MCS connection has been established.

12.5 Domain merger

Domains are merged as a consequence of MCS-CONNECT-PROVIDER. If it is convenient to arrange that one domain or the other be empty at this point, there is little complication in a merger. In the most general case, however, provision must be made for updating the information base at the remaining top provider to contain the information base of the former top provider and for resolving any conflicts that become apparent. The details of this are explained in clause 10.

To aid understanding, an example of domain merger is illustrated in the sequence of Figures 12-1 through 12-4. Here provider E represents a former top that has joined a new domain by the highlighted MCS connection to an intermediate provider F. It is irrelevant whether provider E or provider F initiated the MCS connection.

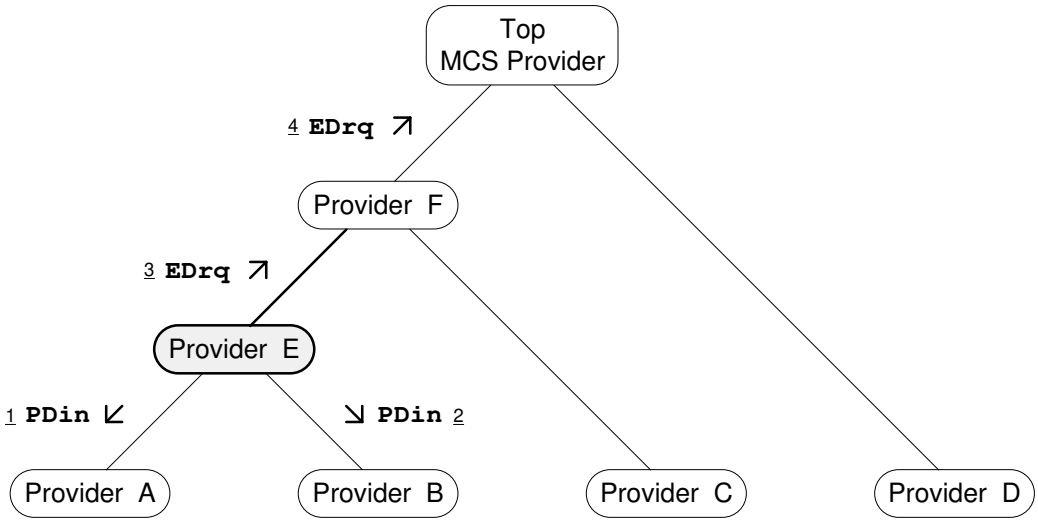


Figure 12-1 – Domain merger step one: establish hierarchy

Provider E, finding itself at the lower end of a connection upward, assumes responsibility for executing the merger. Since it is unsafe to transact other activity while the information base is in flux, provider E stops accepting inputs from its subtree. Any MCSPDUs already in transit from providers A and B or new ones generated before the merger is completed will be safeguarded and processed later. The downward flow of MCSPDUs, however, is not impeded, whether generated by provider E or forwarded from further above.

Until the merger completes, the only confirm MCSPDUs provider E will receive are **MCcf** and **MTcf**, since user requests are not allowed upward. These will confirm or purge channel and token ids already in use. Indication MCSPDUs **PCin**, **PTin**, **DUin**, and **CDin** of the upper domain may also arrive, deleting channel and token ids from the lower domain whose merger has been individually confirmed. Unconfirmed ids of the lower domain are protected against deletion, because as yet they mean something different from the same id of the upper domain. **PDin** must be obeyed to enforce the domain height limit. Remaining indications need not be applied by provider E while merger is in progress. Data transfers, in particular, need not flow between the formerly separate domains until merger is completed; they cannot flow until at least the channels conveying them have been sorted out. Two indications that can place new ids into use may be inconvenient to apply. To keep the information base consistent, if provider E refuses **CAin** or **TVin**, it will react as specified in clause 10.

The first actions of provider E are to send **PDin** downward, to ensure that the latest MCS connection has not created a cycle that would invalidate the principle that each domain hierarchy have exactly one top provider, and to send **EDrq** upward to report its existing height and throughput enforcement interval. Provider F, advancing thereby from height 2 to 3, passes **EDrq** along to the top provider, which then attains height 4.

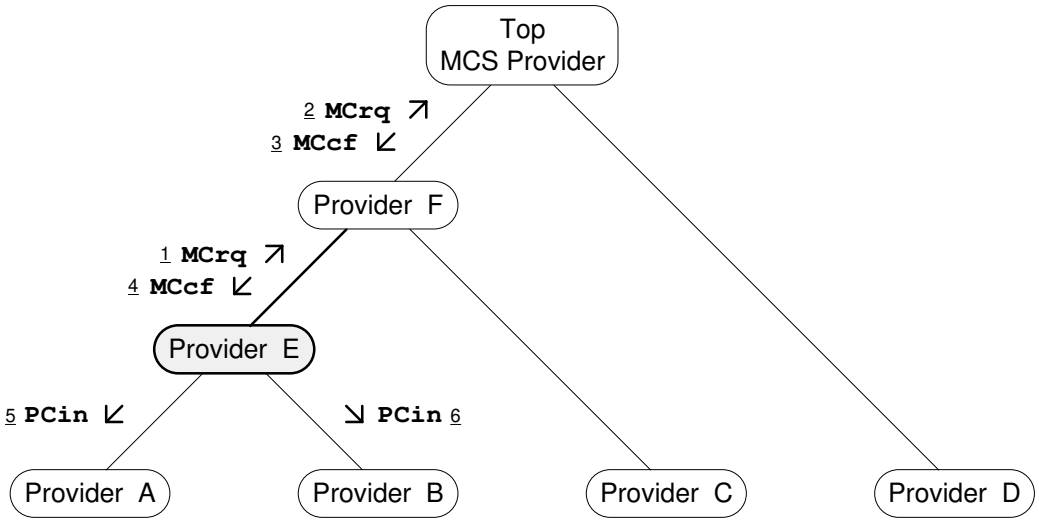


Figure 12-2 – Domain merger step two: merge user id channels

In the second stage of domain merger, provider E sends upward as many instances of **MCrq** as it takes to contain the user ids in its information base. User ids that do not conflict with the upper domain are confirmed and the rest are purged in equally many instances of **MCcf**. Provider E generates **PCin** from **MCcf** to report any purges throughout its subtree. This stage ends when all user ids have either been explicitly confirmed or purged.

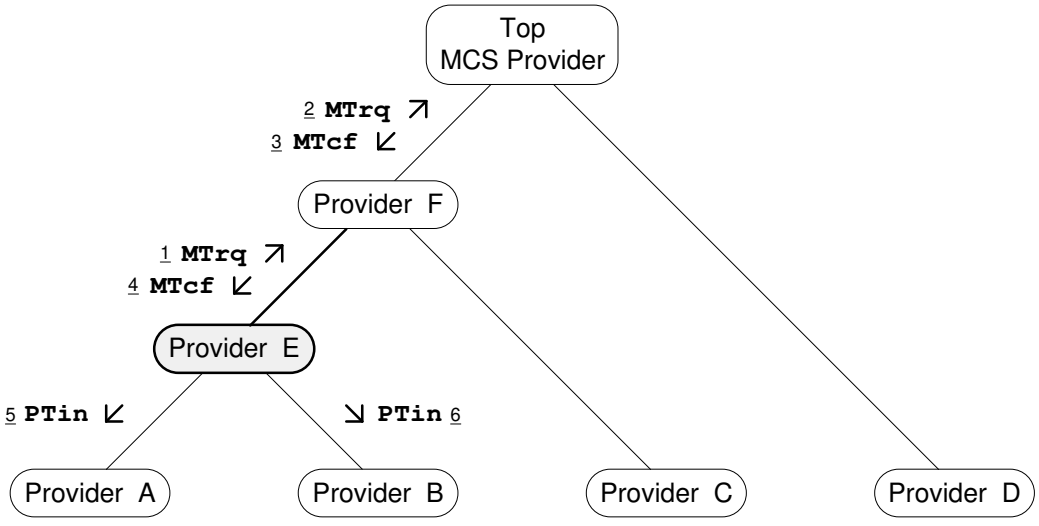


Figure 12-3 – Domain merger step three: merge token ids

Stage three resembles stage two, but concerns token ids rather than user id channels, using a parallel set of MCSPDUs. If user ids had not been merged first, portions of a later **MTrq** could be refused as invalid and the affected token ids would be unnecessarily purged. In the case of an inhibited token, the entire set of inhibiting users may not fit into a single MCSPDU. Provider E waits for confirmation of the first subset it sends upward before sending the inhibited token again with the remaining user ids. This protects against the first set being refusjected due to too many token ids in use but the remainder later being accepted, which would corrupt the information base. This stage ends when all token ids have either been explicitly confirmed or purged.

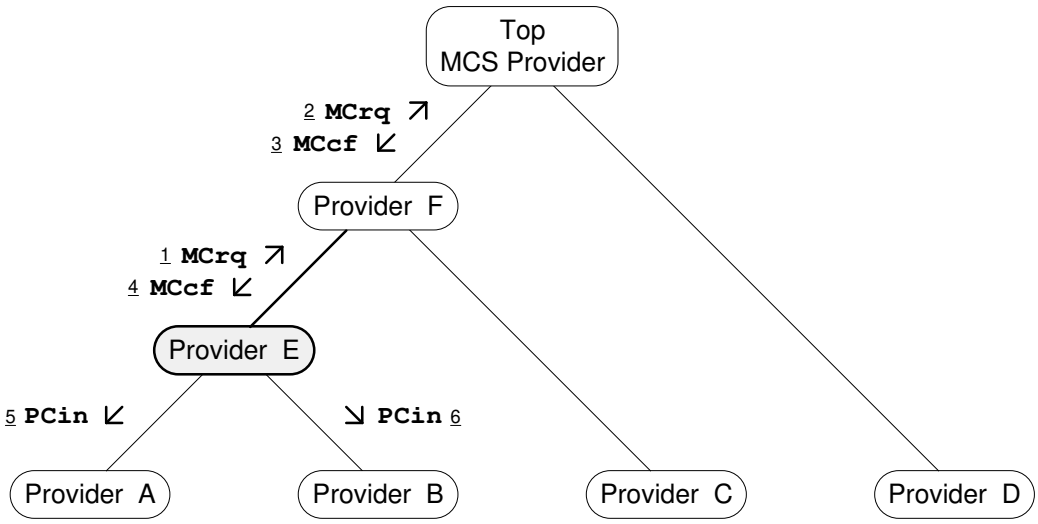


Figure 12-4 – Domain merger step four: merge remaining channel ids

Stage four involves the same MCSPDUs as stage two, but they contain different channel ids. User id channels having been taken care of, what remains are static, private, and assigned channel ids. If user ids had not been merged first, portions of a later **MCrq** could be refused as invalid and the affected channel ids would be unnecessarily purged. In the case of a private channel, the entire set of users admitted by the channel manager may not fit into a single MCSPDU. Provider E waits for confirmation of the first subset it sends upward before sending the private channel again with the remaining user ids. This protects against the first set being refusjected due to too many channel ids in use but the remainder later being accepted, which would corrupt the information base. This stage ends and domain merger completes when all remaining channel ids have either been explicitly confirmed or purged.

NOTE – Merging token ids first makes their exclusive possession more effective in suppressing data flow conflicts. Otherwise, data may leak between domains, as channel ids are confirmed, before the conflict is revealed through a token purge.

12.6 Domain disconnection

When an upward MCS connection is disconnected, an MCS provider shall eradicate its subtree of the domain by detaching all its direct MCS attachments and disconnecting all its other MCS connections. The affected provider cannot in general establish a residual domain in its own subtree, because it has no record of request MCSPDUs that were sent upward for which it will never receive a matching confirm MCSPDU.

When a downward MCS connection is disconnected, an MCS provider shall generate **DUrq** MCSPDUs for all users residing in that portion of its subtree, giving as reason *domain disconnected*.

12.7 Channel id allocation

Channel ids in the range of 1001 and above are allocated dynamically at the top MCS provider during the processing of primitive requests MCS-ATTACH-USER, MCS-CHANNEL-JOIN zero, and MCS-CHANNEL-CONVENE. There is no requirement that the values allocated fit any particular pattern. Rather, it is desirable that the values be randomly dispersed over the allowed range. This makes it more likely that two domains that operate independently for a substantial period of time may later be merged without conflicts in their respective allocation of channel ids. It also defends against recycling ids too quickly within a single domain ids as they are released from one use and then reallocated for a different use. Applications using MCS should have time to adjust to the disappearance of a channel or user id before it returns in a new incarnation.

Token ids, unlike channel ids, are not allocated, and the dividing line of value 1001 has no significance for them. A free token id with a given value is statically available to be grabbed or inhibited at any time, subject only to a domain limit on the total number in use at once.

12.8 Token status

Token status is defined formally in clause 7. It is returned as a component of token confirm MCSPDUs and is used to update the record for a token id in the information base of subordinate providers. Token status is not necessarily reported directly through a confirm primitive to the initiating user but may be reported indirectly through a result value.

When more that one token status value describes the relationship of a given user to a specified token id, the order of preference shall be as follows. *Self-recipient* shall be reported first, if it fits, to remind the user that they must respond to an MCS-TOKEN-GIVE indication. Next preferred is *self-giving*, to remind the user that they are engaged in an uncompleted operation, followed by *self-grabbed* or *self-inhibited*. Last in order are the remaining status values, reflecting the token's current state as a result of its sole use by other parties.

An MCS provider relies on the specified preference for token status values in order to update the token state correctly in its information base.

13 Reference implementation

Appendices II through VII contain an implementation of an MCS provider expressed in SDL. They demonstrate that the protocol specified can be realized with reasonable effort. The provision of this example should save implementors the work of reinventing equivalent logic and should speed the introduction of compatible systems.

SDL is a formal description technique more powerful than conventional state tables and better suited to the complexity of MCS. It admits two equivalent representations, textual and graphical. The reference implementation is cast in the former, which is more concise and easier to translate to a conventional programming language. It makes extensive use of abstract data types like the powerset generator. Those unfamiliar with SDL may wish to consult these references:

- CCITT Recommendation Z.100 (1988), *Specification and Description Language (SDL)*.
- Belina, Hogrefe, and Sarma: *SDL with Applications from Protocol Specification*, (Prentice Hall, 1991), ISBN 0-13-785890-6.
- Belina and Hogrefe: *The CCITT Specification and Description Language SDL*, Computer Networks and ISDN Systems, 16 (1988/89), pages 311-341.

Appendix II begins with a figure that summarizes relationships codified in the system and block definitions that follow. Processes appearing in the figure – control, domain, endpoint, and attachment – are defined individually in subsequent Appendices III through VI. Appendix VII explains assumptions of the model and illustrates key signal flows.

The appendices have been reviewed by technical experts and are believed to constitute a correct implementation of the MCS protocol defined in this specification, but they are not normative. In case of any discrepancy, descriptions contained in the body of this specification take precedence.

The MCS protocol may also be realized by other designs, which need not be based on the reference implementation.

The reference implementation of Appendices II through VII is parameterized to expand to a fully general MCS provider. Alternative implementations that are limited to a special case may be more compact and efficient. Of particular interest is the case of an MCS provider limited to one MCS connection, a condition appropriate for a terminal node. This might be further specialized to simplify mergers by voluntarily purging any ids employed in a lower domain. The selection of important special cases and their implementation in SDL remains a topic for further study.

Appendix I
Alternative encodings of an MCSPDU

(This appendix does not form an integral part of this Recommendation.)

I.1 Send data request

The relevant definitions, extracted from clause 7, are:

```
DomainMCSPDU      ::= CHOICE
{
        ...
        sdrq                    SDrq,
        ...
}

SDrq    ::= [APPLICATION 25] IMPLICIT SEQUENCE
{
        initiator               UserId,
        channelId               ChannelId,
        dataPriority            DataPriority,
        segmentation            Segmentation,
        userData                OCTET STRING
}

UserId          ::= DynamicChannelId

DynamicChannelId::= ChannelId (1001..65535)

ChannelId       ::= INTEGER (0..65535)

DataPriority    ::= ENUMERATED
{
                        top                     (0),
                        high                    (1),
                        medium                  (2),
                        low                     (3)
}

Segmentation    ::= BIT STRING
{
                        begin                   (0),
                        end                     (1)
} (SIZE (2))
```

A sample value of this type is:

```
sdrq
{
        initiator               1701,
        channelId               5,
        dataPriority            high,
        segmentation            {begin},
        userData                '4D4353'H
}
```

I.2 Basic Encoding Rules (BER)

BER applies a scheme of *identifier-length-contents* recursively to component types:

```
SDrq     Length  Contents
79       13

                 INTEGER          Length  Contents
                 02               02      06 A5

                 INTEGER          Length  Contents
                 02               01      05

                 ENUMERATED       Length  Contents
                 0A               01      01

                 BIT STRING       Length  Contents
                 03               02      06 80

                 OCTET STRING     Length  Contents
                 04               03      4D 43 53
```

Header octets exclusive of user data: 18 - 24, depending on MCSPDU tag, channel id, and user data length.

I.3 Packed Encoding Rules (PER)

PER can be decoded only if it is known in advance what ASN.1 type is contained in the encoding, since tags are not conveyed. This is one reason for defining a combined type of domain MCSPDUs. The application tags of domain MCSPDUs advance sequentially beginning from zero. This is convenient, because PER encodes values as offsets from the base of their range. With this arrangement, the value 25 identifies **SDrq** under both BER and PER:

```
CHOICE                                  -- 6 bits + pad
64

INTEGER (1001..65535)                   -- offset 1001
02 BC

INTEGER (0..65535)                      -- offset 0
00 05

ENUMERATED + BIT STRING (SIZE (2))      -- 2 bits + 2 bits + pad
60

OCTET STRING                            -- length + contents
03  4D 43 53
```

Header octets exclusive of user data: 7 - 8, depending on user data length.

PER encodes an unconstrained octet string length up to 127 in one octet and up to 16384 in two octets.

Appendix II
SDL decomposition of an MCS provider

(This appendix does not form an integral part of this Recommendation.)

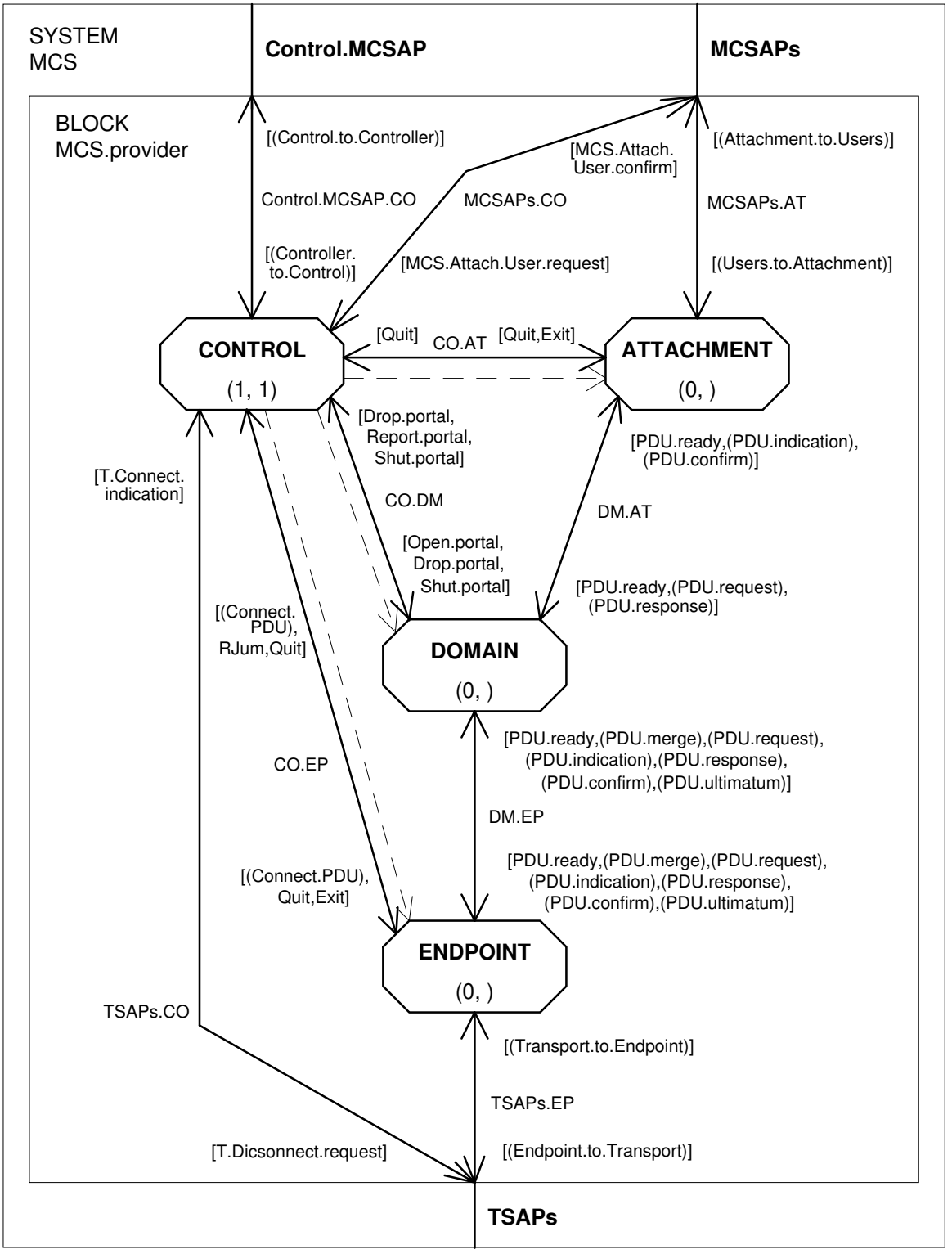


Figure II-1 – SDL decomposition of an MCS provider

```
SYSTEM  MCS;

SYNONYM oneSecond           Duration = 1000;    /* time is in milliseconds */

        /* Type definitions */

SYNTYPE         ChannelId       = Integer CONSTANTS 0:65535
ENDSYNTYPE;

NEWTYPE         UserId          INHERITS ChannelId
OPERATORS ALL;
ADDING
OPERATORS
        UserId:    ChannelId    -> UserId;      /* type cast */
        ChannelId: UserId       -> ChannelId;   /* type cast */
AXIOMS
        UserId(0) = 0;
        FOR ALL c in ChannelId
        (
                ChannelId(UserId(c)) = c;
        );
ENDNEWTYPE;

SYNTYPE         TokenId         = Integer CONSTANTS 1:65535
ENDSYNTYPE;

NEWTYPE         ChannelIdSet    SetOf(ChannelId);
ENDNEWTYPE;
NEWTYPE         UserIdSet       SetOf(UserId);
ENDNEWTYPE;
NEWTYPE         TokenIdSet      SetOf(TokenId);
ENDNEWTYPE;

NEWTYPE         TokenStatus
LITERALS
        NotInUse,
        SelfGrabbed,
        OtherGrabbed,
        SelfInhibited,
        OtherInhibited,
        SelfRecipient,
        SelfGiving,
        OtherGiving;
ENDNEWTYPE;

SYNTYPE         DataPriority    = Integer CONSTANTS 0:3
ENDSYNTYPE;

NEWTYPE         Segmentation
STRUCT
        begin                   Boolean;
        end                     Boolean;
ENDNEWTYPE;

NEWTYPE         DomainParameters
STRUCT
        maxChannelIds           Natural;
        maxUserIds              Natural;
        maxTokenIds             Natural;
        numPriorities           Natural;
        minThroughput           Natural;
        maxHeight               Natural;
        maxMCSPDUsize           Natural;
        protocolVersion         Natural;
ENDNEWTYPE;

SYNTYPE         DomainSelector  = OctetString
ENDSYNTYPE;

SYNTYPE         TSAPAddress     = OctetString
ENDSYNTYPE;

NEWTYPE         TransportQOS            /* quality of service */
STRUCT
        throughput              Natural;        /* octets per second */
        transitDelay            Duration;       /* one-way */
        dataPriority            Natural;        /* 0 is highest */
ENDNEWTYPE;
```

```
 80    NEWTYPE         TransportQOSByPri       Array(DataPriority, TransportQOS);
 81    ENDNEWTYPE;

 83    SYNTYPE         UserData        = OctetString
 84    ENDSYNTYPE;

 86    SYNTYPE         TSDU            = OctetString
 87    ENDSYNTYPE;

 89    SYNTYPE         Octet           = Integer CONSTANTS 0:255
 90    ENDSYNTYPE;

 92    NEWTYPE         OctetString     String(Octet, NullString);
 93    ENDNEWTYPE;

 95    GENERATOR       SetOf (TYPE ItemType)   /* subsets with choice operator */
 96    /*AS*/
 97            Powerset(ItemType);
 98    ADDING
 99    OPERATORS
100            Pick:   SetOf   -> ItemType;    /* chooses any element */
101    AXIOMS
102            Pick(Empty) == ERROR!;
103            FOR ALL s IN SetOf
104            (
105                    s /= Empty ==> Pick(s) in s;
106            );
107    DEFAULT
108            Empty;
109    ENDGENERATOR;

111    NEWTYPE         Reason
112    LITERALS
113            RN_domain_disconnected,
114            RN_provider_initiated,
115            RN_token_purged,
116            RN_user_requested,
117 +          RN_channel_purged,
118 *          RN_channel_disbanded,           /* not in MCSPDUs */
119            RN_domain_not_hierarchical,     /* not in MCSPDUs */
120            RN_parameters_unacceptable,     /* not in MCSPDUs */
121            RN_unspecified;                 /* not in MCSPDUs */
122    ENDNEWTYPE;

124    NEWTYPE         Result
125    LITERALS
126            RT_successful,
127            RT_domain_merging,
128            RT_domain_not_hierarchical,
129            RT_no_such_channel,
130            RT_no_such_domain,
131            RT_no_such_user,
132            RT_not_admitted,
133            RT_other_user_id,
134            RT_parameters_unacceptable,
135            RT_token_not_available,
136            RT_token_not_possessed,
137            RT_too_many_channels,
138            RT_too_many_tokens,
139            RT_too_many_users,
140            RT_unspecified_failure,
141            RT_user_rejected,
142            RT_congested,                   /* not in MCSPDUs */
143            RT_domain_disconnected;         /* not in MCSPDUs */
144    ENDNEWTYPE;

146            /* The next three identifier types distinguish separate instances
147            of communication across the interface between an MCS provider and
148            its environment.  MCSConnectionId maps to some resource within the
149            Control process.  MCSAttachmentId, which equals the process id of
150            an Attachment process, could be considered implicit, since it is
151            the source or destination address of corresponding signals, but
152            the usage is clearer when it is made an explicit signal parameter.
153            The same model is assumed for TCEndpointId. */

155    SYNTYPE         MCSConnectionId = Natural
156    ENDSYNTYPE;

158    SYNTYPE         MCSAttachmentId = PId
```

```
159     ENDSYNTYPE;

161     SYNTYPE         TCEndpointId    = PId
162     ENDSYNTYPE;

164             /* Block decomposition */

166     BLOCK  MCS.provider     REFERENCED;

168     CHANNEL Control.MCSAP
169             FROM ENV TO MCS.provider WITH
170                     (Controller.to.Control);
171             FROM MCS.provider TO ENV WITH
172                     (Control.to.Controller);
173     ENDCHANNEL;

175     SIGNALLIST Controller.to.Control =
176             MCS.Connect.Provider.request,
177             MCS.Connect.Provider.response,
178             MCS.Disconnect.Provider.request;

180     SIGNALLIST Control.to.Controller =
181             MCS.Connect.Provider.indication,
182             MCS.Connect.Provider.confirm,
183             MCS.Disconnect.Provider.indication;

185     SIGNAL  MCS.Connect.Provider.request
186             (
187                     Natural,                /* requester's label */
188                     TSAPAddress,            /* calling */
189                     DomainSelector,
190                     TSAPAddress,            /* called */
191                     DomainSelector,
192                     Boolean,                /* upward */
193                     DomainParameters,       /* target */
194                     DomainParameters,       /* minimum */
195                     DomainParameters,       /* maximum */
196                     TransportQOSByPri,      /* target */
197                     TransportQOSByPri,      /* minimum */
198                     UserData
199             );
200     SIGNAL  MCS.Connect.Provider.indication
201             (
202                     MCSConnectionId,        /* provider-assigned */
203                     TSAPAddress,            /* calling */
204                     DomainSelector,
205                     TSAPAddress,            /* called */
206                     DomainSelector,
207                     Boolean,                /* upward */
208                     DomainParameters,       /* target */
209                     DomainParameters,       /* minimum */
210                     DomainParameters,       /* maximum */
211                     UserData
212             );
213     SIGNAL  MCS.Connect.Provider.response
214             (
215                     MCSConnectionId,
216                     Result,
217                     DomainParameters,
218                     UserData
219             );
220     SIGNAL  MCS.Connect.Provider.confirm
221             (
222                     Natural,                /* requester's label */
223                     Result,
224                     MCSConnectionId,        /* provider-assigned */
225                     DomainParameters,
226                     UserData
227             );

229     SIGNAL  MCS.Disconnect.Provider.request
230             (
231                     MCSConnectionId
232             );
233     SIGNAL  MCS.Disconnect.Provider.indication
234             (
235                     MCSConnectionId,
236                     Reason
237             );
```

```
239     CHANNEL MCSAPs
240             FROM ENV TO MCS.provider WITH
241                     MCS.Attach.User.request,
242                     (Users.to.Attachment);
243             FROM MCS.provider TO ENV WITH
244                     (Attachment.to.Users);
245     ENDCHANNEL;

247     SIGNALLIST Users.to.Attachment =
248             MCS.ready,
249             MCS.Detach.User.request,
250             MCS.Channel.Join.request,
251             MCS.Channel.Leave.request,
252             MCS.Channel.Convene.request,
253             MCS.Channel.Disband.request,
254             MCS.Channel.Admit.request,
255             MCS.Channel.Expel.request,
256             MCS.Send.Data.request,
257             MCS.Uniform.Send.Data.request,
258             MCS.Token.Grab.request,
259             MCS.Token.Inhibit.request,
260             MCS.Token.Give.request,
261             MCS.Token.Give.response,
262             MCS.Token.Please.request,
263             MCS.Token.Release.request,
264             MCS.Token.Test.request;

266     SIGNALLIST Attachment.to.Users =
267             MCS.ready,
268             MCS.Attach.User.confirm,
269             MCS.Detach.User.indication,
270             MCS.Channel.Join.confirm,
271             MCS.Channel.Leave.indication,
272             MCS.Channel.Convene.confirm,
273             MCS.Channel.Disband.indication,
274             MCS.Channel.Admit.indication,
275             MCS.Channel.Expel.indication,
276             MCS.Send.Data.indication,
277             MCS.Uniform.Send.Data.indication,
278             MCS.Token.Grab.confirm,
279             MCS.Token.Inhibit.confirm,
280             MCS.Token.Give.indication,
281             MCS.Token.Give.confirm,
282             MCS.Token.Please.indication,
283             MCS.Token.Release.confirm,
284             MCS.Token.Test.confirm;

286     SIGNAL  MCS.ready                       /* allows one MCS.[Uniform.]Send.Data */
287             (
288                     MCSAttachmentId,
289                     DataPriority
290             );

292     SIGNAL  MCS.Attach.User.request
293             (
294 +                   Natural,                /* requester's label */
295                     DomainSelector
296             );
297     SIGNAL  MCS.Attach.User.confirm
298             (
299 |                   Natural,                /* requester's label */
300                     Result,
301 +                   MCSAttachmentId,        /* provider-assigned */
302                     UserId
303             );

305     SIGNAL  MCS.Detach.User.request
306             (
307                     MCSAttachmentId
308             );
309     SIGNAL  MCS.Detach.User.indication
310             (
311                     MCSAttachmentId,
312                     UserId,
313                     Reason
314             );

316     SIGNAL  MCS.Channel.Join.request
```

```
317             (
318                     MCSAttachmentId,
319                     ChannelId
320             );
321     SIGNAL  MCS.Channel.Join.confirm
322             (
323                     MCSAttachmentId,
324 |                   ChannelId,              /* requested */
325 |                   Result,
326 |                   ChannelId
327             );

329     SIGNAL  MCS.Channel.Leave.request
330             (
331                     MCSAttachmentId,
332                     ChannelId
333             );
334     SIGNAL  MCS.Channel.Leave.indication
335             (
336                     MCSAttachmentId,
337                     ChannelId,
338                     Reason
339             );

341     SIGNAL  MCS.Channel.Convene.request
342             (
343                     MCSAttachmentId
344             );
345     SIGNAL  MCS.Channel.Convene.confirm
346             (
347                     MCSAttachmentId,
348                     Result,
349                     ChannelId
350             );

352     SIGNAL  MCS.Channel.Disband.request
353             (
354                     MCSAttachmentId,
355                     ChannelId
356             );
357     SIGNAL  MCS.Channel.Disband.indication
358             (
359                     MCSAttachmentId,
360                     ChannelId,
361                     Reason
362             );

364     SIGNAL  MCS.Channel.Admit.request
365             (
366                     MCSAttachmentId,
367                     ChannelId,
368                     UserIdSet
369             );
370     SIGNAL  MCS.Channel.Admit.indication
371             (
372                     MCSAttachmentId,
373                     ChannelId,
374                     UserId
375             );

377     SIGNAL  MCS.Channel.Expel.request
378             (
379                     MCSAttachmentId,
380                     ChannelId,
381                     UserIdSet
382             );
383     SIGNAL  MCS.Channel.Expel.indication
384             (
385                     MCSAttachmentId,
386                     ChannelId,
387                     Reason
388             );

390     SIGNAL  MCS.Send.Data.request
391             (
392                     MCSAttachmentId,
393                     ChannelId,
394                     DataPriority,
395                     Segmentation,
```

```
396                     UserData
397             );
398     SIGNAL  MCS.Send.Data.indication
399             (
400                     MCSAttachmentId,
401                     ChannelId,
402                     DataPriority,
403                     UserId,
404                     Segmentation,
405                     UserData
406             );

408     SIGNAL  MCS.Uniform.Send.Data.request
409             (
410                     MCSAttachmentId,
411                     ChannelId,
412                     DataPriority,
413                     Segmentation,
414                     UserData
415             );
416     SIGNAL  MCS.Uniform.Send.Data.indication
417             (
418                     MCSAttachmentId,
419                     ChannelId,
420                     DataPriority,
421                     UserId,
422                     Segmentation,
423                     UserData
424             );

426     SIGNAL  MCS.Token.Grab.request
427             (
428                     MCSAttachmentId,
429                     TokenId
430             );
431     SIGNAL  MCS.Token.Grab.confirm
432             (
433                     MCSAttachmentId,
434                     TokenId,
435                     Result
436             );

438     SIGNAL  MCS.Token.Inhibit.request
439             (
440                     MCSAttachmentId,
441                     TokenId
442             );
443     SIGNAL  MCS.Token.Inhibit.confirm
444             (
445                     MCSAttachmentId,
446                     TokenId,
447                     Result
448             );

450     SIGNAL  MCS.Token.Give.request
451             (
452                     MCSAttachmentId,
453                     TokenId,
454                     UserId
455             );
456     SIGNAL  MCS.Token.Give.indication
457             (
458                     MCSAttachmentId,
459                     TokenId,
460                     UserId
461             );
462     SIGNAL  MCS.Token.Give.response
463             (
464                     MCSAttachmentId,
465                     TokenId,
466                     Result
467             );
468     SIGNAL  MCS.Token.Give.confirm
469             (
470                     MCSAttachmentId,
471                     TokenId,
472                     Result
473             );
```

```
475     SIGNAL  MCS.Token.Please.request
476             (
477                     MCSAttachmentId,
478                     TokenId
479             );
480     SIGNAL  MCS.Token.Please.indication
481             (
482                     MCSAttachmentId,
483                     TokenId,
484                     UserId
485             );

487     SIGNAL  MCS.Token.Release.request
488             (
489                     MCSAttachmentId,
490                     TokenId
491             );
492     SIGNAL  MCS.Token.Release.confirm
493             (
494                     MCSAttachmentId,
495                     TokenId,
496                     Result
497             );

499     SIGNAL  MCS.Token.Test.request
500             (
501                     MCSAttachmentId,
502                     TokenId
503             );
504     SIGNAL  MCS.Token.Test.confirm
505             (
506                     MCSAttachmentId,
507                     TokenId,
508                     TokenStatus
509             );

511     CHANNEL TSAPs
512             FROM MCS.provider TO ENV WITH
513                     (Endpoint.to.Transport);
514             FROM ENV TO MCS.provider WITH
515                     T.Connect.indication,
516                     (Transport.to.Endpoint);
517     ENDCHANNEL;

519     SIGNALLIST Endpoint.to.Transport =
520             T.ready,
521             T.Connect.request,
522             T.Connect.response,
523             T.Data.request,
524             T.Disconnect.request;

526     SIGNALLIST Transport.to.Endpoint =
527             T.ready,
528             T.Connect.confirm,
529             T.Data.indication,
530             T.Disconnect.indication;

532     SIGNAL  T.ready                         /* allows one T.Data */
533             (
534                     TCEndpointId
535             );

537     SIGNAL  T.Connect.request
538             (
539                     Natural,                /* requester's label */
540                     TSAPAddress,            /* calling */
541                     TSAPAddress,            /* called */
542                     TransportQOS,           /* target */
543                     TransportQOS            /* minimum */
544             );
545     SIGNAL  T.Connect.indication
546             (
547                     TCEndpointId,           /* provider-assigned */
548                     TSAPAddress,            /* calling */
549                     TSAPAddress,            /* called */
550                     TransportQOS,           /* offered */
551                     TransportQOS            /* minimum */
552             );
553     SIGNAL  T.Connect.response
```

```
554             (
555                     TCEndpointId,
556                     TransportQOS            /* selected */
557             );
558     SIGNAL  T.Connect.confirm
559             (
560                     Natural,                /* requester's label */
561                     TCEndpointId,           /* provider-assigned */
562                     TransportQOS            /* selected */
563             );

565     SIGNAL  T.Data.request
566             (
567                     TCEndpointId,
568                     TSDU
569             );
570     SIGNAL  T.Data.indication
571             (
572                     TCEndpointId,
573                     TSDU
574             );

576     SIGNAL  T.Disconnect.request
577             (
578                     TCEndpointId
579             );
580     SIGNAL  T.Disconnect.indication
581             (
582                     Natural,                /* requester's label */
583                     TCEndpointId            /* provider-assigned */
584             );

586     ENDSYSTEM;


589     BLOCK  MCS.provider;

591     SYNONYM maxPortalIds    Natural = EXTERNAL;     /* an implementation limit */

593             /* Data type definitions */

595     NEWTYPE         PDUKind         /* domain MCSPDUs */
596     LITERALS
597             PDin,           /* plumb domain indication */
598             EDrq,           /* erect domain request */
599             MCrq,           /* merge channels request */
600             MCcf,           /* merge channels confirm */
601             PCin,           /* purge channels indication */
602             MTrq,           /* merge tokens request */
603             MTcf,           /* merge tokens confirm */
604             PTin,           /* purge tokens indication */
605             DPum,           /* disconnect provider ultimatum */
606             RJum,           /* reject MCSPDU ultimatum */
607             AUrq,           /* attach user request */
608             AUcf,           /* attach user confirm */
609             DUrq,           /* detach user request */
610             DUin,           /* detach user confirm */
611             CJrq,           /* channel join request */
612             CJcf,           /* channel join confirm */
613             CLrq,           /* channel leave request */
614             CCrq,           /* channel convene request */
615             CCcf,           /* channel convene confirm */
616             CDrq,           /* channel disband request */
617             CDin,           /* channel disband confirm */
618             CArq,           /* channel admit request */
619             CAin,           /* channel admit confirm */
620             CErq,           /* channel expel request */
621             CEin,           /* channel expel confirm */
622             SDrq,           /* send data request */
623             SDin,           /* send data indication */
624             USrq,           /* uniform send data request */
625             USin,           /* uniform send data indication */
626             TGrq,           /* token grab request */
627             TGcf,           /* token grab confirm */
628             TIrq,           /* token inhibit request */
629             TIcf,           /* token inhibit confirm */
630             TVrq,           /* token give request */
631             TVin,           /* token give indication */
632             TVrs,           /* token give response */
```

```
633             TVcf,           /* token give confirm */
634             TPrq,           /* token please request */
635             TPin,           /* token please indication */
636             TRrq,           /* token release request */
637             TRcf,           /* token release confirm */
638             TTrq,           /* token test request */
639             TTcf;           /* token test confirm */
640     ENDNEWTYPE;

642     NEWTYPE         PDUStruct
643     STRUCT
644             kind                    PDUKind;
645             /* fields used depend on kind */
646             channelId               ChannelId;
647             channelIds              ChannelIdSet;
648             dataPriority            DataPriority;
649 |           detachUserIds            UserIdSet;
650 *           diagnostic              Diagnostic;
651             heightLimit             Natural;
652             initialOctets           OctetString;
653 *           initiator               UserId;
654             mergeChannels           ChannelAttributesSet;
655             mergeTokens             TokenAttributesSet;
656             purgeChannelIds         ChannelIdSet;
657             purgeTokenIds           TokenIdSet;
658             reason                  Reason;
659             recipient               UserId;
660 +           requested               ChannelId;
661             result                  Result;
662             segmentation            Segmentation;
663             subHeight               Natural;
664             subInterval             Duration;
665             tokenId                 TokenId;
666             tokenStatus             TokenStatus;
667             userData                UserData;
668             userIds                 UserIdSet;
669     ENDNEWTYPE;

671     NEWTYPE         ChannelKind
672     LITERALS
673             Static,         /* range 1:1000 = static: known permanently */
674             UserId,         /* dynamic: Attach-User / Detach-User */
675             Private,        /* dynamic: Channel-Convene / Channel-Disband */
676             Assigned;       /* dynamic: Channel-Join zero / last Channel-Leave */
677     ENDNEWTYPE;

679     NEWTYPE         ChannelAttributes
680     STRUCT
681             channelId       ChannelId;      /* the channel with these attributes */
682             kind            ChannelKind;    /* (Static,UserId,Private,Assigned) */
683             manager         UserId;         /* if (Private): the channel manager */
684             admitted        UserIdSet;      /* if (Private): zero or more users */
685             joined          Boolean;        /* if (UserId,Private): True if joined */
686     ENDNEWTYPE;

688     NEWTYPE         ChannelAttributesSet    SetOf(ChannelAttributes);
689     ENDNEWTYPE;

691     NEWTYPE         TokenKind
692     LITERALS
693             Grabbed,        /* assigned exclusively to one user */
694             Inhibited,      /* inhibited by one or more users */
695             Giving,         /* reassigning grabbed to a new user */
696             Ungivable,      /* the recipient has since detached */
697             Given;          /* donor released token or detached */
698     ENDNEWTYPE;

700     NEWTYPE         TokenAttributes
701     STRUCT
702             tokenId         TokenId;        /* the token with these attributes */
703             kind            TokenKind;      /* (Grabbed,Inhibited,Giving,Ungivable,Given) */
704             grabber         UserId;         /* if (Grabbed,Giving,Ungivable): user */
705             recipient       UserId;         /* if (Giving,Given): an intended user */
706             inhibitors      UserIdSet;      /* if (Inhibited): one or more users */
707     ENDNEWTYPE;

709     NEWTYPE         TokenAttributesSet      SetOf(TokenAttributes);
710     ENDNEWTYPE;
```

```
712     SYNTYPE         PortalId        = Integer CONSTANTS 0:maxPortalIds
713     ENDSYNTYPE;

715     NEWTYPE         PortalIdSet     SetOf(PortalId);
716     ENDNEWTYPE;

718     NEWTYPE         PortalKind
719     LITERALS
720             Attached,       /* MCS Attachment through an MCSAP */
721     |       Downlink,       /* MCS Connection to a provider below */
722     |       Uplink;         /* MCS Connection to a provider above */
723     ENDNEWTYPE;

725     NEWTYPE         PIdByPri        Array(DataPriority, PId);
726     ENDNEWTYPE;

728     NEWTYPE         Diagnostic
729     LITERALS
730             DC_inconsistent_merge,
731             DC_forbidden_PDU_downward,
732             DC_forbidden_PDU_upward,
733             DC_invalid_BER_encoding,
734             DC_invalid_PER_encoding,
735             DC_misrouted_user,
736             DC_unrequested_confirm,
737             DC_wrong_transport_priority,
738 +           DC_channel_id_conflict,
739 +           DC_token_id_conflict,
740 +           DC_not_user_id_channel,
741 +           DC_too_many_channels,
742 +           DC_too_many_tokens,
743 +           DC_too_many_users,
744             DC_OK,                          /* not in MCSPDUs */
745             DC_ignore,                      /* not in MCSPDUs */
746             DC_height_limit_exceeded,       /* not in MCSPDUs */
747             DC_throughput_inadequate;       /* not in MCSPDUs */
748     ENDNEWTYPE;

750             /* Process decomposition */

752     PROCESS  Control        (1,1) REFERENCED;       /* CO */
753     PROCESS  Attachment     (0,)  REFERENCED;       /* AT */
754     PROCESS  Domain         (0,)  REFERENCED;       /* DM */
755     PROCESS  Endpoint       (0,)  REFERENCED;       /* EP */

757     CONNECT Control.MCSAP AND Control.MCSAP.CO;

759     SIGNALROUTE Control.MCSAP.CO
760             FROM ENV TO Control WITH
761                     (Controller.to.Control);
762             FROM Control TO ENV WITH
763                     (Control.to.Controller);

765     CONNECT MCSAPs AND MCSAPs.CO, MCSAPs.AT;

767     SIGNALROUTE MCSAPs.CO
768             FROM ENV TO Control WITH
769                     MCS.Attach.User.request;
770             FROM Control TO ENV WITH
771                     MCS.Attach.User.confirm;

773     SIGNALROUTE MCSAPs.AT
774             FROM ENV TO Attachment WITH
775                     (Users.to.Attachment);
776             FROM Attachment TO ENV WITH
777                     (Attachment.to.Users);

779     CONNECT TSAPs AND TSAPs.CO, TSAPs.EP;

781     SIGNALROUTE TSAPs.CO
782             FROM ENV TO Control WITH
783                     T.Connect.indication;
784             FROM Control TO ENV WITH
785                     T.Disconnect.request;

787     SIGNALROUTE TSAPs.EP
788             FROM ENV TO Endpoint WITH
789                     (Transport.to.Endpoint);
790             FROM Endpoint TO ENV WITH
```

```
791                     (Endpoint.to.Transport);

793     SIGNALROUTE CO.AT
794             FROM Control TO Attachment WITH
795                     Quit,
796                     Exit;
797             FROM Attachment TO Control WITH
798                     Quit;

800     SIGNALROUTE CO.DM
801             FROM Control TO Domain WITH
802                     Open.portal,
803                     Drop.portal,
804                     Shut.portal;
805             FROM Domain TO Control WITH
806                     Drop.portal,
807                     Report.portal,
808                     Shut.portal;

810     SIGNALROUTE CO.EP
811             FROM Control TO Endpoint WITH
812                     (Connect.PDU),
813                     Quit,
814                     Exit;
815             FROM Endpoint TO Control WITH
816                     (Connect.PDU),
817                     RJum,
818                     Quit;

820     SIGNALROUTE DM.AT
821             FROM Domain TO Attachment WITH
822                     PDU.ready,
823                     (PDU.indication),
824                     (PDU.confirm);
825             FROM Attachment TO Domain WITH
826                     PDU.ready,
827                     (PDU.request),
828                     (PDU.response);

830     SIGNALROUTE DM.EP
831             FROM Domain TO Endpoint WITH
832                     PDU.ready,
833                     (PDU.merge),
834                     (PDU.request),
835                     (PDU.indication),
836                     (PDU.response),
837                     (PDU.confirm),
838                     (PDU.ultimatum);
839             FROM Endpoint TO Domain WITH
840                     PDU.ready,
841                     (PDU.merge),
842                     (PDU.request),
843                     (PDU.indication),
844                     (PDU.response),
845                     (PDU.confirm),
846                     (PDU.ultimatum);

848     SIGNAL  Open.portal
849             (
850                     PortalId,
851                     PortalKind,
852                     PIdByPri
853             );

855     SIGNAL  Report.portal
856             (
857                     PortalId,
858                     Diagnostic
859             );

861     SIGNAL  Drop.portal
862             (
863                     PortalId,
864                     Reason
865             );

867     SIGNAL  Shut.portal
868             (
869                     PortalId
```

```
870             );

872     SIGNAL  Quit;

874     SIGNAL  Exit;

876     SIGNALLIST Connect.PDU =
877             Connect.Initial,
878             Connect.Response,
879             Connect.Additional,
880             Connect.Result;

882     SIGNAL  Connect.Initial
883             (
884                     DomainSelector,         /* calling */
885                     DomainSelector,         /* called */
886                     Boolean,                /* upward */
887                     DomainParameters,       /* target */
888                     DomainParameters,       /* minimum */
889                     DomainParameters,       /* maximum */
890                     UserData
891             );
892     SIGNAL  Connect.Response
893             (
894                     Result,
895                     Natural,
896                     DomainParameters,
897                     UserData
898             );
899     SIGNAL  Connect.Additional
900             (
901                     Natural,
902                     DataPriority
903             );
904     SIGNAL  Connect.Result
905             (
906                     Result
907             );

909     SIGNALLIST PDU.merge =
910             PDin, EDrq, MCrq, MCcf, MTrq,
911             MTcf;

913     SIGNALLIST PDU.request =
914             AUrq, DUrq, CJrq, CLrq, CCrq,
915             CDrq, CArq, CErq, SDrq, USrq,
916             TGrq, TIrq, TVrq, TPrq, TRrq,
917             TTrq;

919     SIGNALLIST PDU.indication =
920             PCin, PTin, DUin, CDin, CAin,
921             CEin, SDin, USin, TVin, TPin;

923     SIGNALLIST PDU.response =
924             TVrs;

926     SIGNALLIST PDU.confirm =
927             AUcf, CJcf, CCcf, TGcf, TIcf,
928             TVcf, TRcf, TTcf;

930     SIGNALLIST PDU.ultimatum =
931             DPum, RJum;

933     SIGNAL  PDU.ready                       /* allows one domain MCSPDU */
934             (
935                     DataPriority
936             );

938     SIGNAL  PDin    (PDUStruct);    /* plumb domain indication */
939     SIGNAL  EDrq    (PDUStruct);    /* erect domain request */
940     SIGNAL  MCrq    (PDUStruct);    /* merge channels request */
941     SIGNAL  MCcf    (PDUStruct);    /* merge channels confirm */
942     SIGNAL  PCin    (PDUStruct);    /* purge channels indication */
943     SIGNAL  MTrq    (PDUStruct);    /* merge tokens request */
944     SIGNAL  MTcf    (PDUStruct);    /* merge tokens confirm */
945     SIGNAL  PTin    (PDUStruct);    /* purge tokens indication */
946     SIGNAL  DPum    (PDUStruct);    /* disconnect provider ultimatum */
947     SIGNAL  RJum    (PDUStruct);    /* reject MCSPDU ultimatum */
948     SIGNAL  AUrq    (PDUStruct);    /* attach user request */
```

```
949     SIGNAL  AUcf    (PDUStruct);    /* attach user confirm */
950     SIGNAL  DUrq    (PDUStruct);    /* detach user request */
951     SIGNAL  DUin    (PDUStruct);    /* detach user confirm */
952     SIGNAL  CJrq    (PDUStruct);    /* channel join request */
953     SIGNAL  CJcf    (PDUStruct);    /* channel join confirm */
954     SIGNAL  CLrq    (PDUStruct);    /* channel leave request */
955     SIGNAL  CCrq    (PDUStruct);    /* channel convene request */
956     SIGNAL  CCcf    (PDUStruct);    /* channel convene confirm */
957     SIGNAL  CDrq    (PDUStruct);    /* channel disband request */
958     SIGNAL  CDin    (PDUStruct);    /* channel disband confirm */
959     SIGNAL  CArq    (PDUStruct);    /* channel admit request */
960     SIGNAL  CAin    (PDUStruct);    /* channel admit confirm */
961     SIGNAL  CErq    (PDUStruct);    /* channel expel request */
962     SIGNAL  CEin    (PDUStruct);    /* channel expel confirm */
963     SIGNAL  SDrq    (PDUStruct);    /* send data request */
964     SIGNAL  SDin    (PDUStruct);    /* send data indication */
965     SIGNAL  USrq    (PDUStruct);    /* uniform send data request */
966     SIGNAL  USin    (PDUStruct);    /* uniform send data indication */
967     SIGNAL  TGrq    (PDUStruct);    /* token grab request */
968     SIGNAL  TGcf    (PDUStruct);    /* token grab confirm */
969     SIGNAL  TIrq    (PDUStruct);    /* token inhibit request */
970     SIGNAL  TIcf    (PDUStruct);    /* token inhibit confirm */
971     SIGNAL  TVrq    (PDUStruct);    /* token give request */
972     SIGNAL  TVin    (PDUStruct);    /* token give indication */
973     SIGNAL  TVrs    (PDUStruct);    /* token give response */
974     SIGNAL  TVcf    (PDUStruct);    /* token give confirm */
975     SIGNAL  TPrq    (PDUStruct);    /* token please request */
976     SIGNAL  TPin    (PDUStruct);    /* token please indication */
977     SIGNAL  TRrq    (PDUStruct);    /* token release request */
978     SIGNAL  TRcf    (PDUStruct);    /* token release confirm */
979     SIGNAL  TTrq    (PDUStruct);    /* token test request */
980     SIGNAL  TTcf    (PDUStruct);    /* token test confirm */

982     ENDBLOCK;
```

Appendix III
SDL specification of the Control process

(This appendix does not form an integral part of this Recommendation.)

```
 1    PROCESS  Control;

 3            /* Type definitions */

 5    NEWTYPE         Proc
 6    LITERALS
 7            Nil,
 8            Receiving,              /*            Endpoint */
 9            Responding,             /*            Endpoint */
10            Engaged,                /* Attachment Endpoint */
11            Quitting,               /* Attachment Endpoint */
12            Quit;                   /* Attachment Endpoint */
13    ENDNEWTYPE;

15    NEWTYPE         ProcByPri       Array(DataPriority, Proc);
16    ENDNEWTYPE;

18    NEWTYPE         CallSide
19    LITERALS
20            Calling,
21            Called;
22    ENDNEWTYPE;

24    NEWTYPE         PortalStruct
25    STRUCT
26            mcId            MCSConnectionId;        /* equals PortalId index */
27            ccId            Natural;                /* equals PortalId index */
28 |          kind            PortalKind;             /* (Attached,Downlink,Uplink) */
29            pids            PIdByPri;               /* processes comprising a portal */
30            proc            ProcByPri;              /* state of each created process */
31 +          label           Natural;                /* requester's label for confirm */
32            domain          DomainSelector;         /* domain selected by the portal */
33            opened          Boolean;                /* True if portal has been opened */
34            notify          Boolean;                /* True to notify when portal quit */
35            minParms        DomainParameters;       /* lower limit for negotiation */
36            maxParms        DomainParameters;       /* upper limit for negotiation */
37            parameters      DomainParameters;       /* values negotiated by portal */
38 *          callSide        CallSide;               /* portal is calling or called */
39            localTSAP       TSAPAddress;            /* local address for T.Connect */
40            remoteTSAP      TSAPAddress;            /* remote address for T.Connect */
41 |          targetQOSByPri  TransportQOSByPri;      /* desired quality of service */
42 |          minQOSByPri     TransportQOSByPri;      /* minimum that is acceptable */
43            userData        UserData;               /* of response, pending confirm */
44    ENDNEWTYPE;
45            /* Note:  In a practical implementation, the user data stored from
46            Connect.Response, awaiting establishment of additional TCs, need be
47            only one transport interface data unit, not a complete TSDU.  Any
48            excess can be left in the pipeline of the initial TC to be read out
49            when MCS.Connect.Provider.confirm is issued. */

51    NEWTYPE         Portal          Array(PortalId, PortalStruct);
52    ENDNEWTYPE;

54    NEWTYPE         DomainStruct
55    STRUCT
56            pid             PId;                    /* process for the domain or Null */
57            portals         Natural;                /* number of portals open to domain */
58            upward          portalId;               /* unique connection upward or zero */
59            minParms        DomainParameters;       /* lower limit of configuration */
60            maxParms        DomainParameters;       /* upper limit of configuration */
61            parameters      DomainParameters;       /* values established in domain */
62    ENDNEWTYPE;

64    NEWTYPE         Domain          Array(DomainSelector, DomainStruct);
65    ENDNEWTYPE;

67    NEWTYPE         DomainSelectorSet       SetOf(DomainSelector);
68    ENDNEWTYPE;

70            /* Data declarations */

72    DCL     domain          Domain,                 /* resource arrays */
```

```
 73            portal          Portal;

 75            /* Note: The fields of a domain or portal array element
 76            are undefined if the corresponding index is not in dUsed
 77            or pUsed respectively. */

 79    DCL     dUsed           DomainSelectorSet,      /* indexes used */
 80            pUsed           PortalIdSet;

 82    DCL     pFree           PortalIdSet;            /* indexes free */

 84    DCL     nullParms       DomainParameters;       /* initializer */

 86            /* Procedure decomposition */

 88            /*      Initialize_resources
 89                    Identify_sender                 (p, dp)
 90                    Min_parms                       (min, a, b)
 91                    Max_parms                       (max, a, b)
 92                    Test_parms                      (result, x, min, max)
 93                    MCS_Connect_Provider_request    (...)
 94                    T_Connect_indication            (...)
 95                    Connect_Initial                 (...)
 96                    MCS_Connect_Provider_response   (...)
 97                    Connect_Response                (...)
 98                    Connect_Additional              (...)
 99                    Connect_Result                  (...)
100                    MCS_Disconnect_Provider_request (...)
101                    MCS_Attach_User_request         (...)
102                    Open_portal                     (p)
103                    Drop_portal                     (p, reason)
104                    Report_portal                   (p, diagnostic)
105                    RJum                            (pdu)
106                    Quit_portal                     (p, reason)
107                    Quit
108                    Shut_portal                     (p)
109                    Exit_portal                     (p)     */

111                                                            /*-------------------------*/
112    PROCEDURE       Initialize_resources;                   /*  Initialize_resources  */
113                                                            /*-------------------------*/
114            DCL     p               PortalId,
115                    ds              DomainSelector,
116                    dSet            DomainSelectorSet;
117            START
118            COMMENT'Initialize data structures during process start-up
119                    before accepting the first input signal.
120                    Note that each SetOf automatically defaults to Empty.
121                    Configure one hypothetical domain as an example.
122                    Some limits are determined by the implementation.
123                    ';
124            TASK    nullParms!maxChannelIds := 0,
125                    nullParms!maxUserIds := 0,
126                    nullParms!maxTokenIds := 0,
127                    nullParms!numPriorities := 0,
128                    nullParms!minThroughput := 0,
129                    nullParms!maxHeight := 0,
130                    nullParms!maxMCSPDUsize := 0,
131                    nullParms!protocolVersion := 0,
132                    p := maxPortalIds;
133             1b :   /* for p = ?..1 */
134            DECISION p > 0;
135            (True): TASK    portal(p)!mcId := p,
136                            portal(p)!ccId := p,
137                            pFree := Incl(p, pFree),
138                            p := p - 1;
139                    JOIN 1b;
140            ELSE:ENDDECISION;
141 |          TASK    ds := Mkstring(77) // Mkstring(67) // Mkstring(83),
142                    dUsed := Incl(ds, dUsed),
143                    domain(ds)!Pid := Null,
144                    domain(ds)!portals := 0,
145 *                  domain(ds)!upward := 0,
146                    dSet := dUsed;
147             2b :   /* for ds in dSet */
148            DECISION dSet = Empty;
149            (False):TASK    ds := Pick(dSet),
150                            dSet := Del(ds, dSet),
151 +                          domain(ds)!minParms!maxChannelIds := 0,
```

```
152 +                           domain(ds)!minParms!maxUserIds := 0,
153 +                           domain(ds)!minParms!maxTokenIds := 0,
154                             domain(ds)!minParms!numPriorities := 1,
155 +                           domain(ds)!minParms!minThroughput := 0,
156                             domain(ds)!minParms!maxHeight := 1,
157 |                           domain(ds)!minParms!maxMCSPDUsize := 35,
158                             domain(ds)!minParms!protocolVersion := 1,
159 +                           domain(ds)!minParms!maxChannelIds := 65535,
160 +                           domain(ds)!minParms!maxUserIds := 65535,
161 +                           domain(ds)!minParms!maxTokenIds := 65535,
162                             domain(ds)!maxParms!numPriorities := 4,
163 +                           domain(ds)!minParms!minThroughput := 1000000,
164 +                           domain(ds)!minParms!maxHeight := 1000,
165 +                           domain(ds)!minParms!maxMCSPDUsize := 32768,
166                             domain(ds)!maxParms!protocolVersion := 2;
167                     JOIN 2b;
168             ELSE:ENDDECISION;
169             RETURN;
170             ENDPROCEDURE;

172                                                                     /*-------------------*/
173     PROCEDURE       Identify_sender;                              /*  Identify_sender  */
174     FPAR    IN/OUT  p               PortalId,                     /*-------------------*/
175             IN/OUT  dp              DataPriority;

177             DCL     pSet            PortalIdSet;
178             START
179             COMMENT'An alternative would be to carry this
180                     information explicitly in SDL signals.
181                     ';
182             TASK    pSet := pUsed;
183              1b :   /* for p in pSet */
184             DECISION pSet = Empty;
185             (False):TASK    p := Pick(pSet),
186                             pSet := Del(p, pSet),
187                             dp := 0;
188                      2b :   /* for dp = 0..3 */
189                     DECISION dp < 4;
190                     (True): DECISION portal(p)!pids(dp) = SENDER;
191                             (True): RETURN;
192                             ELSE:ENDDECISION;
193                             TASK    dp := dp + 1;
194                             JOIN 2b;
195                     ELSE:ENDDECISION;
196                     JOIN 1b;
197             ELSE:ENDDECISION;
198             TASK    p := 0,
199                     dp := 0;
200             RETURN;
201             ENDPROCEDURE;

203                                                                     /*-------------*/
204     PROCEDURE       Min_parms;                                    /*  Min_parms  */
205     FPAR    IN/OUT  min             DomainParameters,             /*-------------*/
206                     a               DomainParameters,
207                     b               DomainParameters;

209             START
210 |           COMMENT'Return the minimum of two parameter sets.
211                     ';
212             TASK    min!maxChannelIds := IF a!maxChannelIds < b!maxChannelIds
213 |                                   THEN a!maxChannelIds ELSE b!maxChannelIds FI,
214                     min!maxUserIds := IF a!maxUserIds < b!maxUserIds
215 |                                   THEN a!maxUserIds ELSE b!maxUserIds FI,
216                     min!maxTokenIds := IF a!maxTokenIds < b!maxTokenIds
217 |                                   THEN a!maxTokenIds ELSE b!maxTokenIds FI,
218                     min!numPriorities := IF a!numPriorities < b!numPriorities
219 |                                   THEN a!numPriorities ELSE b!numPriorities FI,
220                     min!minThroughput := IF a!minThroughput < b!minThroughput
221 |                                   THEN a!minThroughput ELSE b!minThroughput FI,
222                     min!maxHeight := IF a!maxHeight < b!maxHeight
223 |                                   THEN a!maxHeight ELSE b!maxHeight FI,
224                     min!maxMCSPDUsize := IF a!maxMCSPDUsize < b!maxMCSPDUsize
225 |                                   THEN a!maxMCSPDUsize ELSE b!maxMCSPDUsize FI,
226                     min!protocolVersion := IF a!protocolVersion < b!protocolVersion
227 |                                   THEN a!protocolVersion ELSE b!protocolVersion FI;
228             RETURN;
229             ENDPROCEDURE;
```

```
                                                                    /*-------------*/
PROCEDURE       Max_parms;                                          /*  Max_parms  */
FPAR    IN/OUT  max             DomainParameters,                   /*-------------*/
                a               DomainParameters,
                b               DomainParameters;

        START
|       COMMENT'Return the maximum of two parameter sets.
                ';
        TASK    max!maxChannelIds := IF a!maxChannelIds > b!maxChannelIds
                                  THEN a!maxChannelIds ELSE b!maxChannelIds FI,
                max!maxUserIds := IF a!maxUserIds > b!maxUserIds
                                  THEN a!maxUserIds ELSE b!maxUserIds FI,
                max!maxTokenIds := IF a!maxTokenIds > b!maxTokenIds
                                  THEN a!maxTokenIds ELSE b!maxTokenIds FI,
                max!numPriorities := IF a!numPriorities > b!numPriorities
                                  THEN a!numPriorities ELSE b!numPriorities FI,
                max!minThroughput := IF a!minThroughput > b!minThroughput
                                  THEN a!minThroughput ELSE b!minThroughput FI,
                max!maxHeight := IF a!maxHeight > b!maxHeight
                                  THEN a!maxHeight ELSE b!maxHeight FI,
                max!maxMCSPDUsize := IF a!maxMCSPDUsize > b!maxMCSPDUsize
                                  THEN a!maxMCSPDUsize ELSE b!maxMCSPDUsize FI,
                max!protocolVersion := IF a!protocolVersion > b!protocolVersion
                                  THEN a!protocolVersion ELSE b!protocolVersion FI;
        RETURN;
        ENDPROCEDURE;

                                                                    /*--------------*/
PROCEDURE       Test_parms;                                         /*  Test_parms  */
FPAR    IN/OUT  result          Result,                             /*--------------*/
                z               DomainParameters,
                min             DomainParameters,
                max             DomainParameters;

        START
|       COMMENT'Check that the parameters lie between min and max.
                ';
|       DECISION (z!maxChannelIds >= min!maxChannelIds)
|            and (z!maxChannelIds <= max!maxChannelIds)
|            and (z!maxUserIds >= min!maxUserIds)
|            and (z!maxUserIds <= max!maxUserIds)
|            and (z!maxTokenIds >= min!maxTokenIds)
|            and (z!maxTokenIds <= max!maxTokenIds)
|            and (z!numPriorities >= min!numPriorities)
|            and (z!numPriorities <= max!numPriorities)
|            and (z!minThroughput >= min!minThroughput)
|            and (z!minThroughput <= max!minThroughput)
|            and (z!maxHeight >= min!maxHeight)
|            and (z!maxHeight <= max!maxHeight)
|            and (z!maxMCSPDUsize >= min!maxMCSPDUsize)
|            and (z!maxMCSPDUsize <= max!maxMCSPDUsize)
|            and (z!protocolVersion >= min!protocolVersion)
|            and (z!protocolVersion <= max!protocolVersion);
        (True): TASK    result := RT_successful;
        (False):TASK    result := RT_parameters_unacceptable;
        ENDDECISION;
        RETURN;
        ENDPROCEDURE;

                                                          /*--------------------------------*/
PROCEDURE       MCS_Connect_Provider_request;             /*  MCS_Connect_Provider_request  */
FPAR            label           Natural,                  /*--------------------------------*/
                localTSAP       TSAPAddress,
                localDomain     DomainSelector,
                remoteTSAP      TSAPAddress,
                remoteDomain    DomainSelector,
                upward          Boolean,
                targetParms     DomainParameters,
                minParms        DomainParameters,
                maxParms        DomainParameters,
|               targetQOSByPri  TransportQOSByPri,
|               minQOSByPri     TransportQOSByPri,
                userData        UserData;

        DCL     result          Result,
                p               PortalId,
                dp              DataPriority,
                ds              DomainSelector;
```

```
310             START
311             COMMENT'Process an MCS.Connect.Provider.request input signal.
312                     Begin parameter negotiation and allocate a portal.
313                     Create an endpoint process for the initial TC and
314                     transmit Connect.Initial through it.
315                     ';
316             DECISION pFree = Empty;
317             (True): TASK    result := RT_congested;
318                     JOIN 1f;
319             ELSE:ENDDECISION;
320             TASK    ds := localDomain;
321             DECISION ds in dUsed;
322             (False):TASK    result := RT_no_such_domain;
323                     JOIN 1f;
324             ELSE:ENDDECISION;
325             DECISION upward and domain(ds)!upward /= 0;
326             (True): TASK    result := RT_domain_not_hierarchical;
327                     JOIN 1f;
328             ELSE:ENDDECISION;
329             CALL    Max_parms(minParms, minParms, domain(ds)!minParms);
330             CALL    Min_parms(maxParms, maxParms, domain(ds)!maxParms);
331             DECISION domain(ds)!portals > 0;
332             (True): TASK    targetParms := domain(ds)!parameters;
333             (False):CALL    Max_parms(targetParms, targetParms, minParms);
334                     CALL    Min_parms(targetParms, targetParms, maxParms);
335             ENDDECISION;
336             CALL    Test_parms(result, targetParms, minParms, maxParms);
337             DECISION result = RT_successful;
338             (False): 1f :
339                     OUTPUT  MCS.Connect.Provider.confirm
340                             (label, result, 0, nullParms, NullString);
341                     RETURN;
342             ELSE:ENDDECISION;
343             DECISION domain(ds)!portals > 0;
344             (True): TASK    minParms := targetParms,
345                             maxParms := targetParms;
346             ELSE:ENDDECISION;
347             CREATE  Endpoint(Null, localTSAP, remoteTSAP,
348 |                   targetQOSByPri(0), minQOSByPri(0),
349 |                   nullParms);
350             OUTPUT  Connect.Initial(localDomain, remoteDomain, upward,
351                     targetParms, minParms, maxParms, userData)
352                     TO OFFSPRING;
353             TASK    p := Pick(pFree),
354                     pFree := Del(p, pFree),
355                     pUsed := Incl(p, pUsed),
356 |                   portal(p)!kind := IF upward THEN Uplink ELSE Downlink FI,
357 |                   portal(p)!label := label,
358                     portal(p)!domain := ds,
359                     portal(p)!opened := False,
360                     portal(p)!notify := True,
361                     portal(p)!minParms := minParms,
362                     portal(p)!maxParms := maxParms,
363                     portal(p)!parameters := nullParms,
364 *                   portal(p)!callSide := Calling,
365                     portal(p)!localTSAP := localTSAP,
366                     portal(p)!remoteTSAP := remoteTSAP,
367 |                   portal(p)!targetQOSByPri := targetQOSByPri,
368 |                   portal(p)!minQOSByPri := minQOSByPri,
369                     portal(p)!pids(0) := OFFSPRING,
370                     portal(p)!proc(0) := Receiving,
371                     dp := 1;
372              2b :   /* for dp = 1..3 */
373             DECISION dp < 4;
374             (True): TASK    portal(p)!pids(dp) := Null,
375                             portal(p)!proc(dp) := Nil,
376                             dp := dp + 1;
377                     JOIN 2b;
378             ELSE:ENDDECISION;
379             RETURN;
380             ENDPROCEDURE;

382                                                             /*-------------------------*/
383     PROCEDURE       T_Connect_indication;                   /*  T_Connect_indication   */
384     FPAR            tcId            TCEndpointId,           /*-------------------------*/
385                     remoteTSAP      TSAPAddress,
386                     localTSAP       TSAPAddress,
387                     offeredQOS      TransportQOS,
388                     minQOS          TransportQOS;
```

```
390             DCL     p               PortalId,
391                     dp              DataPriority;
392             START
393             COMMENT'Process a T.Connect.indication input signal.
394                     Allocate a portal, in case this is an initial TC.
395                     Create an endpoint process to receive Connect.Initial
396                     or Connect.Additional.
397                     ';
398             DECISION pFree = Empty
399                     or offeredQOS!throughput < minQOS!throughput
400                     or offeredQOS!transitDelay > minQOS!transitDelay
401                     or offeredQOS!dataPriority > minQOS!dataPriority;
402             (True): OUTPUT  T.Disconnect.request(tcId);
403                     RETURN;
404             ELSE:ENDDECISION;
405             CREATE  Endpoint(tcId, localTSAP, remoteTSAP,
406 |                   offeredQOS, minQOS,
407 |                   nullParms);
408             TASK    p := Pick(pFree),
409                     pFree := Del(p, pFree),
410                     pUsed := Incl(p, pUsed),
411 |                   portal(p)!kind := Downlink,
412                     portal(p)!opened := False,
413                     portal(p)!notify := False,
414                     portal(p)!parameters := nullParms,
415                     portal(p)!callSide := Called,
416                     portal(p)!localTSAP := localTSAP,
417                     portal(p)!remoteTSAP := remoteTSAP,
418                     portal(p)!pids(0) := OFFSPRING,
419                     portal(p)!proc(0) := Receiving,
420                     dp := 1;
421              1b :   /* for dp = 1..3 */
422             DECISION dp < 4;
423             (True): TASK    portal(p)!pids(dp) := Null,
424                             portal(p)!proc(dp) := Nil,
425                             dp := dp + 1;
426                     JOIN 1b;
427             ELSE:ENDDECISION;
428             RETURN;
429             ENDPROCEDURE;

431                                                                     /*-------------------*/
432     PROCEDURE       Connect_Initial;                                /*  Connect_Initial  */
433     FPAR            remoteDomain    DomainSelector,                 /*-------------------*/
434                     localDomain     DomainSelector,
435 |                   upward          Boolean,
436                     targetParms     DomainParameters,
437                     minParms        DomainParameters,
438                     maxParms        DomainParameters,
439                     userData        UserData;

441             DCL     result          Result,
442                     p               PortalId,
443                     dp              DataPriority,
444                     ds              DomainSelector;
445             START
446             COMMENT'Process a Connect.Initial input signal.
447                     Retain the portal and begin parameter negotiation.
448                     Indicate the connection to the controlling user.
449                     ';
450             CALL    Identify_sender(p, dp);
451             DECISION p in pUsed and portal(p)!callSide = Called
452                     and dp = 0 and portal(p)!proc(0) = Receiving;
453             (False):TASK    result := RT_unspecified_failure;
454                     JOIN 1f;
455             ELSE:ENDDECISION;
456             TASK    ds := localDomain;
457             DECISION ds in dUsed;
458             (False):TASK    result := RT_no_such_domain;
459                     JOIN 1f;
460             ELSE:ENDDECISION;
461 |           DECISION upward or domain(ds)!upward = 0;
462             (False): TASK   result := RT_domain_not_hierarchical;
463                     JOIN 1f;
464             ELSE:ENDDECISION;
465             CALL    Max_parms(minParms, minParms, domain(ds)!minParms);
466             CALL    Min_parms(maxParms, maxParms, domain(ds)!maxParms);
467             DECISION domain(ds)!portals > 0;
```

```
468             (True): TASK    targetParms := domain(ds)!parameters;
469             (False):CALL    Max_parms(targetParms, targetParms, minParms);
470                     CALL    Min_parms(targetParms, targetParms, maxParms);
471             ENDDECISION;
472             CALL    Test_parms(result, targetParms, minParms, maxParms);
473             DECISION result = RT_successful;
474             (False): 1f :
475                     OUTPUT  Connect.Response(result, 0, nullParms, NullString)
476                             TO SENDER;
477                     CALL    Quit_portal(p, RN_unspecified);
478                     RETURN;
479             ELSE:ENDDECISION;
480             DECISION domain(ds)!portals > 0;
481             (True): TASK    minParms := targetParms,
482                             maxParms := targetParms;
483             ELSE:ENDDECISION;
484 |           TASK    portal(p)!kind := IF upward THEN Downlink ELSE Uplink FI,
485                     portal(p)!domain := ds,
486                     portal(p)!notify := True,
487                     portal(p)!minParms := minParms,
488                     portal(p)!maxParms := maxParms,
489                     portal(p)!proc(0) := Responding;
490             OUTPUT  MCS.Connect.Provider.indication
491                     (portal(p)!mcId, portal(p)!localTSAP, localDomain, portal(p)!remoteTSAP,
492 |                   remoteDomain, upward, targetParms, minParms, maxParms, userData);
493             RETURN;
494             ENDPROCEDURE;

496                                                     /*---------------------------------*/
497     PROCEDURE       MCS_Connect_Provider_response;  /*  MCS_Connect_Provider_response  */
498     FPAR            mcId            MCSConnectionId,    /*---------------------------------*/
499                     result          Result,
500                     parameters      DomainParameters,
501                     userData        UserData;

503             DCL     p               PortalId;
504             START
505             COMMENT'Process an MCS.Connect.Provider.response input signal.
506                     Check negotiated parameters and force a preference.
507                     Transmit Connect.Response.
508                     If there are no additional TCs, open the portal.
509                     ';
510             TASK    p := mcId;
511             DECISION p in pUsed and portal(p)!proc(0) = Responding;
512             (False):RETURN;
513             ELSE:ENDDECISION;
514             DECISION result = RT_successful;
515             (False):TASK    result := RT_user_rejected,
516                             portal(p)!notify := False;
517                     JOIN 1f;
518             ELSE:ENDDECISION;
519             CALL    Test_parms(result, parameters, portal(p)!minParms, portal(p)!maxParms);
520             DECISION result = RT_successful;
521             (False): 1f :
522                     OUTPUT  Connect.Response(result, 0, parameters, userData)
523                             TO portal(p)!pids(0);
524                     CALL    Quit_portal(p, RN_parameters_unacceptable);
525                     RETURN;
526 *           ELSE:ENDDECISION;
527             CALL    Max_parms(parameters, parameters, portal(p)!minParms);
528             OUTPUT  Connect.Response(result, portal(p)!ccId, parameters, userData)
529                     TO portal(p)!pids(0);
530             TASK    portal(p)!parameters := parameters,
531                     portal(p)!proc(0) := Engaged;
532             CALL    Open_portal(p);
533             RETURN;
534             ENDPROCEDURE;

536                                                             /*--------------------*/
537     PROCEDURE       Connect_Response;                       /*  Connect_Response  */
538     FPAR            result          Result,                 /*--------------------*/
539                     ccId            Natural,
540                     parameters      DomainParameters,
541                     userData        UserData;

543             DCL     p               PortalId,
544                     dp              DataPriority;
545             START
546             COMMENT'Process a Connect.Response input signal.
```

```
547                     Check the negotiated parameters.
548                     Create endpoint processes for additional TCs and
549                     transmit Connect.Additional through them.
550                     If there are no additional TCs, open the portal.
551                     ';
552             CALL    Identify_sender(p, dp);
553             DECISION p in pUsed and portal(p)!callSide = Calling
554                     and dp = 0 and portal(p)!proc(0) = Receiving;
555             (False):CALL    Quit_portal(p, RN_unspecified);
556                     RETURN;
557             ELSE:ENDDECISION;
558             TASK    portal(p)!parameters := parameters,
559                     portal(p)!userData := userData;
560             DECISION result = RT_successful;
561             (False):JOIN 1f;
562 *           ELSE:ENDDECISION;
563             CALL    Test_parms(result, parameters, portal(p)!minParms, portal(p)!maxParms);
564             DECISION result = RT_successful;
565             (False): 1f :
566                     OUTPUT  MCS.Connect.Provider.confirm
567                             (portal(p)!label, result, 0, parameters, userData);
568                     TASK    portal(p)!notify := False;
569                     CALL    Quit_portal(p, RN_unspecified);
570                     RETURN;
571             ELSE:ENDDECISION;
572             TASK    portal(p)!proc(0) := Engaged,
573                     dp := 1;
574              2b :   /* for dp = 1..? */
575             DECISION dp < parameters!numPriorities;
576             (True): CREATE  Endpoint(Null, portal(p)!localTSAP, portal(p)!remoteTSAP,
577 |                           portal(p)!targetQOSByPri(dp), portal(p)!minQOSByPri(dp),
578 |                           parameters);
579                     OUTPUT  Connect.Additional(ccId, dp)
580                             TO OFFSPRING;
581                     TASK    portal(p)!pids(dp) := OFFSPRING,
582                             portal(p)!proc(dp) := Receiving,
583                             dp := dp + 1;
584                     JOIN 2b;
585             ELSE:ENDDECISION;
586             CALL    Open_portal(p);
587             RETURN;
588             ENDPROCEDURE;

590                                                                     /*----------------------*/
591     PROCEDURE       Connect_Additional;                             /*  Connect_Additional  */
592     FPAR            ccId            Natural,                        /*----------------------*/
593                     dp              DataPriority;

595             DCL     p               PortalId,
596                     r               PortalId,
597                     x               DataPriority;
598             START
599             COMMENT'Process a Connect.Additional input signal.
600                     Release the allocated portal and piggyback onto
601                     the preceding Connect.Initial.
602                     Transmit Connect.Result.
603                     If all TCs are established, open the portal.
604                     ';
605             CALL    Identify_sender(r, x);
606             DECISION r in pUsed and portal(r)!callSide = Called
607                     and x = 0 and portal(r)!proc(0) = Receiving;
608             (False):JOIN 1f;
609             ELSE:ENDDECISION;
610             TASK    p := ccId;
611             DECISION p in pUsed and portal(p)!callSide = Called
612                     and dp > 0 and dp < portal(p)!parameters!numPriorities
613                     and portal(p)!proc(0) = Engaged and portal(p)!proc(dp) = Nil;
614             (False): 1f :
615                     OUTPUT  Connect.Result(RT_unspecified_failure)
616                             TO SENDER;
617                     CALL    Quit_portal(r, RN_unspecified);
618                     RETURN;
619             ELSE:ENDDECISION;
620             TASK    pUsed := Del(r, pUsed),
621                     pFree := Incl(r, pFree);
622             OUTPUT  Connect.Result(RT_successful)
623                     TO SENDER;
624             TASK    portal(p)!pids(dp) := SENDER,
625                     portal(p)!proc(dp) := Engaged;
```

```
626             CALL    Open_portal(p);
627             RETURN;
628             ENDPROCEDURE;

630                                                                     /*------------------*/
631     PROCEDURE       Connect_Result;                                 /*  Connect_Result  */
632     FPAR            result          Result;                         /*------------------*/

634             DCL     p               PortalId,
635                     dp              DataPriority;
636             START
637             COMMENT'Process a Connect.Result input signal.
638                     If all TCs are established, open the portal.
639                     ';
640             CALL    Identify_sender(p, dp);
641             DECISION p in pUsed and portal(p)!callSide = Calling
642                     and dp > 0 and portal(p)!proc(dp) = Receiving;
643             (False):CALL    Quit_portal(p, RN_unspecified);
644                     RETURN;
645             ELSE:ENDDECISION;
646             DECISION result = RT_successful;
647             (False):OUTPUT  MCS.Connect.Provider.confirm
648                             (portal(p)!label, result, 0,
649                             portal(p)!parameters, portal(p)!userData);
650                     TASK    portal(p)!notify := False;
651                     CALL    Quit_portal(p, RN_unspecified);
652                     RETURN;
653             ELSE:ENDDECISION;
654             TASK    portal(p)!proc(dp) := Engaged;
655             CALL    Open_portal(p);
656             RETURN;
657             ENDPROCEDURE;

659                                                 /*-----------------------------------*/
660     PROCEDURE       MCS_Disconnect_Provider_request;  /*  MCS_Disconnect_Provider_request  */
661     FPAR            mcId            MCSConnectionId;  /*-----------------------------------*/

663             DCL     p               PortalId,
664                     ds              DomainSelector;
665             START
666             COMMENT'Process an MCS.Disconnect.Provider.request input signal.
667                     If the portal is open, this can be done gracefully.
668                     ';
669             TASK    p := mcId;
670             DECISION p in pUsed and portal(p)!notify;
671             (True): TASK    portal(p)!notify := False,
672                             ds := portal(p)!domain;
673                     DECISION portal(p)!opened and domain(ds)!portals > 0;
674                     (True): OUTPUT  Drop.portal(p, RN_user_requested) TO domain(ds)!pid;
675                     (False):CALL    Quit_portal(p, RN_unspecified);
676                     ENDDECISION;
677             ELSE:ENDDECISION;
678             RETURN;
679             ENDPROCEDURE;

681                                                             /*--------------------------*/
682     PROCEDURE       MCS_Attach_User_request;                /*  MCS_Attach_User_request  */
683 |   FPAR            label           Natural,                /*--------------------------*/
684 |                   localDomain     DomainSelector;

686             DCL     p               PortalId,
687                     dp              DataPriority,
688                     ds              DomainSelector;
689             START
690             COMMENT'Process an MCS.Attach.User.request input signal.
691                     Allocate a portal, expecting to create an attachment,
692                     and open the portal.
693                     ';
694             DECISION pFree = Empty;
695             (True): TASK    result := RT_congested;
696                     JOIN 1f;
697             ELSE:ENDDECISION;
698             TASK    ds := localDomain;
699             DECISION ds in dUsed;
700             (False):TASK    result := RT_no_such_domain;
701                      1f :
702                     OUTPUT  MCS.Attach.User.confirm
703 |                           (label, result, Null, 0);
704                     RETURN;
```

```
705             ELSE:ENDDECISION;
706             TASK    p := Pick(pFree),
707                     pFree := Del(p, pFree),
708                     pUsed := Incl(p, pUsed),
709                     portal(p)!kind := Attached,
710 +                   portal(p)!label := label,
711                     portal(p)!domain := ds,
712                     portal(p)!opened := False,
713                     portal(p)!notify := False,
714                     portal(p)!pids(0) := Null,
715                     portal(p)!proc(0) := Engaged,
716                     dp := 1;
717              2b :   /* for dp = 1..3 */
718             DECISION dp < 4;
719             (True): TASK    portal(p)!pids(dp) := Null,
720                             portal(p)!proc(dp) := Nil,
721                             dp := dp + 1;
722                     JOIN 2b;
723             ELSE:ENDDECISION;
724             CALL    Open_portal(p);
725             RETURN;
726             ENDPROCEDURE;

728                                                                     /*---------------*/
729     PROCEDURE       Open_portal;                                    /*  Open_portal  */
730     FPAR            p               PortalId;                       /*---------------*/

732             DCL     dp              DataPriority,
733                     ds              DomainSelector,
734                     numP            Natural,
735                     parameters      DomainParameters;
736             START
737             COMMENT'When all TCs have been established, or if this
738                     is a user attachment, open the portal to a domain.
739                     If the domain process is stopping, try again later.
740                     Attachments must wait until domain parameters are set.
741                     ';
742             TASK    ds := portal(p)!domain,
743                     dp := 0;
744             DECISION portal(p)!kind = Attached;
745             (True): TASK    parameters := domain(ds)!minParms,
746                             numP := 1;
747             (False):TASK    parameters := portal(p)!parameters,
748                             numP := parameters!numPriorities;
749             ENDDECISION;
750              1b :   /* for dp = 0..? */
751             DECISION dp < numP;
752             (True): DECISION portal(p)!proc(dp) = Engaged;
753                     (False):RETURN;
754                     ELSE:ENDDECISION;
755                     TASK    dp := dp + 1;
756                     JOIN 1b;
757             ELSE:ENDDECISION;
758             DECISION domain(ds)!pid = Null;
759             (False):DECISION domain(ds)!portals = 0;
760                     (True): RETURN;
761                     ELSE:ENDDECISION;
762             (True): CREATE  Domain(parameters);
763                     TASK    domain(ds)!pid := OFFSPRING,
764                             domain(ds)!portals := 0,
765                             domain(ds)!upward := 0,
766                             domain(ds)!parameters := parameters;
767             ENDDECISION;
768             DECISION portal(p)!kind = Attached;
769 |           (True): CREATE  Attachment(portal(p)!label, domain(ds)!parameters);
770                     TASK    portal(p)!pids(0) := OFFSPRING;
771             (False):DECISION domain(ds)!parameters = parameters;
772                     (False):CALL    Quit_portal(p, RN_parameters_unacceptable);
773                             RETURN;
774                     ELSE:ENDDECISION;
775 |                   DECISION portal(p)!kind = Uplink;
776                     (True): DECISION domain(ds)!upward = 0;
777                             (False):CALL    Quit_portal(p, RN_domain_not_hierarchical);
778                                     RETURN;
779                             (True): TASK    domain(ds)!upward := p;
780                             ENDDECISION;
781                     ELSE:ENDDECISION;
782                     DECISION portal(p)!callSide = Called;
783                     (False):OUTPUT  MCS.Connect.Provider.confirm
```

```
784                                 (portal(p)!label, RT_successful, portal(p)!mcId,
785                                 portal(p)!parameters, portal(p)!userData);
786                 ELSE:ENDDECISION;
787         ENDDECISION;
788         OUTPUT  Open.portal(p, portal(p)!kind, portal(p)!pids) TO domain(ds)!pid;
789         TASK    domain(ds)!portals := domain(ds)!portals + 1,
790                 portal(p)!opened := True;
791         RETURN;
792         ENDPROCEDURE;

794                                                                     /*---------------*/
795 PROCEDURE       Drop_portal;                                        /*  Drop_portal  */
796 FPAR            p               PortalId,                           /*---------------*/
797                 reason          Reason;

799         START
800         COMMENT'Process a Drop.portal input signal.
801                 ';
802         CALL    Quit_portal(p, reason);
803         RETURN;
804         ENDPROCEDURE;

806                                                                     /*-----------------*/
807 PROCEDURE       Report_portal;                                      /*  Report_portal  */
808 FPAR            p               PortalId,                           /*-----------------*/
809                 diagnostic      Diagnostic;

811         DCL     reason          Reason;
812         START
813         COMMENT'Process a Report.portal input signal.
814                 For testing, local diagnostic could be logged.
815                 ';
816         TASK    reason := RN_unspecified;
817         DECISION diagnostic;
818         (DC_throughput_inadequate):
819                 TASK    reason := RN_provider_initiated;
820         (DC_height_limit_exceeded):
821                 TASK    reason := RN_domain_not_hierarchical;
822         ELSE:ENDDECISION;
823         CALL    Quit_portal(p, reason);
824         RETURN;
825         ENDPROCEDURE;

827                                                                     /*--------*/
828 PROCEDURE       RJum;                                               /*  RJum  */
829 FPAR            pdu             PDUStruct;                          /*--------*/

831         DCL     p               PortalId,
832                 dp              DataPriority;
833         START
834         COMMENT'Process an RJum input signal.
835                 For testing, remote pdu!diagnostic could be logged.
836                 ';
837         CALL    Identify_sender(p, dp);
838         CALL    Quit_portal(p, RN_unspecified);
839         RETURN;
840         ENDPROCEDURE;

842                                                                     /*---------------*/
843 PROCEDURE       Quit_portal;                                        /*  Quit_portal  */
844 FPAR            p               PortalId,                           /*---------------*/
845                 reason          Reason;

847         DCL     result          Result,
848                 dp              DataPriority;
849         START
850         COMMENT'If necessary, notify the controlling user.
851                 Quiesce the portal processes.
852                 ';
853         DECISION p in pUsed;
854         (False):RETURN;
855         ELSE:ENDDECISION;
856         DECISION portal(p)!notify;
857         (True): DECISION portal(p)!callSide = Called or portal(p)!opened;
858                 (True): OUTPUT  MCS.Disconnect.Provider.indication
859                                 (portal(p)!mcId, reason);
860                 (False):DECISION reason;
861                         (RN_domain_not_hierarchical):
862                                 TASK    result := RT_domain_not_hierarchical;
```

```
863                             (RN_parameters_unacceptable):
864                                     TASK    result := RT_parameters_unacceptable;
865                             ELSE:
866                                     TASK    result := RT_unspecified_failure;
867                             ENDDECISION;
868                             OUTPUT  MCS.Connect.Provider.confirm
869                                     (portal(p)!label, result, 0, nullParms, NullString);
870                     ENDDECISION;
871             ELSE:ENDDECISION;
872             TASK    portal(p)!notify := False,
873                     dp := 0;
874              1b :   /* for dp = 0..3 */
875             DECISION dp < 4;
876             (True): DECISION portal(p)!proc(dp);
877                     (Receiving, Responding, Engaged):
878                             OUTPUT  Quit TO portal(p)!pids(dp);
879                             TASK    portal(p)!proc(dp) := Quitting;
880                     ELSE:ENDDECISION;
881                     TASK    dp := dp + 1;
882                     JOIN 1b;
883             ELSE:ENDDECISION;
884             RETURN;
885             ENDPROCEDURE;

887                                                                     /*--------*/
888     PROCEDURE       Quit;                                           /*  Quit  */
889                                                                     /*--------*/
890             DCL     p               PortalId,
891                     pSet            PortalIdSet,
892                     dp              DataPriority,
893                     ds              DomainSelector;
894             START
895             COMMENT'Process a Quit input signal.
896                     When all processes are quiesced, it is safe
897                     to shut this portal on the domain.
898                     If an upward portal, quiesce all others too.
899                     ';
900             CALL    Identify_sender(p, dp);
901             DECISION p in pUsed;
902             (False):RETURN;
903             ELSE:ENDDECISION;
904             TASK    portal(p)!proc(dp) := Quit;
905             CALL    Quit_portal(p, RN_unspecified);
906             TASK    dp := 0;
907              1b :   /* for dp = 0..3 */
908             DECISION dp < 4;
909             (True): DECISION portal(p)!proc(dp);
910                     (Receiving, Responding, Engaged, Quitting):
911                             RETURN;
912                     ELSE:ENDDECISION;
913                     TASK    dp := dp + 1;
914                     JOIN 1b;
915             ELSE:ENDDECISION;
916             DECISION portal(p)!opened;
917             (False):CALL    Exit_portal(p);
918                     RETURN;
919             ELSE:ENDDECISION;
920             TASK    ds := portal(p)!domain,
921                     domain(ds)!portals := domain(ds)!portals - 1;
922             OUTPUT  Shut.portal(p) TO domain(ds)!pid;
923 |           DECISION portal(p)!kind = Uplink;
924             (True): TASK    domain(ds)!upward := 0,
925                             pSet := pUsed;
926                      2b :   /* for p in pSet */
927                     DECISION pSet = Empty;
928                     (False):TASK    p := Pick(pSet),
929                                     pSet := Del(p, pSet);
930                             DECISION portal(p)!opened and portal(p)!domain = ds;
931                             (True): CALL    Quit_portal(p, RN_domain_disconnected);
932                             ELSE:ENDDECISION;
933                             JOIN 2b;
934                     ELSE:ENDDECISION;
935             ELSE:ENDDECISION;
936             RETURN;
937             ENDPROCEDURE;

939                                                                     /*---------------*/
940     PROCEDURE       Shut_portal;                                    /*  Shut_portal  */
941     FPAR            p               PortalId;                       /*---------------*/
```

```
            DCL     ds              DomainSelector,
                    pSet            PortalIdSet;
            START
            COMMENT'Process a Shut.portal input signal.
                    It is now safe to stop the portal processes.
                    If this was the last portal, the domain process stops too
                    so that it can be recreated with different parameters.
                    ';
            TASK    ds := portal(p)!domain;
            CALL    Exit_portal(p);
            DECISION domain(ds)!portals = 0;
            (True): TASK    domain(ds)!pid := Null,
                            pSet := pUsed;
                     1b :   /* for p in pSet */
                    DECISION pSet = Empty;
                    (False):TASK    p := Pick(pSet),
                                    pSet := Del(p, pSet);
                            DECISION portal(p)!domain = ds;
                            (True): CALL    Open_portal(p);
                            ELSE:ENDDECISION;
                    ELSE:ENDDECISION;
            ELSE:ENDDECISION;
            RETURN;
            ENDPROCEDURE;

                                                                    /*---------------*/
    PROCEDURE       Exit_portal;                                    /*  Exit_portal  */
    FPAR            p               PortalId;                       /*---------------*/

            DCL     dp              DataPriority;
            START
            COMMENT'Release the portal and stop its processes.
                    ';
            TASK    pUsed := Del(p, pUsed),
                    pFree := Incl(p, pFree),
                    dp := 0;
             1b :   /* for dp = 0..3 */
            DECISION dp < 4;
            (True): DECISION portal(p)!proc(dp);
                    (Quit):
                            OUTPUT  Exit TO portal(p)!pids(dp);
                            TASK    portal(p)!proc(dp) := Nil;
                    ELSE:ENDDECISION;
                    TASK    dp := dp + 1;
                    JOIN 1b;
            ELSE:ENDDECISION;
            RETURN;
            ENDPROCEDURE;

            /* Input transitions */

            DCL     p               PortalId,
                    dp              DataPriority,
                    pdu             PDUStruct,
                    mcId            MCSConnectionId,
                    ccId            Natural,
                    tcId            TCEndpointId,
                    reason          Reason,
                    result          Result,
                    diagnostic      Diagnostic,
                    label           Natural,
                    localTSAP       TSAPAddress,
                    localDomain     DomainSelector,
                    remoteTSAP      TSAPAddress,
                    remoteDomain    DomainSelector,
*                   upward          Boolean,
                    targetParms     DomainParameters,
                    minParms        DomainParameters,
                    maxParms        DomainParameters,
                    parameters      DomainParameters,
|                   targetQOSByPri  TransportQOSByPri,
|                   minQOSByPri     TransportQOSByPri,
|                   offeredQOS      TransportQOS,
|                   minQOS          TransportQOS,
                    userData        UserData;
            START
            COMMENT'The state machine contains a single state.
                    ';
```

```
1021            CALL    Initialize_resources;
1022            NEXTSTATE ~;

1024    STATE ~;INPUT   MCS.Connect.Provider.request(label, localTSAP, localDomain,
1025                    remoteTSAP, remoteDomain, upward, targetParms, minParms, maxParms,
1026 |                  targetQOSByPri, minQOSByPri, userData);
1027            CALL    MCS_Connect_Provider_request(label, localTSAP, localDomain,
1028                    remoteTSAP, remoteDomain, upward, targetParms, minParms, maxParms,
1029 |                  targetQOSByPri, minQOSByPri, userData);
1030            NEXTSTATE -;

1032    STATE ~;INPUT   MCS.Connect.Provider.response(mcId, result, parameters, userData);
1033            CALL    MCS_Connect_Provider_response(mcId, result, parameters, userData);
1034            NEXTSTATE -;

1036    STATE ~;INPUT   MCS.Disconnect.Provider.request(mcId);
1037            CALL    MCS_Disconnect_Provider_request(mcId);
1038            NEXTSTATE -;

1040 |  STATE ~;INPUT   MCS.Attach.User.request(label, localDomain);
1041 |          CALL    MCS_Attach_User_request(label, localDomain);
1042            NEXTSTATE -;

1044    STATE ~;INPUT   T.Connect.indication(tcId, remoteTSAP, localTSAP, offeredQOS, minQOS);
1045            CALL    T_Connect_indication(tcId, remoteTSAP, localTSAP, offeredQOS, minQOS);
1046            NEXTSTATE -;

1048 |  STATE ~;INPUT   Connect.Initial(remoteDomain, localDomain, upward,
1049                    targetParms, minParms, maxParms, userData);
1050 |          CALL    Connect_Initial(remoteDomain, localDomain, upward,
1051                    targetParms, minParms, maxParms, userData);
1052            NEXTSTATE -;

1054    STATE ~;INPUT   Connect.Response(result, ccId, parameters, userData);
1055            CALL    Connect_Response(result, ccId, parameters, userData);
1056            NEXTSTATE -;

1058    STATE ~;INPUT   Connect.Additional(ccId, dp);
1059            CALL    Connect_Additional(ccId, dp);
1060            NEXTSTATE -;

1062    STATE ~;INPUT   Connect.Result(result);
1063            CALL    Connect_Result(result);
1064            NEXTSTATE -;

1066    STATE ~;INPUT   Drop.portal(p, reason);
1067            CALL    Drop_portal(p, reason);
1068            NEXTSTATE -;

1070    STATE ~;INPUT   Report.portal(p, diagnostic);
1071            CALL    Report_portal(p, diagnostic);
1072            NEXTSTATE -;

1074    STATE ~;INPUT   Shut.portal(p);
1075            CALL    Shut_portal(p);
1076            NEXTSTATE -;

1078    STATE ~;INPUT   RJum(pdu);
1079            CALL    RJum(pdu);
1080            NEXTSTATE -;

1082    STATE ~;INPUT   Quit;
1083            CALL    Quit;
1084            NEXTSTATE -;

1086    ENDPROCESS;
```

Appendix IV
SDL specification of the Domain process

(This appendix does not form an integral part of this Recommendation.)

```
 1    PROCESS Domain;
 2    FPAR    parameters      DomainParameters;       /* values established in domain */

 4    SYNONYM maxBufferIds    Natural = EXTERNAL;     /* an implementation limit */

 6    TIMER   Time.portal(PortalId);                  /* at most one timer per portal */

 8            /* Type definitions */

10    NEWTYPE         ChannelStruct
11    STRUCT
12            kind            ChannelKind;    /* (Static,UserId,Private,Assigned) */
13            joined          PortalIdSet;    /* directions where channel is joined */
14            portal          PortalId;       /* if (UserId): the direction to it */
15            manager         UserId;         /* if (Private): channel's manager */
16            admitted        UserIdSet;      /* if (Private): zero or more users */
17            uMerge          UserIdSet;      /* if (Private): still to be merged */
18    ENDNEWTYPE;

20    NEWTYPE         Chan                Array(ChannelId, ChannelStruct);
21    ENDNEWTYPE;

23    NEWTYPE         TokenStruct
24    STRUCT
25            kind            TokenKind;      /* (Grabbed,Inhibited,Giving,Ungivable,Given) */
26            grabber         UserId;         /* if (Grabbed,Giving,Ungivable): user */
27            recipient       UserId;         /* if (Giving,Given): an intended user */
28            inhibitors      UserIdSet;      /* if (Inhibited): one or more users */
29            uMerge          UserIdSet;      /* if (Inhibited): still to be merged */
30    ENDNEWTYPE;

32    NEWTYPE         Token               Array(TokenId, TokenStruct);
33    ENDNEWTYPE;

35    NEWTYPE         BooleanByPri        Array(DataPriority, Boolean);
36    ENDNEWTYPE;
37    NEWTYPE         NaturalByPri        Array(DataPriority, Natural);
38    ENDNEWTYPE;
39    NEWTYPE         BufferIdByPri       Array(DataPriority, BufferId);
40    ENDNEWTYPE;
41    NEWTYPE         BufferIdQueueByPri  Array(DataPriority, BufferIdQueue);
42    ENDNEWTYPE;
43    NEWTYPE         PortalIdSetByPri    Array(DataPriority, PortalIdSet);
44    ENDNEWTYPE;

46    NEWTYPE         NaturalByPriByKind  Array(PortalKind, NaturalByPri);
47    ENDNEWTYPE;

49    NEWTYPE         PortalStruct
50    STRUCT
51            kind            PortalKind;         /* (Attached,Downlink,Uplink) */
52            pids            PIdByPri;           /* processes constituting a portal */
53            inCredit        NaturalByPri;       /* permission to allocate inBuffer */
54            inBuffer        BufferIdByPri;      /* PDU input coming from a process */
55            outReady        BooleanByPri;       /* True if process allows an output */
56            outQueue        BufferIdQueueByPri; /* PDUs awaiting output to process */
57            outCount        Natural;            /* number queued for all priorities */
58            outFlow         Natural;            /* number output since timer was set */
59            elapsed         Duration;           /* interval set for portal timer */
60            interval        Duration;           /* new interval to set for timer */
61            subHeight       Natural;            /* subordinate's height or zero */
62            subInterval     Duration;           /* subordinate's interval or zero */
63    ENDNEWTYPE;

65    NEWTYPE         Portal              Array(PortalId, PortalStruct);
66    ENDNEWTYPE;

68    NEWTYPE         PortalIdQueue       Queue(PortalId);
69    ENDNEWTYPE;

71    SYNTYPE         BufferId            = Integer CONSTANTS 0:maxBufferIds
72    ENDSYNTYPE;
```

```
 74    NEWTYPE         BufferIdSet         SetOf(BufferId);
 75    ENDNEWTYPE;

 77    NEWTYPE         BufferStruct
 78    STRUCT
 79            receiver        PortalId;       /* source of inCredit and input PDU */
 80            dataPriority    DataPriority;   /* index into inBuffer and outQueue */
 81            portals         Natural;        /* number of outQueues buffer is in */
 82            pdu             PDUStruct;      /* the content of one domain MCSPDU */
 83    ENDNEWTYPE;

 85    NEWTYPE         Buffer              Array(BufferId, BufferStruct);
 86    ENDNEWTYPE;

 88    NEWTYPE         BufferIdQueue       Queue(BufferId);
 89    ENDNEWTYPE;

 91    GENERATOR       Queue (TYPE ItemType)   /* a first-in first-out queue */
 92    LITERALS
 93            EmptyQueue;
 94    OPERATORS
 95            Push:   ItemType, Queue -> Queue;       /* appends next item */
 96            Next:   Queue           -> ItemType;    /* reveals first item */
 97            Pull:   Queue           -> Queue;       /* deletes first item */
 98    AXIOMS
 99            Next(EmptyQueue) == ERROR!;
100            Pull(EmptyQueue) == ERROR!;
101            FOR ALL q IN Queue (
102            FOR ALL item IN ItemType
103            (
104                    Next(Push(item,q)) == IF q = EmptyQueue THEN item
105                                                  ELSE Next(q) FI;
106                    Pull(Push(item,q)) == IF q = EmptyQueue THEN q
107                                                  ELSE Push(item,Pull(q)) FI;
108            ));
109    DEFAULT
110            EmptyQueue;
111    ENDGENERATOR;

113            /* Data declarations */

115    DCL     control         PId;                /* the Control process */

117    DCL     upward          PortalId;           /* unique MCS Connection upward or */
118                                                /* zero if this provider is the top */

120    DCL     merging         Boolean,            /* True if domain is merging upward */
121            pdSend          Boolean,            /* True if PDin to be sent downward */
122            edSend          Boolean;            /* True if EDrq to be sent upward */

124    DCL     uMerge          UserIdSet,          /* users still to be merged */
125            uConfirm        UserIdSet,          /* users with merge confirmed */
126            cMerge          ChannelIdSet,       /* channels still to be merged */
127 *          cConfirm        ChannelIdSet,       /* channels with merge confirmed */
128            tMerge          TokenIdSet,         /* tokens still to be merged */
129 *          tConfirm        TokenIdSet,         /* tokens with merge confirmed */

131    DCL     mcrqQueue       PortalIdQueue,      /* origin of each pending MCrq */
132            mtrqQueue       PortalIdQueue,      /* origin of each pending MTrq */
133            aurqQueue       PortalIdQueue;      /* origin of each pending AUrq */

135    DCL     pDrop           PortalIdSet,        /* dropped portals needing DPum */
136            uDetach         UserIdSet,          /* disconnected users needing DUrq */
137            cLeave          ChannelIdSet,       /* unjoined channels needing CLrq */
138            cDisband        ChannelIdSet,       /* unmanaged channels needing CDin */
139            tReject         TokenIdSet;         /* ungivable tokens needing TVcf */

141    DCL     pBufferWait     PortalIdSetByPri,   /* portals requiring an inBuffer */
142            pInputWait      PortalIdSetByPri;   /* inputs suspended during merge */

144    DCL     chan            Chan,               /* resource arrays */
145            token           Token,
146            portal          Portal,
147            buffer          Buffer;

149            /* Note: The fields of a chan, token, portal, or buffer
150            array element are undefined if the corresponding index
151            is not in cUsed, tUsed, pUsed, or bUsed, respectively. */
```

```
153    DCL     cUsed           ChannelIdSet,       /* indexes used */
154            tUsed           TokenIdSet,
155            pUsed           PortalIdSet,
156            bUsed           BufferIdSet;

158    DCL     cFree           ChannelIdSet,       /* indexes free */
159            tFree           TokenIdSet,
160            pFree           PortalIdSet,
161            bFree           BufferIdSet;

163    DCL     uUsed           UserIdSet;          /* subset of cUsed */

165    DCL     numChannelIds   Natural,            /* number in use */
166            numUserIds      Natural,
167            numTokenIds     Natural;

169    DCL     height          Natural,            /* current height of provider */
170            interval        Duration,           /* min throughput of one MCSPDU */
171            maxInterval     Duration;           /* maximum of portal intervals */

173    DCL     inCredit        NaturalByPriByKind; /* initializer */

175            /* Procedure decomposition */

177            /*      Initialize_resources
178 |                  Take_user               (c, diagnostic)
179 |                  Take_channel            (c, diagnostic)
180 |                  Take_token              (t, diagnostic)
181                    New_user                (u, result)
182                    New_channel             (c, result)
183                    Open_portal             (p, pKind, pids)
184                    Time_portal             (p)
185                    Drop_portal             (p, reason)
186                    Shut_portal             (p)
187                    Clean_queue             (p, queue)
188                    Erect_domain
189                    Identify_sender         (p, dp)
190                    PDU_ready               (dp)
191                    Input_PDU               (pdu)
192                    Allocate_buffer         (b)
193                    Cast_buffer             (b, p)
194                    Release_buffer          (b)
195                    Route_user              (u, p)
196                    Multicast_buffer        (b, uSet)
197                    Broadcast_buffer        (b)
198                    Crank_engine
199                    Drop_portals
200                    Merge_users
201                    Merge_channels
202                    Merge_tokens
203                    Detach_users
204                    Leave_channels
205                    Disband_channels
206                    Reject_tokens
207 |                  Process_PDU             (r, dp)
208                    Validate_input          (r, b, diagnostic)
209                    Top_provider            (r, b)
210 |                  Apply_PDU               (r, b, diagnostic)
211                    Token_status            (pdu)
212                    Delete_user             (u)
213                    Delete_channel          (c)
214                    Delete_token            (t)
215 +                  Purge_users             (uSet)
216                    Purge_channels          (cSet)
217                    Purge_tokens            (tSet)
218                    Output_buffer           (b, p)          */

220                                                            /*-------------------------*/
221    PROCEDURE       Initialize_resources;                   /*  Initialize_resources   */
222                                                            /*-------------------------*/
223            DCL     c               ChannelId,
224                    t               TokenId,
225                    p               PortalId,
226                    b               BufferId,
227                    n               NaturalByPri;
228            START
229            COMMENT'Initialize data structures during process start-up
230                    before accepting the first input signal.
```

```
231                     Note that each SetOf automatically defaults to Empty
232                     and each Queue to EmptyQueue.
233                     Fixed buffer credits are an example; other values could
234                     be computed from maxPortalIds and maxBufferIds.
235                     ';
236             TASK    control := PARENT,
237                     upward := 0,
238                     merging := False,
239 +                   pdSend := False,
240 +                   edSend := False,
241                     numChannelIds := 0,
242                     numUserIds := 0,
243                     numTokenIds := 0,
244                     height := 1,
245 |                   interval := IF parameters!minThroughput = 0 THEN 0 ELSE oneSecond *
246 |                    (Float(parameters!maxMCSPDUsize) / Float(parameters!minThroughput)) FI,
247                     maxInterval := 0,
248                     c := 65535,
249                     t := 65535,
250                     p := maxPortalIds,
251                     b := maxBufferIds;
252              1b :   /* for c = ?..1 */
253             DECISION c > 0;
254             (True): TASK    cFree := Incl(c, cFree),
255                             c := c - 1;
256                     JOIN 1b;
257             ELSE:ENDDECISION;
258              2b :   /* for t = ?..1 */
259 |           TASK    tFree := Incl(t, tFree);
260 |           DECISION t > 1;
261 |           (True): TASK    t := t - 1;
262                     JOIN 2b;
263             ELSE:ENDDECISION;
264              3b :   /* for p = ?..1 */
265             DECISION p > 0;
266             (True): TASK    pFree := Incl(p, pFree),
267                             p := p - 1;
268                     JOIN 3b;
269             ELSE:ENDDECISION;
270              4b :   /* for b = ?..1 */
271             DECISION b > 0;
272             (True): TASK    bFree := Incl(b, bFree),
273                             b := b - 1;
274                     JOIN 4b;
275             ELSE:ENDDECISION;
276             TASK    n(0) := 2, n(1) := 1, n(2) := 1, n(3) := 1,
277                     inCredit(Attached) := n,
278                     n(0) := 3, n(1) := 2, n(2) := 1, n(3) := 1,
279 |                   inCredit(Downlink) := n,
280                     n(0) := 4, n(1) := 3, n(2) := 2, n(3) := 1,
281 |                   inCredit(Uplink) := n;
282             RETURN;
283             ENDPROCEDURE;

285                                                                     /*-------------*/
286     PROCEDURE       Take_user;                                      /*  Take_user  */
287 |   FPAR            c               ChannelId,                      /*-------------*/
288 |           IN/OUT  diagnostic      Diagnostic;

290             DCL     u               UserId;
291             START
292             COMMENT'Put the free channel id to use as a user id.
293                     ';
294 |           DECISION numUserIds < parameters!maxUserIds;
295 |           (False):TASK    diagnostic := DC_too_many_users;
296 |                   RETURN;
297 |           ELSE:ENDDECISION;
298 |           CALL    Take_channel(c, diagnostic);
299 |           DECISION diagnostic = DC_OK;
300 |           (True): TASK    u := UserId(c),
301 |                           numUserIds := numUserIds + 1,
302 |                           uUsed := Incl(u, uUsed),
303 |                           chan(c)!kind := UserId;
304 |           ELSE:ENDDECISION;
305             RETURN;
306             ENDPROCEDURE;

308                                                                     /*----------------*/
309     PROCEDURE       Take_channel;                                   /*  Take_channel  */
```

```
310 |  FPAR           c               ChannelId,                      /*----------------*/
311 |          IN/OUT  diagnostic      Diagnostic;

313             START
314             COMMENT'Put the free channel id to unspecified use.
315                     ';
316 |           DECISION c in cFree;
317 |           (False):TASK    diagnostic := DC_channel_id_conflict;
318 |                   RETURN;
319 |           ELSE:ENDDECISION;
320 |           DECISION numChannelIds < parameters!maxChannelIds;
321 |           (False):TASK    diagnostic := DC_too_many_channels;
322 |                   RETURN;
323 |           ELSE:ENDDECISION;
324 |           TASK    diagnostic := DC_OK,
325 |                   numChannelIds := numChannelIds + 1,
326                     cFree := Del(c, cFree),
327                     cUsed := Incl(c, cUsed),
328                     chan(c)!joined := Empty,
329                     chan(c)!admitted := Empty;
330             RETURN;
331             ENDPROCEDURE;

333                                                                     /*--------------*/
334     PROCEDURE       Take_token;                                     /*  Take_token  */
335 |   FPAR            t               TokenId,                        /*--------------*/
336 |           IN/OUT  diagnostic      Diagnostic;

338             START
339             COMMENT'Put the free token id to unspecified use.
340                     ';
341 |           DECISION t in tFree;
342 |           (False):TASK    diagnostic := DC_token_id_conflict;
343 |                   RETURN;
344 |           ELSE:ENDDECISION;
345 |           DECISION numTokenIds < parameters!maxTokenIds;
346 |           (False):TASK    diagnostic := DC_too_many_tokens;
347 |                   RETURN;
348 |           ELSE:ENDDECISION;
349 |           TASK    diagnostic := DC_OK,
350 |                   numTokenIds := numTokenIds + 1,
351                     tFree := Del(t, tFree),
352                     tUsed := Incl(t, tUsed),
353                     token(t)!inhibitors := Empty;
354             RETURN;
355             ENDPROCEDURE;

357                                                                     /*------------*/
358     PROCEDURE       New_user;                                       /*  New_user  */
359     FPAR    IN/OUT  u               UserId,                         /*------------*/
360             IN/OUT  result          Result;

362             DCL     c               ChannelId;
363             START
364             COMMENT'Allocate a new user id or fail and return 0.
365                     ';
366             TASK    u := 0;
367             DECISION numUserIds < parameters!maxUserIds;
368             (False):TASK    result := RT_too_many_users;
369                     RETURN;
370             ELSE:ENDDECISION;
371             CALL    New_channel(c, result);
372             DECISION result = RT_successful;
373             (True): TASK    u := UserId(c),
374                             numUserIds := numUserIds + 1,
375                             uUsed := Incl(u, uUsed),
376                             chan(c)!kind := UserId;
377             ELSE:ENDDECISION;
378             RETURN;
379             ENDPROCEDURE;

381                                                                     /*---------------*/
382     PROCEDURE       New_channel;                                    /*  New_channel  */
383     FPAR    IN/OUT  c               ChannelId,                      /*---------------*/
384             IN/OUT  result          Result;

386 +           DCL     diagnostic      Diagnostic;
387             START
388             COMMENT'Allocate a new channel id or fail and return 0.
```

```
389                     ';
390             TASK    c := 0;
391             DECISION numChannelIds < parameters!maxChannelIds;
392             (False):TASK    result := RT_too_many_channels;
393                     RETURN;
394             ELSE:ENDDECISION;
395              1b :   /* keep trying */
396             TASK    c := ANY(ChannelId);    /* randomize */
397             DECISION c >= 1001 and c in cFree;
398             (False):JOIN 1b;
399             ELSE:ENDDECISION;
400 |           CALL    Take_channel(c, diagnostic);
401             TASK    result := RT_successful;
402             RETURN;
403             ENDPROCEDURE;

405                                                                     /*---------------*/
406     PROCEDURE       Open_portal;                                    /*  Open_portal  */
407     FPAR            p               PortalId,                       /*---------------*/
408                     pKind           PortalKind,
409                     pids            PIdByPri;

411             DCL     pid             PId,
412                     dp              DataPriority,
413                     c               ChannelId,
414                     cSet            ChannelIdSet,
415                     t               TokenId,
416                     tSet            TokenIdSet;
417             START
418             COMMENT'Process an Open.portal input signal.
419                     Accept a new MCS connection or attachment to the domain.
420                     If this is an upward connection, prepare for merge.
421                     ';
422             DECISION pKind = Attached;
423             (True): TASK    pid := pids(0),
424                             pids(1) := pid,
425                             pids(2) := pid,
426                             pids(3) := pid;
427             ELSE:ENDDECISION;
428             TASK    pFree := Del(p, pFree),
429                     pUsed := Incl(p, pUsed),
430                     portal(p)!kind := pKind,
431                     portal(p)!pids := pids,
432                     portal(p)!inCredit := inCredit(pKind),
433                     portal(p)!outCount := 0,
434 |                   portal(p)!interval := IF pKind = Uplink THEN 0 ELSE interval FI,
435                     portal(p)!subHeight := 0,
436                     portal(p)!subInterval := 0,
437                     dp := 0;
438              1b :   /* for dp = 0..? */
439             DECISION dp < parameters!numPriorities;
440             (True): TASK    portal(p)!inBuffer(dp) := 0,
441                             portal(p)!outReady(dp) := False,
442                             portal(p)!outQueue(dp) := EmptyQueue,
443                             pBufferWait(dp) := Incl(p, pBufferWait(dp)),
444                             dp := dp + 1;
445                     JOIN 1b;
446             ELSE:ENDDECISION;
447 |           DECISION portal(p)!kind = Uplink;
448             (True): TASK    upward := p,
449                             merging := True,
450                             pdSend := True,
451                             edSend := True,
452                             uMerge := uUsed,
453                             uConfirm := Empty,
454                             cMerge := cUsed,
455 *                           cConfirm := Empty,
456                             tMerge := tUsed,
457 *                           tConfirm := Empty,
458                             cLeave := Empty,
459                             cDisband := Empty,
460                             tReject := Empty,
461                             cSet := cMerge;
462                      2b :   /* for c in cSet */
463                     DECISION cSet = Empty;
464                     (False):TASK    c := Pick(cSet),
465                                     cSet := Del(c, cSet),
466                                     chan(c)!uMerge := chan(c)!admitted;
467                             JOIN 2b;
```

```
468                     ELSE:ENDDECISION;
469                     TASK    tSet := tMerge;
470                      3b :   /* for t in tSet */
471                     DECISION tSet = Empty;
472                     (False):TASK    t := Pick(tSet),
473                                     tSet := Del(t, tSet),
474                                     token(t)!uMerge := token(t)!inhibitors;
475                             JOIN 3b;
476                     ELSE:ENDDECISION;
477             ELSE:ENDDECISION;
478             CALL    Erect_domain;
479             CALL    Crank_engine;
480             RETURN;
481             ENDPROCEDURE;

483                                                                     /*---------------*/
484     PROCEDURE       Time_portal;                                    /*  Time_portal  */
485     FPAR            p               PortalId;                       /*---------------*/

487             START
488             COMMENT'Process a Time.portal input signal.
489                     Ensure that minimum throughput is maintained.
490                     Allow for the fact that outFlow takes integer steps.
491                     ';
492 |           DECISION portal(p)!elapsed > interval * (Float(portal(p)!outFlow) + 0.99);
493 |           (True): OUTPUT  Report.portal(p, DC_throughput_inadequate) TO control;
494 |           (False):TASK    portal(p)!outFlow := 0,
495                             portal(p)!elapsed := portal(p)!interval;
496 *                   SET(NOW + portal(p)!interval, Time.portal(p));
497             ENDDECISION;
498             CALL    Crank_engine;
499             RETURN;
500             ENDPROCEDURE;

502                                                                     /*---------------*/
503     PROCEDURE       Drop_portal;                                    /*  Drop_portal  */
504     FPAR            p               PortalId,                       /*---------------*/
505                     reason          Reason;

507             START
508             COMMENT'Process a Drop.portal input signal,
509                     knowing the only reason is user-requested.
510                     ';
511             DECISION p in pUsed and portal(p)!kind /= Attached;
512             (True): TASK    pDrop := Incl(p, pDrop);
513             ELSE:ENDDECISION;
514             CALL    Crank_engine;
515             RETURN;
516             ENDPROCEDURE;

518                                                                     /*---------------*/
519     PROCEDURE       Shut_portal;                                    /*  Shut_portal  */
520     FPAR            p               PortalId;                       /*---------------*/

522             DCL     dp              DataPriority,
523                     b               BufferId,
524                     bSet            BufferIdSet,
525                     c               ChannelId,
526                     cSet            ChannelIdSet,
527                     u               UserId;
528             START
529             COMMENT'Process a Shut.portal input signal.
530                     Remove the corresponding MCS connection or attachment.
531                     When the last one is gone, stop the domain process.
532                     ';
533             RESET(Time.portal(p));
534             TASK    pUsed := Del(p, pUsed),
535                     pFree := Incl(p, pFree),
536                     pDrop := Del(p, pDrop),
537                     dp := 0;
538              1b :   /* for dp = 0..? */
539             DECISION dp < parameters!numPriorities;
540             (True):  2b :   /* for b in outQueue(dp) */
541                     DECISION portal(p)!outQueue(dp) = EmptyQueue;
542                     (False):TASK    b := Next(portal(p)!outQueue(dp)),
543                                     portal(p)!outQueue(dp) := Pull(portal(p)!outQueue(dp)),
544                                     buffer(b)!portals := buffer(b)!portals - 1;
545                             CALL    Release_buffer(b);
546                             JOIN 2b;
```

```
547                     ELSE:ENDDECISION;
548                     TASK    b := portal(p)!inBuffer(dp),
549                             portal(p)!inBuffer(dp) := 0;
550                     CALL    Release_buffer(b);
551                     TASK    pBufferWait(dp) := Del(p, pBufferWait(dp)),
552                             pInputWait(dp) := Del(p, pInputWait(dp)),
553                             dp := dp + 1;
554                     JOIN 1b;
555             ELSE:ENDDECISION;
556             TASK    bSet := bUsed;
557              3b :   /* for b in bSet */
558             DECISION bSet = Empty;
559             (False):TASK    b := Pick(bSet),
560                             bSet := Del(b, bSet);
561                     DECISION buffer(b)!receiver = p;
562                     (True): TASK    buffer(b)!receiver := 0;
563                     ELSE:ENDDECISION;
564                     JOIN 3b;
565             ELSE:ENDDECISION;
566             TASK    cSet := cUsed;
567              4b :   /* for c in cSet */
568             DECISION cSet = Empty;
569             (False):TASK    c := Pick(cSet),
570                             cSet := Del(c, cSet);
571                     DECISION p in chan(c)!joined;
572                     (True): TASK    chan(c)!joined := Del(p, chan(c)!joined);
573                             DECISION chan(c)!joined = Empty;
574                             (True): TASK    cLeave := Incl(c, cLeave);
575                             ELSE:ENDDECISION;
576                     ELSE:ENDDECISION;
577                     DECISION chan(c)!kind = UserId and chan(c)!portal = p;
578                     (True): TASK    chan(c)!portal := 0,
579                                     u := UserId(c),
580                                     uDetach := Incl(u, uDetach),
581                                     cLeave := Del(c, cLeave);
582                     ELSE:ENDDECISION;
583                     JOIN 4b;
584             ELSE:ENDDECISION;
585 |           DECISION portal(p)!kind = Uplink;
586             (True): TASK    upward := 0,
587                             merging := False,
588                             mcrqQueue := EmptyQueue,
589                             mtrqQueue := EmptyQueue,
590                             aurqQueue := EmptyQueue;
591             (False):CALL    Clean_queue(p, mcrqQueue);
592                     CALL    Clean_queue(p, mtrqQueue);
593                     CALL    Clean_queue(p, aurqQueue);
594             ENDDECISION;
595 *           OUTPUT  Shut.portal(p) TO control;
596             CALL    Erect_domain;
597             CALL    Crank_engine;
598             RETURN;
599             ENDPROCEDURE;

601                                                                     /*---------------*/
602     PROCEDURE       Clean_queue;                                    /*  Clean_queue  */
603     FPAR            p               PortalId,                       /*---------------*/
604             IN/OUT  queue           PortalIdQueue;

606             DCL     x               PortalId,
607                     clean           PortalIdQueue;
608             START
609             COMMENT'Remove a shut portal id from a queue.
610                     ';
611             TASK    clean := EmptyQueue;
612              1b :   /* for x in queue */
613             DECISION queue = EmptyQueue;
614             (False):TASK    x := Next(queue),
615                             queue := Pull(queue),
616                             x := IF x = p THEN 0 ELSE x FI,
617                             clean := Push(x, clean);
618                     JOIN 1b;
619             ELSE:ENDDECISION;
620             TASK    queue := clean;
621             RETURN;
622             ENDPROCEDURE;

624                                                                     /*---------------*/
625     PROCEDURE       Erect_domain;                                   /*  Erect_domain  */
```

```
626                                                                     /*----------------*/

628             DCL     p               PortalId,
629                     pSet            PortalIdSet,
630                     h               Natural,
631                     hmax            Natural,
632                     i               Duration,
633                     imax            Duration;
634             START
635             COMMENT'Recalculate the provider height and maxInterval.
636                     If either changed, communicate them upward.
637                     ';
638             TASK    hmax := 1,
639                     imax := 0,
640                     pSet := pUsed;
641              1b :   /* for p in pSet */
642             DECISION pSet = Empty;
643             (False):TASK    p := Pick(pSet),
644                             pSet := Del(p, pSet),
645                             h := portal(p)!subHeight + 1,
646                             hmax := IF h > hmax THEN h ELSE hmax FI,
647                             i := portal(p)!interval,
648                             imax := IF i > imax THEN i ELSE imax FI;
649                     JOIN 1b;
650             ELSE:ENDDECISION;
651             DECISION height = hmax and maxInterval = imax;
652             (False):TASK    height := hmax,
653                             maxInterval := imax,
654                             edSend := True;
655             ELSE:ENDDECISION;
656             RETURN;
657             ENDPROCEDURE;

659                                                                     /*--------------------*/
660     PROCEDURE       Identify_sender;                                /*  Identify_sender  */
661     FPAR    IN/OUT  p               PortalId,                       /*--------------------*/
662             IN/OUT  dp              DataPriority;

664             DCL     pSet            PortalIdSet;
665             START
666             COMMENT'An alternative would be to carry this
667                     information explicitly in SDL signals.
668                     ';
669             TASK    pSet := pUsed;
670              1b :   /* for p in pSet */
671             DECISION pSet = Empty;
672             (False):TASK    p := Pick(pSet),
673                             pSet := Del(p, pSet),
674                             dp := 0;
675                      2b :   /* for dp = 0..? */
676                     DECISION dp < parameters!numPriorities;
677                     (True): DECISION portal(p)!pids(dp) = SENDER;
678                             (True): RETURN;
679                             ELSE:ENDDECISION;
680                             TASK    dp := dp + 1;
681                             JOIN 2b;
682                     ELSE:ENDDECISION;
683                     JOIN 1b;
684             ELSE:ENDDECISION;
685             TASK    p := 0,
686                     dp := 0;
687             RETURN;
688             ENDPROCEDURE;

690                                                                     /*--------------*/
691     PROCEDURE       PDU_ready;                                      /*  PDU_ready  */
692     FPAR            dp              DataPriority;                   /*--------------*/
693             DCL     p               PortalId,
694                     x               DataPriority,
695                     b               BufferId;
696             START
697             COMMENT'Process a PDU.ready input signal.
698                     If a buffer is waiting, it can be output.
699                     ';
700             CALL    Identify_sender(p, x);
701             DECISION p in pUsed;
702             (False):RETURN;
703             ELSE:ENDDECISION;
704             DECISION portal(p)!outQueue(dp) = EmptyQueue;
```

```
705             (True): TASK    portal(p)!outReady(dp) := True;
706             (False):TASK    b := Next(portal(p)!outQueue(dp)),
707                             portal(p)!outQueue(dp) := Pull(portal(p)!outQueue(dp)),
708                             buffer(b)!portals := buffer(b)!portals - 1;
709                     CALL    Output_buffer(b, p);
710                     TASK    portal(p)!outReady(dp) := False;
711                     CALL    Release_buffer(b);
712                     TASK    portal(p)!outFlow := portal(p)!outFlow + 1,
713                             portal(p)!outCount := portal(p)!outCount - 1;
714                     DECISION portal(p)!outCount = 0;
715                     (True): RESET(Time.portal(p));
716                     ELSE:ENDDECISION;
717             ENDDECISION;
718             CALL    Crank_engine;
719             RETURN;
720             ENDPROCEDURE;

722                                                                     /*-------------*/
723     PROCEDURE       Input_PDU;                                      /*  Input_PDU  */
724     FPAR            pdu             PDUStruct;                      /*-------------*/

726             DCL     p               PortalId,
727                     dp              DataPriority,
728                     b               BufferId;
729             START
730             COMMENT'Process an MCSPDU input signal.
731                     If merging, requests and responses must wait.
732                     ';
733             CALL    Identify_sender(p, dp);
734             DECISION p in pUsed;
735             (False):RETURN;
736             ELSE:ENDDECISION;
737             DECISION portal(p)!kind = Attached;
738             (True): DECISION pdu!kind;
739                     (SDrq, SDin, USrq, USin):
740                             TASK    dp := pdu!dataPriority,
741                                     dp := IF dp < parameters!numPriorities THEN dp
742                                                 ELSE parameters!numPriorities - 1 FI;
743                     ELSE:ENDDECISION;
744             ELSE:ENDDECISION;
745             TASK    b := portal(p)!inBuffer(dp),
746                     buffer(b)!pdu := pdu;
747             DECISION p = upward or not merging;
748 |           (True): CALL    Process_PDU(p, dp);
749             (False):TASK    pInputWait(dp) := Incl(p, pInputWait(dp));
750             ENDDECISION;
751             CALL    Crank_engine;
752             RETURN;
753             ENDPROCEDURE;

755                                                                     /*-------------------*/
756     PROCEDURE       Allocate_buffer;                                /*  Allocate_buffer  */
757     FPAR    IN/OUT  b               BufferId;                       /*-------------------*/

759             START
760             COMMENT'Allocate a free buffer id or fail and return 0.
761                     ';
762             DECISION b /= 0 and buffer(b)!portals = 0;
763             (True): RETURN;
764             ELSE:ENDDECISION;
765             TASK    b := 0;
766             DECISION bFree = Empty;
767             (False):TASK    b := Pick(bFree),
768                             bFree := Del(b, bFree),
769                             bUsed := Incl(b, bUsed),
770                             buffer(b)!receiver := 0,
771                             buffer(b)!dataPriority := 0,
772                             buffer(b)!portals := 0;
773             ELSE:ENDDECISION;
774             RETURN;
775             ENDPROCEDURE;

777                                                                     /*---------------*/
778     PROCEDURE       Cast_buffer;                                    /*  Cast_buffer  */
779     FPAR            b               BufferId,                       /*---------------*/
780                     p               PortalId;

782             DCL     dp              DataPriority;
783             START
```

```
784             COMMENT'Send buffer containing an MCSPDU to a single output.
785                     ';
786             DECISION p = 0;
787             (True): RETURN;
788             ELSE:ENDDECISION;
789             TASK    dp := buffer(b)!dataPriority;
790             DECISION portal(p)!outReady(dp);
791             (True): CALL    Output_buffer(b, p);
792                     TASK    portal(p)!outReady(dp) := False;
793             (False):DECISION portal(p)!outCount = 0 and portal(p)!interval > 0;
794                     (True): TASK    portal(p)!outFlow := 0,
795                                     portal(p)!elapsed := portal(p)!interval;
796                             SET(NOW + portal(p)!interval, Time.portal(p));
797                     ELSE:ENDDECISION;
798                     TASK    portal(p)!outQueue(dp) := Push(b, portal(p)!outQueue(dp)),
799                             buffer(b)!portals := buffer(b)!portals + 1,
800                             portal(p)!outCount := portal(p)!outCount + 1;
801             ENDDECISION;
802             RETURN;
803             ENDPROCEDURE;

805                                                                     /*------------------*/
806     PROCEDURE       Release_buffer;                                 /*  Release_buffer  */
807     FPAR            b               BufferId;                       /*------------------*/

809             DCL     p               PortalId,
810                     dp              DataPriority;
811             START
812             COMMENT'Release a buffer that is no longer needed.
813                     ';
814             DECISION b = 0 or buffer(b)!portals > 0;
815             (True): RETURN;
816             ELSE:ENDDECISION;
817             TASK    bUsed := Del(b, bUsed),
818                     bFree := Incl(b, bFree),
819                     p := buffer(b)!receiver,
820                     dp := buffer(b)!dataPriority;
821             DECISION p = 0;
822             (False):TASK    portal(p)!inCredit(dp) := portal(p)!inCredit(dp) + 1;
823 |                   DECISION portal(p)!inBuffer(dp) = 0;
824                     (True): TASK    pBufferWait(dp) := Incl(p, pBufferWait(dp));
825                     ELSE:ENDDECISION;
826             ELSE:ENDDECISION;
827             RETURN;
828             ENDPROCEDURE;

830                                                                     /*--------------*/
831     PROCEDURE       Route_user;                                     /*  Route_user  */
832     FPAR            u               UserId,                         /*--------------*/
833             IN/OUT  p               PortalId;

835             DCL     c               ChannelId;
836             START
837             COMMENT'Return the portal id that leads toward a user.
838                     ';
839             TASK    p := 0;
840             DECISION u in uUsed;
841             (True): TASK    c := ChannelId(u),
842                             p := chan(c)!portal;
843             ELSE:ENDDECISION;
844             RETURN;
845             ENDPROCEDURE;

847                                                                     /*--------------------*/
848     PROCEDURE       Multicast_buffer;                               /*  Multicast_buffer  */
849     FPAR            b               BufferId,                       /*--------------------*/
850                     uSet            UserIdSet;

852             DCL     u               UserId,
853                     p               PortalId,
854                     pSet            PortalIdSet;
855             START
856             COMMENT'Send buffer containing an MCSPDU to multiple users.
857                     ';
858             TASK    pSet := Empty;
859              1b :   /* for u in uSet */
860             DECISION uSet = Empty;
861             (False):TASK    u := Pick(uSet),
862                             uSet := Del(u, uSet);
```

```
863                     CALL    Route_user(u, p);
864                     TASK    pSet := Incl(p, pSet);
865                     JOIN 1b;
866             ELSE:ENDDECISION;
867              2b :   /* for p in pSet */
868             DECISION pSet = Empty;
869             (False):TASK    p := Pick(pSet),
870                             pSet := Del(p, pSet);
871                     CALL    Cast_buffer(b, p);
872                     JOIN 2b;
873             ELSE:ENDDECISION;
874             RETURN;
875             ENDPROCEDURE;

877                                                                     /*--------------------*/
878     PROCEDURE       Broadcast_buffer;                               /*  Broadcast_buffer  */
879     FPAR            b               BufferId;                       /*--------------------*/

881             DCL     pdu             PDUStruct,
882                     p               PortalId,
883                     pSet            PortalIdSet;
884             START
885             COMMENT'Send buffer containing an MCSPDU to the whole subtree.
886                     ';
887             TASK    pdu := buffer(b)!pdu;
888 |           DECISION (pdu!kind = PCin and pdu!detachUserIds = Empty
889 |                                 and pdu!purgeChannelIds = Empty)
890                     or (pdu!kind = PTin and pdu!purgeTokenIds = Empty)
891                     or (pdu!kind = DUin and pdu!userIds = Empty);
892             (True): RETURN;
893             ELSE:ENDDECISION;
894             TASK    pSet := pUsed;
895              1b :   /* for p in pSet */
896             DECISION pSet = Empty;
897             (False):TASK    p := Pick(pSet),
898                             pSet := Del(p, pSet);
899                     DECISION portal(p)!kind;
900                     (Attached):
901                             DECISION pdu!kind = PDin;
902                             (False):CALL    Cast_buffer(b, p);
903                             ELSE:ENDDECISION;
904 |                   (Downlink):
905                             CALL    Cast_buffer(b, p);
906                     ELSE:ENDDECISION;
907                     JOIN 1b;
908             ELSE:ENDDECISION;
909             RETURN;
910             ENDPROCEDURE;

912                                                                     /*----------------*/
913     PROCEDURE       Crank_engine;                                   /*  Crank_engine  */
914                                                                     /*----------------*/
915             DCL     b               BufferId,
916                     p               PortalId,
917                     dp              DataPriority;
918             START
919             COMMENT'After individual processing of each input signal,
920                     look for the most deserving work to do next.
921                     This advances incrementally through stages of a merge;
922                     else it cleans up users, channels, tokens left behind.
923 |                   Buffers are assigned to accept new inputs, with preference
924 |                   to PDUs flowing downward and to higher priorities first.
925 *                   ';
926             TASK    b := 0;
927             DECISION pdSend;
928             (True): CALL    Allocate_buffer(b);
929                     DECISION b = 0;
930                     (True): RETURN;
931                     ELSE:ENDDECISION;
932                     TASK    pdSend := False,
933                             buffer(b)!pdu!kind := PDin,
934                             buffer(b)!pdu!heightLimit := parameters!maxHeight - 1;
935                     CALL    Broadcast_buffer(b);
936             ELSE:ENDDECISION;
937             DECISION edSend;
938             (True): CALL    Allocate_buffer(b);
939                     DECISION b = 0;
940                     (True): RETURN;
941                     ELSE:ENDDECISION;
```

```
                        TASK    edSend := False,
                                buffer(b)!pdu!kind := EDrq,
                                buffer(b)!pdu!subHeight := height,
                                buffer(b)!pdu!subInterval := maxInterval;
                        CALL    Cast_buffer(b, upward);
                ELSE:ENDDECISION;
+               CALL    Release_buffer(b);
+               TASK    dp := 0;
+                1b :   /* for dp = 0..? */
+               DECISION dp < parameters!numPriorities;
+               (True): DECISION upward in pBufferWait(dp);
+                       (False):TASK    dp := dp + 1;
+                               JOIN 1b;
+                       ELSE:ENDDECISION;
+                       TASK    p := upward,
+                               b := 0;
+                       CALL    Allocate_buffer(b);
+                       DECISION b = 0;
+                       (True): RETURN;
+                       ELSE:ENDDECISION;
+                       TASK    pBufferWait(dp) := Del(p, pBufferWait(dp)),
+                               buffer(b)!receiver := p,
+                               buffer(b)!dataPriority := dp,
+                               portal(p)!inCredit(dp) := portal(p)!inCredit(dp) - 1,
+                               portal(p)!inBuffer(dp) := b;
+                       OUTPUT  PDU.ready(dp) TO portal(p)!pids(dp);
+                       JOIN 1b;
+               ELSE:ENDDECISION;
+               CALL    Drop_portals;
                DECISION merging;
                (True): CALL    Merge_users;
                        DECISION uConfirm = uUsed;
                        (True): CALL    Merge_tokens;
|                               DECISION tConfirm = tUsed;
                                (True): CALL    Merge_channels;
+                                       DECISION cConfirm = cUsed;
+                                       (True): TASK    merging := False;
+                                       ELSE:ENDDECISION;
                                ELSE:ENDDECISION;
                        ELSE:ENDDECISION;
|                       DECISION merging;
|                       (True): RETURN;
|                       (False):TASK    dp := 0;
|                                2b :   /* for dp = 0..? */
                                DECISION dp < parameters!numPriorities;
|                               (True): /* for p in pInputWait(dp) */
                                        DECISION pInputWait(dp) = Empty;
|                                       (True): TASK    dp := dp + 1;
                                                JOIN 2b;
                                        ELSE:ENDDECISION;
|                                       TASK    p := Pick(pInputWait(dp)),
|                                               pInputWait(dp) := Del(p, pInputWait(dp));
|                                       CALL    Process_PDU(p, dp);
|                                       JOIN 2b;
                                ELSE:ENDDECISION;
|               ENDDECISION;
                ELSE:ENDDECISION;
|               CALL    Detach_users;
|               CALL    Leave_channels;
|               CALL    Disband_channels;
|               CALL    Reject_tokens;
                TASK    dp := 0;
                 3b :   /* for dp = 0..? */
                DECISION dp < parameters!numPriorities;
|               (True): /* for p in pBufferWait(dp) */
|                       DECISION pBufferWait(dp) = Empty;
|                       (True): TASK    dp := dp + 1;
|                               JOIN 3b;
|                       ELSE:ENDDECISION;
|                       TASK    p := Pick(pBufferWait(dp)),
|                               b := 0;
                        CALL    Allocate_buffer(b);
                        DECISION b = 0;
                        (True): RETURN;
                        ELSE:ENDDECISION;
                        TASK    pBufferWait(dp) := Del(p, pBufferWait(dp)),
                                buffer(b)!receiver := p,
                                buffer(b)!dataPriority := dp,
                                portal(p)!inCredit(dp) := portal(p)!inCredit(dp) - 1,
```

```
1021                            portal(p)!inBuffer(dp) := b;
1022                    OUTPUT  PDU.ready(dp) TO portal(p)!pids(dp);
1023                    JOIN 3b;
1024 *          ELSE:ENDDECISION;
1025            RETURN;
1026            ENDPROCEDURE;

1028                                                                    /*----------------*/
1029    PROCEDURE       Drop_portals;                                   /*  Drop_portals  */
1030                                                                    /*----------------*/
1031            DCL     b               BufferId,
1032                    p               PortalId,
1033                    pdu             PDUStruct;
1034            START
1035            COMMENT'Generate DPum MCSPDUs requested by controller.
1036                    ';
1037            TASK    b := 0,
1038                    pdu!kind := DPum,
1039                    pdu!reason := RN_user_requested;
1040             1b :   /* for p in pDrop */
1041            DECISION pDrop = Empty;
1042            (False):CALL    Allocate_buffer(b);
1043                    DECISION b = 0;
1044                    (True): RETURN;
1045                    ELSE:ENDDECISION;
1046                    TASK    p := Pick(pDrop),
1047                            pDrop := Del(p, pDrop),
1048                            buffer(b)!pdu := pdu;
1049                    CALL    Cast_buffer(b, p);
1050                    JOIN 1b;
1051            ELSE:ENDDECISION;
1052            CALL    Release_buffer(b);
1053            RETURN;
1054            ENDPROCEDURE;

1056                                                                    /*---------------*/
1057    PROCEDURE       Merge_users;                                    /*  Merge_users  */
1058                                                                    /*---------------*/
1059            DCL     b               BufferId,
1060                    u               UserId,
1061                    c               ChannelId,
1062                    pdu             PDUStruct,
1063                    cAttr           ChannelAttributes;
1064            START
1065            COMMENT'Generate MCrq MCSPDUs to merge user ids upward.
1066                    Fill them to maximum size that encoding allows.
1067                    ';
1068            TASK    b := 0,
1069                    pdu!kind := MCrq,
1070                    pdu!mergeChannels := Empty;
1071             1b :   /* for u in uMerge */
1072            DECISION uMerge = Empty;
1073            (False):CALL    Allocate_buffer(b);
1074                    DECISION b = 0;
1075                    (True): RETURN;
1076                    ELSE:ENDDECISION;
1077                    TASK    u := Pick(uMerge),
1078                            uMerge := Del(u, uMerge),
1079                            c := ChannelId(u),
1080                            cMerge := Del(c, cMerge),
1081                            cAttr!channelId := c,
1082                            cAttr!kind := UserId,
1083                            cAttr!joined := (chan(c)!joined /= Empty),
1084                            pdu!mergeChannels := Incl(cAttr, pdu!mergeChannels);
1085                    DECISION'encoding of pdu + 9 <= maxMCSPDUsize';
1086 |                  ('True'):JOIN 1b;
1087 *                  ELSE:ENDDECISION;
1088            ELSE:ENDDECISION;
1089            DECISION pdu!mergeChannels = Empty;
1090 |          (False):TASK    buffer(b)!pdu := pdu,
1091 |                          pdu!mergeChannels := Empty;
1092                    CALL    Cast_buffer(b, upward);
1093 +                  JOIN 1b;
1094            ELSE:ENDDECISION;
1095            CALL    Release_buffer(b);
1096            RETURN;
1097            ENDPROCEDURE;

1099                                                                    /*----------------*/
```

```
1100    PROCEDURE       Merge_channels;                                 /*  Merge_channels  */
1101                                                                    /*------------------*/
1102            DCL     b               BufferId,
1103 *                  c               ChannelId,
1104                    u               UserId,
1105                    pdu             PDUStruct,
1106                    cAttr           ChannelAttributes;
1107            START
1108            COMMENT'Generate MCrq MCSPDUs to merge channel ids upward.
1109                    Fill them to maximum size that encoding allows.
1110                    ';
1111            TASK    b := 0,
1112                    pdu!kind := MCrq,
1113                    pdu!mergeChannels := Empty,
1114 |                  pdu!purgeChannelIds := Empty;
1115 |           1b :   /* for c in cMerge */
1116 |          DECISION cMerge = Empty;
1117 |          (False):CALL    Allocate_buffer(b);
1118                    DECISION b = 0;
1119                    (True): RETURN;
1120                    ELSE:ENDDECISION;
1121 |                  TASK    c := Pick(cMerge),
1122 |                          cMerge := Del(c, cMerge),
1123                            cAttr!channelId := c,
1124                            cAttr!kind := chan(c)!kind,
1125                            cAttr!manager := chan(c)!manager,
1126                            cAttr!admitted := Empty,
1127                            cAttr!joined := (chan(c)!joined /= Empty);
1128 +                  DECISION (chan(c)!kind = Static or chan(c)!kind = Assigned)
1129 +                          and chan(c)!joined = Empty;
1130 +                  (True): CALL    Delete_channel(c);
1131 +                          JOIN 1b;
1132 +                  ELSE:ENDDECISION;
1133                    DECISION chan(c)!kind = Private;
1134                    (True): DECISION chan(c)!manager in uUsed;
1135                            (False):TASK    pdu!purgeChannelIds :=
1136 |                                                   Incl(c, pdu!purgeChannelIds);
1137 |                                   JOIN 3f;
1138                            ELSE:ENDDECISION;
1139 |                           2b :   /* for u in chan(c)!uMerge */
1140                            DECISION chan(c)!uMerge = Empty;
1141                            (False):TASK    u := Pick(chan(c)!uMerge),
1142                                            chan(c)!uMerge := Del(u, chan(c)!uMerge),
1143                                            cAttr!admitted := Incl(u, cAttr!admitted),
1144                                            pdu!mergeChannels :=       /* try this many */
1145                                                   Incl(cAttr, pdu!mergeChannels);
1146                                   DECISION'encoding of pdu + 4 <= maxMCSPDUsize';
1147                                   ('True'):TASK   pdu!mergeChannels := /* try another */
1148                                                   Del(cAttr, pdu!mergeChannels);
1149 |                                         JOIN 2b;
1150 *                                 ELSE:ENDDECISION;
1151                            ELSE:ENDDECISION;
1152                    ELSE:ENDDECISION;
1153                    TASK    pdu!mergeChannels := Incl(cAttr, pdu!mergeChannels);
1154 |                   3f :
1155 |                  DECISION'encoding of pdu + 23 <= maxMCSPDUsize';
1156 |                  ('True'):JOIN 1b;
1157 *                  ELSE:ENDDECISION;
1158            ELSE:ENDDECISION;
1159            DECISION pdu!mergeChannels = Empty and pdu!purgeChannelIds = Empty;
1160 |          (False):TASK    buffer(b)!pdu := pdu,
1161 |                          pdu!mergeChannels := Empty,
1162 |                          pdu!purgeChannelIds := Empty;
1163                    CALL    Cast_buffer(b, upward);
1164 +                  JOIN 1b;
1165            ELSE:ENDDECISION;
1166            CALL    Release_buffer(b);
1167            RETURN;
1168            ENDPROCEDURE;

1170                                                                    /*----------------*/
1171    PROCEDURE       Merge_tokens;                                   /*  Merge_tokens  */
1172                                                                    /*----------------*/
1173            DCL     b               BufferId,
1174 *                  t               TokenId,
1175                    u               UserId,
1176                    pdu             PDUStruct,
1177                    tAttr           TokenAttributes;
1178            START
```

```
1179            COMMENT'Generate MTrq MCSPDUs to merge token ids upward.
1180                    Fill them to maximum size that encoding allows.
1181                    ';
1182            TASK    b := 0,
1183                    pdu!kind := MTrq,
1184                    pdu!mergeTokens := Empty,
1185                    pdu!purgeTokenIds := Empty,
1186 |           1b :   /* for t in tMerge */
1187 |          DECISION tMerge = Empty;
1188 |          (False):CALL    Allocate_buffer(b);
1189                    DECISION b = 0;
1190                    (True): RETURN;
1191                    ELSE:ENDDECISION;
1192 |                  TASK    t := Pick(tMerge),
1193 |                          tMerge := Del(t, tMerge),
1194                            tAttr!tokenId := t,
1195                            tAttr!kind := token(t)!kind,
1196                            tAttr!grabber := token(t)!grabber,
1197                            tAttr!recipient := token(t)!recipient,
1198                            tAttr!inhibitors := Empty;
1199                  DECISION token(t)!kind = Inhibited;
1200 |                (True):  2b :   /* for u in token(t)!uMerge */
1201                            DECISION token(t)!uMerge = Empty;
1202                            (False):TASK    u := Pick(token(t)!uMerge),
1203                                            token(t)!uMerge := Del(u, token(t)!uMerge),
1204                                            tAttr!inhibitors := Incl(u, tAttr!inhibitors),
1205                                            pdu!mergeTokens :=       /* try this many */
1206                                                              Incl(tAttr, pdu!mergeTokens);
1207                                    DECISION'encoding of pdu + 4 <= maxMCSPDUsize';
1208                                    ('True'):TASK   pdu!mergeTokens := /* try another */
1209                                                              Del(tAttr, pdu!mergeTokens);
1210 |                                          JOIN 2b;
1211 *                                  ELSE:ENDDECISION;
1212                            ELSE:ENDDECISION;
1213                  ELSE:ENDDECISION;
1214 |                TASK    pdu!mergeTokens := Incl(tAttr, pdu!mergeTokens);
1215 |                DECISION'encoding of pdu + 16 <= maxMCSPDUsize';
1216 |                ('True'):JOIN 1b;
1217 *                ELSE:ENDDECISION;
1218           ELSE:ENDDECISION;
1219           DECISION pdu!mergeTokens = Empty;
1220 |         (False):TASK    buffer(b)!pdu := pdu,
1221 |                         pdu!mergeTokens := Empty;
1222                   CALL    Cast_buffer(b, upward);
1223 +                 JOIN 1b;
1224           ELSE:ENDDECISION;
1225           CALL    Release_buffer(b);
1226           RETURN;
1227           ENDPROCEDURE;

1229                                                                   /*----------------*/
1230   PROCEDURE       Detach_users;                                   /*  Detach_users  */
1231                                                                   /*----------------*/
1232           DCL     b               BufferId,
1233                   u               UserId,
1234 |                 pdu             PDUStruct,
1235 |                 diagnostic      Diagnostic;
1236           START
1237           COMMENT'Generate DUrq MCSPDUs to detach users.
1238                   Fill them to maximum size that encoding allows.
1239                   ';
1240           TASK    b := 0,
1241                   pdu!kind := DUrq,
1242                   pdu!reason := RN_domain_disconnected,
1243                   pdu!userIds := Empty;
1244            1b :   /* for u in uDetach */
1245           DECISION uDetach = Empty;
1246           (False):CALL    Allocate_buffer(b);
1247                   DECISION b = 0;
1248                   (True): RETURN;
1249                   ELSE:ENDDECISION;
1250                   TASK    u := Pick(uDetach),
1251                           uDetach := Del(u, uDetach),
1252                           pdu!userIds := Incl(u, pdu!userIds);
1253                   DECISION'encoding of pdu + 4 <= maxMCSPDUsize';
1254 |                 ('True'):JOIN 1b;
1255 *                 ELSE:ENDDECISION;
1256           ELSE:ENDDECISION;
1257           DECISION pdu!userIds = Empty;
```

```
1258 |          (False):TASK    buffer(b)!pdu := pdu,
1259 |                          pdu!userIds := Empty;
1260                    DECISION upward = 0;
1261                    (False):CALL    Cast_buffer(b, upward);
1262                    (True): TASK    buffer(b)!pdu!kind := DUin;
1263 |                          CALL    Apply_PDU(0, b, diagnostic);
1264                    ENDDECISION;
1265 +                  JOIN 1b;
1266            ELSE:ENDDECISION;
1267            CALL    Release_buffer(b);
1268            RETURN;
1269            ENDPROCEDURE;

1271                                                                    /*------------------*/
1272    PROCEDURE       Leave_channels;                                 /*  Leave_channels  */
1273                                                                    /*------------------*/
1274            DCL     b               BufferId,
1275                    c               ChannelId,
1276                    pdu             PDUStruct;
1277            START
1278            COMMENT'Generate CLrq MCSPDUs to leave channels.
1279                    Fill them to maximum size that encoding allows.
1280                    ';
1281            TASK    b := 0,
1282                    pdu!kind := CLrq,
1283                    pdu!channelIds := Empty;
1284             1b :   /* for c in cLeave */
1285            DECISION cLeave = Empty;
1286            (False):CALL    Allocate_buffer(b);
1287                    DECISION b = 0;
1288                    (True): RETURN;
1289                    ELSE:ENDDECISION;
1290                    TASK    c := Pick(cLeave),
1291                            cLeave := Del(c, cLeave);
1292                    DECISION chan(c)!kind;
1293                    (Static, Assigned):
1294                            CALL    Delete_channel(c);
1295                    ELSE:ENDDECISION;
1296                    TASK    pdu!channelIds := Incl(c, pdu!channelIds);
1297                    DECISION'encoding of pdu + 4 <= maxMCSPDUsize';
1298 |                  ('True'):JOIN 1b;
1299 *                  ELSE:ENDDECISION;
1300            ELSE:ENDDECISION;
1301            DECISION pdu!channelIds = Empty;
1302 |          (False):TASK    buffer(b)!pdu := pdu,
1303 |                          pdu!channelIds := Empty;
1304                    CALL    Cast_buffer(b, upward);
1305 +                  JOIN 1b;
1306            ELSE:ENDDECISION;
1307            CALL    Release_buffer(b);
1308            RETURN;
1309            ENDPROCEDURE;

1311                                                                    /*--------------------*/
1312    PROCEDURE       Disband_channels;                               /*  Disband_channels  */
1313                                                                    /*--------------------*/
1314            DCL     b               BufferId,
1315                    c               ChannelId,
1316                    pdu             PDUStruct,
1317                    uSet            UserIdSet;
1318            START
1319            COMMENT'Generate CDin MCSPDUs to disband channels.
1320                    ';
1321            TASK    b := 0,
1322                    pdu!kind := CDin;
1323             1b :   /* for c in cDisband */
1324            DECISION cDisband = Empty;
1325            (False):CALL    Allocate_buffer(b);
1326                    DECISION b = 0;
1327                    (True): RETURN;
1328                    ELSE:ENDDECISION;
1329                    TASK    c := Pick(cDisband),
1330                            cDisband := Del(c, cDisband),
1331                            pdu!channelId := c;
1332                    TASK    buffer(b)!pdu := pdu,
1333                            uSet := chan(c)!admitted;
1334                    CALL    Delete_channel(c);
1335                    CALL    Multicast_buffer(b, uSet);
1336                    JOIN 1b;
```

```
1337            ELSE:ENDDECISION;
1338            CALL    Release_buffer(b);
1339            RETURN;
1340            ENDPROCEDURE;

1342                                                                    /*-----------------*/
1343    PROCEDURE       Reject_tokens;                                  /*  Reject_tokens  */
1344                                                                    /*-----------------*/
1345            DCL     b               BufferId,
1346                    t               TokenId,
1347                    u               UserId,
1348                    p               PortalId,
1349                    pdu             PDUStruct;
1350            START
1351            COMMENT'Generate TVcf MCSPDUs to reject tokens.
1352                    ';
1353            TASK    b := 0,
1354                    pdu!kind := TVcf,
1355                    pdu!result := RT_no_such_user;
1356             1b :   /* for t in tReject */
1357            DECISION tReject = Empty;
1358            (False):CALL    Allocate_buffer(b);
1359                    DECISION b = 0;
1360                    (True): RETURN;
1361                    ELSE:ENDDECISION;
1362                    TASK    t := Pick(tReject),
1363                            tReject := Del(t, tReject),
1364                            token(t)!kind := Grabbed,
1365                            u := token(t)!grabber,
1366                            pdu!initiator := u,
1367                            pdu!tokenId := t;
1368                    CALL    Token_status(pdu);
1369                    TASK    buffer(b)!pdu := pdu;
1370                    CALL    Route_user(u, p);
1371                    CALL    Cast_buffer(b, p);
1372                    JOIN 1b;
1373            ELSE:ENDDECISION;
1374            CALL    Release_buffer(b);
1375            RETURN;
1376            ENDPROCEDURE;

1378                                                                    /*---------------*/
1379    PROCEDURE       Process_PDU;                                    /*  Process_PDU  */
1380    FPAR            r               PortalId,                       /*---------------*/
1381 |                  dp              DataPriority;

1383 |          DCL     b               BufferId,
1384 |                  pdu             PDUStruct,
1385                    diagnostic      Diagnostic;
1386            START
1387            COMMENT'This is the main line of processing an MCSPDU.
1388                    If its content is invalid, ignore or reject it.
1389                    Else if this is the top provider, take the special
1390                    actions of Top_provider.  Then whether or not this
1391                    is the top provider, call Apply_PDU.
1392                    ';
1393 |          TASK    b := portal(r)!inBuffer(dp),
1394 |                  pdu := buffer(b)!pdu;
1395            DECISION pdu!kind = RJum;
1396            (True): TASK    diagnostic := pdu!diagnostic;
1397            (False):CALL    Validate_input(r, b, diagnostic);
1398 +                  DECISION diagnostic = DC_OK;
1399 +                  (True): DECISION buffer(b)!pdu!kind;
1400 +                          (MCrq): TASK    mcrqQueue := Push(r, mcrqQueue);
1401 +                          (MTrq): TASK    mtrqQueue := Push(r, mtrqQueue);
1402 +                          (AUrq): TASK    aurqQueue := Push(r, aurqQueue);
1403 +                          ELSE:ENDDECISION;
1404 +                          DECISION upward = 0;
1405 +                          (True): CALL    Top_provider(r, b);
1406 +                          ELSE:ENDDECISION;
1407 +                          CALL    Apply_PDU(r, b, diagnostic);
1408 +                  ELSE:ENDDECISION;
1409            ENDDECISION;
1410            DECISION diagnostic;
1411 |          (DC_OK, DC_ignore):
1412                    /* no action */
1413            ELSE:
1414                    OUTPUT  Report.portal(r, diagnostic) TO control;
1415                    DECISION pdu!kind = RJum;
```

```
1416                    (False):TASK'pdu!initialOctets := truncate pdu';
1417                    ELSE:ENDDECISION;
1418                    TASK    pdu!kind := RJum,
1419                            pdu!diagnostic := diagnostic,
1420                            buffer(b)!pdu := pdu,
1421                            dp := buffer(b)!dataPriority,
1422                            buffer(b)!dataPriority := 0;
1423                    CALL    Cast_buffer(b, r);
1424                    TASK    buffer(b)!dataPriority := dp;
1425            ENDDECISION;
1426            CALL    Release_buffer(b);
1427 +          TASK    portal(r)!inBuffer(dp) := 0;
1428 +          DECISION portal(r)!inCredit(dp) > 0;
1429 +          (True): TASK    pBufferWait(dp) := Incl(r, pBufferWait(dp));
1430 +          ELSE:ENDDECISION;
1431            RETURN;
1432            ENDPROCEDURE;

1434                                                                    /*------------------*/
1435    PROCEDURE       Validate_input;                                 /*  Validate_input  */
1436    FPAR            r               PortalId,                       /*------------------*/
1437                    b               BufferId,
1438            IN/OUT  diagnostic      Diagnostic;

1440            DCL     pdu             PDUStruct,
1441 +                  p               PortalId,
1442                    dp              DataPriority,
1443                    u               UserId,
1444                    uSet            UserIdSet,
1445                    c               ChannelId,
1446                    cSet            ChannelIdSet,
1447                    cAttr           ChannelAttributes,
1448                    cAttrSet        ChannelAttributesSet,
1449                    t               TokenId,
1450                    tSet            TokenIdSet,
1451                    tAttr           TokenAttributes,
1452                    tAttrSet        TokenAttributesSet;
1453            START
1454            COMMENT'Validate the direction and user ids of an MCSPDU
1455                    and perform other checks, depending on its type.
1456                    Top_provider and Apply_PDU trust this procedure
1457                    to catch important anomalies and ease their logic.
1458                    ';
1459            TASK    pdu := buffer(b)!pdu,
1460                    diagnostic := DC_OK;
1461            DECISION portal(r)!kind;
1462 |          (Downlink):
1463                    DECISION pdu!kind;
1464                    (EDrq, MCrq, MTrq, DPum, RJum,
1465                     AUrq, DUrq, CJrq, CLrq, CCrq,
1466                     CDrq, CArq, CErq, SDrq, USrq,
1467                     TGrq, TIrq, TVrq, TVrs, TPrq,
1468                     TRrq, TTrq):
1469                            /* OK */
1470                    ELSE:
1471                            TASK    diagnostic := DC_forbidden_PDU_upward;
1472                            RETURN;
1473                    ENDDECISION;
1474 |          (Uplink):
1475                    DECISION pdu!kind;
1476                    (PDin, MCcf, PCin, MTcf, PTin,
1477                     DPum, RJum, AUcf, DUin, CJcf,
1478                     CCcf, CDin, CAin, CEin, SDin,
1479                     USin, TGcf, TIcf, TVin, TVcf,
1480                     TPin, TRcf, TTcf):
1481                            /* OK */
1482                    ELSE:
1483                            TASK    diagnostic := DC_forbidden_PDU_downward;
1484                            RETURN;
1485                    ENDDECISION;
1486            ELSE:ENDDECISION;
1487            TASK    dp := 0;
1488            DECISION pdu!kind;
1489            (SDrq, SDin, USrq, USin):
1490                    TASK    dp := pdu!dataPriority,
1491                            dp := IF dp < parameters!numPriorities THEN dp
1492                                            ELSE parameters!numPriorities - 1 FI;
1493            ELSE:ENDDECISION;
1494            DECISION buffer(b)!dataPriority = dp;
```

```
1495            (False):TASK    diagnostic := DC_wrong_transport_priority;
1496                    RETURN;
1497            ELSE:ENDDECISION;
1498            DECISION pdu!kind;
1499            (MCrq): TASK    cAttrSet := pdu!mergeChannels,
1500                            cSet := pdu!purgeChannelIds;
1501                     1b :   /* for cAttr in cAttrSet */
1502                    DECISION cAttrSet = Empty;
1503                    (False):TASK    cAttr := Pick(cAttrSet),
1504                                    cAttrSet := Del(cAttr, cAttrSet),
1505                                    c := cAttr!channelId;
1506                            DECISION c in cSet;
1507                            (True): TASK    diagnostic := DC_inconsistent_merge;
1508                                    RETURN;
1509                            ELSE:ENDDECISION;
1510                            TASK    cSet := Incl(c, cSet);
1511                            DECISION cAttr!kind = Private;
1512                            (False):JOIN 1b;
1513                            ELSE:ENDDECISION;
1514                            TASK    pdu!mergeChannels := Del(cAttr, pdu!mergeChannels);
1515                            CALL    Route_user(cAttr!manager, p);
1516                            DECISION p = r;
1517 |                          (False):TASK    pdu!purgeChannelIds :=
1518 |                                                  Incl(c, pdu!purgeChannelIds);
1519                                    JOIN 1b;
1520                            ELSE:ENDDECISION;
1521                            TASK    uSet := cAttr!admitted;
1522                             2b :   /* for u in uSet */
1523                            DECISION uSet = Empty;
1524                            (False):TASK    u := Pick(uSet),
1525                                            uSet := Del(u, uSet);
1526                                    CALL    Route_user(u, p);
1527                                    DECISION p = r;
1528                                    (False):TASK    cAttr!admitted :=
1529                                                    Del(u, cAttr!admitted);
1530                                    ELSE:ENDDECISION;
1531                                    JOIN 2b;
1532                            ELSE:ENDDECISION;
1533                            TASK    pdu!mergeChannels := Incl(cAttr, pdu!mergeChannels);
1534                            JOIN 1b;
1535                    ELSE:ENDDECISION;
1536            (MTrq): TASK    tAttrSet := pdu!mergeTokens,
1537                            tSet := pdu!purgeTokenIds;
1538 |                   3b :   /* for tAttr in tAttrSet */
1539                    DECISION tAttrSet = Empty;
1540                    (False):TASK    tAttr := Pick(tAttrSet),
1541                                    tAttrSet := Del(tAttr, tAttrSet),
1542                                    t := tAttr!tokenId;
1543                            DECISION t in tSet;
1544                            (True): TASK    diagnostic := DC_inconsistent_merge;
1545                                    RETURN;
1546                            ELSE:ENDDECISION;
1547                            TASK    tSet := Incl(t, tSet),
1548                                    pdu!mergeTokens := Del(tAttr, pdu!mergeTokens);
1549                            DECISION tAttr!kind;
1550                            (Grabbed, Ungivable):
1551                                    CALL    Route_user(tAttr!grabber, p);
1552                                    DECISION p = r;
1553 |                                  (False):JOIN 4f;
1554                                    ELSE:ENDDECISION;
1555                            (Given):
1556                                    CALL    Route_user(tAttr!recipient, p);
1557                                    DECISION p = r;
1558 |                                  (False):JOIN 4f;
1559                                    ELSE:ENDDECISION;
1560                            (Giving):
1561                                    CALL    Route_user(tAttr!recipient, p);
1562 *                                  DECISION p = r;
1563                                    (True): CALL    Route_user(tAttr!grabber, p);
1564                                            DECISION p = r;
1565                                            (False):TASK    tAttr!kind := Given;
1566                                            ELSE:ENDDECISION;
1567 +                                  (False):TASK    tAttr!kind := Ungivable;
1568 +                                          CALL    Route_user(tAttr!grabber, p);
1569 +                                          DECISION p = r;
1570 +                                          (False): 4f :
1571 +                                                  TASK    pdu!purgeTokenIds :=
1572 +                                                               Incl(t, pdu!purgeTokenIds);
1573 +                                                  JOIN 3b;
```

```
1574 +                                          ELSE:ENDDECISION;
1575                                    ENDDECISION;
1576                            (Inhibited):
1577                                    TASK    uSet := tAttr!inhibitors;
1578                                     5b :   /* for u in uSet */
1579                                    DECISION uSet = Empty;
1580                                    (False):TASK    u := Pick(uSet),
1581                                                    uSet := Del(u, uSet);
1582                                            CALL    Route_user(u, p);
1583                                            DECISION p = r;
1584                                            (False):TASK    tAttr!inhibitors :=
1585                                                               Del(u, tAttr!inhibitors);
1586                                            ELSE:ENDDECISION;
1587                                            JOIN 5b;
1588 *                                  ELSE:ENDDECISION;
1589                            ENDDECISION;
1590                            TASK    pdu!mergeTokens := Incl(tAttr, pdu!mergeTokens);
1591 |                          JOIN 3b;
1592                    ELSE:ENDDECISION;
1593            (DUrq): TASK    uSet := pdu!userIds;
1594 |                   6b :   /* for u in uSet */
1595                    DECISION uSet = Empty;
1596                    (False):TASK    u := Pick(uSet),
1597                                    uSet := Del(u, uSet);
1598                            CALL    Route_user(u, p);
1599                            DECISION p = r;
1600                            (False):TASK    pdu!userIds := Del(u, pdu!userIds);
1601                            ELSE:ENDDECISION;
1602 |                          JOIN 6b;
1603                    ELSE:ENDDECISION;
1604 +                  DECISION pdu!userIds = Empty;
1605 +                  (True): TASK    diagnostic := DC_ignore;
1606 +                          RETURN;
1607 +                  ELSE:ENDDECISION;
1608            (CJrq, CCrq, CDrq, CArq, CErq,
1609             SDrq, USrq, TGrq, TIrq, TVrq,
1610             TPrq, TRrq, TTrq):
1611                    CALL    Route_user(pdu!initiator, p);
1612                    DECISION p = r;
1613                    (False):TASK    diagnostic := DC_ignore;
1614                            RETURN;
1615                    ELSE:ENDDECISION;
1616            (TVin): DECISION pdu!recipient in uUsed;
1617                    (False):TASK    diagnostic := DC_misrouted_user;
1618                            RETURN;
1619                    ELSE:ENDDECISION;
1620            (TVrs): CALL    Route_user(pdu!recipient, p);
1621                    DECISION p = r;
1622                    (False):TASK    diagnostic := DC_ignore;
1623                            RETURN;
1624                    ELSE:ENDDECISION;
1625                    DECISION portal(p)!kind = Attached and pdu!result /= RT_successful;
1626                    (True): TASK    pdu!result := RT_user_rejected;
1627                    ELSE:ENDDECISION;
1628            (CJcf, CCcf, TGcf, TIcf, TVcf,
1629             TRcf, TTcf):
1630                    DECISION pdu!initiator in uUsed;
1631                    (False):TASK    diagnostic := DC_misrouted_user;
1632                            RETURN;
1633                    ELSE:ENDDECISION;
1634            ELSE:ENDDECISION;
1635            DECISION pdu!kind;
1636            (CDrq, CArq, CErq):
1637                    TASK    c := pdu!channelId;
1638                    DECISION c in cUsed and chan(c)!kind = Private
1639                            and pdu!initiator = chan(c)!manager;
1640                    (False):TASK    diagnostic := DC_ignore;
1641                            RETURN;
1642                    ELSE:ENDDECISION;
1643            (SDrq, USrq):
1644                    TASK    c := pdu!channelId;
1645                    DECISION c in cUsed and chan(c)!kind = Private;
1646                    (True): DECISION pdu!initiator in chan(c)!admitted;
1647                            (False):TASK    diagnostic := DC_ignore;
1648                                    RETURN;
1649                            ELSE:ENDDECISION;
1650                    ELSE:ENDDECISION;
1651 |          (MCcf): DECISION not merging and mcrqQueue = EmptyQueue;
1652                    (True): TASK    diagnostic := DC_unrequested_confirm;
```

```
1653                            RETURN;
1654                    ELSE:ENDDECISION;
1655 |          (MTcf): DECISION not merging and mtrqQueue = EmptyQueue;
1656                    (True): TASK    diagnostic := DC_unrequested_confirm;
1657                            RETURN;
1658                    ELSE:ENDDECISION;
1659 |          (AUcf): DECISION aurqQueue = EmptyQueue;
1660                    (True): TASK    diagnostic := DC_unrequested_confirm;
1661                            RETURN;
1662                    ELSE:ENDDECISION;
1663            (CJcf, CCcf, TGcf, TIcf, TVcf,
1664             TRcf, TTcf):
1665                    DECISION merging;
1666                    (True): TASK    diagnostic := DC_unrequested_confirm;
1667 |                          RETURN;
1668                    ELSE:ENDDECISION;
1669            ELSE:ENDDECISION;
1670            TASK    buffer(b)!pdu := pdu;
1671            RETURN
1672 |          COMMENT'Additional tests might be applied to check
1673 |                  that MCSPDUs flowing downward are consistent
1674 |                  with user or channel states already recorded.
1675 |                  But stronger assertions could contain flaws,
1676 |                  and such defenses are not necessary for the
1677                    continued functioning of this MCS provider.
1678 |                  The value of an MCS domain involves trust in
1679                    the correct operation of superior providers.
1680                    ';
1681            ENDPROCEDURE;

1683                                                                    /*----------------*/
1684    PROCEDURE       Top_provider;                                   /*  Top_provider  */
1685    FPAR            r               PortalId,                       /*----------------*/
1686                    b               BufferId;

1688            DCL     pdu             PDUStruct,
1689 *                  new             PDUStruct,
1690                    u               UserId,
1691 +                  c               ChannelId,
1692                    cAttr           ChannelAttributes,
1693 +                  t               TokenId,
1694                    tAttr           TokenAttributes,
1695 |                  result          Result,
1696 |                  diagnostic      Diagnostic;
1697            START
1698            COMMENT'The buffer containing an MCSPDU will be processed next
1699                    by Apply_PDU.  Its contents may first be modified here.
1700                    Changing pdu!kind results in "turning the corner".
1701                    ';
1702            TASK    pdu := buffer(b)!pdu;
1703            DECISION pdu!kind;
1704            (EDrq): DECISION pdu!subHeight < parameters!maxHeight;
1705                    (False):TASK    pdSend := True;
1706                    ELSE:ENDDECISION;
1707            (MCrq): TASK    new!kind := MCcf,
1708                            new!mergeChannels := Empty,
1709                            new!purgeChannelIds := pdu!purgeChannelIds;
1710                     1b :   /* for cAttr in mergeChannels */
1711                    DECISION pdu!mergeChannels = Empty;
1712                    (False):TASK    cAttr := Pick(pdu!mergeChannels),
1713                                    pdu!mergeChannels := Del(cAttr, pdu!mergeChannels),
1714                                    c := cAttr!channelId;
1715                            DECISION c in cUsed;
1716                            (True): DECISION chan(c)!kind;
1717                                    (Static):
1718                                            JOIN 2f;
1719                                    (Private):
1720                                            DECISION cAttr!kind = Private
1721                                                    and cAttr!manager = chan(c)!manager;
1722                                            (True): JOIN 2f;
1723                                            ELSE:ENDDECISION;
1724                                    ELSE:ENDDECISION;
1725 |                          (False):DECISION cAttr!kind = UserId;
1726 |                                  (True): CALL    Take_user(c, diagnostic);
1727 |                                  (False):CALL    Take_channel(c, diagnostic);
1728 |                                  ENDDECISION;
1729 |                                  DECISION diagnostic = DC_OK;
1730 |                                  (True): TASK    chan(c)!kind := cAttr!kind;
1731 |                                           2f :
```

```
1732 |                                          TASK    new!mergeChannels :=
1733 |                                                  Incl(cAttr, new!mergeChannels);
1734 |                                          JOIN 1b;
1735                                    ELSE:ENDDECISION;
1736                            ENDDECISION;
1737                            TASK    new!purgeChannelIds := Incl(c, new!purgeChannelIds);
1738                            JOIN 1b;
1739                    ELSE:ENDDECISION;
1740                    TASK    pdu := new;
1741            (MTrq): TASK    new!kind := MTcf,
1742                            new!mergeTokens := Empty,
1743                            new!purgeTokenIds := pdu!purgeTokenIds;
1744                     3b :   /* for tAttr in mergeTokens */
1745                    DECISION pdu!mergeTokens = Empty;
1746                    (False):TASK    tAttr := Pick(pdu!mergeTokens),
1747                                    pdu!mergeTokens := Del(tAttr, pdu!mergeTokens),
1748                                    t := tAttr!tokenId;
1749                            DECISION t in tUsed;
1750                            (True): DECISION token(t)!kind;
1751                                    (Inhibited):
1752                                            DECISION tAttr!kind = Inhibited;
1753                                            (True): JOIN 4f;
1754                                            ELSE:ENDDECISION;
1755                                    ELSE:ENDDECISION;
1756 |                          (False):CALL    Take_token(t, diagnostic);
1757 |                                  DECISION diagnostic = DC_OK;
1758 |                                  (True):  4f :
1759 |                                          TASK    new!mergeTokens :=
1760 |                                                  Incl(tAttr, new!mergeTokens);
1761                                          JOIN 3b;
1762                                  ELSE:ENDDECISION;
1763                          ENDDECISION;
1764                          TASK    new!purgeTokenIds := Incl(t, new!purgeTokenIds);
1765                          JOIN 3b;
1766                  ELSE:ENDDECISION;
1767                  TASK    pdu := new;
1768          (AUrq): CALL    New_user(u, result);
1769                  TASK    pdu!kind := AUcf,
1770                          pdu!result := result,
1771                          pdu!initiator := u;
1772          (DUrq): TASK    pdu!kind := DUin;
1773          (CJrq): TASK    pdu!kind := CJcf,
1774 |                        c := pdu!channelId,
1775 |                        pdu!requested := c,
1776 |                        result := RT_successful;
1777                  DECISION c = 0;
1778                  (True): CALL    New_channel(c, result);
1779                          DECISION result = RT_successful;
1780 |                        (True): TASK    chan(c)!kind := Assigned;
1781                          ELSE:ENDDECISION;
1782 |                        TASK    pdu!channelId := c;
1783 |                 (False):DECISION c in cUsed;
1784                          (False):DECISION c < 1001;
1785                                  (False):TASK    result := RT_no_such_channel;
1786 |                                (True): CALL    Take_chanel(c, diagnostic);
1787 |                                        DECISION diagnostic = DC_OK;
1788                                        (False):TASK    result := RT_too_many_channels;
1789 |                                      (True): TASK    chan(c)!kind := Static;
1790 |                                      ENDDECISION;
1791                                  ENDDECISION;
1792                          (True): DECISION chan(c)!kind;
1793                                  (UserId):
1794                                          TASK    u := UserId(c);
1795                                          DECISION pdu!initiator = u;
1796                                          (False):TASK    result := RT_other_user_id;
1797                                          ELSE:ENDDECISION;
1798                                  (Private):
1799                                          DECISION pdu!initiator in chan(c)!admitted;
1800                                          (False):TASK    result := RT_not_admitted;
1801                                          ELSE:ENDDECISION;
1802                                  ELSE:ENDDECISION;
1803                          ENDDECISION;
1804 +                        DECISION result = RT_successful;
1805 +                        (False):TASK    pdu!channelId := 0;
1806 +                        ELSE:ENDDECISION;
1807                  ENDDECISION;
1808                  TASK    pdu!result := result;
1809          (CLrq): /* no special action */
1810          (CCrq): CALL    New_channel(c, result);
```

```
1811                    TASK    pdu!kind := CCcf,
1812                            pdu!result := result,
1813                            pdu!channelId := c;
1814            (CDrq): TASK    pdu!kind := CDin;
1815            (CArq): TASK    pdu!kind := CAin;
1816            (CErq): TASK    pdu!kind := CEin;
1817            (SDrq): TASK    pdu!kind := SDin;
1818            (USrq): TASK    pdu!kind := USin;
1819            (TGrq): TASK    pdu!kind := TGcf,
1820                            pdu!result := RT_successful,
1821                            t := pdu!tokenId,
1822                            u := pdu!initiator;
1823 |                  DECISION t in tUsed;
1824 |                  (False):CALL    Take_token(t, diagnostic);
1825 |                          DECISION diagnostic = DC_OK;
1826                            (False):TASK    pdu!result := RT_too_many_tokens;
1827 |                          (True): TASK    token(t)!kind := Grabbed,
1828                                            token(t)!grabber := u;
1829                            ENDDECISION;
1830 |                  (True): DECISION token(t)!kind = Inhibited
1831                                    and token(t)!inhibitors = Incl(u, Empty);
1832                            (False):TASK    pdu!result := RT_token_not_available;
1833                            (True): TASK    token(t)!kind := Grabbed,
1834                                            token(t)!grabber := u;
1835                            ENDDECISION;
1836                   ENDDECISION;
1837                   CALL     Token_status(pdu);
1838            (TIrq): TASK    pdu!kind := TIcf,
1839                            pdu!result := RT_successful,
1840                            t := pdu!tokenId,
1841                            u := pdu!initiator;
1842 |                  DECISION t in tUsed;
1843 |                  (False):CALL    Take_token(t, diagnostic);
1844 |                          DECISION diagnostic = DC_OK;
1845                            (False):TASK    pdu!result := RT_too_many_tokens;
1846 |                          (True): TASK    token(t)!kind := Inhibited,
1847                                            token(t)!inhibitors := Incl(u, Empty);
1848                            ENDDECISION;
1849 |                  (True): DECISION token(t)!kind = Grabbed and token(t)!grabber = u;
1850                            (True): TASK    token(t)!kind := Inhibited,
1851                                            token(t)!inhibitors := Incl(u, Empty);
1852                            ELSE:ENDDECISION;
1853                            DECISION token(t)!kind = Inhibited;
1854                            (False):TASK    pdu!result := RT_token_not_available;
1855                            (True): TASK    token(t)!inhibitors :=
1856                                                    Incl(u, token(t)!inhibitors);
1857                            ENDDECISION;
1858                   ENDDECISION;
1859                   CALL     Token_status(pdu);
1860            (TVrq): TASK    pdu!kind := TVcf,
1861                            pdu!result := RT_token_not_possessed,
1862                            t := pdu!tokenId,
1863                            u := pdu!initiator;
1864                   DECISION t in tUsed and token(t)!kind = Grabbed and token(t)!grabber = u;
1865                   (True): DECISION pdu!recipient in uUsed;
1866                            (False):TASK    pdu!result := RT_no_such_user;
1867                            (True): TASK    pdu!kind := TVin,
1868                                            token(t)!kind := Giving,
1869                                            token(t)!recipient := pdu!recipient;
1870                            ENDDECISION;
1871                   ELSE:ENDDECISION;
1872                   CALL     Token_status(pdu);
1873            (TPrq): TASK    pdu!kind := TPin;
1874            (TRrq): TASK    pdu!kind := TRcf,
1875                            pdu!result := RT_token_not_possessed,
1876                            t := pdu!tokenId,
1877                            u := pdu!initiator;
1878                   DECISION t in tUsed;
1879                   (True): DECISION token(t)!kind;
1880 |                          (Grabbed, Ungivable):
1881                                    DECISION token(t)!grabber = u;
1882                                    (True): TASK    pdu!result := RT_successful;
1883                                            CALL    Delete_token(t);
1884                                    ELSE:ENDDECISION;
1885                            (Giving):
1886                                    DECISION token(t)!grabber = u;
1887                                    (True): TASK    pdu!result := RT_successful,
1888                                                    token(t)!kind := Given;
1889 *                                  ELSE:ENDDECISION;
```

```
1890                            (Inhibited):
1891                                    DECISION u in token(t)!inhibitors;
1892                                    (True): TASK    pdu!result := RT_successful,
1893                                                    token(t)!inhibitors :=
1894                                                            Del(u, token(t)!inhibitors);
1895                                    ELSE:ENDDECISION;
1896                                    DECISION token(t)!inhibitors = Empty;
1897                                    (True): CALL    Delete_token(t);
1898                                    ELSE:ENDDECISION;
1899                            ENDDECISION;
1900                    ELSE:ENDDECISION;
1901                    CALL    Token_status(pdu);
1902            (TTrq): TASK    pdu!kind := TTcf;
1903                    CALL    Token_status(pdu);
1904            (TVrs): TASK    t := pdu!tokenId,
1905                            u := pdu!recipient;
1906                    DECISION t in tUsed and token(t)!recipient = u;
1907                    (True): DECISION token(t)!kind;
1908                            (Giving):
1909                                    TASK    token(t)!kind := Grabbed,
1910                                            pdu!kind := TVcf,
1911                                            pdu!initiator := token(t)!grabber;
1912                                    DECISION pdu!result = RT_successful;
1913                                    (True): TASK    token(t)!grabber := u;
1914                                    ELSE:ENDDECISION;
1915 *                                  CALL    Token_status(pdu);
1916                            ELSE:ENDDECISION;
1917                    ELSE:ENDDECISION;
1918            ELSE:ENDDECISION;
1919            TASK    buffer(b)!pdu := pdu;
1920            RETURN;
1921            ENDPROCEDURE;

1923                                                                    /*-------------*/
1924    PROCEDURE       Apply_PDU;                                      /*  Apply_PDU  */
1925    FPAR            r               PortalId,                       /*-------------*/
1926 |                  b               BufferId,
1927 |          IN/OUT  diagnostic      Diagnostic;

1929            DCL     pdu             PDUStruct,
1930                    new             PDUStruct,
1931                    p               PortalId,
1932 *                  pSet            PortalIdSet,
1933                    u               UserId,
1934                    uSet            UserIdSet,
1935 +                  c               ChannelId,
1936 +                  cSet            ChannelIdSet,
1937                    cAttr           ChannelAttributes,
1938                    cAttrSet        ChannelAttributesSet,
1939 +                  t               TokenId,
1940 +                  tSet            TokenIdSet,
1941                    tAttr           TokenAttributes,
1942                    tAttrSet        TokenAttributesSet,
1943                    result          Result;
1944            START
1945            COMMENT'This is a large case selection based on MCSPDU kind.
1946                    These actions and updates to the information base
1947                    take place in both the top and subordinate providers.
1948                    ';
1949            TASK    pdu := buffer(b)!pdu;
1950            DECISION pdu!kind;
1951            (MCrq, MTrq, AUrq, DUrq, CCrq,
1952             CDrq, CArq, CErq, USrq, TGrq,
1953             TIrq, TVrq, TPrq, TRrq, TTrq):
1954                    CALL    Cast_buffer(b, upward);
1955            (PDin): DECISION pdu!heightLimit > 0;
1956                    (True):TASK     buffer(b)!pdu!heightLimit := pdu!heightLimit - 1;
1957                    (False):OUTPUT  Report.portal(r, DC_height_limit_exceeded) TO control;
1958                    ENDDECISION;
1959                    CALL    Broadcast_buffer(b);
1960            (EDrq): TASK    portal(r)!subHeight := pdu!subHeight,
1961                            portal(r)!subInterval := pdu!subInterval,
1962                            portal(r)!interval := IF pdu!subInterval = 0 THEN interval
1963                                            ELSE oneSecond + (pdu!subInterval * 3) FI;
1964                    CALL    Erect_domain;
1965            (MCcf): DECISION merging;
1966                    (False):TASK    p := Next(mcrqQueue),
1967                                    mcrqQueue := Pull(mcrqQueue);
1968                    ELSE:ENDDECISION;
```

```
1969                    TASK    cAttrSet := pdu!mergeChannels;
1970                     1b :   /* for cAttr in cAttrSet */
1971                    DECISION cAttrSet = Empty;
1972                    (False):TASK    cAttr := Pick(cAttrSet),
1973                                    cAttrSet := Del(cAttr, cAttrSet),
1974                                    c := cAttr!channelId;
1975 |                          DECISION c in cUsed and chan(c)!kind = cAttr!kind;
1976 |                          (False):DECISION cAttr!kind = UserId;
1977 |                                  (True): CALL    Take_user(c, diagnostic);
1978 |                                  (False):CALL    Take_channel(c, diagnostic);
1979                                    ENDDECISION;
1980 +                                  DECISION diagnostic = DC_OK;
1981 +                                  (False):RETURN;
1982 +                                  ELSE:ENDDECISION;
1983 +                                  TASK    chan(c)!kind := cAttr!kind;
1984                            ELSE:ENDDECISION;
1985                            DECISION merging;
1986 |                          (True): DECISION chan(c)!kind = UserId;
1987                                    (True): TASK    u := UserId(c),
1988                                                    uConfirm := Incl(u, uConfirm);
1989                                    ELSE:ENDDECISION;
1990 |                                  DECISION chan(c)!kind = Private
1991 |                                          and chan(c)!uMerge /= Empty;
1992 |                                  (True): TASK    cMerge := Incl(c, cMerge);
1993 |                                  (False):TASK    cConfirm := Incl(c, cConfirm);
1994 |                                  ENDDECISION;
1995 |                                  TASK    p := chan(c)!portal;
1996                            ELSE:ENDDECISION;
1997 |                          TASK    chan(c)!portal := p,
1998                                    chan(c)!manager := cAttr!manager,
1999                                    uSet := cAttr!admitted and uUsed,
2000                                    chan(c)!admitted := chan(c)!admitted or uSet;
2001 +                          DECISION (not merging or c in cConfirm)
2002 +                                  and chan(c)!kind = Private
2003 +                                  and chan(c)!admitted = Empty
2004 +                                  and not chan(c)!manager in uUsed
2005 +                          (True): CALL    Delete_channel(c);
2006 +                          ELSE:ENDDECISION;
2007 +                          DECISION merging;
2008 +                          (True): JOIN 1b;
2009 +                          ELSE:ENDDECISION;
2010                            DECISION cAttr!kind = Static or cAttr!kind = Assigned
2011                                    or cAttr!joined;
2012                            (True): DECISION p = 0;
2013                                    (False):TASK    chan(c)!joined :=
2014                                                            Incl(p, chan(c)!joined),
2015                                                    cLeave := Del(c, cLeave);
2016                                    (True): DECISION chan(c)!joined = Empty;
2017                                            (True): TASK    cLeave := Incl(c, cLeave);
2018                                            ELSE:ENDDECISION;
2019                                    ENDDECISION;
2020                            ELSE:ENDDECISION;
2021                            DECISION cAttr!kind = UserId and p = 0;
2022                            (True): TASK    u := UserId(c),
2023                                            uDetach := Incl(u, uDetach),
2024                                            cLeave := Del(c, cLeave);
2025                            ELSE:ENDDECISION;
2026                            JOIN 1b;
2027                    ELSE:ENDDECISION;
2028                    DECISION merging;
2029                    (False):CALL    Cast_buffer(b, p);
2030                    (True): TASK    new!kind := PCin,
2031 |                                  new!detachUserIds := Empty,
2032 |                                  new!purgeChannelIds := Empty,
2033                                    cSet := pdu!purgeChannelIds and cUsed;
2034                             2b :   /* for c in cSet */
2035                            DECISION cSet = Empty;
2036                            (False):TASK    c := Pick(cSet),
2037                                            cSet := Del(c, cSet);
2038                                    DECISION chan(c)!kind;
2039                                    (UserId):
2040 |                                          TASK    u := UserId(c),
2041 |                                                  new!detachUserIds :=
2042 |                                                          Incl(u, new!detachUserIds);
2043 |                                  (Static, Assigned, Private):
2044 |                                          TASK    new!purgeChannelIds :=
2045 |                                                          Incl(c, new!purgeChannelIds);
2046 |                                  ENDDECISION;
2047                                    JOIN 2b;
```

```
2048                            ELSE:ENDDECISION;
2049                            CALL    Purge_channels(pdu!purgeChannelIds);
2050                            TASK    buffer(b)!pdu := new;
2051                            CALL    Broadcast_buffer(b);
2052                    ENDDECISION;
2053            (PCin): DECISION merging;
2054 |                  (True): TASK    cSet := buffer(b)!pdu!purgeChannelIds and cUsed;
2055 |                           3b :   /* for c in cSet */
2056 |                          DECISION cSet = Empty;
2057 |                          (False):TASK    c := Pick(cSet),
2058 |                                          cSet := Del(c, cSet);
2059 |                                  DECISION c in cConfirm;
2060 |                                  (False):TASK    chan(c)!uMerge := chan(c)!admitted;
2061 |                                  ELSE:ENDDECISION;
2062 |                                  JOIN 3b;
2063 |                          ELSE:ENDDECISION;
2064 |                          TASK    buffer(b)!pdu!detachUserIds := uConfirm
2065 |                                          and buffer(b)!pdu!detachUserIds,
2066 |                                  buffer(b)!pdu!purgeChannelIds := cConfirm
2067 |                                          and buffer(b)!pdu!purgeChannelIds;
2068                    ELSE:ENDDECISION;
2069 |                  CALL    Purge_users(buffer(b)!pdu!detachUserIds);
2070 |                  CALL    Purge_channels(buffer(b)!pdu!purgeChannelIds);
2071                    CALL    Broadcast_buffer(b);
2072            (MTcf): DECISION merging;
2073                    (False):TASK    p := Next(mtrqQueue),
2074                                    mtrqQueue := Pull(mtrqQueue);
2075                    ELSE:ENDDECISION;
2076                    TASK    tAttrSet := pdu!mergeTokens;
2077 |                   4b :   /* for tAttr in tAttrSet */
2078                    DECISION tAttrSet = Empty;
2079                    (False):TASK    tAttr := Pick(tAttrSet),
2080                                    tAttrSet := Del(tAttr, tAttrSet),
2081                                    t := tAttr!tokenId;
2082 |                          DECISION t in tUsed;
2083 |                          (False):CALL    Take_token(t, diagnostic);
2084 |                                  DECISION diagnostic = DC_OK;
2085 |                                  (False):RETURN;
2086 |                                  ELSE:ENDDECISION;
2087                            ELSE:ENDDECISION;
2088                            DECISION merging;
2089 |                          (True): DECISION tAttr!kind = Inhibited
2090 |                                          and token(t)!uMerge /= Empty;
2091 |                                  (True): TASK    tMerge := Incl(t, tMerge);
2092 |                                  (False):TASK    tConfirm := Incl(t, tConfirm);
2093 |                                  ENDDECISION;
2094                            ELSE:ENDDECISION;
2095                            TASK    token(t)!kind := tAttr!kind,
2096                                    token(t)!grabber := tAttr!grabber,
2097                                    token(t)!recipient := tAttr!recipient,
2098                                    uSet := tAttr!inhibitors and uUsed,
2099                                    token(t)!inhibitors := token(t)!inhibitors or uSet;
2100 +                          DECISION (not merging or t in tConfirm)
2101 +                                  and token(t)!kind = Inhibited
2102 +                                  and token(t)!inhibitors = Empty;
2103 +                          (True): CALL    Delete_token(t);
2104 +                          ELSE:ENDDECISION;
2105                            DECISION upward = 0 and token(t)!kind = Ungivable;
2106                            (True): TASK    tReject := Incl(t, tReject);
2107                            ELSE:ENDDECISION;
2108 |                          JOIN 4b;
2109                    ELSE:ENDDECISION;
2110                    DECISION merging;
2111                    (False):CALL    Cast_buffer(b, p);
2112                    (True): CALL    Purge_tokens(pdu!purgeTokenIds);
2113                            TASK    buffer(b)!pdu!kind := PTin;
2114                            CALL    Broadcast_buffer(b);
2115                    ENDDECISION;
2116            (PTin): DECISION merging;
2117 |                  (True): TASK    tSet := buffer(b)!pdu!purgeTokenIds and tUsed;
2118 |                           5b :   /* for t in tSet */
2119 |                          DECISION tSet = Empty;
2120 |                          (False):TASK    t := Pick(tSet),
2121 |                                          tSet := Del(t, tSet);
2122 |                                  DECISION t in tConfirm;
2123 |                                  (False):TASK    token(t)!uMerge := token(t)!inhibitors;
2124 |                                  ELSE:ENDDECISION;
2125 |                                  JOIN 5b;
2126 |                          ELSE:ENDDECISION;
```

```
2127 |                          TASK    buffer(b)!pdu!purgeTokenIds := tConfirm
2128                                       and buffer(b)!pdu!purgeTokenIds;
2129                     ELSE:ENDDECISION;
2130                     CALL    Purge_tokens(buffer(b)!pdu!purgeTokenIds);
2131                     CALL    Broadcast_buffer(b);
2132 |           (DPum): OUTPUT  Drop.portal(r, pdu!reason) TO control;
2133             (AUcf): TASK    p := Next(aurqQueue),
2134 |                          aurqQueue := Pull(aurqQueue),
2135 |                          c := ChannelId(u),
2136 |                          u := pdu!initiator;
2137                     DECISION pdu!result = RT_successful;
2138 |                   (True): DECISION upward = 0;
2139 |                           (False):CALL    Take_user(c, diagnostic);
2140 |                                   DECISION diagnostic = DC_OK;
2141 |                                   (False):RETURN;
2142 |                                   ELSE:ENDDECISION;
2143                             ELSE:ENDDECISION;
2144 |                           TASK    chan(c)!portal := p;
2145                             DECISION p = 0;
2146 |                           (True): TASK    uDetach := Incl(u, uDetach),
2147 |                                           cLeave := Del(c, cLeave);
2148                             ELSE:ENDDECISION;
2149                     ELSE:ENDDECISION;
2150                     CALL    Cast_buffer(b, p);
2151 |           (DUin): DECISION merging;
2152 |                   (True): TASK    buffer(b)!pdu!userIds := uConfirm
2153 |                                   and buffer(b)!pdu!userIds;
2154                     ELSE:ENDDECISION;
2155 |                   CALL    Purge_users(buffer(b)!pdu!userIds);
2156                     CALL    Broadcast_buffer(b);
2157             (CJrq): TASK    c := pdu!channelId,
2158                             result := RT_successful,
2159                             p := upward;
2160                     DECISION c in cUsed;
2161                     (True): DECISION chan(c)!kind;
2162                             (UserId):
2163                                     TASK    u := UserId(c);
2164                                     DECISION pdu!initiator = u;
2165                                     (False):TASK    result := RT_other_user_id;
2166                                     ELSE:ENDDECISION;
2167                             (Private):
2168                                     DECISION pdu!initiator in chan(c)!admitted;
2169                                     (False):TASK    result := RT_not_admitted;
2170                                     ELSE:ENDDECISION;
2171                             ELSE:ENDDECISION;
2172 |                           DECISION result = RT_successful;
2173 |                           (False):TASK    buffer(b)!pdu!kind := CJcf,
2174 |                                           buffer(b)!pdu!requested := c,
2175                                             buffer(b)!pdu!result := result,
2176 +                                           buffer(b)!pdu!channelId := 0,
2177                                             p := r;
2178 |                           (True): DECISION chan(c)!joined /= Empty or c in cLeave;
2179 |                                   (True): TASK    chan(c)!joined :=
2180 |                                                   Incl(r, chan(c)!joined),
2181 |                                           cLeave := Del(c, cLeave),
2182 |                                           buffer(b)!pdu!kind := CJcf,
2183 |                                           buffer(b)!pdu!requested := c,
2184 |                                           buffer(b)!pdu!result := result,
2185 |                                           p := r;
2186 |                                   ELSE:ENDDECISION;
2187 |                           ENDDECISION;
2188 |                   ELSE:ENDDECISION;
2189                     CALL    Cast_buffer(b, p);
2190             (CJcf): TASK    c := pdu!channelId,
2191                             u := pdu!initiator;
2192                     CALL    Route_user(u, p);
2193                     DECISION pdu!result = RT_successful;
2194 |                   (True): DECISION c in cUsed;
2195 |                           (False):CALL    Take_channel(c, diagnostic);
2196 |                                   DECISION diagnostic = DC_OK;
2197 |                                   (False):RETURN;
2198 |                                   ELSE:ENDDECISION;
2199                                     TASK    chan(c)!kind := IF c < 1001 THEN Static
2200                                                             ELSE Assigned FI;
2201                             ELSE:ENDDECISION;
2202                             DECISION p = 0;
2203                             (False):TASK    chan(c)!joined := Incl(p, chan(c)!joined),
2204                                             cLeave := Del(c, cLeave);
2205                             (True): DECISION chan(c)!joined = Empty;
```

```
2206                                    (True): TASK    cLeave := Incl(c, cLeave);
2207                                    ELSE:ENDDECISION;
2208                            ENDDECISION;
2209                    ELSE:ENDDECISION;
2210                    CALL    Cast_buffer(b, p);
2211            (CLrq): TASK    cSet := pdu!channelIds and cUsed;
2212 |                   6b :   /* for c in cSet */
2213                    DECISION cSet = Empty;
2214                    (False):TASK    c := Pick(cSet),
2215                                    cSet := Del(c, cSet);
2216                            DECISION r in chan(c)!joined;
2217                            (True): TASK    chan(c)!joined := Del(r, chan(c)!joined);
2218                                    DECISION chan(c)!joined = Empty;
2219                                    (True): TASK    cLeave := Incl(c, cLeave);
2220                                    ELSE:ENDDECISION;
2221                            ELSE:ENDDECISION;
2222 |                          JOIN 6b;
2223                    ELSE:ENDDECISION;
2224            (CCcf): TASK    c := pdu!channelId,
2225                            u := pdu!initiator;
2226                    DECISION pdu!result = RT_successful;
2227 |                  (True): DECISION upward = 0;
2228 |                          (False):CALL    Take_channel(c, diagnostic);
2229 |                                  DECISION diagnostic = DC_OK;
2230 |                                  (False):RETURN;
2231 |                                  ELSE:ENDDECISION;
2232                            ELSE:ENDDECISION;
2233 *                          TASK    chan(c)!kind := Private,
2234                                    chan(c)!manager := u,
2235                                    chan(c)!admitted := Incl(u, Empty);
2236                    ELSE:ENDDECISION;
2237                    CALL    Route_user(u, p);
2238                    CALL    Cast_buffer(b, p);
2239            (CDin): TASK    c := pdu!channelId;
2240 |                  DECISION c in cUsed;
2241                    (False):RETURN;
2242 |                  (True): DECISION merging and not c in cConfirm;
2243 |                          (True): TASK    chan(c)!uMerge := chan(c)!admitted;
2244 |                                  RETURN;
2245 |                          ELSE:ENDDECISION;
2246 |                          DECISION chan(c)!kind = Private;
2247 |                          (False):TASK    diagnostic := DC_channel_id_conflict;
2248 |                                  RETURN;
2249 |                          ELSE:ENDDECISION;
2250 |                  ENDDECISION;
2251                    TASK    uSet := Incl(chan(c)!manager, chan(c)!admitted);
2252                    CALL    Delete_channel(c);
2253                    CALL    Multicast_buffer(b, uSet);
2254            (CAin): TASK    c := pdu!channelId,
2255                            uSet := pdu!userIds and uUsed,
2256                            buffer(b)!pdu!userIds := uSet;
2257                    DECISION merging;
2258 |                  (True): TASK    uSet := uSet and uConfirm,
2259                                    buffer(b)!pdu!userIds := uSet;
2260 +                           7b :   /* for u in uSet */
2261 +                          DECISION uSet = Empty;
2262 +                          (False):TASK    u := Pick(uSet),
2263 +                                          uSet := Del(u, uSet),
2264 +                                          c := ChannelId(u),
2265 +                                          chan(c)!portal = 0;
2266 +                                  JOIN 7b;
2267 +                          ELSE:ENDDECISION;
2268 +                          TASK    buffer(b)!pdu!kind := DUrq,
2269 +                                  buffer(b)!pdu!reason := RN_channel_purged;
2270 +                          CALL    Cast_buffer(b, r);
2271 +                          RETURN;
2272                    ELSE:ENDDECISION;
2273                    DECISION uSet = Empty;
2274 |                  (False):DECISION c in cUsed and chan(c)!kind = Private;
2275 |                          (False):CALL    Take_channel(c, diagnostic);
2276 |                                  DECISION diagnostic = DC_OK;
2277 |                                  (False):RETURN;
2278 |                                  ELSE:ENDDECISION;
2279                            ELSE:ENDDECISION;
2280                            TASK    chan(c)!kind := Private,
2281                                    chan(c)!manager := pdu!initiator,
2282                                    chan(c)!admitted := chan(c)!admitted or uSet;
2283                            CALL    Multicast_buffer(b, uSet);
2284                    ELSE:ENDDECISION;
```

```
(CEin): TASK    c := pdu!channelId;
        DECISION c in cUsed;
        (False):RETURN;
        (True): DECISION merging and not c in cConfirm;
                (True): TASK    chan(c)!uMerge := chan(c)!admitted;
                        RETURN;
                ELSE:ENDDECISION;
                DECISION chan(c)!kind = Private;
                (False):TASK    diagnostic := DC_channel_id_conflict;
                        RETURN;
                ELSE:ENDDECISION;
        ENDDECISION;
        TASK    uSet := chan(c)!admitted,
                pSet := Empty;
         8b :   /* for u in uSet */
        DECISION uSet = Empty;
        (False):TASK    u := Pick(uSet),
                        uSet := Del(u, uSet);
                DECISION u in pdu!userIds;
                (False):CALL    Route_user(u, p);
                        TASK    pSet := Incl(p, pSet);
                ELSE:ENDDECISION;
                JOIN 8b;
        ELSE:ENDDECISION;
        TASK    uSet := pdu!userIds and chan(c)!admitted,
                buffer(b)!pdu!userIds := uSet;
         9b :   /* for u in uSet */
        DECISION uSet = Empty;
        (False):TASK    u := Pick(uSet),
                        uSet := Del(u, uSet),
                        chan(c)!admitted := Del(u, chan(c)!admitted);
                CALL    Route_user(u, p);
                DECISION p in chan(c)!joined and not p in pSet;
                (False):TASK    chan(c)!joined := Del(p, chan(c)!joined);
                        DECISION chan(c)!joined = Empty;
                        (True): TASK    cLeave := Incl(c, cLeave);
                        ELSE:ENDDECISION;
                ELSE:ENDDECISION;
                JOIN 9b;
        ELSE:ENDDECISION;
        DECISION chan(c)!admitted /= Empty or chan(c)!manager in uUsed;
        (False):CALL    Delete_channel(c);
        ELSE:ENDDECISION;
        TASK    uSet := buffer(b)!pdu!userIds;
        CALL    Multicast_buffer(b, uSet);
(SDrq): TASK    buffer(b)!pdu!kind := SDin,
                pSet := Incl(upward, Empty);
        JOIN 10f;
(SDin): TASK    pSet := Empty;
        JOIN 10f;
(USin): TASK    pSet := Empty;
         10f :
        TASK    c := pdu!channelId;
        DECISION c in cUsed and not merging;
        (True): TASK    pSet := pSet or chan(c)!joined;
                DECISION chan(c)!kind = UserId;
                (True): TASK    pSet := Del(upward, pSet);
                ELSE:ENDDECISION;
        ELSE:ENDDECISION;
        DECISION buffer(b)!pdu!kind = SDin;
        (True): TASK    pSet := Del(r, pSet);
        ELSE:ENDDECISION;
         11b :  /* for p in pSet */
        DECISION pSet = Empty;
        (False):TASK    p := Pick(pSet),
                        pSet := Del(p, pSet);
                CALL    Cast_buffer(b, p);
                JOIN 11b;
        ELSE:ENDDECISION;
(TGcf, TIcf, TVcf, TRcf, TTcf):
        TASK    t := pdu!tokenId,
                u := pdu!initiator;
        DECISION pdu!tokenStatus;
        (SelfGrabbed):
                DECISION t in tUsed;
                (False):CALL    Take_token(t, diagnostic);
                        DECISION diagnostic = DC_OK;
                        (False):RETURN;
                        ELSE:ENDDECISION;
```

```
2364                            ELSE:ENDDECISION;
2365                            TASK    token(t)!kind := Grabbed,
2366                                    token(t)!grabber := u;
2367                    (SelfInhibited):
2368 |                          DECISION t in tUsed;
2369 |                          (False):CALL    Take_token(t, diagnostic);
2370 |                                  DECISION diagnostic = DC_OK;
2371 |                                  (False):RETURN;
2372 |                                  ELSE:ENDDECISION;
2373                            ELSE:ENDDECISION;
2374                            TASK    token(t)!kind := Inhibited,
2375                                    token(t)!inhibitors := Incl(u, token(t)!inhibitors);
2376                    (SelfRecipient):
2377                            TASK    token(t)!recipient := u;
2378                    (SelfGiving):
2379                            TASK    token(t)!grabber := u;
2380                    (NotInUse):
2381                            CALL    Delete_token(t);
2382                    ELSE:
2383                            DECISION token(t)!kind;
2384                            (Grabbed, Ungivable):
2385                                    DECISION token(t)!grabber = u;
2386                                    (True): CALL    Delete_token(t);
2387                                    ELSE:ENDDECISION;
2388                            (Giving, Given):
2389                                    DECISION token(t)!grabber = u;
2390                                    (True): TASK    token(t)!kind := Given;
2391                                    ELSE:ENDDECISION;
2392                                    DECISION token(t)!recipient = u;
2393                                    (True): CALL    Delete_token(t);
2394                                    ELSE:ENDDECISION;
2395                            (Inhibited):
2396                                    TASK    token(t)!inhibitors :=
2397                                                    Del(u, token(t)!inhibitors);
2398                                    DECISION token(t)!inhibitors = Empty;
2399                                    (True): CALL    Delete_token(t);
2400                                    ELSE:ENDDECISION;
2401                            ENDDECISION;
2402                    ENDDECISION;
2403                    CALL    Route_user(u, p);
2404                    CALL    Cast_buffer(b, p);
2405            (TVin): TASK    t := pdu!tokenId,
2406                            u := pdu!recipient;
2407                    DECISION merging;
2408 |                  (True): TASK    buffer(b)!pdu!kind := TVrs,
2409 |                                  buffer(b)!pdu!result := RT_domain_merging;
2410 |                          CALL    Cast_buffer(b, r);
2411 |                          RETURN;
2412 |                  ELSE:ENDDECISION;
2413 |                  DECISION t in tUsed;
2414 |                  (False):CALL    Take_token(t, diagnostic);
2415 |                          DECISION diagnostic = DC_OK;
2416 |                          (False):RETURN;
2417                            ELSE:ENDDECISION;
2418 *                  ELSE:ENDDECISION;
2419                    TASK    token(t)!kind := Giving,
2420                            token(t)!grabber := pdu!initiator,
2421                            token(t)!recipient := u;
2422                    CALL    Route_user(u, p);
2423                    CALL    Cast_buffer(b, p);
2424            (TVrs): TASK    t := pdu!tokenId,
2425                            u := pdu!recipient;
2426                    DECISION t in tUsed and token(t)!recipient = u;
2427 |                  (True): DECISION token(t)!kind;
2428 |                          (Giving):
2429 |                                  TASK    token(t)!kind := Grabbed;
2430 |                                  DECISION pdu!result = RT_successful;
2431 |                                  (True): TASK    token(t)!grabber := u;
2432 |                                  (False):DECISION token(t)!grabber in uUsed;
2433 |                                          (False):CALL    Delete_token(t);
2434 |                                          ELSE:ENDDECISION;
2435 |                                  ENDDECISION;
2436 |                                  CALL    Cast_buffer(b, upward);
2437 |                          (Given):
2438 |                                  TASK    token(t)!kind := Grabbed,
2439 |                                          token(t)!grabber := u;
2440 |                                  DECISION pdu!result = RT_successful;
2441                                    (False):CALL    Delete_token(t);
2442                                    ELSE:ENDDECISION;
```

```
2443 |                                 CALL    Cast_buffer(b, upward);
2444 *                         ELSE:ENDDECISION;
2445                    ELSE:ENDDECISION;
2446            (TPin): TASK    t := pdu!tokenId;
2447 |                  DECISION t in tUsed and not merging;
2448 |                  (True): DECISION token(t)!kind;
2449 |                          (Grabbed, Ungivable):
2450 |                                  CALL    Route_user(token(t)!grabber, p);
2451 |                                  CALL    Cast_buffer(b, p);
2452 |                          (Given):
2453 |                                  CALL    Route_user(token(t)!recipient, p);
2454 |                                  CALL    Cast_buffer(b, p);
2455 |                          (Giving):
2456 |                                  TASK    uSet := Incl(token(t)!grabber, Empty);
2457 |                                  TASK    uSet := Incl(token(t)!recipient, uSet);
2458 |                                  CALL    Multicast_buffer(b, uSet);
2459 |                          (Inhibited):
2460 |                                  TASK    uSet := token(t)!inhibitors;
2461 |                                  CALL    Multicast_buffer(b, uSet);
2462 |                          ENDDECISION;
2463 *                   ELSE:ENDDECISION;
2464            ENDDECISION;
2465            RETURN;
2466            ENDPROCEDURE;

2468                                                                    /*----------------*/
2469    PROCEDURE       Token_status;                                   /*  Token_status  */
2470    FPAR    IN/OUT  pdu             PDUStruct;                      /*----------------*/

2472            DCL     t               TokenId,
2473                    u               UserId,
2474                    status          TokenStatus;
2475            START
2476            COMMENT'Calculate for an MCSPDU the relationship between
2477                    the initiator user id and token id it contains.
2478                    ';
2479            TASK    t := pdu!tokenId,
2480                    u := pdu!initiator;
2481            DECISION t in tUsed;
2482            (False):TASK    status := NotInUse;
2483            (True): DECISION token(t)!kind;
2484                    (Grabbed):
2485                            DECISION token(t)!grabber = u;
2486                            (True): TASK    status := SelfGrabbed;
2487                            (False):TASK    status := OtherGrabbed;
2488                            ENDDECISION;
2489                    (Ungivable):
2490                            DECISION token(t)!grabber = u;
2491                            (True): TASK    status := SelfGiving;
2492                            (False):TASK    status := OtherGiving;
2493                            ENDDECISION;
2494                    (Given):
2495                            DECISION token(t)!recipient = u;
2496                            (True): TASK    status := SelfRecipient;
2497                            (False):TASK    status := OtherGiving;
2498                            ENDDECISION;
2499                    (Giving):
2500                            DECISION token(t)!recipient = u;
2501                            (True): TASK    status := SelfRecipient;
2502                            (False):DECISION token(t)!grabber = u;
2503                                    (True): TASK    status := SelfGiving;
2504                                    (False):TASK    status := OtherGiving;
2505                                    ENDDECISION;
2506                            ENDDECISION;
2507                    (Inhibited):
2508                            DECISION u in token(t)!inhibitors;
2509                            (True): TASK    status := SelfInhibited;
2510                            (False):TASK    status := OtherInhibited;
2511                            ENDDECISION;
2512                    ENDDECISION;
2513            ENDDECISION;
2514            TASK    pdu!tokenStatus := status;
2515            RETURN;
2516            ENDPROCEDURE;

2518                                                                    /*---------------*/
2519    PROCEDURE       Delete_user;                                    /*  Delete_user  */
2520    FPAR            u               UserId;                         /*---------------*/
```

```
2522 |          DCL     p               PortalId,
2523 |                  c               ChannelId,
2524                    cSet            ChannelIdSet,
2525 +                  a               UserId,
2526 +                  aSet            UserIdSet,
2527 +                  q               PortalId,
2528                    t               TokenId,
2529 |                  tSet            TokenIdSet;
2530            START
2531            COMMENT'Update the information base to delete a user id.
2532                    This has consequences for its channels and tokens.
2533                    Any data still in transit is left undisturbed.
2534 *                  ';
2535            DECISION u in uUsed;
2536 |          (False):RETURN;
2537            ELSE:ENDDECISION;
2538 |          TASK    c := ChannelId(u),
2539 |                  p := chan(c)!portal,
2540 |                  uUsed := Del(u, uUsed),
2541 |                  numUserIds := numUserIds - 1,
2542 |                  cUsed := Del(c, cUsed),
2543 |                  cFree := Incl(c, cFree),
2544 |                  numChannelIds := numChannelIds - 1,
2545 |                  uConfirm := Del(u, uConfirm),
2546 |                  cConfirm := Del(c, cConfirm),
2547 |                  uDetach := Del(u, uDetach),
2548 |                  cLeave := Del(c, cLeave);
2549            TASK    cSet := cUsed;
2550             1b :   /* for c in cSet */
2551            DECISION cSet = Empty;
2552            (False):TASK    c := Pick(cSet),
2553                            cSet := Del(c, cSet);
2554 |                  DECISION portal(p)!kind = Attached and p in chan(c)!joined;
2555                    (True): TASK    chan(c)!joined := Del(p, chan(c)!joined);
2556                            DECISION chan(c)!joined = Empty;
2557                            (True): TASK    cLeave := Incl(c, cLeave);
2558                            ELSE:ENDDECISION;
2559                    ELSE:ENDDECISION;
2560                    DECISION chan(c)!kind = Private;
2561                    (True): DECISION chan(c)!manager = u;
2562                            (True): TASK    chan(c)!manager := 0;
2563                                    DECISION upward = 0;
2564                                    (True): TASK    cDisband := Incl(c, cDisband);
2565                                    ELSE:ENDDECISION;
2566                            ELSE:ENDDECISION;
2567                            TASK    chan(c)!admitted := Del(u, chan(c)!admitted),
2568 *                                  chan(c)!uMerge := Del(u, chan(c)!uMerge);
2569                            DECISION chan(c)!admitted /= Empty or chan(c)!manager in uUsed;
2570 |                          (False):DECISION not merging or c in cConfirm;
2571 |                                  (True): CALL    Delete_channel(c);
2572 |                                  ELSE:ENDDECISION;
2573 |                          (True): DECISION p in chan(c)!joined;
2574 |                                  (True): TASK    aSet := chan(c)!admitted;
2575 |                                           2b :   /* for a in aSet */
2576 |                                          DECISION aSet = Empty;
2577 |                                          (False):TASK    a := Pick(aSet),
2578 |                                                          aSet := Del(a, aSet);
2579 |                                                  CALL    Route_user(a, q);
2580 |                                                  DECISION q = p;
2581 |                                                  (False):JOIN 2b;
2582 |                                                  (True): JOIN 1b;
2583 |                                                  ENDDECISION;
2584 |                                          ELSE:ENDDECISION;
2585 |                                          TASK    chan(c)!joined := Del(p, chan(c)!joined);
2586 |                                          DECISION chan(c)!joined = Empty;
2587 |                                          (True): TASK    cLeave := Incl(c, cLeave);
2588 |                                          ELSE:ENDDECISION;
2589 |                                  ELSE:ENDDECISION;
2590 |                          ENDDECISION;
2591                    ELSE:ENDDECISION;
2592                    JOIN 1b;
2593            ELSE:ENDDECISION;
2594            TASK    tSet := tUsed;
2595 |           3b :   /* for t in tSet */
2596            DECISION tSet = Empty;
2597            (False):TASK    t := Pick(tSet),
2598                            tSet := Del(t, tSet);
2599                    DECISION token(t)!kind;
2600                    (Grabbed, Ungivable):
```

```
2601                            DECISION token(t)!grabber = u;
2602 |                          (True): JOIN 3f;
2603                            ELSE:ENDDECISION;
2604                    (Given):
2605                            DECISION token(t)!recipient = u;
2606 |                          (True): JOIN 3f;
2607                            ELSE:ENDDECISION;
2608                    (Giving):
2609                            DECISION token(t)!recipient = u;
2610                            (True): TASK    token(t)!kind := Ungivable;
2611                                    DECISION token(t)!grabber in uUsed;
2612 |                                  (False):JOIN 3f;
2613                                    (True): DECISION upward = 0;
2614                                            (True): TASK    tReject := Incl(t, tReject);
2615                                            ELSE:ENDDECISION;
2616                                    ENDDECISION;
2617                            (False):DECISION token(t)!grabber = u;
2618                                    (True): TASK    token(t)!kind := Given;
2619                                    ELSE:ENDDECISION;
2620                            ELSE:ENDDECISION;
2621                    (Inhibited):
2622                            TASK    token(t)!inhibitors := Del(u, token(t)!inhibitors),
2623 *                                  token(t)!uMerge := Del(u, token(t)!uMerge);
2624                            DECISION token(t)!inhibitors = Empty;
2625 |                          (True):  3f :
2626 |                                  DECISION not merging or t in tConfirm;
2627 |                                  (True): CALL    Delete_token(t);
2628 |                                  (False):TASK    token(t)!kind := Inhibited,
2629 |                                                  token(t)!inhibitors := Empty;
2630 |                                  ENDDECISION;
2631                            ELSE:ENDDECISION;
2632                    ENDDECISION;
2633 |                  JOIN 3b;
2634            ELSE:ENDDECISION;
2635            RETURN;
2636            ENDPROCEDURE;

2638                                                                    /*------------------*/
2639    PROCEDURE       Delete_channel;                                 /*  Delete_channel  */
2640    FPAR            c               ChannelId;                      /*------------------*/

2642            DCL     u               UserId;
2643            START
2644            COMMENT'Update the information base to delete a channel id.
2645                    ';
2646            DECISION c in cUsed;
2647            (False):RETURN;
2648            ELSE:ENDDECISION;
2649            DECISION chan(c)!kind = UserId;
2650            (True): TASK    u := UserId(c);
2651                    CALL    Delete_user(u);
2652            (False):TASK    cUsed := Del(c, cUsed),
2653                            cFree := Incl(c, cFree),
2654                            numChannelIds := numChannelIds - 1,
2655 *                          cConfirm := Del(c, cConfirm),
2656                            cLeave := Del(c, cLeave),
2657                            cDisband := Del(c, cDisband);
2658            ENDDECISION;
2659            RETURN;
2660            ENDPROCEDURE;

2662                                                                    /*----------------*/
2663    PROCEDURE       Delete_token;                                   /*  Delete_token  */
2664    FPAR            t               TokenId;                        /*----------------*/

2666            START
2667            COMMENT'Update the information base to delete a token id.
2668                    ';
2669            DECISION t in tUsed;
2670            (False):RETURN;
2671            ELSE:ENDDECISION;
2672            TASK    tUsed := Del(t, tUsed),
2673                    tFree := Incl(t, tFree),
2674                    numTokenIds := numTokenIds - 1,
2675 *                  tConfirm := Del(t, tConfirm),
2676                    tReject := Del(t, tReject);
2677            RETURN;
2678            ENDPROCEDURE;
```

```
2680 +                                                                  /*---------------*/
2681 +  PROCEDURE       Purge_users;                                    /*  Purge_users  */
2682 +  FPAR            uSet            UserIdSet;                      /*---------------*/

2684 +          DCL     u               UserId;
2685 +          START
2686 +          COMMENT'Delete a set of user ids.
2687 +                  ';
2688 +           1b :   /* for u in uSet */
2689 +          DECISION uSet = Empty;
2690 +          (False):TASK    u := Pick(uSet),
2691 +                          uSet := Del(u, uSet);
2692 +                  CALL    Delete_user(u);
2693 +                  JOIN 1b;
2694 +          ELSE:ENDDECISION;
2695 +          RETURN;
2696 +          ENDPROCEDURE;

2698                                                                   /*------------------*/
2699     PROCEDURE       Purge_channels;                             /*  Purge_channels  */
2700     FPAR            cSet            ChannelIdSet;               /*------------------*/

2702            DCL     c               ChannelId;
2703            START
2704            COMMENT'Delete a set of channel ids.
2705                    ';
2706             1b :   /* for c in cSet */
2707            DECISION cSet = Empty;
2708            (False):TASK    c := Pick(cSet),
2709                            cSet := Del(c, cSet);
2710                    CALL    Delete_channel(c);
2711                    JOIN 1b;
2712            ELSE:ENDDECISION;
2713            RETURN;
2714            ENDPROCEDURE;

2716                                                                   /*----------------*/
2717     PROCEDURE       Purge_tokens;                               /*  Purge_tokens  */
2718     FPAR            tSet            TokenIdSet;                 /*----------------*/

2720            DCL     t               TokenId;
2721            START
2722            COMMENT'Delete a set of token ids.
2723                    ';
2724             1b :   /* for t in tSet */
2725            DECISION tSet = Empty;
2726            (False):TASK    t := Pick(tSet),
2727                            tSet := Del(t, tSet);
2728                    CALL    Delete_token(t);
2729                    JOIN 1b;
2730            ELSE:ENDDECISION;
2731            RETURN;
2732            ENDPROCEDURE;

2734                                                                   /*-----------------*/
2735     PROCEDURE       Output_buffer;                              /*  Output_buffer  */
2736     FPAR            b               BufferId,                   /*-----------------*/
2737                     p               PortalId;

2739            DCL     dp              DataPriority,
2740                    pdu             PDUStruct,
2741                    pid             PId;
2742            START
2743            COMMENT'Send the output signal representing an MCSPDU.
2744                    The next must wait until PDU.ready is received.
2745                    ';
2746            TASK    dp := buffer(b)!dataPriority,
2747                    pdu := buffer(b)!pdu,
2748                    pid := portal(p)!pids(dp);
2749            DECISION p = upward and pdu!kind = SDin;
2750            (True): TASK    pdu!kind := SDrq;
2751            ELSE:ENDDECISION;
2752            DECISION pdu!kind;
2753            (PDin): OUTPUT  PDin(pdu) TO pid;
2754            (EDrq): OUTPUT  EDrq(pdu) TO pid;
2755            (MCrq): OUTPUT  MCrq(pdu) TO pid;
2756            (MCcf): OUTPUT  MCcf(pdu) TO pid;
2757            (PCin): OUTPUT  PCin(pdu) TO pid;
2758            (MTrq): OUTPUT  MTrq(pdu) TO pid;
```

```
2759            (MTcf): OUTPUT  MTcf(pdu) TO pid;
2760            (PTin): OUTPUT  PTin(pdu) TO pid;
2761            (DPum): OUTPUT  DPum(pdu) TO pid;
2762            (RJum): OUTPUT  RJum(pdu) TO pid;
2763            (AUrq): OUTPUT  AUrq(pdu) TO pid;
2764            (AUcf): OUTPUT  AUcf(pdu) TO pid;
2765            (DUrq): OUTPUT  DUrq(pdu) TO pid;
2766            (DUin): OUTPUT  DUin(pdu) TO pid;
2767            (CJrq): OUTPUT  CJrq(pdu) TO pid;
2768            (CJcf): OUTPUT  CJcf(pdu) TO pid;
2769            (CLrq): OUTPUT  CLrq(pdu) TO pid;
2770            (CCrq): OUTPUT  CCrq(pdu) TO pid;
2771            (CCcf): OUTPUT  CCcf(pdu) TO pid;
2772            (CDrq): OUTPUT  CDrq(pdu) TO pid;
2773            (CDin): OUTPUT  CDin(pdu) TO pid;
2774            (CArq): OUTPUT  CArq(pdu) TO pid;
2775            (CAin): OUTPUT  CAin(pdu) TO pid;
2776            (CErq): OUTPUT  CErq(pdu) TO pid;
2777            (CEin): OUTPUT  CEin(pdu) TO pid;
2778            (SDrq): OUTPUT  SDrq(pdu) TO pid;
2779            (SDin): OUTPUT  SDin(pdu) TO pid;
2780            (USrq): OUTPUT  USrq(pdu) TO pid;
2781            (USin): OUTPUT  USin(pdu) TO pid;
2782            (TGrq): OUTPUT  TGrq(pdu) TO pid;
2783            (TGcf): OUTPUT  TGcf(pdu) TO pid;
2784            (TIrq): OUTPUT  TIrq(pdu) TO pid;
2785            (TIcf): OUTPUT  TIcf(pdu) TO pid;
2786            (TVrq): OUTPUT  TVrq(pdu) TO pid;
2787            (TVin): OUTPUT  TVin(pdu) TO pid;
2788            (TVrs): OUTPUT  TVrs(pdu) TO pid;
2789            (TVcf): OUTPUT  TVcf(pdu) TO pid;
2790            (TPrq): OUTPUT  TPrq(pdu) TO pid;
2791            (TPin): OUTPUT  TPin(pdu) TO pid;
2792            (TRrq): OUTPUT  TRrq(pdu) TO pid;
2793            (TRcf): OUTPUT  TRcf(pdu) TO pid;
2794            (TTrq): OUTPUT  TTrq(pdu) TO pid;
2795            (TTcf): OUTPUT  TTcf(pdu) TO pid;
2796          ENDDECISION;
2797          RETURN;
2798          ENDPROCEDURE;

2800          /* Input transitions */

2802          DCL     p               PortalId,
2803                  dp              DataPriority,
2804                  pdu             PDUStruct,
2805                  pKind           PortalKind,
2806                  pids            PIdByPri,
2807                  reason          Reason;
2808          START
2809          COMMENT'The state machine contains a single state.
2810                  ';
2811          CALL    Initialize_resources;
2812          NEXTSTATE ~;

2814    STATE ~;INPUT   Open.portal(p, pKind, pids);
2815          CALL    Open_portal(p, pKind, pids);
2816          NEXTSTATE -;

2818    STATE ~;INPUT   Time.portal(p);
2819          CALL    Time_portal(p);
2820          NEXTSTATE -;

2822    STATE ~;INPUT   Drop.portal(p, reason);
2823          CALL    Drop_portal(p, reason);
2824          NEXTSTATE -;

2826    STATE ~;INPUT   Shut.portal(p);
2827          CALL    Shut_portal(p);
2828 |        DECISION pUsed = Empty;
2829 |        (False):NEXTSTATE -;
2830 |        (True): STOP;
2831 |        ENDDECISION;

2833    STATE ~;INPUT   PDU.ready(dp);
2834          CALL    PDU_ready(dp);
2835          NEXTSTATE -;

2837    STATE ~;INPUT   PDin(pdu); TASK pdu!kind := PDin; CALL Input_PDU(pdu); NEXTSTATE -;
```

```
2838    STATE ~;INPUT   EDrq(pdu); TASK pdu!kind := EDrq; CALL Input_PDU(pdu); NEXTSTATE -;
2839    STATE ~;INPUT   MCrq(pdu); TASK pdu!kind := MCrq; CALL Input_PDU(pdu); NEXTSTATE -;
2840    STATE ~;INPUT   MCcf(pdu); TASK pdu!kind := MCcf; CALL Input_PDU(pdu); NEXTSTATE -;
2841    STATE ~;INPUT   PCin(pdu); TASK pdu!kind := PCin; CALL Input_PDU(pdu); NEXTSTATE -;
2842    STATE ~;INPUT   MTrq(pdu); TASK pdu!kind := MTrq; CALL Input_PDU(pdu); NEXTSTATE -;
2843    STATE ~;INPUT   MTcf(pdu); TASK pdu!kind := MTcf; CALL Input_PDU(pdu); NEXTSTATE -;
2844    STATE ~;INPUT   PTin(pdu); TASK pdu!kind := PTin; CALL Input_PDU(pdu); NEXTSTATE -;
2845    STATE ~;INPUT   DPum(pdu); TASK pdu!kind := DPum; CALL Input_PDU(pdu); NEXTSTATE -;
2846    STATE ~;INPUT   RJum(pdu); TASK pdu!kind := RJum; CALL Input_PDU(pdu); NEXTSTATE -;
2847    STATE ~;INPUT   AUrq(pdu); TASK pdu!kind := AUrq; CALL Input_PDU(pdu); NEXTSTATE -;
2848    STATE ~;INPUT   AUcf(pdu); TASK pdu!kind := AUcf; CALL Input_PDU(pdu); NEXTSTATE -;
2849    STATE ~;INPUT   DUrq(pdu); TASK pdu!kind := DUrq; CALL Input_PDU(pdu); NEXTSTATE -;
2850    STATE ~;INPUT   DUin(pdu); TASK pdu!kind := DUin; CALL Input_PDU(pdu); NEXTSTATE -;
2851    STATE ~;INPUT   CJrq(pdu); TASK pdu!kind := CJrq; CALL Input_PDU(pdu); NEXTSTATE -;
2852    STATE ~;INPUT   CJcf(pdu); TASK pdu!kind := CJcf; CALL Input_PDU(pdu); NEXTSTATE -;
2853    STATE ~;INPUT   CLrq(pdu); TASK pdu!kind := CLrq; CALL Input_PDU(pdu); NEXTSTATE -;
2854    STATE ~;INPUT   CCrq(pdu); TASK pdu!kind := CCrq; CALL Input_PDU(pdu); NEXTSTATE -;
2855    STATE ~;INPUT   CCcf(pdu); TASK pdu!kind := CCcf; CALL Input_PDU(pdu); NEXTSTATE -;
2856    STATE ~;INPUT   CDrq(pdu); TASK pdu!kind := CDrq; CALL Input_PDU(pdu); NEXTSTATE -;
2857    STATE ~;INPUT   CDin(pdu); TASK pdu!kind := CDin; CALL Input_PDU(pdu); NEXTSTATE -;
2858    STATE ~;INPUT   CArq(pdu); TASK pdu!kind := CArq; CALL Input_PDU(pdu); NEXTSTATE -;
2859    STATE ~;INPUT   CAin(pdu); TASK pdu!kind := CAin; CALL Input_PDU(pdu); NEXTSTATE -;
2860    STATE ~;INPUT   CErq(pdu); TASK pdu!kind := CErq; CALL Input_PDU(pdu); NEXTSTATE -;
2861    STATE ~;INPUT   CEin(pdu); TASK pdu!kind := CEin; CALL Input_PDU(pdu); NEXTSTATE -;
2862    STATE ~;INPUT   SDrq(pdu); TASK pdu!kind := SDrq; CALL Input_PDU(pdu); NEXTSTATE -;
2863    STATE ~;INPUT   SDin(pdu); TASK pdu!kind := SDin; CALL Input_PDU(pdu); NEXTSTATE -;
2864    STATE ~;INPUT   USrq(pdu); TASK pdu!kind := USrq; CALL Input_PDU(pdu); NEXTSTATE -;
2865    STATE ~;INPUT   USin(pdu); TASK pdu!kind := USin; CALL Input_PDU(pdu); NEXTSTATE -;
2866    STATE ~;INPUT   TGrq(pdu); TASK pdu!kind := TGrq; CALL Input_PDU(pdu); NEXTSTATE -;
2867    STATE ~;INPUT   TGcf(pdu); TASK pdu!kind := TGcf; CALL Input_PDU(pdu); NEXTSTATE -;
2868    STATE ~;INPUT   TIrq(pdu); TASK pdu!kind := TIrq; CALL Input_PDU(pdu); NEXTSTATE -;
2869    STATE ~;INPUT   TIcf(pdu); TASK pdu!kind := TIcf; CALL Input_PDU(pdu); NEXTSTATE -;
2870    STATE ~;INPUT   TVrq(pdu); TASK pdu!kind := TVrq; CALL Input_PDU(pdu); NEXTSTATE -;
2871    STATE ~;INPUT   TVin(pdu); TASK pdu!kind := TVin; CALL Input_PDU(pdu); NEXTSTATE -;
2872    STATE ~;INPUT   TVrs(pdu); TASK pdu!kind := TVrs; CALL Input_PDU(pdu); NEXTSTATE -;
2873    STATE ~;INPUT   TVcf(pdu); TASK pdu!kind := TVcf; CALL Input_PDU(pdu); NEXTSTATE -;
2874    STATE ~;INPUT   TPrq(pdu); TASK pdu!kind := TPrq; CALL Input_PDU(pdu); NEXTSTATE -;
2875    STATE ~;INPUT   TPin(pdu); TASK pdu!kind := TPin; CALL Input_PDU(pdu); NEXTSTATE -;
2876    STATE ~;INPUT   TRrq(pdu); TASK pdu!kind := TRrq; CALL Input_PDU(pdu); NEXTSTATE -;
2877    STATE ~;INPUT   TRcf(pdu); TASK pdu!kind := TRcf; CALL Input_PDU(pdu); NEXTSTATE -;
2878    STATE ~;INPUT   TTrq(pdu); TASK pdu!kind := TTrq; CALL Input_PDU(pdu); NEXTSTATE -;
2879    STATE ~;INPUT   TTcf(pdu); TASK pdu!kind := TTcf; CALL Input_PDU(pdu); NEXTSTATE -;

2881    ENDPROCESS;
```

Appendix V
SDL specification of the Endpoint process

(This appendix does not form an integral part of this Recommendation.)

```
 1    PROCESS Endpoint;
 2    FPAR    tcId            TCEndpointId,           /* incoming TC if not Null */
 3            localTSAP       TSAPAddress,            /* own transport address */
 4            remoteTSAP      TSAPAddress,            /* other transport address */
 5            givenQOS        TransportQOS,           /* target or offered QOS */
 6            minQOS          TransportQOS,           /* minimum acceptable QOS */
 7            parameters      DomainParameters;       /* values established or null */

 9            /* Data declarations */

11    DCL     control         PId,                /* single Control process */
12            domain          PId;                /* selected Domain process */

14            /* Input transitions */

16            DCL     localDomain     DomainSelector,
17                    remoteDomain    DomainSelector,
18                    upward          Boolean,
19                    targetParms     DomainParameters,
20                    minParms        DomainParameters,
21                    maxParms        DomainParameters,
22                    userData        UserData,
23                    result          Result,
24                    ccId            Natural,
25                    label           Natural,
26                    tsdu            TSDU,
27                    dp              DataPriority,
28 |                  pdu             PDUStruct;
29            START
30            COMMENT'State machine:              NEXTSTATE
31                            1  connecting   * . 2 . . . 6
32                            2  connbusy     * . 2 3 . 5 6
33                            3  connready      . 2 3 . 5 6
34                            4  busy           . . . 4 5 6
35                            5  ready          . . . 4 5 6
36                            6  disconnected   . . . . . 6
37                    ';
38            TASK    control := PARENT,
39                    label := 0;
40            DECISION tcId = Null;
41            (True): OUTPUT  T.Connect.request(label, localTSAP, remoteTSAP,
42                                                    givenQOS, minQOS);
43                    NEXTSTATE connecting;
44            (False):OUTPUT  T.Connect.response(tcId, givenQOS);
45                    OUTPUT  T.ready(tcId);
46                    NEXTSTATE connbusy;
47            ENDDECISION;

49    STATE *;
50            INPUT   Exit;
51            STOP;

53    STATE *;
54            INPUT   T.Disconnect.indication(label, tcId);
55            OUTPUT  Quit TO control;
56            NEXTSTATE disconnected;

58    STATE connecting;
59            INPUT   T.Connect.confirm(label, tcId, givenQOS);
60            OUTPUT  T.ready(tcId);
61            NEXTSTATE connbusy;

63    STATE connecting;
64            SAVE    *;  /* await outcome of TC */

66    STATE connbusy, connready, busy, ready;
67            INPUT   *;
68            OUTPUT  T.Disconnect.request(tcId);
69            OUTPUT  Quit TO control;
70            NEXTSTATE disconnected;

72    STATE connbusy;
```

```
 73          INPUT   T.ready(tcId);
 74          NEXTSTATE connready;

 76    STATE connbusy;
 77          SAVE    Connect.Initial, Connect.Response, Connect.Additional, Connect.Result;

 79    STATE connready;
 80          INPUT   Connect.Initial(localDomain, remoteDomain, upward,
 81                  targetParms, minParms, maxParms, userData);
 82          TASK'tsdu := encode Connect-Initial using BER';
 83           1b :
 84          OUTPUT  T.Data.request(tcId, tsdu);
 85          NEXTSTATE connbusy;

 87    STATE connready;
 88          INPUT   Connect.Response(result, ccId, parameters, userData);
 89          TASK'tsdu := encode Connect-Response using BER';
 90          JOIN 1b;

 92    STATE connready;
 93          INPUT   Connect.Additional(ccId, dp);
 94          TASK'tsdu := encode Connect-Additional using BER';
 95          JOIN 1b;

 97    STATE connready;
 98          INPUT   Connect.Result(result);
 99          TASK'tsdu := encode Connect-Result using BER';
100          JOIN 1b;

102   STATE connbusy, connready;
103         INPUT   T.Data.indication(tcId, tsdu);
104         TASK'connect MCSPDU := decode tsdu using BER';
105         DECISION'connect MCSPDU';
106         ('Connect-Initial'):
107                 OUTPUT  Connect.Initial(localDomain, remoteDomain, upward,
108                         targetParms, minParms, maxParms, userData) TO control;
109                 NEXTSTATE -;
110         ('Connect-Response'):
111                 OUTPUT  Connect.Response(result, ccId, parameters, userData) TO control;
112                 NEXTSTATE -;
113         ('Connect-Additional'):
114                 OUTPUT  Connect.Additional(ccId, dp) TO control;
115                 NEXTSTATE -;
116         ('Connect-Result'):
117                 OUTPUT  Connect.Result(result) TO control;
118                 NEXTSTATE -;
119         ELSE:
120                 OUTPUT  T.Disconnect.request(tcId);
121                 OUTPUT  Quit TO control;
122                 NEXTSTATE disconnected;
123         ENDDECISION;

125   STATE connbusy;
126         INPUT   PDU.ready(dp);
127         TASK    domain := SENDER;
128         OUTPUT  T.ready(tcId);
129         NEXTSTATE ready;

131   STATE connready;
132         INPUT   PDU.ready(dp);
133         TASK    domain := SENDER;
134         OUTPUT  PDU.ready(dp) TO domain;
135         OUTPUT  T.ready(tcId);
136         NEXTSTATE ready;

138   STATE busy;
139         INPUT   PDU.ready(dp);
140         OUTPUT  T.ready(tcId);
141         NEXTSTATE ready;

143   STATE busy, ready;
144         INPUT   T.ready(tcId);
145         OUTPUT  PDU.ready(dp) TO domain;
146         NEXTSTATE -;

148   STATE busy, ready;
149         INPUT   PDin(pdu); TASK pdu!kind := PDin; JOIN 2f;
150         INPUT   EDrq(pdu); TASK pdu!kind := EDrq; JOIN 2f;
151         INPUT   MCrq(pdu); TASK pdu!kind := MCrq; JOIN 2f;
```

```
152           INPUT   MCcf(pdu); TASK pdu!kind := MCcf; JOIN 2f;
153           INPUT   PCin(pdu); TASK pdu!kind := PCin; JOIN 2f;
154           INPUT   MTrq(pdu); TASK pdu!kind := MTrq; JOIN 2f;
155           INPUT   MTcf(pdu); TASK pdu!kind := MTcf; JOIN 2f;
156           INPUT   PTin(pdu); TASK pdu!kind := PTin; JOIN 2f;
157           INPUT   DPum(pdu); TASK pdu!kind := DPum; JOIN 2f;
158           INPUT   RJum(pdu); TASK pdu!kind := RJum; JOIN 2f;
159           INPUT   AUrq(pdu); TASK pdu!kind := AUrq; JOIN 2f;
160           INPUT   AUcf(pdu); TASK pdu!kind := AUcf; JOIN 2f;
161           INPUT   DUrq(pdu); TASK pdu!kind := DUrq; JOIN 2f;
162           INPUT   DUin(pdu); TASK pdu!kind := DUin; JOIN 2f;
163           INPUT   CJrq(pdu); TASK pdu!kind := CJrq; JOIN 2f;
164           INPUT   CJcf(pdu); TASK pdu!kind := CJcf; JOIN 2f;
165           INPUT   CLrq(pdu); TASK pdu!kind := CLrq; JOIN 2f;
166           INPUT   CCrq(pdu); TASK pdu!kind := CCrq; JOIN 2f;
167           INPUT   CCcf(pdu); TASK pdu!kind := CCcf; JOIN 2f;
168           INPUT   CDrq(pdu); TASK pdu!kind := CDrq; JOIN 2f;
169           INPUT   CDin(pdu); TASK pdu!kind := CDin; JOIN 2f;
170           INPUT   CArq(pdu); TASK pdu!kind := CArq; JOIN 2f;
171           INPUT   CAin(pdu); TASK pdu!kind := CAin; JOIN 2f;
172           INPUT   CErq(pdu); TASK pdu!kind := CErq; JOIN 2f;
173           INPUT   CEin(pdu); TASK pdu!kind := CEin; JOIN 2f;
174           INPUT   SDrq(pdu); TASK pdu!kind := SDrq; JOIN 2f;
175           INPUT   SDin(pdu); TASK pdu!kind := SDin; JOIN 2f;
176           INPUT   USrq(pdu); TASK pdu!kind := USrq; JOIN 2f;
177           INPUT   USin(pdu); TASK pdu!kind := USin; JOIN 2f;
178           INPUT   TGrq(pdu); TASK pdu!kind := TGrq; JOIN 2f;
179           INPUT   TGcf(pdu); TASK pdu!kind := TGcf; JOIN 2f;
180           INPUT   TIrq(pdu); TASK pdu!kind := TIrq; JOIN 2f;
181           INPUT   TIcf(pdu); TASK pdu!kind := TIcf; JOIN 2f;
182           INPUT   TVrq(pdu); TASK pdu!kind := TVrq; JOIN 2f;
183           INPUT   TVin(pdu); TASK pdu!kind := TVin; JOIN 2f;
184           INPUT   TVrs(pdu); TASK pdu!kind := TVrs; JOIN 2f;
185           INPUT   TVcf(pdu); TASK pdu!kind := TVcf; JOIN 2f;
186           INPUT   TPrq(pdu); TASK pdu!kind := TPrq; JOIN 2f;
187           INPUT   TPin(pdu); TASK pdu!kind := TPin; JOIN 2f;
188           INPUT   TRrq(pdu); TASK pdu!kind := TRrq; JOIN 2f;
189           INPUT   TRcf(pdu); TASK pdu!kind := TRcf; JOIN 2f;
190           INPUT   TTrq(pdu); TASK pdu!kind := TTrq; JOIN 2f;
191           INPUT   TTcf(pdu); TASK pdu!kind := TTcf;
192            2f :
193           TASK'tsdu := encode pdu using BER or PER,
194                depending on parameters!protocolVersion';
195           OUTPUT  T.Data.request(tcId, tsdu);
196           NEXTSTATE -;

198   STATE ready;
199           INPUT   T.Data.indication(tcId, tsdu);
200           TASK'pdu := decode tsdu using BER or PER,
201                depending on parameters!protocolVersion';
202           DECISION pdu!kind;
203           (RJum): OUTPUT  RJum(pdu) TO control;
204                   OUTPUT  T.ready(tcId);
205                   NEXTSTATE -;
206           (PDin): OUTPUT  PDin(pdu) TO domain; NEXTSTATE busy;
207           (EDrq): OUTPUT  EDrq(pdu) TO domain; NEXTSTATE busy;
208           (MCrq): OUTPUT  MCrq(pdu) TO domain; NEXTSTATE busy;
209           (MCcf): OUTPUT  MCcf(pdu) TO domain; NEXTSTATE busy;
210           (PCin): OUTPUT  PCin(pdu) TO domain; NEXTSTATE busy;
211           (MTrq): OUTPUT  MTrq(pdu) TO domain; NEXTSTATE busy;
212           (MTcf): OUTPUT  MTcf(pdu) TO domain; NEXTSTATE busy;
213           (PTin): OUTPUT  PTin(pdu) TO domain; NEXTSTATE busy;
214           (DPum): OUTPUT  DPum(pdu) TO domain; NEXTSTATE busy;
215           (AUrq): OUTPUT  AUrq(pdu) TO domain; NEXTSTATE busy;
216           (AUcf): OUTPUT  AUcf(pdu) TO domain; NEXTSTATE busy;
217           (DUrq): OUTPUT  DUrq(pdu) TO domain; NEXTSTATE busy;
218           (DUin): OUTPUT  DUin(pdu) TO domain; NEXTSTATE busy;
219           (CJrq): OUTPUT  CJrq(pdu) TO domain; NEXTSTATE busy;
220           (CJcf): OUTPUT  CJcf(pdu) TO domain; NEXTSTATE busy;
221           (CLrq): OUTPUT  CLrq(pdu) TO domain; NEXTSTATE busy;
222           (CCrq): OUTPUT  CCrq(pdu) TO domain; NEXTSTATE busy;
223           (CCcf): OUTPUT  CCcf(pdu) TO domain; NEXTSTATE busy;
224           (CDrq): OUTPUT  CDrq(pdu) TO domain; NEXTSTATE busy;
225           (CDin): OUTPUT  CDin(pdu) TO domain; NEXTSTATE busy;
226           (CArq): OUTPUT  CArq(pdu) TO domain; NEXTSTATE busy;
227           (CAin): OUTPUT  CAin(pdu) TO domain; NEXTSTATE busy;
228           (CErq): OUTPUT  CErq(pdu) TO domain; NEXTSTATE busy;
229           (CEin): OUTPUT  CEin(pdu) TO domain; NEXTSTATE busy;
230           (SDrq): OUTPUT  SDrq(pdu) TO domain; NEXTSTATE busy;
```

```
231             (SDin): OUTPUT  SDin(pdu) TO domain; NEXTSTATE busy;
232             (USrq): OUTPUT  USrq(pdu) TO domain; NEXTSTATE busy;
233             (USin): OUTPUT  USin(pdu) TO domain; NEXTSTATE busy;
234             (TGrq): OUTPUT  TGrq(pdu) TO domain; NEXTSTATE busy;
235             (TGcf): OUTPUT  TGcf(pdu) TO domain; NEXTSTATE busy;
236             (TIrq): OUTPUT  TIrq(pdu) TO domain; NEXTSTATE busy;
237             (TIcf): OUTPUT  TIcf(pdu) TO domain; NEXTSTATE busy;
238             (TVrq): OUTPUT  TVrq(pdu) TO domain; NEXTSTATE busy;
239             (TVin): OUTPUT  TVin(pdu) TO domain; NEXTSTATE busy;
240             (TVrs): OUTPUT  TVrs(pdu) TO domain; NEXTSTATE busy;
241             (TVcf): OUTPUT  TVcf(pdu) TO domain; NEXTSTATE busy;
242             (TPrq): OUTPUT  TPrq(pdu) TO domain; NEXTSTATE busy;
243             (TPin): OUTPUT  TPin(pdu) TO domain; NEXTSTATE busy;
244             (TRrq): OUTPUT  TRrq(pdu) TO domain; NEXTSTATE busy;
245             (TRcf): OUTPUT  TRcf(pdu) TO domain; NEXTSTATE busy;
246             (TTrq): OUTPUT  TTrq(pdu) TO domain; NEXTSTATE busy;
247             (TTcf): OUTPUT  TTcf(pdu) TO domain; NEXTSTATE busy;
248             ELSE:
249                     TASK'pdu!initialOctets := truncate tsdu';
250                     TASK'pdu!diagnostic := DC_invalid_?ER_encoding';
251                     OUTPUT  RJum(pdu) TO domain;
252                     NEXTSTATE busy;
253             ENDDECISION;

255     STATE disconnected;
256             INPUT   Quit;
257             OUTPUT  Quit TO control;
258             NEXTSTATE disconnected;

260     ENDPROCESS;
```

Appendix VI
SDL specification of the Attachment process

(This appendix does not form an integral part of this Recommendation.)

```
 1    PROCESS Attachment;
 2 |  FPAR    label           Natural,                /* for MCS.Attach.User.confirm */
 3           parameters      DomainParameters;       /* values established in domain */

 5           /* Type definitions */

 7 |  NEWTYPE         MCSRequest
 8    STRUCT
 9 |         kind            PDUKind;            /* SDrq, USrq, CArq, CErq */
10 *         channelId       ChannelId;          /* parameter of request */
11          segmentation    Segmentation;       /* parameter of request */
12          userData        UserData;           /* parameter of request */
13          offset          Integer;            /* octets sent so far */
14 +         userIds         UserIdSet;          /* remaining to affect */
15   ENDNEWTYPE;

17   NEWTYPE         PrioritySet         SetOf(DataPriority);
18   ENDNEWTYPE;

20 |  NEWTYPE         MCSRequestByPri     Array(DataPriority, MCSRequest);
21   ENDNEWTYPE;
22   NEWTYPE         PDUStructByPri      Array(DataPriority, PDUStruct);
23   ENDNEWTYPE;

25          /* Data declarations */

27   DCL     maId            MCSAttachmentId,    /* this Attachment process */
28           control         PId,                /* single Control process */
29           domain          PId;                /* selected Domain process */

31   DCL     user            UserId;             /* unique id of attached user */

33   DCL     cJoined         ChannelIdSet,       /* channels the user has joined */
34           cConvened       ChannelIdSet,       /* channels the user has convened */
35           cAdmitted       ChannelIdSet;       /* channels user was admitted to */

37   DCL     tPossessed      TokenIdSet,         /* tokens grabbed or inhibited */
38           tRecipient      TokenIdSet;         /* tokens given waiting response */

40 |  DCL     uPending        PrioritySet,        /* request is pending at 0..3 */
41 |         mcsreq          MCSRequestByPri,    /* content of segmented request */
42 |         dReady          PrioritySet;        /* domain MCSPDU allowed 0..? */

44   DCL     dPending        PrioritySet,        /* SDin or USin pending at 0..3 */
45           mcspdu          PDUStructByPri,     /* content of the domain MCSPDU */
46           uReady          PrioritySet;        /* data indication allowed 0..3 */

48           /* Procedure decomposition */

50 |         /*      Segment_request         (dp)
51                   Indicate_data
52                   Track_token             (t, status)     */

54 |                                                          /*-------------------*/
55 |  PROCEDURE       Segment_request;                        /*  Segment_request  */
56 |  FPAR            dp      DataPriority;                   /*-------------------*/

58           DCL     udp     DataPriority,
59 |                 req     MCSRequest,
60                   pdu     PDUStruct,
61                   b       Boolean,
62                   m       Integer,
63 |                 n       Integer,
64 |                 u       UserId;
65           START
66 |         COMMENT'Feed domain process the next segment of user data
67 |                 or the next subset of private channel user ids,
68                   if ready, for the specified transport priority.
69                   ';
70           DECISION dp in dReady;
71           (False):RETURN;
72           ELSE:ENDDECISION;
```

```
          TASK    udp := dp;
           1b :   /* for udp = dp..? */
          DECISION udp = dp or (udp >= parameters!numPriorities and udp < 4);
          (False):RETURN;
          (True): DECISION udp in uPending;
                  (False):TASK    udp := udp + 1;
                          JOIN 1b;
                  ELSE:ENDDECISION;
          ENDDECISION;
          TASK    dReady := Del(dp, dReady),
                  dp := udp,
                  req := mcsreq(dp),
                  pdu!initiator := user,
                  pdu!channelId := req!channelId;
          DECISION req!kind;
          (SDrq, USrq):
                  TASK    pdu!dataPriority := dp,
                          b := req!segmentation!begin,
                          pdu!segmentation!begin := IF req!offset = 0 THEN b ELSE False FI,
                          m := parameters!maxMCSPDUsize,
                          m := IF parameters!protocolVersion = 1 THEN m - 24 ELSE m - 8 FI,
                          n := Length(req!userData) - req!offset,
                          n := IF n > m THEN m ELSE n FI,
                          pdu!userData := Substring(req!userData, 1 + req!offset, n),
                          req!offset := req!offset + n,
                          n := Length(req!userData) - req!offset,
                          b := req!segmentation!end,
                          pdu!segmentation!end := IF n = 0 THEN b ELSE False FI,
                          mcsreq(dp)!offset := req!offset;
          (CArq, CErq):
                  TASK    pdu!userIds := Empty,
                          n := 0,
                          m := parameters!maxMCSPDUsize,
                          m := IF parameters!protocolVersion = 1 THEN (m - 16) / 4
                                          ELSE (m - 7) / 2 FI;
                   2b :   /* while n < m */
                  DECISION req!userIds = Empty;
                  (True): TASK    n := 0;
                  (False):TASK    u := Pick(req!userIds),
                                  req!userIds := Del(u, req!userIds);
                          DECISION u >= 1001;
                          (True): TASK    pdu!userIds := Incl(u, pdu!userIds);
                          ELSE:ENDDECISION;
                          TASK    n := n + 1;
                          DECISION n < m;
                          (True): JOIN 2b;
                          ELSE:ENDDECISION;
                  ENDDECISION;
          ENDDECISION;
          DECISION req!kind;
          (SDrq): OUTPUT  SDrq(pdu) TO domain;
          (USrq): OUTPUT  USrq(pdu) TO domain;
          (CArq): OUTPUT  CArq(pdu) TO domain;
          (CErq): OUTPUT  CErq(pdu) TO domain;
          ENDDECISION;
          DECISION n = 0;
          (True): TASK    uPending := Del(dp, uPending);
                  DECISION req!kind;
                  (SDrq, USrq):
                          OUTPUT  MCS.ready(maId, dp);
                  ELSE:ENDDECISION;
          ELSE:ENDDECISION;
          RETURN;
          ENDPROCEDURE;

                                                                  /*-----------------*/
   PROCEDURE      Indicate_data;                                  /*  Indicate_data  */
                                                                  /*-----------------*/
          DCL     dp      DataPriority,
                  pdu     PDUStruct;
          START
          COMMENT'Indicate the receipt of user data, if ready
                  and none pending at higher priorities.
                  ';
          TASK    dp := 0;
           1b :   /* for dp = 0..3 */
          DECISION dp < 4 and (dp in uReady or not dp in dPending);
          (False):RETURN;
          (True): DECISION dp in dPending;
```

```
152                     (False):TASK    dp := dp + 1;
153                             JOIN 1b;
154                     ELSE:ENDDECISION;
155             ENDDECISION;
156             TASK    dPending := Del(dp, dPending),
157                     pdu := mcspdu(dp);
158             DECISION pdu!kind;
159             (SDin): OUTPUT  MCS.Send.Data.indication(maId, pdu!channelId,
160                             pdu!dataPriority, pdu!initiator, pdu!segmentation, pdu!userData);
161             (USin): OUTPUT  MCS.Uniform.Send.Data.indication(maId, pdu!channelId,
162                             pdu!dataPriority, pdu!initiator, pdu!segmentation, pdu!userData);
163             ENDDECISION;
164             TASK    uReady := Del(dp, uReady),
165                     dp := IF dp < parameters!numPriorities THEN dp
166                                     ELSE parameters!numPriorities - 1 FI;
167             OUTPUT  PDU.ready(dp) TO domain;
168 |           JOIN 1b;
169             ENDPROCEDURE;

171                                                                     /*---------------*/
172     PROCEDURE       Track_token;                                    /*  Track_token  */
173     FPAR            t       TokenId,                                /*---------------*/
174                     status  TokenStatus;

176             START
177             COMMENT'Condense status into possessed or recipient.
178                     ';
179             TASK    tPossessed := Del(t, tPossessed),
180                     tRecipient := Del(t, tRecipient);
181             DECISION status;
182             (SelfGrabbed, SelfInhibited, SelfGiving):
183                     TASK    tPossessed := Incl(t, tPossessed);
184             (SelfRecipient):
185                     TASK    tRecipient := Incl(t, tRecipient);
186             ELSE:ENDDECISION;
187             RETURN;
188             ENDPROCEDURE;

190             /* Input transitions */

192             DCL     dp              DataPriority,
193                     pdu             PDUStruct,
194                     c               ChannelId,
195                     cSet            ChannelIdSet,
196                     t               TokenId,
197                     u               UserId,
198                     uSet            UserIdSet,
199                     segmentation    Segmentation,
200                     userData        UserData,
201                     result          Result,
202                     kind            PDUKind,
203                     reason          Reason;
204             START
205             COMMENT'State machine:          NEXTSTATE
206                             1  initial    * . 2 . . . 6
207                             2  attaching    . 2 3 4 . 6
208                             3  busy         . . 3 4 5 6
209                             4  ready        . . 3 4 . 6
210                             5  detaching    . . . . 5 6
211                             6  detached     . . . . . 6
212                     ';
213             TASK    maId := SELF,
214 |                   control := PARENT,
215 |                   user := 0;
216             NEXTSTATE initial;

218     STATE *;
219             INPUT   Exit;
220             STOP;

222     STATE initial, attaching;
223             INPUT   *;
224 |           OUTPUT  MCS.Attach.User.confirm(label, RT_unspecified_failure, maId, user);
225             OUTPUT  Quit TO control;
226             NEXTSTATE detached;

228     STATE initial, attaching;
229             INPUT   Quit;
230 |           OUTPUT  MCS.Attach.User.confirm(label, RT_domain_disconnected, maId, user);
```

```
231             OUTPUT  Quit TO control;
232             NEXTSTATE detached;

234     STATE initial;
235             INPUT   PDU.ready(dp);
236             TASK    domain := SENDER;
237             DECISION dp = 0;
238             (False):TASK    dReady := Incl(dp, dReady);
239                     NEXTSTATE -;
240             ELSE:ENDDECISION;
241             OUTPUT  PDU.ready(0) TO domain;
242             OUTPUT  AUrq(pdu) TO domain;
243             NEXTSTATE attaching;

245     STATE attaching;
246             INPUT   PDU.ready(dp);
247             TASK    dReady := Incl(dp, dReady);
248             NEXTSTATE -;

250     STATE attaching;
251             INPUT   PCin(pdu), PTin(pdu), DUin(pdu);
252             /* no action */
253             NEXTSTATE -;

255     STATE attaching;
256             INPUT   AUcf(pdu);
257             TASK    user := pdu!initiator;
258 |           OUTPUT  MCS.Attach.User.confirm(label, pdu!result, maId, user);
259             DECISION pdu!result = RT_successful;
260             (False):OUTPUT  Quit TO control;
261                     NEXTSTATE detached;
262             (True): TASK    dp := 0;
263                      1b :   /* for dp = 0..3 */
264                     DECISION dp < 4;
265                     (True): OUTPUT  MCS.ready(maId, dp);
266                             DECISION dp < parameters!numPriorities;
267                             (True): OUTPUT  PDU.ready(dp) TO domain;
268                             ELSE:ENDDECISION;
269                             TASK    dp := dp + 1;
270                             JOIN 1b;
271                     ELSE:ENDDECISION;
272                     DECISION 0 in dReady;
273                     (False):NEXTSTATE busy;
274                     (True): NEXTSTATE ready;
275                     ENDDECISION;
276             ENDDECISION;

278     STATE busy, ready;
279             INPUT   Quit;
280             OUTPUT  MCS.Detach.User.indication(maId, user, RN_provider_initiated);
281             OUTPUT  Quit TO control;
282             NEXTSTATE detached;

284     STATE busy, ready;
285             INPUT   MCS.ready(maId, dp);
286             TASK    uReady := Incl(dp, uReady);
287             CALL    Indicate_data;
288             NEXTSTATE -;

290     STATE busy;
291             INPUT   MCS.Detach.User.request(maId);
292             TASK    reason := RN_user_requested;
293             NEXTSTATE detaching;

295     STATE busy;
296             SAVE    *;  /* defer MCS other */

298     STATE ready;
299             INPUT   MCS.Detach.User.request(maId);
300             TASK    pdu!reason := RN_user_requested,
301                     pdu!userIds := Incl(user, Empty);
302             OUTPUT  DUrq(pdu) TO domain;
303             NEXTSTATE detached;

305     STATE ready;
306             INPUT   MCS.Channel.Join.request(maId, c);
307             TASK    pdu!initiator := user,
308                     pdu!channelId := c;
309             OUTPUT  CJrq(pdu) TO domain;
```

```
310             2b :
311             TASK    dReady := Del(0, dReady);
312             NEXTSTATE busy;

314     STATE ready;
315             INPUT   MCS.Channel.Leave.request(maId, c);
316             TASK    cJoined := Del(c, cJoined),
317                     pdu!channelIds := Incl(c, Empty);
318             OUTPUT  CLrq(pdu) TO domain;
319             JOIN 2b;

321     STATE ready;
322             INPUT   MCS.Channel.Convene.request(maId);
323             TASK    pdu!initiator := user;
324             OUTPUT  CCrq(pdu) TO domain;
325             JOIN 2b;

327     STATE ready;
328             INPUT   MCS.Channel.Disband.request(maId, c);
329             DECISION c in cConvened;
330             (True): TASK    cAdmitted := Del(c, cAdmitted),
331                             cJoined := Del(c, cJoined);
332             ELSE:ENDDECISION;
333             TASK    cConvened := Del(c, cConvened),
334                     pdu!initiator := user,
335                     pdu!channelId := c;
336             OUTPUT  CDrq(pdu) TO domain;
337             JOIN 2b;

339     STATE ready;
340             INPUT   MCS.Channel.Admit.request(maId, c, uSet);
341 |           TASK    kind := CArq;
342 |           JOIN 3f;

344     STATE ready;
345             INPUT   MCS.Channel.Expel.request(maId, c, uSet);
346 |           TASK    kind := CErq;
347 |            3f :
348 |           TASK    mcsreq(0)!kind := kind,
349 |                   mcsreq(0)!channelId := c,
350 |                   mcsreq(0)!userIds := uSet,
351 |                   uPending := Incl(0, uPending);
352 |           CALL    Segment_request(0);
353 |           DECISION 0 in dReady;
354 |           (False):NEXTSTATE busy;
355 |           (True): NEXTSTATE ready;
356 |           ENDDECISION;

358     STATE ready;
359             INPUT   MCS.Send.Data.request(maId, c, dp, segmentation, userData);
360             TASK    kind := SDrq;
361 |           JOIN 4f;

363     STATE ready;
364             INPUT   MCS.Uniform.Send.Data.request(maId, c, dp, segmentation, userData);
365             TASK    kind := USrq;
366 |           4f :
367 |           DECISION dp >= 4 or dp in uPending;
368             (True): OUTPUT  MCS.Detach.User.indication(maId, user, RN_provider_initiated);
369                     TASK    pdu!reason := RN_provider_initiated,
370                             pdu!userIds := Incl(user, Empty);
371                     OUTPUT  DUrq(pdu) TO domain;
372                     NEXTSTATE detached;
373 |           (False):TASK    mcsreq(dp)!kind := kind,
374 |                           mcsreq(dp)!channelId := c,
375 |                           mcsreq(dp)!segmentation := segmentation,
376 |                           mcsreq(dp)!userData := userData,
377 |                           mcsreq(dp)!offset := 0,
378                             uPending := Incl(dp, uPending),
379                             dp := IF dp < parameters!numPriorities THEN dp
380                                             ELSE parameters!numPriorities - 1 FI;
381 |                   CALL    Segment_request(dp);
382                     DECISION 0 in dReady;
383                     (False):NEXTSTATE busy;
384                     (True): NEXTSTATE ready;
385                     ENDDECISION;
386             ENDDECISION;

388     STATE ready;
```

```
389             INPUT   MCS.Token.Grab.request(maId, t);
390             TASK    pdu!initiator := user,
391                     pdu!tokenId := t;
392             OUTPUT  TGrq(pdu) TO domain;
393             JOIN 2b;

395     STATE ready;
396             INPUT   MCS.Token.Inhibit.request(maId, t);
397             TASK    pdu!initiator := user,
398                     pdu!tokenId := t;
399             OUTPUT  TIrq(pdu) TO domain;
400             JOIN 2b;

402     STATE ready;
403             INPUT   MCS.Token.Give.request(maId, t, u);
404 |           DECISION u >= 1001;
405 |           (False):OUTPUT  MCS.Token.Give.confirm(maId, t, RT_no_such_user);
406 |                   NEXTSTATE -;
407 |           (True): TASK    pdu!initiator := user,
408 |                           pdu!tokenId := t,
409 |                           pdu!recipient := u;
410 |                   OUTPUT  TVrq(pdu) TO domain;
411 |                   JOIN 2b;
412 |           ENDDECISION;

414     STATE ready;
415             INPUT   MCS.Token.Give.response(maId, t, result);
416             DECISION t in tRecipient and result = RT_successful;
417             (True): TASK    tPossessed := Incl(t, tPossessed);
418             ELSE:ENDDECISION;
419             TASK    tRecipient := Del(t, tRecipient),
420                     pdu!result := result,
421                     pdu!recipient := user,
422                     pdu!tokenId := t;
423             OUTPUT  TVrs(pdu) TO domain;
424             JOIN 2b;

426     STATE ready;
427             INPUT   MCS.Token.Please.request(maId, t);
428             TASK    pdu!initiator := user,
429                     pdu!tokenId := t;
430             OUTPUT  TPrq(pdu) TO domain;
431             JOIN 2b;

433     STATE ready;
434             INPUT   MCS.Token.Release.request(maId, t);
435             TASK    tPossessed := Del(t, tPossessed),
436                     pdu!initiator := user,
437                     pdu!tokenId := t;
438             OUTPUT  TRrq(pdu) TO domain;
439             JOIN 2b;

441     STATE ready;
442             INPUT   MCS.Token.Test.request(maId, t);
443             TASK    pdu!initiator := user,
444                     pdu!tokenId := t;
445             OUTPUT  TTrq(pdu) TO domain;
446             JOIN 2b;

448     STATE busy, ready;
449             INPUT   PDU.ready(dp);
450             TASK    dReady := Incl(dp, dReady);
451 |           CALL    Segment_request(dp);
452             DECISION 0 in dReady;
453             (False):NEXTSTATE busy;
454             (True): NEXTSTATE ready;
455             ENDDECISION;

457     STATE busy, ready;
458             INPUT   PCin(pdu);
459 |           TASK    reason := RN_channel_purged;
460 |           DECISION user in pdu!detachUserIds;
461             (True): OUTPUT  MCS.Detach.User.indication(maId, user, reason);
462                     OUTPUT  Quit TO control;
463                     NEXTSTATE detached;
464 |           (False):TASK    uSet := pdu!detachUserIds;
465 |                    5b :   /* for u in uSet */
466 |                   DECISION uSet = Empty;
467 |                   (False):TASK    u := Pick(uSet),
```

```
468 |                                   uSet := Del(u, uSet);
469 *                           OUTPUT  MCS.Detach.User.indication(maId, u, reason);
470                             JOIN 5b;
471                     ELSE:ENDDECISION;
472 |                   TASK    cSet := pdu!purgeChannelIds;
473                      6b :   /* for c in cSet */
474                     DECISION cSet = Empty;
475                     (False):TASK    c := Pick(cSet),
476                                     cSet := Del(c, cSet);
477                             DECISION c in cConvened;
478                             (True): TASK    cConvened := Del(c, cConvened),
479                                             cAdmitted := Del(c, cAdmitted),
480                                             cJoined := Del(c, cJoined);
481                                     OUTPUT  MCS.Channel.Disband.indication(maId, c, reason);
482                             ELSE:ENDDECISION;
483                             DECISION c in cAdmitted;
484                             (True): TASK    cAdmitted := Del(c, cAdmitted),
485                                             cJoined := Del(c, cJoined);
486                                     OUTPUT  MCS.Channel.Expel.indication(maId, c, reason);
487                             ELSE:ENDDECISION;
488 +                           DECISION c in cJoined;
489 +                           (True): TASK    cJoined := Del(c, cJoined);
490 +                                   OUTPUT  MCS.Channel.Leave.indication(maId, c, reason);
491 +                           ELSE:ENDDECISION;
492                             JOIN 6b;
493                     ELSE:ENDDECISION;
494                     OUTPUT  PDU.ready(0) TO domain;
495                     NEXTSTATE -;
496             ENDDECISION;

498     STATE busy;
499             INPUT   PTin(pdu);
500             DECISION (pdu!purgeTokenIds and (tPossessed or tRecipient)) = Empty;
501             (False):OUTPUT  MCS.Detach.User.indication(maId, user, RN_token_purged);
502                     TASK    reason := RN_token_purged;
503                     NEXTSTATE detaching;
504             (True): OUTPUT  PDU.ready(0) TO domain;
505                     NEXTSTATE -;
506             ENDDECISION;

508     STATE ready;
509             INPUT   PTin(pdu);
510             DECISION (pdu!purgeTokenIds and (tPossessed or tRecipient)) = Empty;
511             (False):OUTPUT  MCS.Detach.User.indication(maId, user, RN_token_purged);
512                     TASK    pdu!reason := RN_token_purged,
513                             pdu!userIds := Incl(user, Empty);
514                     OUTPUT  DUrq(pdu) TO domain;
515                     NEXTSTATE detached;
516             (True): OUTPUT  PDU.ready(0) TO domain;
517                     NEXTSTATE -;
518             ENDDECISION;

520     STATE busy, ready;
521             INPUT   AUcf(pdu);
522             OUTPUT  MCS.Detach.User.indication(maId, user, RN_unspecified);
523             OUTPUT  Quit TO control;
524             NEXTSTATE detached;

526     STATE busy, ready;
527             INPUT   DUin(pdu);
528             DECISION user in pdu!userIds;
529             (True): OUTPUT  MCS.Detach.User.indication(maId, user, pdu!reason);
530                     OUTPUT  Quit TO control;
531                     NEXTSTATE detached;
532             (False):TASK    uSet := pdu!userIds;
533                      7b :   /* for u in uSet */
534                     DECISION uSet = Empty;
535                     (False):TASK    u := Pick(uSet),
536                                     uSet := Del(u, uSet);
537                             OUTPUT  MCS.Detach.User.indication(maId, u, pdu!reason);
538                             JOIN 7b;
539                     ELSE:ENDDECISION;
540                     OUTPUT  PDU.ready(0) TO domain;
541                     NEXTSTATE -;
542             ENDDECISION;

544     STATE busy, ready;
545             INPUT   CJcf(pdu);
546             TASK    c := pdu!channelId;
```

```
547             DECISION pdu!result = RT_successful;
548             (True): TASK    cJoined := Incl(c, cJoined);
549             ELSE:ENDDECISION;
550 |           OUTPUT  MCS.Channel.Join.confirm(maId, pdu!requested, pdu!result, c);
551              8b :
552             OUTPUT  PDU.ready(0) TO domain;
553             NEXTSTATE -;

555     STATE busy, ready;
556             INPUT   CCcf(pdu);
557             TASK    c := pdu!channelId;
558             DECISION pdu!result = RT_successful;
559             (True): TASK    cConvened := Incl(c, cConvened),
560                             cAdmitted := Incl(c, cAdmitted);
561             ELSE:ENDDECISION;
562             OUTPUT  MCS.Channel.Convene.confirm(maId, pdu!result, c);
563             JOIN 8b;

565     STATE busy, ready;
566             INPUT   CDin(pdu);
567             TASK    c := pdu!channelId;
568             DECISION c in cAdmitted;
569             (True): OUTPUT  MCS.Channel.Expel.indication(maId, c, RN_channel_disbanded);
570             ELSE:ENDDECISION;
571             TASK    cAdmitted := Del(c, cAdmitted),
572                     cJoined := Del(c, cJoined);
573             JOIN 8b;

575     STATE busy, ready;
576             INPUT   CAin(pdu);
577             TASK    c := pdu!channelId,
578                     cAdmitted := Incl(c, cAdmitted);
579             OUTPUT  MCS.Channel.Admit.indication(maId, c, pdu!initiator);
580             JOIN 8b;

582     STATE busy, ready;
583             INPUT   CEin(pdu);
584 |           TASK    c := pdu!channelId;
585 |           DECISION c in cAdmitted;
586 |           (True): OUTPUT  MCS.Channel.Expel.indication(maId, c, RN_user_requested);
587 |           ELSE:ENDDECISION;
588 |           TASK    cAdmitted := Del(c, cAdmitted),
589 *                   cJoined := Del(c, cJoined);
590             JOIN 8b;

592     STATE busy, ready;
593             INPUT   SDin(pdu);
594             TASK    pdu!kind := SDin;
595             JOIN 9f;

597     STATE busy, ready;
598             INPUT   USin(pdu);
599             TASK    pdu!kind := USin;
600              9f :
601 |           TASK    c := pdu!channelId,
602 |                   dp := pdu!dataPriority;
603 |           DECISION c in cJoined;
604 |           (True): TASK    mcspdu(dp) := pdu,
605 |                           dPending := Incl(dp, dPending);
606 |                   CALL    Indicate_data;
607 |           (False):TASK    dp := IF dp < parameters!numPriorities THEN dp
608 |                                       ELSE parameters!numPriorities - 1 FI;
609 |                   OUTPUT  PDU.ready(dp) TO domain;
610 |           ENDDECISION;
611 |           NEXTSTATE -;

613     STATE busy, ready;
614             INPUT   TGcf(pdu);
615             CALL    Track_token(pdu!tokenId, pdu!tokenStatus);
616             OUTPUT  MCS.Token.Grab.confirm(maId, pdu!tokenId, pdu!result);
617             JOIN 8b;

619     STATE busy, ready;
620             INPUT   TIcf(pdu);
621             CALL    Track_token(pdu!tokenId, pdu!tokenStatus);
622             OUTPUT  MCS.Token.Inhibit.confirm(maId, pdu!tokenId, pdu!result);
623             JOIN 8b;

625     STATE busy, ready;
```

```
626             INPUT   TVin(pdu);
627             TASK    tRecipient := Incl(pdu!tokenId, tRecipient);
628             OUTPUT  MCS.Token.Give.indication(maId, pdu!tokenId, pdu!initiator);
629             JOIN 8b;

631     STATE busy, ready;
632             INPUT   TVcf(pdu);
633             CALL    Track_token(pdu!tokenId, pdu!tokenStatus);
634             OUTPUT  MCS.Token.Give.confirm(maId, pdu!tokenId, pdu!result);
635             JOIN 8b;

637     STATE busy, ready;
638             INPUT   TPin(pdu);
639 |           TASK    t := pdu!tokenId;
640 |           DECISION t in tPossessed or t in tRecipient;
641 |           (True): OUTPUT  MCS.Token.Please.indication(maId, t, pdu!initiator);
642 |           ELSE:ENDDECISION;
643             JOIN 8b;

645     STATE busy, ready;
646             INPUT   TRcf(pdu);
647             CALL    Track_token(pdu!tokenId, pdu!tokenStatus);
648             OUTPUT  MCS.Token.Release.confirm(maId, pdu!tokenId, pdu!result);
649             JOIN 8b;

651     STATE busy, ready;
652             INPUT   TTcf(pdu);
653             OUTPUT  MCS.Token.Test.confirm(maId, pdu!tokenId, pdu!tokenStatus);
654             JOIN 8b;

656     STATE detaching, detached;
657             INPUT   *;
658             NEXTSTATE -;

660     STATE detaching, detached;
661             INPUT   Quit;
662             OUTPUT  Quit TO control;
663             NEXTSTATE detached;

665     STATE detaching;
666             INPUT   PDU.ready(dp);
667             DECISION dp = 0;
668             (False):NEXTSTATE -;
669             (True): TASK    pdu!reason := reason,
670                             pdu!userIds := Incl(user, Empty);
671                     OUTPUT  DUrq(pdu) TO domain;
672                     NEXTSTATE detached;
673             ENDDECISION;

675     STATE detaching, detached;
676 +           INPUT   PCin(pdu);
677 +           DECISION user in pdu!detachUserIds;
678 +           (False):NEXTSTATE -;
679 +           (True): OUTPUT  Quit TO control;
680 +                   NEXTSTATE detached;
681 +           ENDDECISION;

683 +   STATE detaching, detached;
684             INPUT   DUin(pdu);
685             DECISION user in pdu!userIds;
686             (False):NEXTSTATE -;
687             (True): OUTPUT  Quit TO control;
688                     NEXTSTATE detached;
689             ENDDECISION;

691     ENDPROCESS;
```

Appendix VII
Characteristics of the reference implementation

(This appendix does not form an integral part of this Recommendation.)

VII.1 SDL decomposition

Figure II-1 depicts an MCS provider in the context of a system, with three channels binding it to its environment: Control.MCSAP to a single controller application, MCSAPs to zero or more attached users, and TSAPs to zero or more transport service providers. In a decomposition of the provider, these external channels connect with signal routes to and from component processes.

Each octagonal element represents a process type and gives bounds (minimum, maximum) on its number of instances. One Control process exists permanently. Dashed lines show that it creates instances of the remaining three processes. Attachment, Domain, and Endpoint do not exist initially, but their potential number is unlimited.

Signal routes to and from component processes convey SDL signals. The set of signals transmitted in a given direction is shown near the corresponding arrow head. Identifiers in parentheses are lists of related signals. The expansion of these lists is to long to include in the figure. Details appear in the text of Appendix II.

VII.2 Service definitions

Signals over the external channels represent uses of MCS and the transport services defined abstractly in ITU-T Rec. T.122 and CCITT Rec. X.214. Modeling these in SDL requires further assumptions about details of the interactions. Two aspects are worth special note: a use of endpoints identifiers to specify the context of a service primitive and a use of flow control ready signals to invite data transfer.

The three identifiers involved are: MCSConnectionId, MCSAttachmentId, and TCEndpointId. These parameters are implicit (not depicted) in the service definition documents. The model here assumes that they are assigned by the responsible provider. For example, TCEndpointId appears in T.Connect.indication and T.Connect.confirm. To enable the user of a service to relate a confirm to a preceding request, the user may specify an arbitrary label to be echoed back. ~~This facility is omitted only from MCS.Attach.User, where the domain selector is enough of a key.~~

Flow control is modeled here by a window-of-one mechanism. A ready signal must be sent in one direction before a data transfer signal may be sent in the opposite direction. Before the next transfer, another ready must be received. In a practical implementation, ready might take the form of moving a pointer or incrementing a counter. Casting ready as a signal does not imply it must be as ponderous as data transfer. There are three ready signals: T.ready, PDU.ready, and MCS.ready. These allow the transfer of, respectively, one TSDU, one domain MCSPDU, and one MCS service data unit. Flow control applies independently to the two directions of a signal route.

Transport quality of service on a TC is modeled here with parameters for throughput, transit delay, and data priority. This is an amalgam of standards are not consistent among themselves. In practice, the details will surely differ. The specification of QOS is a burden that controller applications must bear.

VII.3 Portals onto a domain

The Domain process is central to this decomposition of an MCS provider. Different instances represent different domains hosted by the same provider. A Domain process is created with a set of parameters that remain frozen for its lifetime. These domain parameters are provided by the Control process. The Control process decides which domain selectors are valid and how the corresponding Domain processes shall be configured. If there is room for parameter negotiation during MCS.Connect.Provider, the Control process supervises the interaction.

To keep the Domain process as simple as possible, the MCS attachments and MCS connections it interfaces to are abstracted to appear as portals with a number of common features. A portal is either a single Attachment process or a set of Endpoint processes, one for each constituent TC of an MCS connection. The Control process assigns portal ids and opens and shuts their associations to a Domain process. It hides details of building up and tearing down processes and hides the exchange of connect MCSPDUs.

To the Domain process, a portal looks like a set of signal routes, one for each data priority implemented. Each conveys MCSPDUs subject to PDU.ready flow control. Where distinctions in processing cannot be avoided, they are invoked by classifying a portal as one of three kinds: Attached, Up<u>link</u>~~ward~~, or Down<u>link</u>~~ward~~.

Figures VII-1 through VII-8 illustrate signal flows for typical operations. They assume a single user attached locally to a domain with one MCS connection to another provider. Endpoint1 and Endpoint2 arise from an assumption that two data priorities are implemented in the domain.

The signaling to shut a portal is rather lengthy. It defends against the possibility that one of the processes involved might send a signal to another that has already stopped, which would be a run-time error in SDL. Attachment and Endpoint processes make Quit the last signal they send, and they stop on receipt of Exit. Domain sends no further signals to a portal after replying Shut.portal to Control, and it stops after indicating thus that the last portal is shut.

Of remaining miscellaneous signals, Drop.portal is bidirectional and corresponds to **DPum**. Report.portal alerts Control that a diagnostic was issued ~~in~~ towards another MCS provider. An **RJum** that originates in Endpoint is sent through Domain to get onto the initial TC. An **RJum** received externally is dispatched directly from Endpoint to Control.

Note that MCS.Attach.User.request and T.Connect.indication are directed to Control, since they are stimuli to create corresponding portals. Control can decline with a negative MCS.Attach.User.confirm or T.Disconnect.request.

The present model does not limit the number of MCS attachments or MCS connections per domain, except through a local limit (maxPortalIds) on the number of portals across all domains hosted by the provider. Many aspects of domain configuration must be managed locally until further standardization is undertaken.

VII.4 Alignment of MCSPDUs

Clarity requires that the MCSPDUs defined in clause 7 be represented individually by SDL signals. But simplicity of the Domain process depends on a more unified treatment. The resolution here is to define a PDUStruct data type, with components that can be selected to match any one of the domain MCSPDUs. Connect MCSPDUs, which are handled outside the Domain process, continue each to have its own structure.

A key component of PDUStruct is kind, which identifies the intended domain MCSPDU. Based on kind, a subset of other fields then become relevant: those listed in ASN.1 as components of the MCSPDU. The SDL-defined data types match as closely as possible the ASN.1-defined data types of clause 7. The intention should be clear that a PDUStruct is the internal, decoded representation of a domain MCSPDU.

Three~~wo~~ MCSPDUs have optional components, **AUcf**, **CJcf**, and **CCcf**. In SDL their absence is indicated by coding zero for the corresponding field. This is an illegal value for a static or dynamic channel id.

The signals sent between Attachment, Domain, and Endpoint processes, with the exception of PDU.ready, take a PDUStruct as their single parameter. In a practical implementation, this kind of communication might be as simple as moving a buffer pointer.

VII.5 Method of using SDL

The reference implementation uses the textual representation of SDL as a programming language whose power is enhanced by the ability to define abstract data types. In Control and Domain, the objects managed are too complex and numerous to benefit from the kind of finite state machine that SDL supports. These processes remain in a single state processing input signals. Attachment and Endpoint, with smaller scope, can use a handful of states to better advantage.

Appendix II contains a definition of the SetOf generator, whose scope is the entire provider. This takes any other defined type, such as channel ids, and formalizes the concept of subsets of values of that type. SetOf extends the built-in generator Powerset by adding a new operator that chooses an arbitrary element from a non-empty subset. The meaning of this Pick operator is defined through simple axioms.

Sets are used heavily throughout the implementation. Part of the information base, for example, is to record which subset of channel ids are in use and which subset of portal ids have joined a given channel.

In SDL, arrays are deployed over the full range of the indexing data type. Channel structures, for example, exist for each channel id from 0 to 65535. A practical implementation must deal in much sparser arrays. To this end, separate id sets are kept to record which channels and other resources are in use at a given time. A principle of the design is that array values need not be stored in the information base for ids that are not explicitly marked as in use.

Many iterations follow a familiar pattern: create a candidate set of interest, then Pick and delete one member of it at a time, until no more remain. Such iterations are built out of decisions and joins. Comments show the higher-level control structure intended. Labels are numbered sequentially within a procedure, with one letter appended to indicate whether branches to the label are backward or forward.

A table of contents, which shows calling sequences, occurs before the first procedure definition in each process. SDL procedures may return results through formal parameters that are declared to be in/out.

A few constants appear dispersed throughout the code, like the maximum number of data priorities (4) and the dividing line between static and dynamic channel ids (1001). These are not parameters whose value can be redefined.

It is an important idea in MCS that user ids are a subset of channel ids. But the contexts in which each is used differ, so two separate types are appropriate. Appendix II formalizes a pair of operators for casting between them.

The processes of Appendices III through VI cooperate in the framework specified by Appendix II. Signals are constrained to the specified routes, and misbehavior across internal interfaces is not allowed. Defensive coding focuses its attention on controller and user applications residing in the environment outside the provider and on MCSPDUs received~~communicated by~~ externally from peer providers.

The reference implementation allows input signals to be scheduled in arbitrary sequence if more than one are pending. It makes no assumption about the relative priority of processes. But it does rely on individual input transitions executing to completion without preemption.

VII.6 Remarks on the Domain process

Significant use is made of the Queue generator in addition to SetOf. Buffers are queued in user-transmitted sequence to an output portal, and portal ids are queued in the order of unanswered **`MCrq`**, **`MTrq`**, and **`AUrq`** MCSPDUs.

To avoid copying user data, MCSPDUs are manipulated in buffers, which may be output to multiple ports. The number of buffers available to a Domain process is modeled as a fixed external parameter. This could easily be changed to vary by configuration from domain to domain. The implementation degrades gracefully, through global flow control, if the number of buffers provided is low. It applies buffers, as they become free, to the highest priority data flow.

When an MCSPDU arrives as input, the Domain process needs to know the portal of origin and data priority. The procedure Identify_sender does an exhaustive search of opened portals based on the process id reported by SDL. The needed information could be defined to be part of the signal parameter set, except that the intuitive meaning of signals would then be obscured. In a practical implementation this modeling is not a significant issue.

The heart of the Domain process is the procedure Process_PDU, which calls in sequence Validate_input, Top_provider, and Apply_PDU. Each of these is a large case selection based on the kind of PDU at hand. The details for any given kind are mostly straightforward. A good way to approach these procedures may be to cut across them horizontally, following the course of especially important MCSPDUs, like **`SDrq`**, one by one.

The encoding of MCSPDUs, using either BER or PER depending on the protocol version, is a detail left mostly to the Endpoint process. All Domain needs to know is how large it can make MCSPDUs with variable content, like **`DUrq`** with its multiple user ids. Such considerations are expressed through decision statements with informal text, using worst-case constants for the BER encoding. In practice, some limits on variable content may be precalculated in terms that Domain understands, like maximum numbers of ids. Others may require more direct knowledge of the encoding.

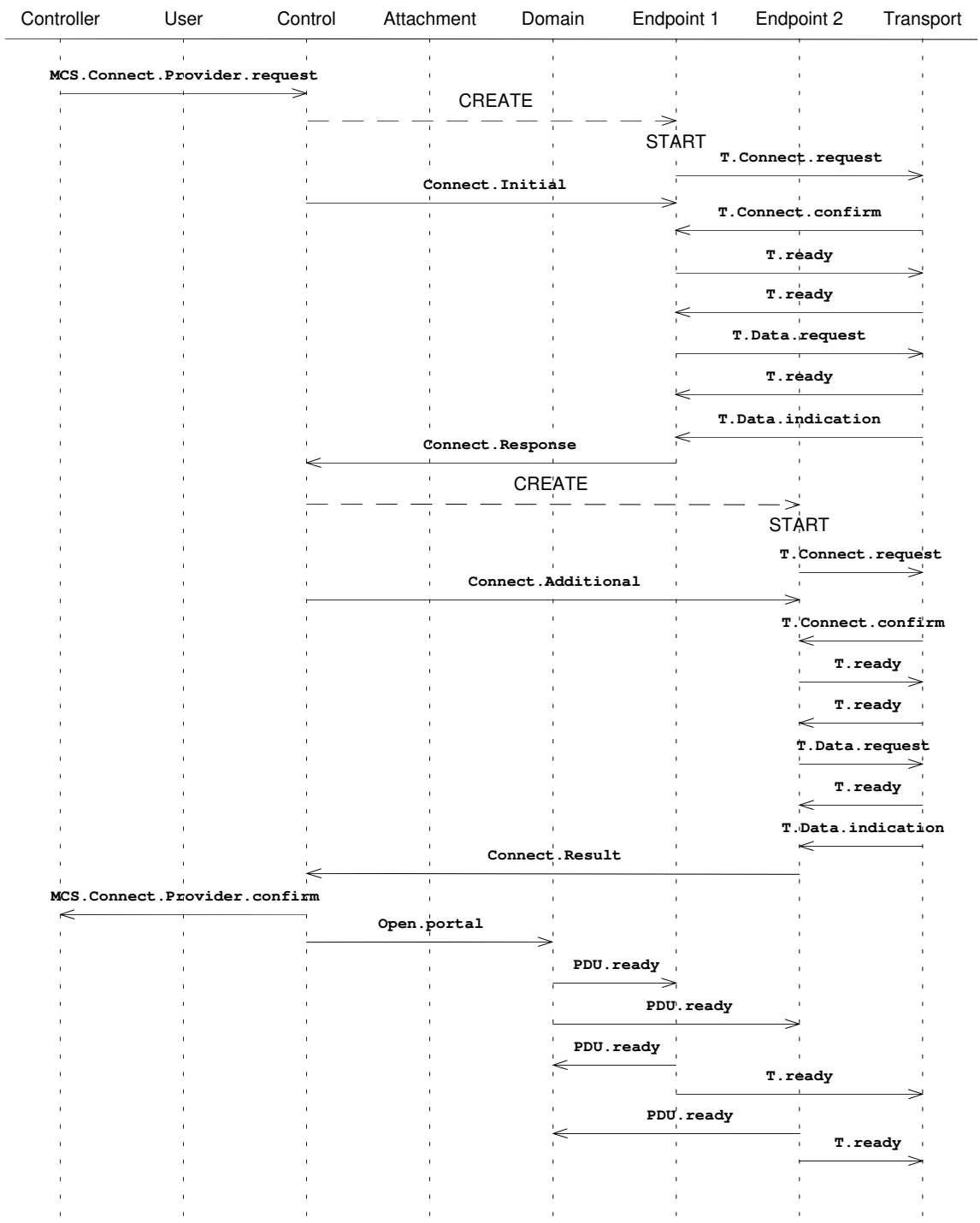


Figure VII-1 – Calling MCS provider

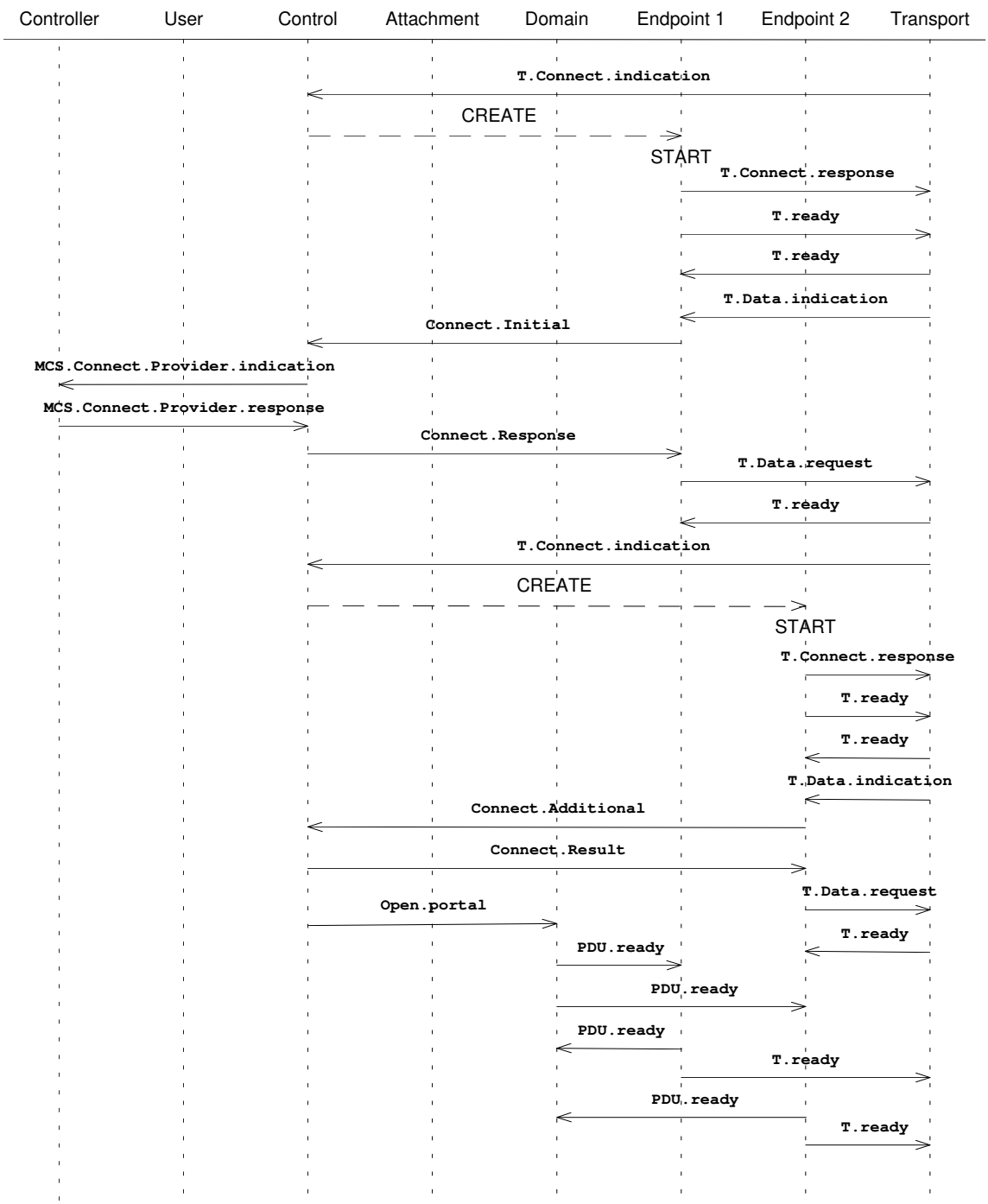


Figure VII-2 – Called MCS provider

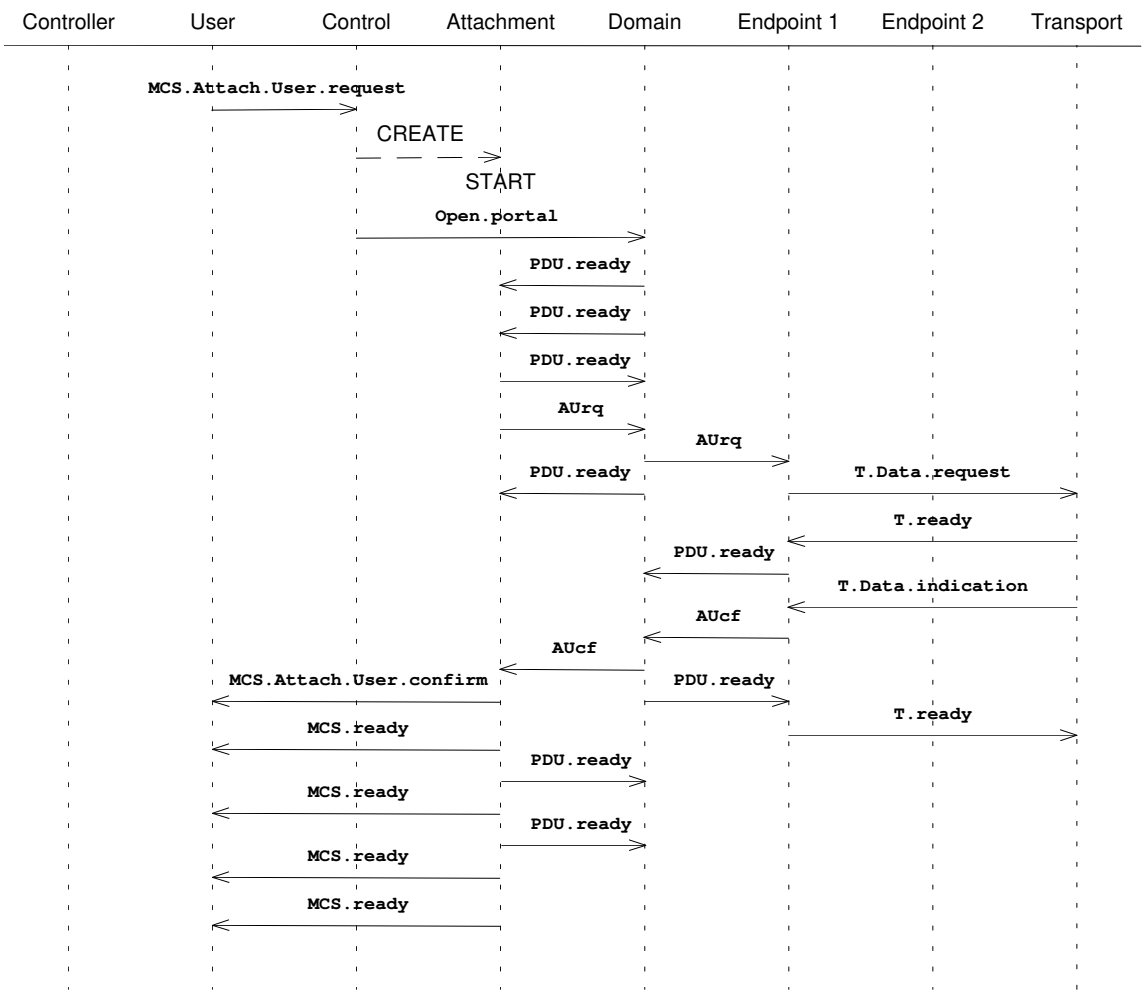


Figure VII-3 – User attach

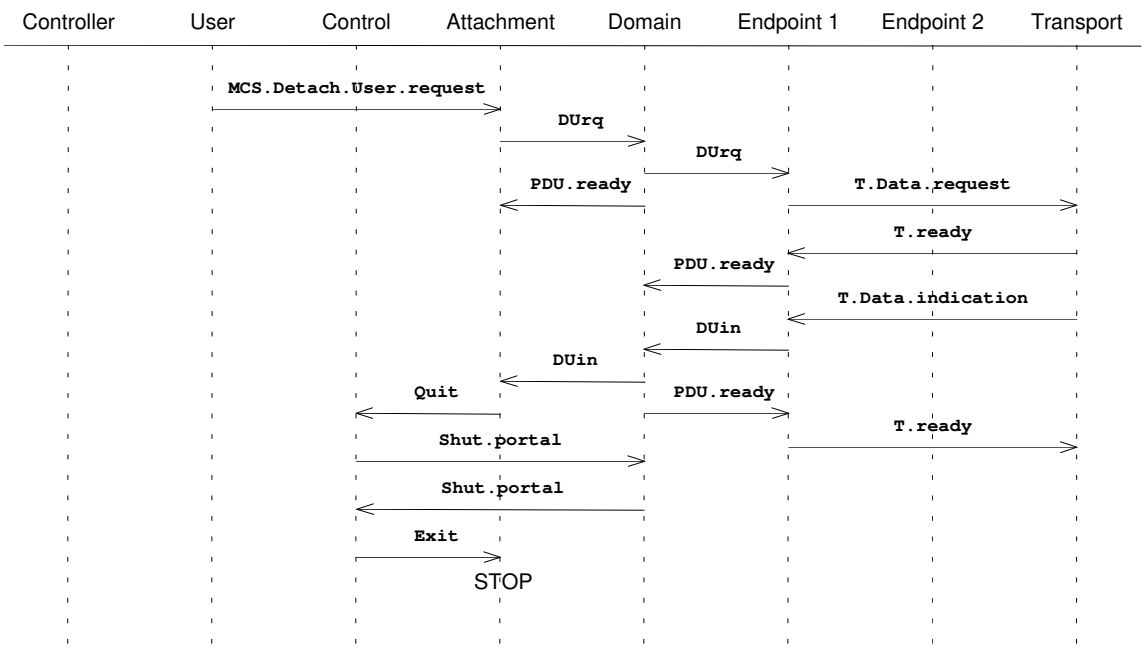


Figure VII-4 – User-requested detach

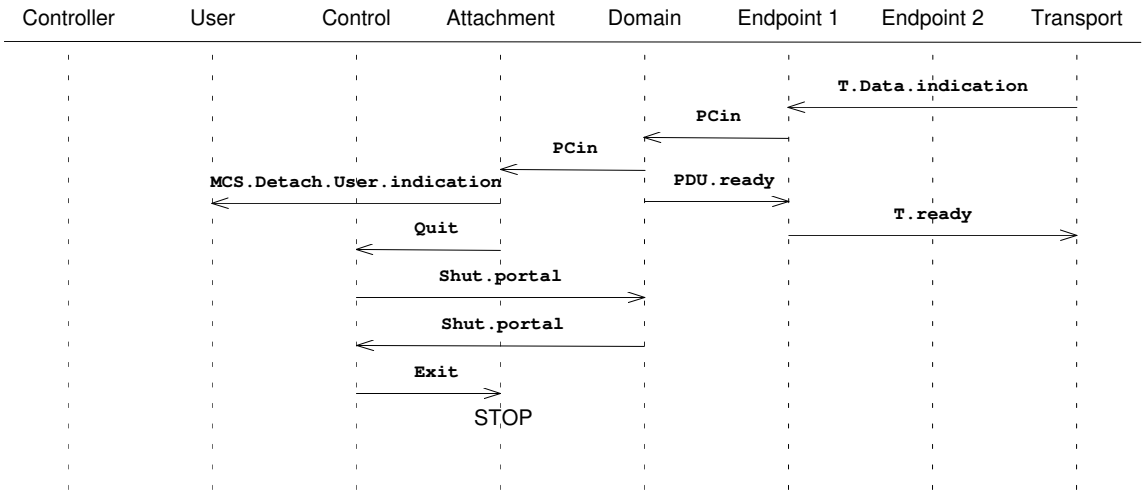


Figure VII-5 – Provider-initiated detach

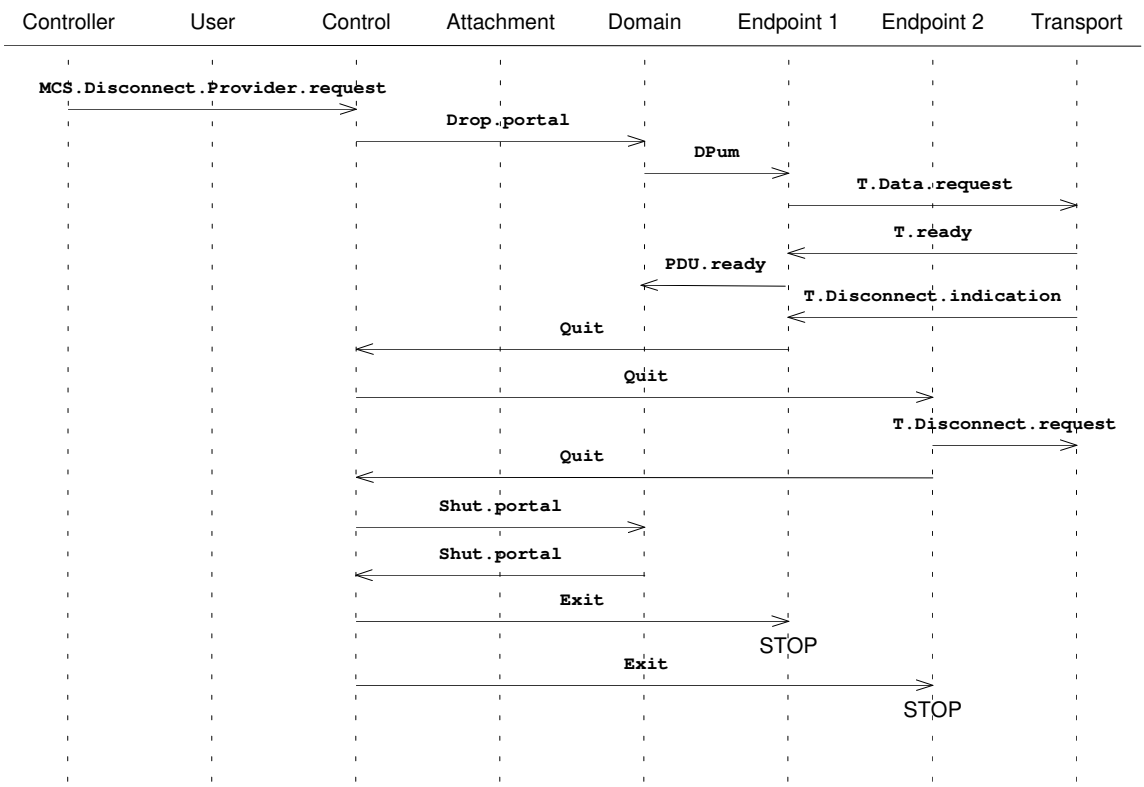
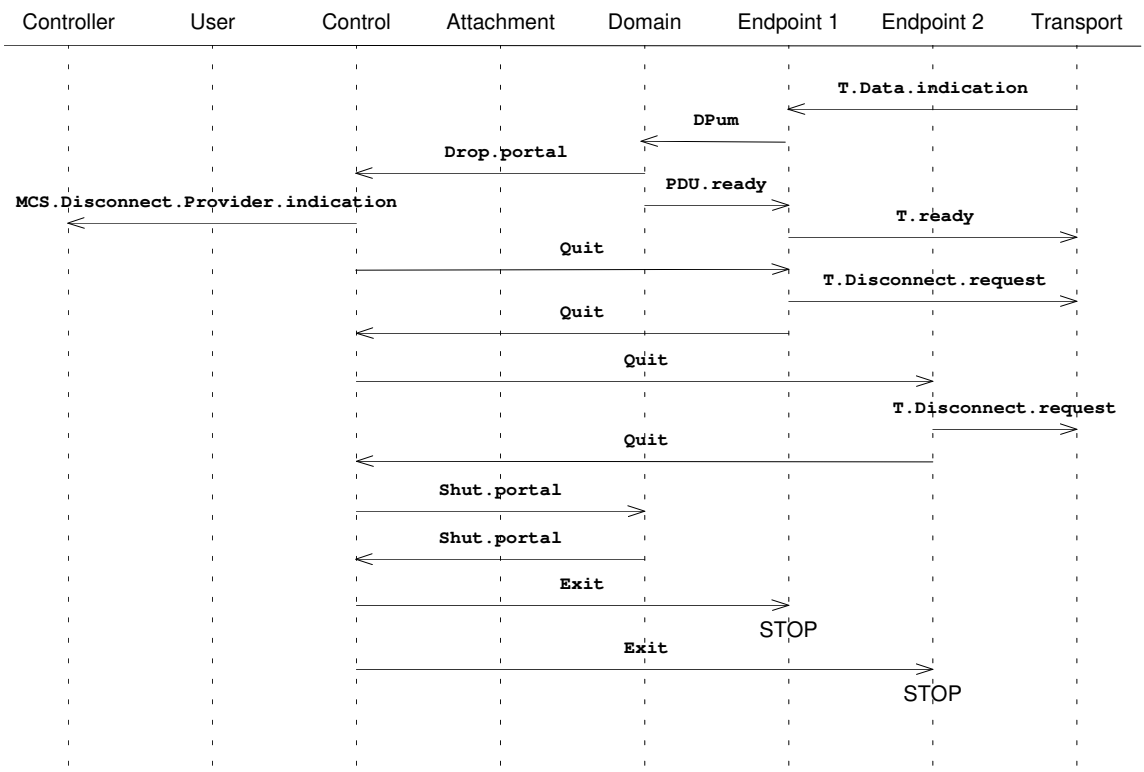


Figure VII-6 – Local controller-requested disconnect



FFigure VII-7 – Remote controller-requested disconnect

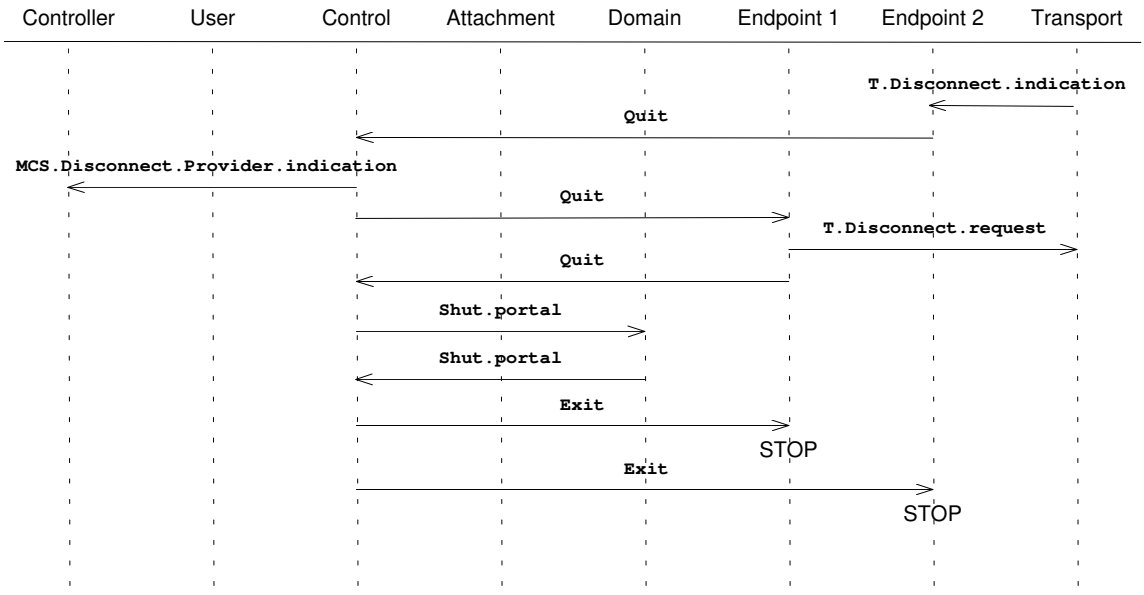


Figure VII-8 – Provider-initiated disconnect

Summary ... 1

1 Scope ... 1

2 Normative references ... 1

3 Definitions ... 2

4 Abbreviations ... 3

5 Overview of the MCS protocol ... 3

5.1 Model of the MCS layer ... 3
5.2 Services provided by the MCS layer ... 4
5.3 Services assumed from the transport layer ... 6
5.4 Functions of the MCS layer ... 7
5.4.1 Domain management ... 7
5.4.2 Channel management ... 7
5.4.3 Data transfer ... 8
5.4.4 Token management ... 8
5.5 Hierarchical processing ... 8
5.6 Domain parameters ... 9

6 Use of the transport service ... 10

6.1 Model of the transport service ... 10
6.2 Use of multiple connections ... 10
6.3 Transport connection release ... 11

7 Structure of MCSPDUs ... 11

8 Encoding of MCSPDUs ... 20

9 Routing of MCSPDUs ... 21

9.1 Connect MCSPDUs ... 21
9.2 Domain MCSPDUs ... 22

10 Meaning of MCSPDUs ... 24

10.1 Connect-Initial ... 24
10.2 Connect-Response ... 25
10.3 Connect-Additional ... 26
10.4 Connect-Result ... 26
10.5 PDin ... 27
10.6 EDrq ... 27
10.7 MCrq ... 28
10.8 MCcf ... 28
10.9 PCin ... 29
10.10 MTrq ... 29
10.11 MTcf ... 30
10.12 PTin ... 31
10.13 DPum ... 31
10.14 RJum ... 31
10.15 AUrq ... 32
10.16 AUcf ... 32
10.17 DUrq ... 32
10.18 DUin ... 33
10.19 CJrq ... 33
10.20 CJcf ... 34
10.21 CLrq ... 35
10.22 CCrq ... 35

10.23 CCcf ... 35
10.24 CDrq ... 36
10.25 CDin ... 36
10.26 CArq ... 37
10.27 CAin ... 37
10.28 CErq ... 37
10.29 CEin ... 38
10.30 SDrq ... 38
10.31 SDin ... 39
10.32 USrq ... 40
10.33 USin ... 40
10.34 TGrq ... 41
10.35 TGcf ... 41
10.36 TIrq ... 42
10.37 TIcf ... 42
10.38 TVrq ... 43
10.39 TVin ... 43
10.40 TVrs ... 44
10.41 TVcf ... 44
10.42 TPrq ... 45
10.43 TPin ... 45
10.44 TRrq ... 46
10.45 TRcf ... 46
10.46 TTrq ... 46
10.47 TTcf ... 47

11 MCS provider information base ... 47

11.1 Hierarchical replication ... 47
11.2 Channel information ... 48
11.3 Token information ... 49

12 Elements of procedure ... 50

12.1 MCSPDU sequencing ... 50
12.2 Input flow control ... 51
12.3 Throughput enforcement ... 51
12.4 Domain configuration ... 52
12.5 Domain merger ... 52
12.6 Domain disconnection ... 55
12.7 Channel id allocation ... 55
12.8 Token status ... 56

13 Reference implementation ... 56

I Alternative encodings of an MCSPDU ... 57

II SDL decomposition of an MCS provider ... 59

III SDL specification of the Control process ... 73

IV SDL specification of the Domain process ... 87

V SDL specification of the Endpoint process ... 123

VI SDL specification of the Attachment process ... 127

VII Characteristics of the reference implementation ... 136