INTERNATIONAL TELECOMMUNICATION UNION

ITU-T T.125

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

(04/94)

TELEMATIC SERVICES

TERMINAL EQUIPMENTS AND PROTOCOLS FOR TELEMATIC SERVICES

MULTIPOINT COMMUNICATION SERVICE PROTOCOL SPECIFICATION

ITU-T Recommendation T.125

(Previously “CCITT Recommendation”)

FOREWORD

The ITU-T (Telecommunication Standardization Sector) is a permanent organ of the International Telecommunication Union (ITU). The ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis..

The World Telecommunication Standardization Conference (WTSC), which meets every four years, etablishes the topics for study by the ITU-T Study Groups which, in their turn, produce Recommendations on these topics.

The approval of Recommendations by the Members of the ITU-T is covered by the procedure laid down in WTSC Resolution No. 1 (Helsinki, March 1-12, 1993).

ITU-T Recommendation T.125 was prepared by ITU-T Study Group 8 (1993-1996) and was approved under the WTSC Resolution No. 1 procedure on the 7th of April 1994.

NOTE

In this Recommendation, the expression “Administration” is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

© ITU 1994

All rights reserved. No part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the ITU.

CONTENTS

			Page
1	Scope		1
2	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	4
	5.4	Functions of the MCS layer	4
	5.5	Hierarchical processing	7
	5.6	Domain parameters	9
6	Use of the transport service		9
	6.1	Model of the transport service	9
	6.2	Use of multiple connections	10
	6.3	Transport connection release	11
7	Structure of MCSPDUs		11
8	Encoding of MCSPDUs		21
9	Routing of MCSPDUs		21
	9.1	Connect MCSPDUs	21
	9.2	Domain MCSPDUs	22
10	Meaning of MCSPDUs		25
	10.1	Connect-Initial	25
	10.2	Connect-Response	26
	10.3	Connect-Additional	27
	10.4	Connect-Result	27
	10.5	PDin	28
	10.6	EDrq	28
	10.7	MCrq	29
	10.8	MCcf	30
	10.9	PCin	30
	10.10	MTrq	31
	10.11	MTcf	32
	10.12	PTin	33
	10.13	DPum	33
	10.14	RJum	33
	10.15	AUrq	34
	10.16	AUcf	34
	10.17	DUrq	35
	10.18	DUin	35
	10.19	CJrq	36
	10.20	CJcf	37
	10.21	CLrq	37
	10.22	CCrq	38
	10.23	CCcf	38
	10.24	CDrq	39
	10.25	CDin	39
	10.26	CArq	39
	10.27	CAin	40

Page

10.28 CErq 40
10.29 CEin 41
10.30 SDrq 41
10.31 SDin 42
10.32 USrq 43
10.33 USin 43
10.34 TGrq 44
10.35 TGcf 44
10.36 TIrq 45
10.37 TIcf 45
10.38 TVrq 46
10.39 TVin 46
10.40 TVrs 47
10.41 TVcf 48
10.42 TPrq 48
10.43 TPin 48
10.44 TRrq 49
10.45 TRcf 49
10.46 TTrq 50
10.47 TTcf 50

11 MCS provider information base 50
11.1 Hierarchical replication 50
11.2 Channel information 52
11.3 Token information 52

12 Elements of procedure 54
12.1 MCSPDU sequencing 54
12.2 Input flow control 55
12.3 Throughput enforcement 56
12.4 Domain configuration 57
12.5 Domain merger 57
12.6 Domain disconnection 59
12.7 Channel id allocation 60
12.8 Token status 61

13 Reference implementation 61

Appendix I – Alternative encodings of an MCSPDU 62
I.1 Send Data Request 62
I.2 Basic Encoding Rules (BER) 62
I.3 Packed Encoding Rules (PER) 63

Appendix II – SDL decomposition of an MCS provider 64

Appendix III – SDL specification of the Control process 80

Appendix IV – SDL specification of the Domain process 97

Appendix V – SDL specification of the Endpoint process 142

Appendix VI – SDL specification of the Attachment process 146

Appendix VII – Characteristics of the reference implementation 156
VII.1 SDL decomposition 156
VII.2 Service definitions 157
VII.3 Portals onto a domain 157
VII.4 Alignment of MCSPDUs 158
VII.5 Method of using SDL 158
VII.6 Remarks on the Domain process 159

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 Recommendation T.122.

MULTIPOINT COMMUNICATION SERVICE PROTOCOL SPECIFICATION

(Geneva, 1994)

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 References

The following Recommendations and other references 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 other references are subject to revision: all users of this Recommendation are therefore encouraged to investigate the possibility of applying the most recent edition of the Recommendations and other references listed below. A list of the currently valid ITU-T Recommendations is regularly published.

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

– Draft ITU-T Recommendation X.691 | ISO/IEC DIS 8825-2, *Information Technology – Open Systems Interconnection – ASN.1 encoding rules – Specification of Packed Encoding Rules (PER)*.

3 Definitions

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

This Recommendation is based on the concepts developed in CCITT Recommendation 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 Recommendation is also based on the concepts developed in ITU-T Recommendation 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 Recommendation, 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 Recommendation.

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 Recommendation.

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

For the purposes of this Recommendation, the following abbreviations apply.

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 Recommendation 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.

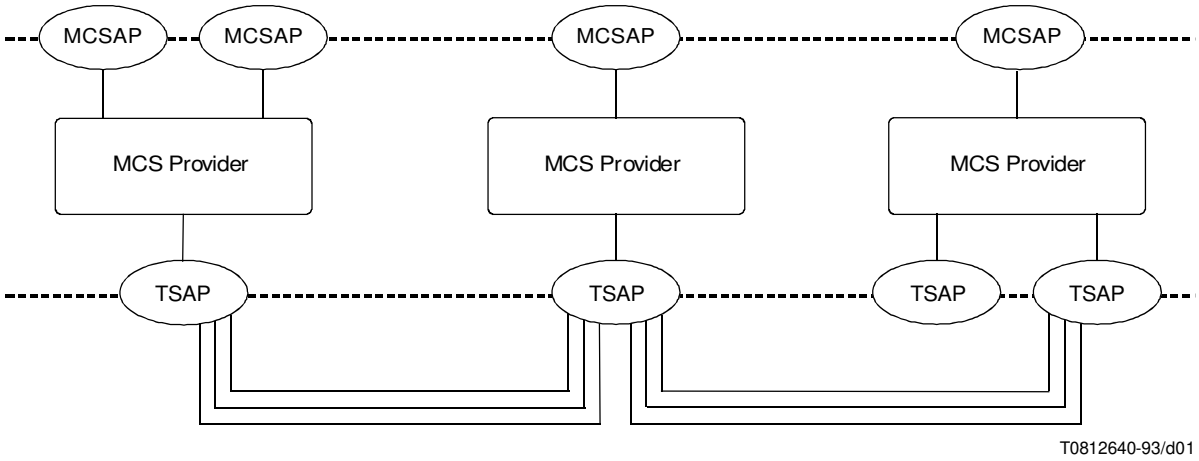


FIGURE 5-1/T.125

Model of the MCS layer

5.2 Services provided by the MCS layer

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

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 Recommendation X.214. Information is transferred to and from a TS provider by the primitives listed in Table 5-2.

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.

TABLE 5-1/T.125

MCS primitives

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

TABLE 5-1/T.125 *(end)*

MCS primitives

Functional Unit	Primitives	Associated MCSPDUs
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**

5.4.2 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 remain 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.

TABLE 5-2/T.125

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.			

5.4.4 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.

The nodes in the figure represent MCS providers, and the labelled 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.

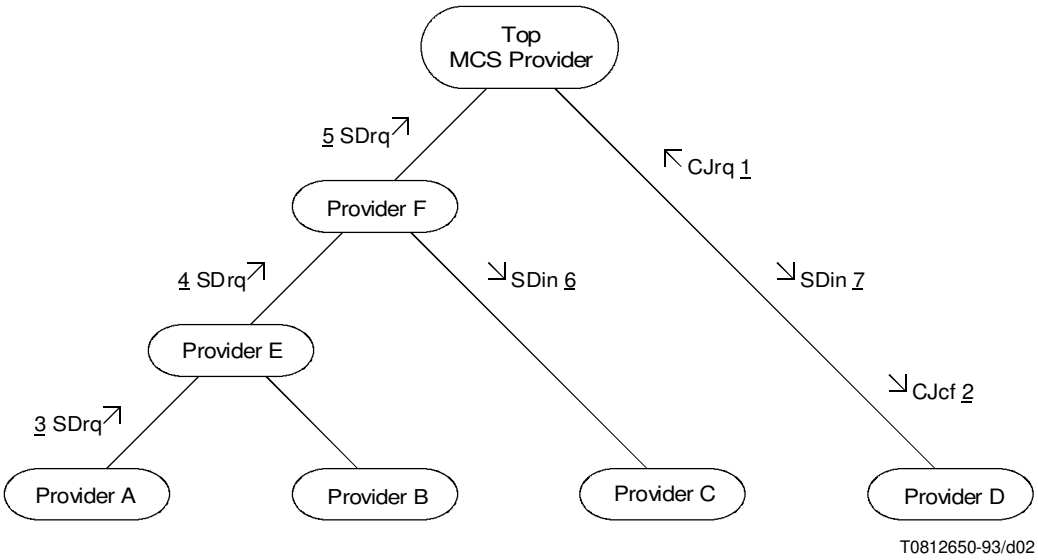


FIGURE 5-2/T.125

Hierarchical processing in 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 Recommendation 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 modelled 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 modelled 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 behaviour 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 favour low transit delay for real-time response but may not require high throughput. Low priority data, on the other hand, may favour 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 Recommendation 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
        {
            joined                        BOOLEAN,
            userId                        UserId
                                          -- TRUE if user is joined to its user id
        },
        private                     [2]   IMPLICIT SEQUENCE
        {
            joined                        BOOLEAN,
                                          -- TRUE if channel id is joined below
            channelId                     PrivateChannelId,
            manager                       UserId,
            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
{
        detachUserIds               SET OF UserId,
                                    -- purge user id channels
        purgeChannelIds             SET OF ChannelId
                                    -- purge other 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
}

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 Recommendation 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 an ITU-T Recommendation or ISO/IEC International Standard.

NOTES

1 The Packed Encoding Rules yield more compact MCSPDU headers.

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

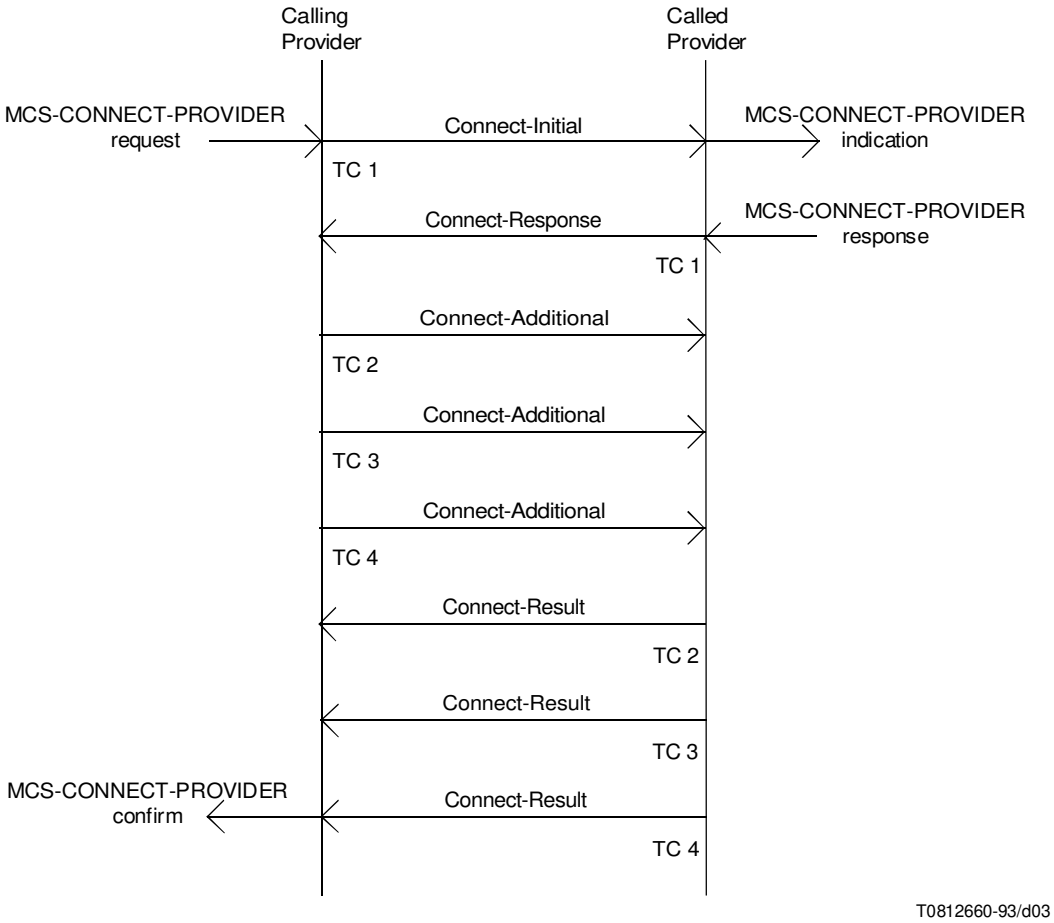
9.1 Connect MCSPDUs

Figure 9-1 specifies the exchange of connect MCSPDUs.

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.



NOTE – The number and relative order of additional TCs may vary

FIGURE 9-1/T.125

Message flow of connect MCSPDUs

9.2 Domain MCSPDUs

Table 9-1 specifies the routing of domain MCSPDUs.

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.

TABLE 9-1/T.125

Routing of domain MCSPDUs

Category	MCSPDUs			TC	Direction
Request	**EDrq**	**MCrq**	**MTrq**	I	Up
	AUrq	**DUrq**			
	CJrq	**CLrq**	**CCrq**		
	CDrq	**CArq**	**CErq**		
	TGrq	**TIrq**	**TVrq**		
	TPrq	**TRrq**	**TTrq**		
	SDrq	**USrq**		A	
Indication	**PDin**	**PCin**	**PTin**	I	Down
		DUin			
	CDin	**CAin**	**CEin**		
			TVin		
	TPin				
	SDin	**USin**		A	
Response			**TVrs**	I	Up
Confirm		**MCcf**	**MTcf**	I	Down
	AUcf				
	CJcf		**CCcf**		
	TGcf	**TIcf**	**TVcf**		
		TRcf	**TTCF**		
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 *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 set of user ids forwarded is unchanged, so that all detached users will be announced to those that remain. The set of other channel ids forwarded may be restricted to those in use in a subtree. At a former top provider that is still merging into an upper domain, both user ids and other channel ids are restricted to those whose acceptance into the upper domain has been confirmed.

c) **PTin** is sent on all MCS connections downward. The set of token ids forwarded may be restricted to those in use in a subtree. At a former top provider that is still merging into an upper domain, token ids are restricted to those 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, so that all detached users will be announced to those that remain. At a former top provider that is still merging into an upper domain, user ids are restricted to those whose acceptance into the upper domain has been confirmed.

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

Tables 10-1 through 10-47 reiterate the contents of individual MCSPDUs as defined in clause 8.

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/T.125

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 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 two connections to higher providers. A provider without a 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 honour 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/T.125

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.

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

TABLE 10-3/T.125

Connect-Additional MCSPDU

Contents	Source	Sink
Called Connect Id	Calling provider	Called provider
Data Priority	Calling provider	Called provider

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/T.125

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/T.125

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.

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.

TABLE 10-6/T.125

EDrq MCSPDU

Contents	Source	Sink
Height in Domain	Subordinate	Higher provider
Throughput Enforcement Interval	Subordinate	Higher provider

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/T.125

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.

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.

TABLE 10-8/T.125

MCcf MCSPDU

Contents	Source	Sink
Merge Channels	Top provider	Intermediates
Purge Channel Ids	Top provider	Former top

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**.

Intermediate MCS providers shall confirm that user ids proclaimed in private channel attributes are assigned to the subtree to which **MCcf** is routed. If a private channel manager has been detached and reassigned elsewhere in the time since the antecedent **MCrq** was validated, an intermediate provider shall generate a **CDrq** making the private channel a casualty of domain merger and shall move the channel id into the purged set. If any admitted users have been reassigned elsewhere, it shall exclude them from the channel.

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/T.125

PCin MCSPDU

Contents	Source	Sink
Detach User Ids	Former top	Subordinates
Purge 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 if a user id channel; MCS-CHANNEL-LEAVE indication to the joined users if a static or assigned channel id; MCS-CHANNEL-DISBAND indication to the manager and MCS-CHANNEL-EXPEL indication to the admitted users 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, **PCin** divides the channel ids to be purged into these two categories.

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/T.125

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. 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/T.125

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**.

Intermediate MCS providers shall confirm that the user ids proclaimed in token attributes are assigned to the subtree to which **MTcf** is routed. If any user ids have been detached and reassigned elsewhere in the time since the antecedent **MTrq** was validated, an intermediate provider shall generate for them a **DUrq** with reason code *channel purged*, making them casualties of domain merger. If a non-inhibited token id becomes released through this deletion of user ids, it shall be moved into the purged set.

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/T.125

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 Recommendation 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.

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.*

TABLE 10-13/T.125

DPum MCSPDU

Contents	Source	Sink
Reason	Requesting provider	Indication

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.

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.

TABLE 10-14/T.125

RJum MCSPDU

Contents	Source	Sink
Diagnostic	Rejecting provider	Rejected provider
Initial Octets	Rejecting provider	Rejected provider

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/T.125

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/T.125

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/T.125

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.

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**.

TABLE 10-18/T.125

DUin MCSPDU

Contents	Source	Sink
Reason	Top provider	Indication
User Ids	Top provider	Indication

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/T.125

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/T.125

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 requested is the same as in **CJrq**. This helps the initiating attachment relate MCS-CHANNEL-JOIN confirm to an antecedent request. 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.

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**.

TABLE 10-21/T.125

CLrq MCSPDU

Contents	Source	Sink
Channel Ids	Requesting provider	Higher provider

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.

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.

TABLE 10-22/T.125

CCrq MCSPDU

Contents	Source	Sink
Initiator	Requesting provider	Higher provider

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/T.125

CCcf MCSPDU

Contents	Source	Sink
Result	Top provider	Confirm
Initiator	Top provider	MSCPDU 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/T.125

CDrq MCSPDU

Contents	Source	Sink
Initiator	Requesting provider	Top provider
Channel Id	Request	Top provider

CDrq may also be generated by an MCS provider on its own initiative to disband a channel.

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 to admitted users with reason *channel disbanded* and, if provider initiated, an MCS-CHANNEL-DISBAND indication to the manager.

TABLE 10-25/T.125

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.

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**.

TABLE 10-26/T.125

CArq MCSPDU

Contents	Source	Sink
Initiator	Requesting provider	Top provider
Channel Id	Request	Top provider
User Ids	Request	Top provider

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.

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.*

TABLE 10-27/T.125

CAin MCSPDU

Contents	Source	Sink
Initiator	Top provider	Indication
Channel Id	Top provider	Indication
User Ids	Top provider	MCSPDU routing

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.

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.

TABLE 10-28/T.125

CErq MCSPDU

Contents	Source	Sink
Initiator	Requesting provider	Top provider
Channel Id	Request	Top provider
User Ids	Request	Top 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-requested*.

TABLE 10-29/T.125

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.

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.

TABLE 10-30/T.125

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

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/T.125

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 a **USin** with identical contents and multicasts this downward.

TABLE 10-32/T.125

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/T.125

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/T.125

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/T.125

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/T.125

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.

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*.

TABLE 10-37/T.125

TIcf MCSPDU

Contents	Source	Sink
Result	Top provider	Confirm
Initiator	Top provider	MCSPDU routing
Token Id	Top provider	Confirm
Token Status	Top provider	Confirm

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/T.125

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.

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 the offered token by generating **TVrs** with reason *domain merging*.

TABLE 10-39/T.125

TVin MCSPDU

Contents	Source	Sink
Initiator	Top provider	Indication
Token Id	Top provider	Indication
Recipient	Top provider	MCSPDU routing

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/T.125

TVrs MCSPDU

Contents	Source	Sink
Result	Response	Top provider
Recipien	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/T.125

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.

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

TABLE 10-42/T.125

TPrq MCSPDU

Contents	Source	Sink
Initiator	Requesting provider	Top provider
Token Id	Request	Top provider

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.

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.

TABLE 10-43/T.125

TPin MCSPDU

Contents	Source	Sink
Initiator	Top provider	Indication
Token Id	Top provider	Indication

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/T.125

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/T.125

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.

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

TABLE 10-46/T.125

TTrq MCSPDU

Contents	Source	Sink
Initiator	Requesting provider	Top provider
Token Id	Request	Top provider

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.

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**.

TABLE 10-47/T.125

TTcf MCSPDU

Contents	Source	Sink
Initiator	Top provider	MCSPDU
Token Id	Top provider	Confirm
Token Status	Top provider	Confirm

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 subclauses.

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.

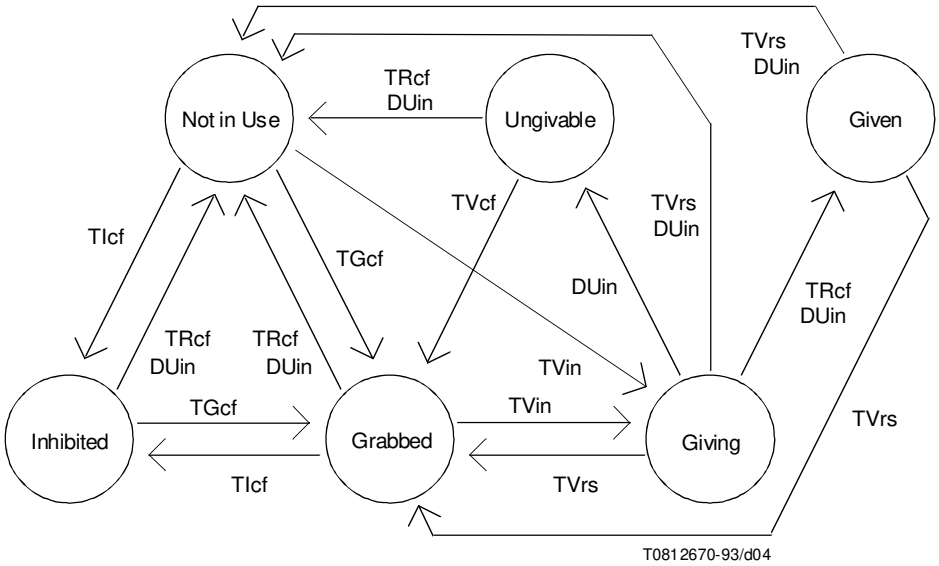


FIGURE 11-1/T.125
State transitions of a token id

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.

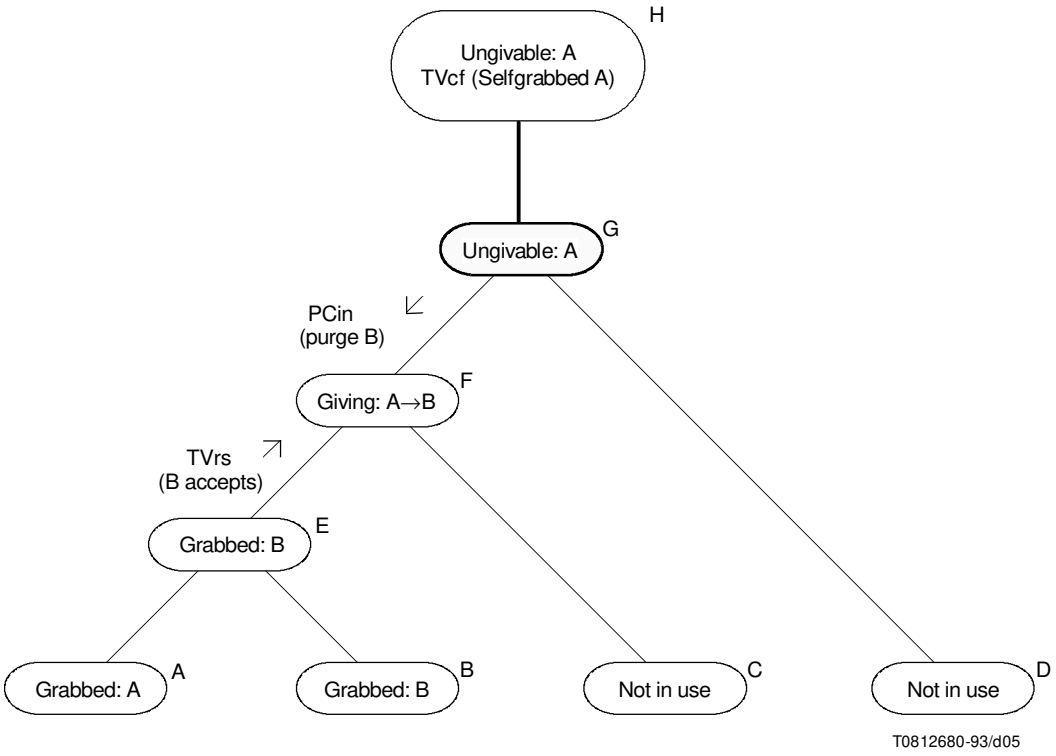


FIGURE 11-2/T.125

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 Recommendation 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 signalled 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. 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 behaviour. Still, enforcement remains a heuristic technique with room for invention.

Enforcing a minimum input rate at each receiver is an option available to controller 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 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 behaviour 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 Recommendation 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.

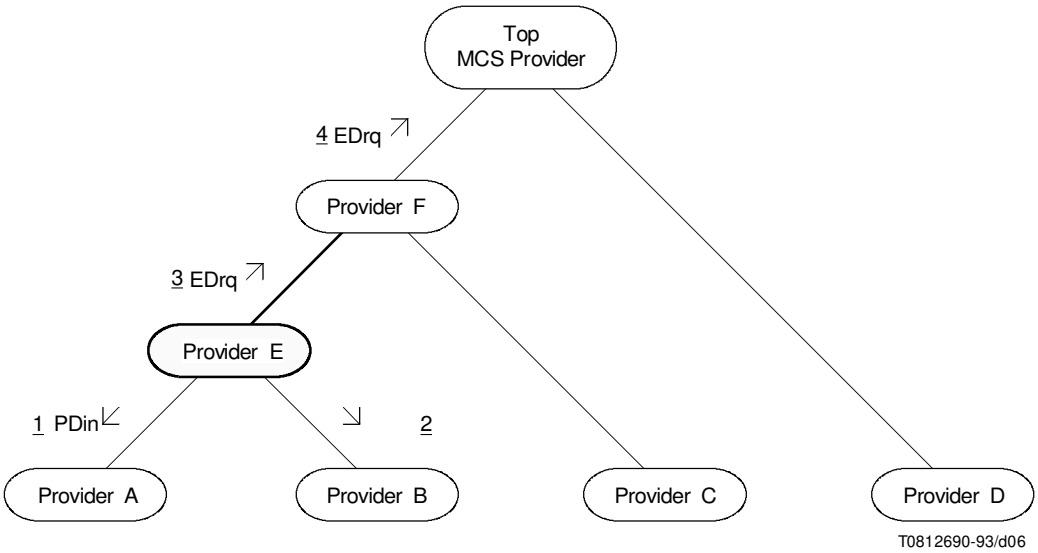


FIGURE 12-1/T.125

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.

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.

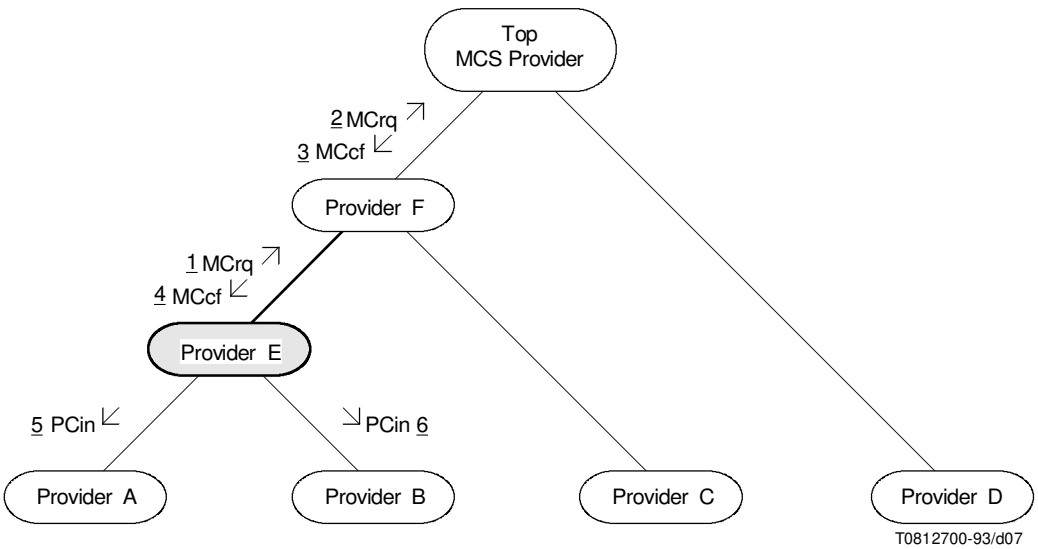


FIGURE 12-2/T.125

Domain merger step two – Merge user id chanels

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 refused 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.

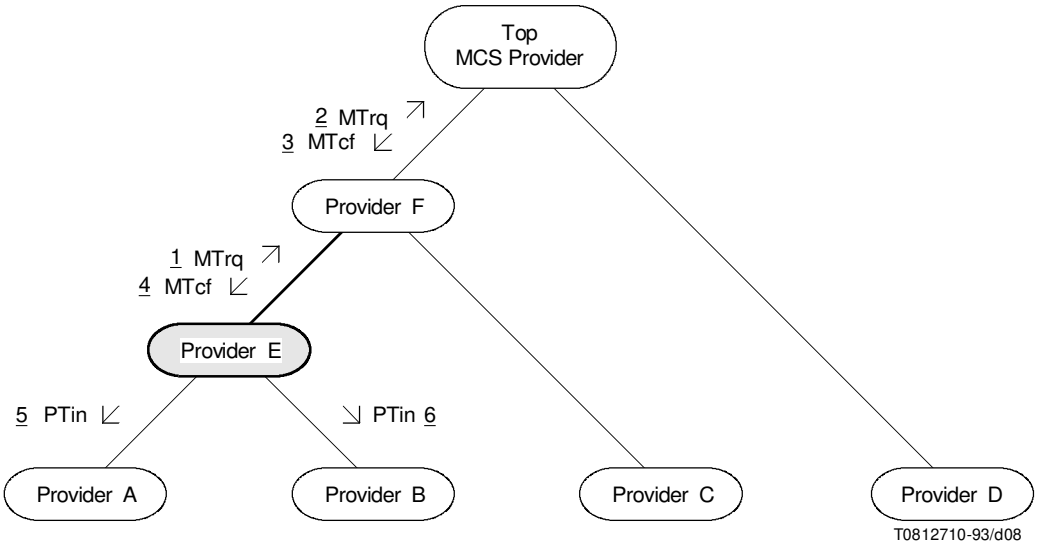


FIGURE 12-3/T.125

Domain merger step three – Merge token 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 refused 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*.

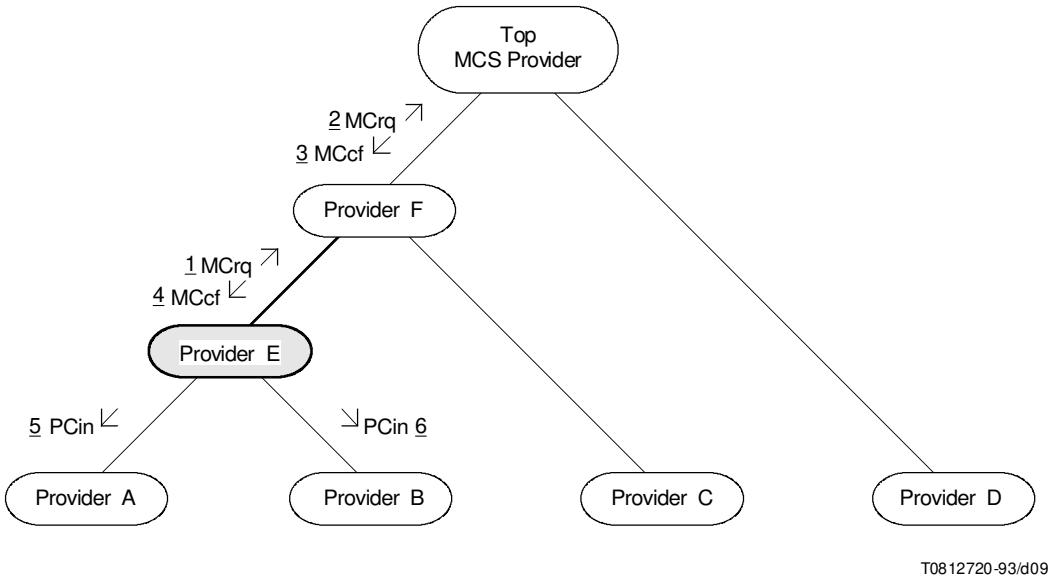


FIGURE 12-4/T.125

Domain merger step four – Merge remaining channel ids

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 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.

In situations where the seamless merging of active domains is required, conflicts may be avoided by selecting channel ids from a subrange unique to each provider. Subranges may be created by dividing the dynamically allocated channel ids 1001..65535 into several bands. Furthermore, within a subrange, ids may be allocated sequentially in one direction, from top or bottom. Providers that employ subrange allocation shall still obey all aspects of this protocol, including the procedure for domain merger. They shall thereby accommodate peer providers that may lack unique subranges.

NOTE – It is a matter of local implementation how a provider chooses a subrange from which to allocate. Pre-arranged conferences may be banded by a network management system.

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	
CTET 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 16 384 in two octets.

Appendix II
SDL decomposition of an MCS provider

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

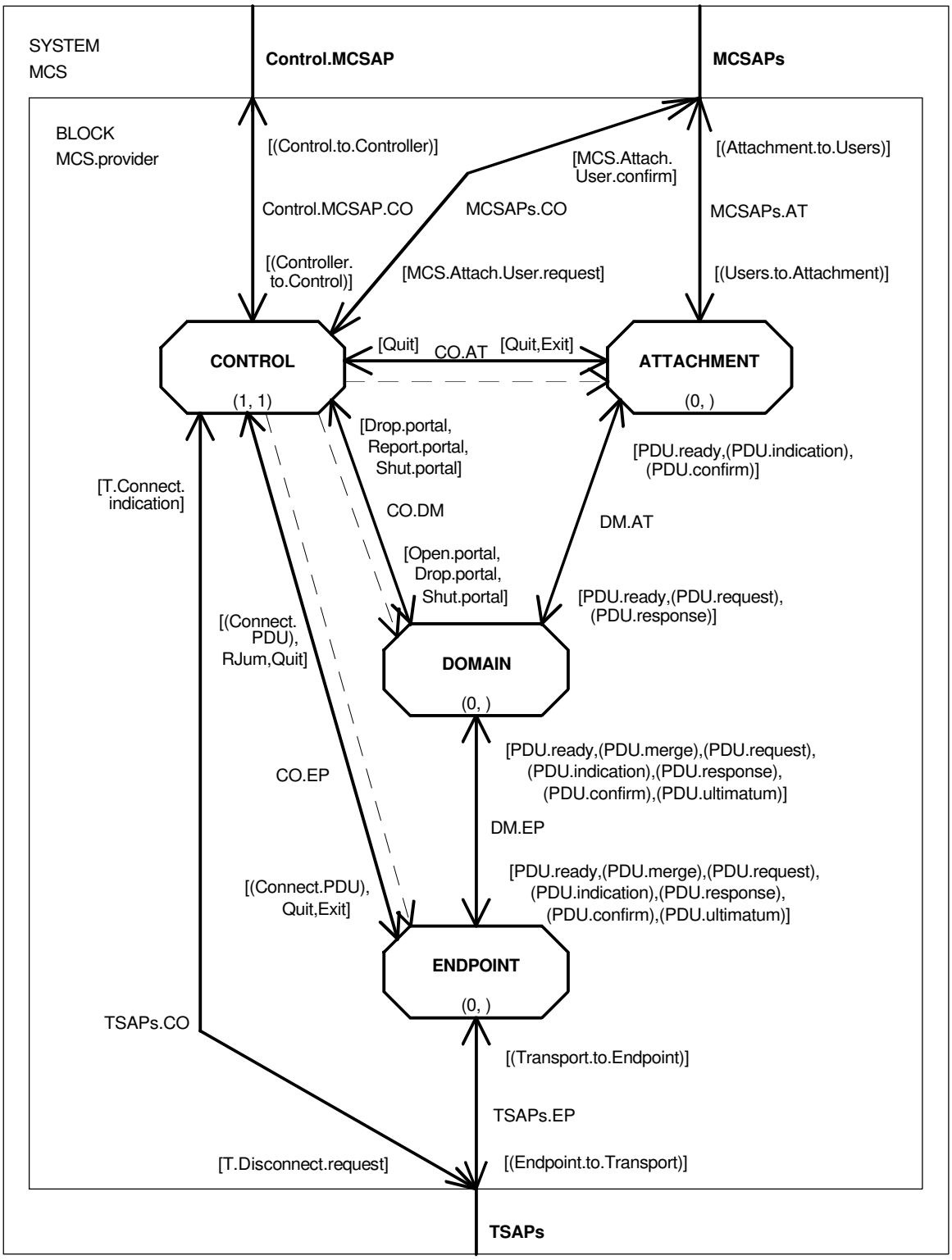


FIGURE II.1/T.125

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;

NEWTYPE    TransportQOSByPri        Array(DataPriority, TransportQOS);
ENDNEWTYPE;

SYNTYPE    UserData           =  OctetString
ENDSYNTYPE;

SYNTYPE    TSDU               =  OctetString
ENDSYNTYPE;

SYNTYPE    Octet              =  Integer  CONSTANTS  0:255
ENDSYNTYPE;

NEWTYPE    OctetString        String(Octet, NullString);
ENDNEWTYPE;

GENERATOR SetOf (TYPE ItemType)      /* subsets with choice operator */
/*AS*/
        Powerset(ItemType);
ADDING
OPERATORS
        Pick:    SetOf          –> ItemType;    /* chooses any element */
AXIOMS
        Pick(Empty)  ==  ERROR!;
        FOR ALL s IN SetOf
        (
                s /= Empty ==> Pick(s) in s;
        );
DEFAULT
        Empty;
ENDGENERATOR;

NEWTYPE    Reason
LITERALS
        RN_domain_disconnected,
        RN_provider_initiated,
        RN_token_purged,
        RN_user_requested,
        RN_channel_purged,
        RN_channel_disbanded,          /* not in MCSPDUs */
        RN_domain_not_hierarchical,    /* not in MCSPDUs */
        RN_parameters_unacceptable,    /* not in MCSPDUs */
        RN_unspecified;                /* not in MCSPDUs */
ENDNEWTYPE;

NEWTYPE    Result

LITERALS
        RT_successful,
        RT_domain_merging,
        RT_domain_not_hierarchical,
        RT_no_such_channel,
        RT_no_such_domain,
        RT_no_such_user,
        RT_not_admitted,
        RT_other_user_id,
        RT_parameters_unacceptable,
        RT_token_not_available,
        RT_token_not_possessed,
        RT_too_many_channels,
        RT_too_many_tokens,
        RT_too_many_users,
```

```
        RT_unspecified_failure,
        RT_user_rejected,
        RT_congested,                   /* not in MCSPDUs */
        RT_domain_disconnected;         /* not in MCSPDUs */
ENDNEWTYPE;

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

SYNTYPE         MCSConnectionId  =  Natural
ENDSYNTYPE;

SYNTYPE         MCSAttachmentId =  PId
ENDSYNTYPE;

SYNTYPE         TCEndpointId   =  PId
ENDSYNTYPE;

        /* Block decomposition */

BLOCK   MCS.provider            REFERENCED;

CHANNEL Control.MCSAP
        FROM ENV TO MCS.provider WITH
                (Controller.to.Control);
        FROM MCS.provider TO ENV WITH
                (Control.to.Controller);

ENDCHANNEL;

SIGNALLIST Controller.to.Control  =
        MCS.Connect.Provider.request,
        MCS.Connect.Provider.response,
        MCS.Disconnect.Provider.request;

SIGNALLIST Control.to.Controller  =
        MCS.Connect.Provider.indication,
        MCS.Connect.Provider.confirm,
        MCS.Disconnect.Provider.indication;

SIGNAL  MCS.Connect.Provider.request
        (
                Natural,                        /* requester's label */
                TSAPAddress,                    /* calling */
                DomainSelector,
                TSAPAddress,                    /* called */
                DomainSelector,
                Boolean,                        /* upward */
                DomainParameters                /* target */
                DomainParameters,               /* minimum */
                DomainParameters,               /* maximum */
                TransportQOSByPri,              /* target */
                TransportQOSByPri,              /* minimum */
                UserData

        );
SIGNAL  MCS.Connect.Provider.indication
        (
                MCSConnectionId,                /* provider-assigned */
                TSAPAddress,                    /* calling */
                DomainSelector,
                TSAPAddress,                    /* called */
                DomainSelector,
```

```
                Boolean,                    /* upward */
                DomainParameters,           /* target */
                DomainParameters,           /* minimum */
                DomainParameters,           /* maximum */
                UserData
        );
SIGNAL  MCS.Connect.Provider.response
        (
                MCSConnectionId,
                Result,
                DomainParameters,
                UserData
        );
SIGNAL  MCS.Connect.Provider.confirm
        (
                Natural,                    /* requester's label */
                Result,
                MCSConnectionId,            /* provider-assigned */
                DomainParameters,
                UserData
        );
SIGNAL  MCS.Disconnect.Provider.request
        (
                MCSConnectionId
        );
SIGNAL  MCS.Disconnect.Provider.indication
        (
                MCSConnectionId,
                Reason
        );
CHANNEL MCSAPs
        FROM ENV TO MCS.provider WITH
                MCS.Attach.User.request,
                (Users.to.Attachment);
        FROM MCS.provider TO ENV WITH
                (Attachment.to.Users);
ENDCHANNEL;

SIGNALLIST Users.to.Attachment  =
        MCS.ready,
        MCS.Detach.User.request,
        MCS.Channel.Join.request,
        MCS.Channel.Leave.request,
        MCS.Channel.Convene.request,
        MCS.Channel.Disband.request,
        MCS.Channel.Admit.request,
        MCS.Channel.Expel.request,
        MCS.Send.Data.request,
        MCS.Uniform.Send.Data.request,
        MCS.Token.Grab.request,
        MCS.Token.Inhibit.request,
        MCS.Token.Give.request,
        MCS.Token.Give.response,
        MCS.Token.Please.request,
        MCS.Token.Release.request,
        MCS.Token.Test.request;

SIGNALLIST Attachment.to.Users  =
        MCS.ready,
        MCS.Attach.User.confirm,
        MCS.Detach.User.indication,
        MCS.Channel.Join.confirm,
        MCS.Channel.Leave.indication,
        MCS.Channel.Convene.confirm,
```

```
        MCS.Channel.Disband.indication,
        MCS.Channel.Admit.indication,
        MCS.Channel.Expel.indication,
        MCS.Send.Data.indication,
        MCS.Uniform.Send.Data.indication,
        MCS.Token.Grab.confirm,
        MCS.Token.Inhibit.confirm,
        MCS.Token.Give.indication,
        MCS.Token.Give.confirm,
        MCS.Token.Please.indication,
        MCS.Token.Release.confirm,
        MCS.Token.Test.confirm;

SIGNAL  MCS.ready                       /* allows one MCS.[Uniform.]Send.Data */
        (
                MCSAttachmentId,
                DataPriority

        );

SIGNAL  MCS.Attach.User.request
        (
                Natural,                /* requester's label */
                DomainSelector

        );

SIGNAL  MCS.Attach.User.confirm
        (
                Natural,                /* requester's label */
                Result,
                MCSAttachmentId,        /* provider-assigned */
                UserId

        );

SIGNAL  MCS.Detach.User.request
        (
                MCSAttachmentId
        );
SIGNAL  MCS.Detach.User.indication
        (
                MCSAttachmentId,
                UserId,
                Reason

        );
SIGNAL  MCS.Channel.Join.request
        (
                MCSAttachmentId,
                ChannelId

        );
SIGNAL  MCS.Channel.Join.confirm
        (
                MCSAttachmentId,
                ChannelId,              /* requested */
                Result,
                ChannelId

        );
SIGNAL  MCS.Channel.Leave.request
        (
                MCSAttachmentId,
                ChannelId

        );
```

```
SIGNAL  MCS.Channel.Leave.indication
        (
                MCSAttachmentId,
                ChannelId,
                Reason

        );
SIGNAL  MCS.Channel.Convene.request
        (
                MCSAttachmentId
        );
SIGNAL  MCS.Channel.Convene.confirm
        (
                MCSAttachmentId,
                Result,
                ChannelId

        );
SIGNAL  MCS.Channel.Disband.request
        (
                MCSAttachmentId,
                ChannelId

        );
SIGNAL  MCS.Channel.Disband.indication
        (
                MCSAttachmentId,
                ChannelId,
                Reason

        );
SIGNAL  MCS.Channel.Admit.request
        (
                MCSAttachmentId,
                ChannelId,
                UserIdSet

        );
SIGNAL  MCS.Channel.Admit.indication
        (
                MCSAttachmentId,
                ChannelId,
                UserId

        );
SIGNAL  MCS.Channel.Expel.request
        (
                MCSAttachmentId,
                ChannelId,
                UserIdSet

        );
SIGNAL  MCS.Channel.Expel.indication
        (
                MCSAttachmentId,
                ChannelId,
                Reason

        );
```

```
SIGNAL  MCS.Send.Data.request
        (
                MCSAttachmentId,
                ChannelId,
                DataPriority,
                Segmentation,
                UserData

        );
SIGNAL  MCS.Send.Data.indication
        (
                MCSAttachmentId,
                ChannelId,
                DataPriority,
                UserId,
                Segmentation,
                UserData

        );
SIGNAL  MCS.Uniform.Send.Data.request
        (
                MCSAttachmentId,
                ChannelId,
                DataPriority,
                Segmentation,
                UserData

        );
SIGNAL  MCS.Uniform.Send.Data.indication
        (
                MCSAttachmentId,
                ChannelId,
                DataPriority,
                UserId,
                Segmentation,
                UserData

        );
SIGNAL  MCS.Token.Grab.request
        (
                MCSAttachmentId,
                TokenId

        );
SIGNAL  MCS.Token.Grab.confirm
        (
                MCSAttachmentId,
                TokenId,
                Result

        );
SIGNAL  MCS.Token.Inhibit.request
        (
                MCSAttachmentId,
                TokenId

        );
SIGNAL  MCS.Token.Inhibit.confirm
        (
                MCSAttachmentId,
                TokenId,
                Result

        );
```

```
SIGNAL  MCS.Token.Give.request
        (
                MCSAttachmentId,
                TokenId,
                UserId
        );
SIGNAL  MCS.Token.Give.indication
        (
                MCSAttachmentId,
                TokenId,
                UserId
        );
SIGNAL  MCS.Token.Give.response
        (
                MCSAttachmentId,
                TokenId,
                Result
        );
SIGNAL  MCS.Token.Give.confirm
        (
                MCSAttachmentId,
                TokenId,
                Result
        );
SIGNAL  MCS.Token.Please.request
        (
                MCSAttachmentId,
                TokenId
        );
SIGNAL  MCS.Token.Please.indication
        (
                MCSAttachmentId,
                TokenId,
                UserId
        );
SIGNAL  MCS.Token.Release.request
        (
                MCSAttachmentId,
                TokenId
        );
SIGNAL  MCS.Token.Release.confirm
        (
                MCSAttachmentId,
                TokenId,
                Result
        );
SIGNAL  MCS.Token.Test.request
        (
                MCSAttachmentId,
                TokenId
        );
```

```
SIGNAL  MCS.Token.Test.confirm
        (
                MCSAttachmentId,
                TokenId,
                TokenStatus

        );

CHANNEL TSAPs
        FROM MCS.provider TO ENV WITH
                (Endpoint.to.Transport);
        FROM ENV TO MCS.provider WITH
                T.Connect.indication,
                (Transport.to.Endpoint);
ENDCHANNEL;

SIGNALLIST Endpoint.to.Transport  =
        T.ready,
        T.Connect.request,
        T.Connect.response,
        T.Data.request,
        T.Disconnect.request;

SIGNALLIST Transport.to.Endpoint  =
        T.ready,
        T.Connect.confirm,
        T.Data.indication,
        T.Disconnect.indication;

SIGNAL  T.ready                         /* allows one T.Data */
        (
                TCEndpointId

        );

SIGNAL  T.Connect.request
        (
                Natural,                /* requester's label */
                TSAPAddress,            /* calling */
                TSAPAddress,            /* called */
                TransportQOS,                   /* target */
                TransportQOS            /* minimum */

        );

SIGNAL  T.Connect.indication
        (
                TCEndpointId,           /* provider-assigned */
                TSAPAddress,            /* calling */
                TSAPAddress,            /* called */
                TransportQOS,                   /* offered */
                TransportQOS            /* minimum */

        );

SIGNAL  T.Connect.response
        (
                TCEndpointId,
                TransportQOS            /* selected */

        );

SIGNAL  T.Connect.confirm
        (
                Natural,                /* requester's label */
                TCEndpointId,           /* provider-assigned */
                TransportQOS            /* selected */

        );
```

```
SIGNAL  T.Data.request
        (
                TCEndpointId,
                TSDU

        );
SIGNAL  T.Data.indication
        (
                TCEndpointId,
                TSDU

        );
SIGNAL  T.Disconnect.request
        (
                TCEndpointId

        );
SIGNAL  T.Disconnect.indication
        (
                Natural,                        /* requester's label */
                TCEndpointId                    /* provider-assigned */

        );
ENDSYSTEM;

BLOCK   MCS.provider;

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

        /* Data type definitions */

NEWTYPE         PDUKind                 /* domain MCSPDUs */
LITERALS
        PDin,           /* plumb domain indication */
        EDrq,           /* erect domain request */
        MCrq,           /* merge channels request */
        MCcf,           /* merge channels confirm */
        PCin,           /* purge channels indication */
        MTrq,           /* merge tokens request */
        MTcf,           /* merge tokens confirm */
        PTin,           /* purge tokens indication */
        DPum,           /* disconnect provider ultimatum */
        RJum,           /* reject MCSPDU ultimatum */
        AUrq,           /* attach user request */
        AUcf,           /* attach user confirm */
        DUrq,           /* detach user request */
        DUin,           /* detach user confirm */
        CJrq,           /* channel join request */
        CJcf,           /* channel join confirm */
        CLrq,           /* channel leave request */
        CCrq,           /* channel convene request */
        CCcf,           /* channel convene confirm */
        CDrq,           /* channel disband request */
        CDin,           /* channel disband confirm */
        CArq,           /* channel admit request */
        CAin,           /* channel admit confirm */
        CErq,           /* channel expel request */
        CEin,           /* channel expel confirm */
        SDrq,           /* send data request */
        SDin,           /* send data indication */
        USrq,           /* uniform send data request */
        USin,           /* uniform send data indication */
        TGrq,           /* token grab request */
        TGcf,           /* token grab confirm */
        TIrq,                   /* token inhibit request */
```

```
        Tlcf,           /* token inhibit confirm */
        TVrq,           /* token give request */
        TVin,           /* token give indication */
        TVrs,           /* token give response */
        TVcf,           /* token give confirm */
        TPrq,           /* token please request */
        TPin,           /* token please indication */
        TRrq,           /* token release request */
        TRcf,           /* token release confirm */
        TTrq,           /* token test request */
        TTcf;           /* token test confirm */

ENDNEWTYPE;

NEWTYPE     PDUStruct
STRUCT
        kind                    PDUKind;
        /* fields used depend on kind */
        channelId               ChannelId;
        channelIds              ChannelIdSet;
        dataPriority            DataPriority;
        detachUserIds           UserIdSet;
        diagnostic              Diagnostic;
        heightLimit             Natural;
        initialOctets           OctetString;
        initiator               UserId;
        mergeChannels                   ChannelAttributesSet;
        mergeTokens             TokenAttributesSet;
        purgeChannelIds         ChannelIdSet;
        purgeTokenIds           TokenIdSet;
        reason                  Reason;
        recipient               UserId;
        requested               ChannelId;
        result                  Result;
        segmentation                    Segmentation;
        subHeight               Natural;
        subInterval             Duration;
        tokenId                 TokenId;
        tokenStatus             TokenStatus;
        userData                UserData;
        userIds                 UserIdSet;
ENDNEWTYPE;

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

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

NEWTYPE     ChannelAttributesSet  SetOf(ChannelAttributes);
ENDNEWTYPE;
```

```
NEWTYPE         TokenKind
LITERALS
        Grabbed,                /* assigned exclusively to one user */
        Inhibited,      /* inhibited by one or more users */
        Giving,         /* reassigning grabbed to a new user */
        Ungivable,      /* the recipient has since detached */
        Given;          /* donor released token or detached */
ENDNEWTYPE;

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

NEWTYPE         TokenAttributesSet      SetOf(TokenAttributes);
ENDNEWTYPE;

SYNTYPE         PortalId        =  Integer CONSTANTS 0:maxPortalIds
ENDSYNTYPE;

NEWTYPE         PortalIdSet     SetOf(PortalId);
ENDNEWTYPE;

NEWTYPE         PortalKind
LITERALS
        Attached,       /* MCS Attachment through an MCSAP */
        Downlink,       /* MCS Connection to a provider below */
        Uplink;         /* MCS Connection to a provider above */
ENDNEWTYPE;

NEWTYPE         PIdByPri        Array(DataPriority, PId);
ENDNEWTYPE;

NEWTYPE         Diagnostic
LITERALS
        DC_inconsistent_merge,
        DC_forbidden_PDU_downward,
        DC_forbidden_PDU_upward,
        DC_invalid_BER_encoding,
        DC_invalid_PER_encoding,
        DC_misrouted_user,
        DC_unrequested_confirm,
        DC_wrong_transport_priority,
        DC_channel_id_conflict,
        DC_token_id_conflict,
        DC_not_user_id_channel,
        DC_too_many_channels,
        DC_too_many_tokens,
        DC_too_many_users,
        DC_OK,                                  /* not in MCSPDUs */
        DC_ignore,                              /* not in MCSPDUs */
        DC_height_limit_exceeded,               /* not in MCSPDUs */
        DC_throughput_inadequate;               /* not in MCSPDUs */
ENDNEWTYPE;

        /* Process decomposition */
PROCESS     Control             (1,1)     REFERENCED;   /* CO */
PROCESS     Attachment                    (0,)      REFERENCED;   /* AT */
PROCESS     Domain              (0,)      REFERENCED;   /* DM */
PROCESS     Endpoint            (0,)      REFERENCED;   /* EP */
```

```
CONNECT Control.MCSAP AND Control.MCSAP.CO;

SIGNALROUTE Control.MCSAP.CO
        FROM ENV TO Control WITH
            (Controller.to.Control);
        FROM Control TO ENV WITH
            (Control.to.Controller);

CONNECT MCSAPs AND MCSAPs.CO, MCSAPs.AT;

SIGNALROUTE MCSAPs.CO
        FROM ENV TO Control WITH
            MCS.Attach.User.request;
        FROM Control TO ENV WITH
            MCS.Attach.User.confirm;

SIGNALROUTE MCSAPs.AT
        FROM ENV TO Attachment WITH
            (Users.to.Attachment);
        FROM Attachment TO ENV WITH
            (Attachment.to.Users);

CONNECT TSAPs AND TSAPs.CO, TSAPs.EP;

SIGNALROUTE TSAPs.CO
        FROM ENV TO Control WITH
            T.Connect.indication;
        FROM Control TO ENV WITH
            T.Disconnect.request;

SIGNALROUTE TSAPs.EP
        FROM ENV TO Endpoint WITH
            (Transport.to.Endpoint);
        FROM Endpoint TO ENV WITH
            (Endpoint.to.Transport);

SIGNALROUTE CO.AT
        FROM Control TO Attachment WITH
            Quit,
            Exit;
        FROM Attachment TO Control WITH
            Quit;

SIGNALROUTE CO.DM
        FROM Control TO Domain WITH
                Open.portal,
                Drop.portal,
                Shut.portal;
        FROM Domain TO Control WITH
                Drop.portal,
                Report.portal,
                Shut.portal;

SIGNALROUTE CO.EP
        FROM Control TO Endpoint WITH
                (Connect.PDU),
                Quit,
                Exit;
        FROM Endpoint TO Control WITH
                (Connect.PDU),
                RJum,
                Quit;

SIGNALROUTE DM.AT
        FROM Domain TO Attachment WITH
                PDU.ready,
                (PDU.indication),
                (PDU.confirm);
        FROM Attachment TO Domain WITH
                PDU.ready,
                (PDU.request),
                (PDU.response);
```

```
SIGNALROUTE DM.EP
        FROM Domain TO Endpoint WITH
                PDU.ready,
                (PDU.merge),
                (PDU.request),
                (PDU.indication),
                (PDU.response),
                (PDU.confirm),
                (PDU.ultimatum);
        FROM Endpoint TO Domain WITH
                PDU.ready,
                (PDU.merge),
                (PDU.request),
                (PDU.indication),
                (PDU.response),
                (PDU.confirm),
                (PDU.ultimatum);

SIGNAL  Open.portal
        (
            PortalId,
            PortalKind,
            PldByPri
        );

SIGNAL  Report.portal
        (
            PortalId,
            Diagnostic
        );

SIGNAL  Drop.portal
        (
                PortalId,
                Reason
        );

SIGNAL  Shut.portal
        (
                PortalId
        );

SIGNAL  Quit;

SIGNAL  Exit;

SIGNALLIST Connect.PDU  =
            Connect.Initial,
            Connect.Response,
            Connect.Additional,
            Connect.Result;

SIGNAL  Connect.Initial
        (
                DomainSelector,             /* calling */
                DomainSelector,             /* called */
                Boolean,                     /* upward */
                DomainParameters,            /* target */
                DomainParameters,            /* minimum */
                DomainParameters,            /* maximum */
                UserData
        );

SIGNAL  Connect.Response
        (
                Result,
                Natural,
                DomainParameters,
                UserData
        );
```

```
SIGNAL  Connect.Additional
        (
                Natural,
                DataPriority
        );

SIGNAL  Connect.Result
        (
                Result
        );

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

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

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

SIGNALLIST PDU.response  =
        TVrs;

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

SIGNALLIST PDU.ultimatum  =
        DPum, RJum;

SIGNAL  PDU.ready                       /* allows one domain MCSPDU */
        (
                DataPriority
        );

SIGNAL  PDin    (PDUStruct);    /* plumb domain indication */
SIGNAL  EDrq    (PDUStruct);    /* erect domain request */
SIGNAL  MCrq    (PDUStruct);    /* merge channels request */
SIGNAL  MCcf    (PDUStruct);    /* merge channels confirm */
SIGNAL  PCin    (PDUStruct);    /* purge channels indication */
SIGNAL  MTrq    (PDUStruct);    /* merge tokens request */
SIGNAL  MTcf    (PDUStruct);    /* merge tokens confirm */
SIGNAL  PTin    (PDUStruct);    /* purge tokens indication */
SIGNAL  DPum    (PDUStruct);    /* disconnect provider ultimatum */
SIGNAL  RJum    (PDUStruct);    /* reject MCSPDU ultimatum */
SIGNAL  AUrq    (PDUStruct);    /* attach user request */
SIGNAL  AUcf    (PDUStruct);    /* attach user confirm */
SIGNAL  DUrq    (PDUStruct);    /* detach user request */
SIGNAL  DUin    (PDUStruct);    /* detach user confirm */
SIGNAL  CJrq    (PDUStruct);    /* channel join request */
SIGNAL  CJcf    (PDUStruct);    /* channel join confirm */
SIGNAL  CLrq    (PDUStruct);    /* channel leave request */
SIGNAL  CCrq    (PDUStruct);    /* channel convene request */
SIGNAL  CCcf    (PDUStruct);    /* channel convene confirm */
SIGNAL  CDrq    (PDUStruct);    /* channel disband request */
```

```
SIGNAL  CDin     (PDUStruct);     /* channel disband confirm */
SIGNAL  CArq     (PDUStruct);     /* channel admit request */
SIGNAL  CAin     (PDUStruct);     /* channel admit confirm */
SIGNAL  CErq     (PDUStruct);     /* channel expel request */
SIGNAL  CEin     (PDUStruct);     /* channel expel confirm */
SIGNAL  SDrq     (PDUStruct);     /* send data request */
SIGNAL  SDin     (PDUStruct);     /* send data indication */
SIGNAL  USrq     (PDUStruct);     /* uniform send data request */
SIGNAL  USin     (PDUStruct);     /* uniform send data indication */
SIGNAL  TGrq     (PDUStruct);     /* token grab request */
SIGNAL  TGcf     (PDUStruct);     /* token grab confirm */
SIGNAL  TIrq     (PDUStruct);     /* token inhibit request */
SIGNAL  TIcf     (PDUStruct);     /* token inhibit confirm */
SIGNAL  TVrq     (PDUStruct);     /* token give request */
SIGNAL  TVin     (PDUStruct);     /* token give indication */
SIGNAL  TVrs     (PDUStruct);     /* token give response */
SIGNAL  TVcf     (PDUStruct);     /* token give confirm */
SIGNAL  TPrq     (PDUStruct);     /* token please request */
SIGNAL  TPin     (PDUStruct);     /* token please indication */
SIGNAL  TRrq     (PDUStruct);     /* token release request */
SIGNAL  TRcf     (PDUStruct);     /* token release confirm */
SIGNAL  TTrq     (PDUStruct);     /* token test request */
SIGNAL  TTcf     (PDUStruct);     /* token test confirm */

ENDBLOCK;
```

Appendix III

SDL specification of the Control process

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

```
PROCESS     Control;

        /* Type definitions */

NEWTYPE         Proc
LITERALS
        Nil,
        Receiving,              /*          Endpoint */
        Responding,             /*          Endpoint */
        Engaged,                /* Attachment Endpoint */
        Quitting,               /* Attachment Endpoint */
        Quit;                   /* Attachment Endpoint */

ENDNEWTYPE;


NEWTYPE         ProcByPri       Array(DataPriority, Proc);
ENDNEWTYPE;

NEWTYPE         CallSide
LITERALS
        Calling,
        Called;
ENDNEWTYPE;
```

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

NEWTYPE         Portal          Array(PortalId, PortalStruct);
ENDNEWTYPE;

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

NEWTYPE         Domain          Array(DomainSelector, DomainStruct);
ENDNEWTYPE;

NEWTYPE         DomainSelectorSet           SetOf(DomainSelector);
ENDNEWTYPE;

        /* Data declarations */

DCL     domain          Domain,                         /* resource arrays */
        portal          Portal;

        /* Note: The fields of a domain or portal array element
        are undefined if the corresponding index is not in dUsed
        or pUsed respectively. */
DCL     dUsed           DomainSelectorSet,              /* indexes used */
        pUsed           PortalIdSet;

DCL     pFree           PortalIdSet;                    /* indexes free */

DCL     nullParms       DomainParameters;               /* initializer */

        /* Procedure decomposition */
        /*          Initialize_resources
                    Identify_sender                     (p, dp)
                    Min_parms                           (min, a, b)
```

```
                Max_parms                           (max, a, b)
                Test_parms                          (result, x, min, max)
                MCS_Connect_Provider_request        (...)
                T_Connect_indication                (...)
                Connect_Initial                     (...)
                MCS_Connect_Provider_response       (...)
                Connect_Response                    (...)
                Connect_Additional                  (...)
                Connect_Result                      (...)
                MCS_Disconnect_Provider_request     (...)
                MCS_Attach_User_request             (...)
                Open_portal                         (p)
                Drop_portal                         (p, reason)
                Report_portal                       (p, diagnostic)
                RJum                                (pdu)
                Quit_portal                         (p, reason)
                Quit
                Shut_portal                         (p)
                Exit_portal                         (p)       */
                                                                /*---------------------------------*/
PROCEDURE       Initialize_resources;                           /*  Initialize_resources  */
                                                                /*---------------------------------*/
    DCL         p           PortalId,
                ds          DomainSelector,
                dSet        DomainSelectorSet;
    START
    COMMENT     'Initialize data structures during process start-up
                before accepting the first input signal.
                Note that each SetOf automatically defaults to Empty.
                Configure one hypothetical domain as an example.
                Some limits are determined by the implementation.
                ';
    TASK        nullParms!maxChannelIds := 0,
                nullParms!maxUserIds := 0,
                nullParms!maxTokenIds := 0,
                nullParms!numPriorities := 0,
                nullParms!minThroughput := 0,
                nullParms!maxHeight := 0,
                nullParms!maxMCSPDUsize := 0,
                nullParms!protocolVersion := 0,
                p := maxPortalIds;
     1b :       /* for p = ?..1 */
    DECISION p > 0;
     (True):    TASK    portal(p)!mcId := p,
                        portal(p)!ccId := p,
                        pFree := Incl(p, pFree),
                        p := p - 1;
                JOIN 1b;
    ELSE:ENDDECISION;
    TASK        ds := Mkstring(77) // Mkstring(67) // Mkstring(83),
                dUsed := Incl(ds, dUsed),
                domain(ds)!Pid := Null,
                domain(ds)!portals := 0,
                domain(ds)!upward := 0,
                dSet := dUsed;
     2b :       /* for ds in dSet */
    DECISION dSet =     Empty;
     (False):   TASK    ds := Pick(dSet),
                        dSet := Del(ds, dSet),
                        domain(ds)!minParms!maxChannelIds := 0,
                        domain(ds)!minParms!maxUserIds := 0,
                        domain(ds)!minParms!maxTokenIds := 0,
                        domain(ds)!minParms!numPriorities := 1,
                        domain(ds)!minParms!minThroughput := 0,
                        domain(ds)!minParms!maxHeight := 1,
                        domain(ds)!minParms!maxMCSPDUsize := 35,
```

```
                        domain(ds)!minParms!protocolVersion := 1,
                        domain(ds)!maxParms!maxChannelIds := 65535,
                        domain(ds)!maxParms!maxUserIds := 65535,
                        domain(ds)!maxParms!maxTokenIds := 65535,
                        domain(ds)!maxParms!numPriorities := 4,
                        domain(ds)!maxParms!minThroughput := 1000000,
                        domain(ds)!maxParms!maxHeight := 1000,
                        domain(ds)!maxParms!maxMCSPDUsize := 32768,
                        domain(ds)!maxParms!protocolVersion := 2;
                JOIN 2b;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;

                                                                        /*-------------------------*/
PROCEDURE       Identify_sender;                                        /*  Identify_sender  */
FPAR    IN/OUT  p               PortalId,                               /*-------------------------*/
        IN/OUT  dp              DataPriority;

        DCL     pSet            PortalIdSet;
        START
        COMMENT 'An alternative would be to carry this
                information explicitly in SDL signals.
                ';
        TASK    pSet := pUsed;
         1b :   /* for p in pSet */
        DECISION pSet =  Empty;
         (False): TASK  p := Pick(pSet),
                        pSet := Del(p, pSet),
                        dp := 0;
                 2b :   /* for dp = 0..3 */
                DECISION dp < 4;
                 (True):    DECISION portal(p)!pids(dp) = SENDER;
                             (True): RETURN;
                            ELSE:ENDDECISION;
                            TASK dp := dp + 1;
                            JOIN 2b;
                ELSE:ENDDECISION;
                JOIN 1b;
        ELSE:ENDDECISION;
        TASK    p := 0,
                dp := 0;
        RETURN;
        ENDPROCEDURE;

                                                                        /*--------------------*/
PROCEDURE       Min_parms;                                              /*  Min_parms  */
FPAR    IN/OUT  min             DomainParameters,                       /*--------------------*/
                a               DomainParameters,
                b               DomainParameters;

        START
        COMMENT 'Return the minimum of two parameter sets.
                ';
        TASK    min!maxChannelIds := IF a!maxChannelIds < b!maxChannelIds
                                THEN a!maxChannelIds ELSE b!maxChannelIds FI,
                min!maxUserIds := IF a!maxUserIds < b!maxUserIds
                                THEN a!maxUserIds ELSE b!maxUserIds FI,
                min!maxTokenIds := IF a!maxTokenIds < b!maxTokenIds
                                THEN a!maxTokenIds ELSE b!maxTokenIds FI,
                min!numPriorities := IF a!numPriorities < b!numPriorities
                                THEN a!numPriorities ELSE b!numPriorities FI,
                min!minThroughput := IF a!minThroughput < b!minThroughput
                                THEN a!minThroughput ELSE b!minThroughput FI,
                min!maxHeight := IF a!maxHeight < b!maxHeight
                                THEN a!maxHeight ELSE b!maxHeight FI,
```

```
                    min!maxMCSPDUsize := IF a!maxMCSPDUsize < b!maxMCSPDUsize
                                THEN a!maxMCSPDUsize ELSE b!maxMCSPDUsize FI,
                    min!protocolVersion := IF a!protocolVersion < b!protocolVersion
                                THEN a!protocolVersion ELSE b!protocolVersion FI;
        RETURN;
        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;
        START
        COMMENT 'Process an MCS.Connect.Provider.request input signal.
                Begin parameter negotiation and allocate a portal.
                Create an endpoint process for the initial TC and
                transmit Connect.Initial through it.
                ';
        DECISION pFree = Empty;
         (True):        TASK    result := RT_congested;
                        JOIN 1f;
        ELSE:ENDDECISION;
        TASK            ds := localDomain;
        DECISION ds in dUsed;
         (False):       TASK    result := RT_no_such_domain;
                        JOIN 1f;
        ELSE:ENDDECISION;
        DECISION upward and domain(ds)!upward /= 0;
         (True):        TASK    result := RT_domain_not_hierarchical;
                        JOIN 1f;
        ELSE:ENDDECISION;
        CALL            Max_parms(minParms, minParms, domain(ds)!minParms);
        CALL            Min_parms(maxParms, maxParms, domain(ds)!maxParms);
        DECISION domain(ds)!portals > 0;
         (True):        TASK    targetParms := domain(ds)!parameters;
         (False):       CALL    Max_parms(targetParms, targetParms, minParms);
                        CALL    Min_parms(targetParms, targetParms, maxParms);
        ENDDECISION;
        CALL            Test_parms(result, targetParms, minParms, maxParms);
        DECISION result = RT_successful;
         (False): 1f :
                        OUTPUT  MCS.Connect.Provider.confirm
                                (label, result, 0, nullParms, NullString);
                        RETURN;
        ELSE:ENDDECISION;
        DECISION domain(ds)!portals > 0;
         (True):        TASK    minParms := targetParms,
                                maxParms := targetParms;
        ELSE:ENDDECISION;
        CREATE          Endpoint(Null, localTSAP, remoteTSAP,
                        targetQOSByPri(0), minQOSByPri(0),
                        nullParms);
        OUTPUT          Connect.Initial(localDomain, remoteDomain, upward,
                        targetParms, minParms, maxParms, userData)
                        TO OFFSPRING;
        TASK            p := Pick(pFree),
                        pFree := Del(p, pFree),
                        pUsed := Incl(p, pUsed),
                        portal(p)!kind := IF upward THEN Uplink ELSE Downlink FI,
```

```
                    portal(p)!label := label,
                    portal(p)!domain := ds,
                    portal(p)!opened := False,
                    portal(p)!notify := True,
                    portal(p)!minParms := minParms,
                    portal(p)!maxParms := maxParms,
                    portal(p)!parameters := nullParms,
                    portal(p)!callSide := Calling,
                    portal(p)!localTSAP := localTSAP,
                    portal(p)!remoteTSAP := remoteTSAP,
                    portal(p)!targetQOSByPri := targetQOSByPri,
                    portal(p)!minQOSByPri := minQOSByPri,
                    portal(p)!pids(0) := OFFSPRING,
                    portal(p)!proc(0) := Receiving,
                    dp := 1;
     2b :           /* for dp = 1..3 */
    DECISION dp < 4;
     (True):        TASK     portal(p)!pids(dp) := Null,
                              portal(p)!proc(dp) := Nil,
                              dp := dp + 1;
                    JOIN 2b;
    ELSE:ENDDECISION;
    RETURN;
    ENDPROCEDURE;


                                                                    /*-----------------------------------*/
PROCEDURE     T_Connect_indication;                                 /*  T_Connect_indication  */
FPAR          tcId                TCEndpointId,                     /*-----------------------------------*/
              remoteTSAP          TSAPAddress,
              localTSAP           TSAPAddress,
              offeredQOS          TransportQOS,
              minQOS              TransportQOS;

    DCL       p                   PortalId,
              dp                  DataPriority;
    START
    COMMENT   'Process a T.Connect.indication input signal.
              Allocate a portal, in case this is an initial TC.
              Create an endpoint process to receive Connect.Initial
              or Connect.Additional.
              ';
    DECISION pFree = Empty
              or offeredQOS!throughput < minQOS!throughput
              or offeredQOS!transitDelay > minQOS!transitDelay
              or offeredQOS!dataPriority > minQOS!dataPriority;
     (True):  OUTPUT  T.Disconnect.request(tcId);
              RETURN;
    ELSE:ENDDECISION;
    CREATE    Endpoint(tcId, localTSAP, remoteTSAP,
              offeredQOS, minQOS,
              nullParms);
    TASK      p := Pick(pFree),
              pFree := Del(p, pFree),
              pUsed := Incl(p, pUsed),
              portal(p)!kind := Downlink,
              portal(p)!opened := False,
              portal(p)!notify := False,
              portal(p)!parameters := nullParms,
              portal(p)!callSide := Called,
              portal(p)!localTSAP := localTSAP,
              portal(p)!remoteTSAP := remoteTSAP,
              portal(p)!pids(0) := OFFSPRING,
              portal(p)!proc(0) := Receiving,
              dp := 1;
     1b :     /* for dp = 1..3 */
```

```
        DECISION dp < 4;
         (True):        TASK      portal(p)!pids(dp) := Null,
                                  portal(p)!proc(dp) := Nil,
                                  dp := dp + 1;
                        JOIN 1b;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;

                                                                        /*------------------------*/
PROCEDURE               Connect_Initial;                                /*  Connect_Initial  */
FPAR                    remoteDomain  DomainSelector,                   /*------------------------*/
                        localDomain   DomainSelector,
                        upward        Boolean,
                        targetParms   DomainParameters,
                        minParms      DomainParameters,
                        maxParms      DomainParameters,
                        userData      UserData;

        DCL             result        Result,
                        p             PortalId,
                        dp            DataPriority,
                        ds            DomainSelector;
        START
        COMMENT         'Process a Connect.Initial input signal.
                        Retain the portal and begin parameter negotiation.
                        Indicate the connection to the controlling user.
                        ';
        CALL            Identify_sender(p, dp);
        DECISION p in pUsed and portal(p)!callSide = Called
                        and dp = 0 and portal(p)!proc(0) = Receiving;
         (False):       TASK      result := RT_unspecified_failure;
                        JOIN 1f;
        ELSE:ENDDECISION;
        TASK            ds := localDomain;
        DECISION ds in dUsed;
         (False):       TASK      result := RT_no_such_domain;
                        JOIN 1f;
        ELSE:ENDDECISION;
        DECISION upward or domain(ds)!upward = 0;
         (False):       TASK      result := RT_domain_not_hierarchical;
                        JOIN 1f;
        ELSE:ENDDECISION;
        CALL            Max_parms(minParms, minParms, domain(ds)!minParms);
        CALL            Min_parms(maxParms, maxParms, domain(ds)!maxParms);
        DECISION domain(ds)!portals > 0;
         (True):        TASK      targetParms := domain(ds)!parameters;
         (False):       CALL      Max_parms(targetParms, targetParms, minParms);
                        CALL      Min_parms(targetParms, targetParms, maxParms);
        ENDDECISION;
        CALL            Test_parms(result, targetParms, minParms, maxParms);
        DECISION result = RT_successful;
         (False): 1f :
                        OUTPUT    Connect.Response(result, 0, nullParms, NullString)
                                  TO SENDER;
                        CALL      Quit_portal(p, RN_unspecified);
                        RETURN;
        ELSE:ENDDECISION;
        DECISION domain(ds)!portals > 0;
         (True):        TASK      minParms := targetParms,
                                  maxParms := targetParms;
        ELSE:ENDDECISION;
        TASK            portal(p)!kind := IF upward THEN Downlink ELSE Uplink FI,
                        portal(p)!domain := ds,
                        portal(p)!notify := True,
                        portal(p)!minParms := minParms,
```

```
                portal(p)!maxParms := maxParms,
                portal(p)!proc(0) := Responding;
        OUTPUT  MCS.Connect.Provider.indication
                (portal(p)!mcId, portal(p)!localTSAP, localDomain, portal(p)!remoteTSAP,
                remoteDomain, upward, targetParms, minParms, maxParms, userData);
        RETURN;
        ENDPROCEDURE;

                                                        /*-------------------------------------------------------*/
PROCEDURE       MCS_Connect_Provider_response;          /*  MCS_Connect_Provider_response  */
FPAR            mcId            MCSConnectionId,        /*-------------------------------------------------------*/
                result          Result,
                parameters      DomainParameters,
                userData        UserData;

        DCL     p               PortalId;
        START
        COMMENT 'Process an MCS.Connect.Provider.response input signal.
                Check negotiated parameters and force a preference.
                Transmit Connect.Response.
                If there are no additional TCs, open the portal.
                ';
        TASK    p := mcId;
        DECISION p in pUsed and portal(p)!proc(0) = Responding;
         (False):   RETURN;
        ELSE:ENDDECISION;
        DECISION result = RT_successful;
         (False):   TASK    result := RT_user_rejected,
                            portal(p)!notify := False;
                    JOIN 1f;
        ELSE:ENDDECISION;
        CALL    Test_parms(result, parameters, portal(p)!minParms, portal(p)!maxParms);
        DECISION result = RT_successful;
         (False): 1f :
                    OUTPUT  Connect.Response(result, 0, parameters, userData)
                            TO portal(p)!pids(0);
                    CALL    Quit_portal(p, RN_parameters_unacceptable);
                    RETURN;
        ELSE:ENDDECISION;
        CALL    Max_parms(parameters, parameters, portal(p)!minParms);
        OUTPUT  Connect.Response(result, portal(p)!ccId, parameters, userData)
                TO portal(p)!pids(0);
        TASK    portal(p)!parameters := parameters,
                portal(p)!proc(0) := Engaged;
        CALL    Open_portal(p);
        RETURN;
        ENDPROCEDURE;

                                                        /*--------------------------------*/
PROCEDURE       Connect_Response;                       /*  Connect_Response  */
FPAR            result          Result,                 /*--------------------------------*/
                ccId            Natural,
                parameters      DomainParameters,
                userData        UserData;

        DCL     p               PortalId,
                dp              DataPriority;
        START
        COMMENT 'Process a Connect.Response input signal.
                Check the negotiated parameters.
                Create endpoint processes for additional TCs and
                transmit Connect.Additional through them.
                If there are no additional TCs, open the portal.
                ';
        CALL    Identify_sender(p, dp);
```

```
        DECISION p in pUsed and portal(p)!callSide = Calling
                        and dp = 0 and portal(p)!proc(0) = Receiving;
         (False):       CALL    Quit_portal(p, RN_unspecified);
                        RETURN;
        ELSE:ENDDECISION;
        TASK            portal(p)!parameters := parameters,
                        portal(p)!userData := userData;
        DECISION result = RT_successful;
         (False):       JOIN 1f;
        ELSE:ENDDECISION;
        CALL            Test_parms(result, parameters, portal(p)!minParms, portal(p)!maxParms);
        DECISION result = RT_successful;
         (False): 1f :
                        OUTPUT  MCS.Connect.Provider.confirm
                                (portal(p)!label, result, 0, parameters, userData);
                        TASK    portal(p)!notify := False;
                        CALL    Quit_portal(p, RN_unspecified);
                        RETURN;
        ELSE:ENDDECISION;
        TASK            portal(p)!proc(0) := Engaged,
                        dp := 1;
         2b :           /* for dp = 1..? */
        DECISION dp < parameters!numPriorities;
         (True):        CREATE  Endpoint(Null, portal(p)!localTSAP, portal(p)!remoteTSAP,
                                portal(p)!targetQOSByPri(dp), portal(p)!minQOSByPri(dp),
                                parameters);
                        OUTPUT  Connect.Additional(ccId, dp)
                                TO OFFSPRING;
                        TASK    portal(p)!pids(dp) := OFFSPRING,
                                portal(p)!proc(dp) := Receiving,
                                dp := dp + 1;
                        JOIN 2b;
        ELSE:ENDDECISION;
        CALL            Open_portal(p);
        RETURN;
        ENDPROCEDURE;
                                                                        /*--------------------------------*/
PROCEDURE               Connect_Additional;                             /*  Connect_Additional  */
FPAR                    ccId            Natural,                        /*--------------------------------*/
                        dp              DataPriority;

        DCL             p               PortalId,
                        r               PortalId,
                        x               DataPriority;
        START
        COMMENT         'Process a Connect.Additional input signal.
                        Release the allocated portal and piggyback onto
                        the preceding Connect.Initial.
                        Transmit Connect.Result.
                        If all TCs are established, open the portal.
                        ';
        CALL            Identify_sender(r, x);
        DECISION r in pUsed and portal(r)!callSide = Called
                        and x = 0 and portal(r)!proc(0) = Receiving;
         (False):       JOIN 1f;
        ELSE:ENDDECISION;
        TASK            p := ccId;
        DECISION p in pUsed and portal(p)!callSide = Called
                        and dp > 0 and dp < portal(p)!parameters!numPriorities
                        and portal(p)!proc(0) = Engaged and portal(p)!proc(dp) = Nil;
         (False): 1f :
                        OUTPUT  Connect.Result(RT_unspecified_failure)
                                TO SENDER;
                        CALL    Quit_portal(r, RN_unspecified);
                        RETURN;
        ELSE:ENDDECISION;
```

```
        TASK        pUsed := Del(r, pUsed),
                    pFree := Incl(r, pFree);
        OUTPUT      Connect.Result(RT_successful)
                    TO SENDER;
        TASK        portal(p)!pids(dp) := SENDER,
                    portal(p)!proc(dp) := Engaged;
        CALL        Open_portal(p);
        RETURN;
        ENDPROCEDURE;
                                                                    /*--------------------------*/
PROCEDURE           Connect_Result;                                 /*  Connect_Result  */
FPAR                result          Result;                         /*--------------------------*/
        DCL         p               PortalId,
                    dp              DataPriority;
        START
        COMMENT     'Process a Connect.Result input signal.
                    If all TCs are established, open the portal.
                    ';
        CALL        Identify_sender(p, dp);
        DECISION p in pUsed and portal(p)!callSide = Calling
                    and dp > 0 and portal(p)!proc(dp) = Receiving;
         (False):   CALL    Quit_portal(p, RN_unspecified);
                    RETURN;
        ELSE:ENDDECISION;
        DECISION result = RT_successful;
         (False):   OUTPUT  MCS.Connect.Provider.confirm
                            (portal(p)!label, result, 0,
                            portal(p)!parameters, portal(p)!userData);
                    TASK    portal(p)!notify := False;
                    CALL    Quit_portal(p, RN_unspecified);
                    RETURN;
        ELSE:ENDDECISION;
        TASK        portal(p)!proc(dp) := Engaged;
        CALL        Open_portal(p);
        RETURN;
        ENDPROCEDURE;
                                                                    /*-------------------------------------------------------*/
PROCEDURE           MCS_Disconnect_Provider_request;                /*  MCS_Disconnect_Provider_request  */
FPAR                mcId            MCSConnectionId;                /*-------------------------------------------------------*/
        DCL         p               PortalId,
                    ds              DomainSelector;
        START
        COMMENT     'Process an MCS.Disconnect.Provider.request input signal.
                    If the portal is open, this can be done gracefully.
                    ';
        TASK        p := mcId;
        DECISION p in pUsed and portal(p)!notify;
         (True):    TASK    portal(p)!notify := False,
                            ds := portal(p)!domain;
                    DECISION portal(p)!opened and domain(ds)!portals > 0;
                     (True):   OUTPUT Drop.portal(p, RN_user_requested) TO domain(ds)!pid;
                     (False):  CALL  Quit_portal(p, RN_unspecified);
                    ENDDECISION;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;
                                                                    /*-----------------------------------------*/
PROCEDURE           MCS_Attach_User_request;                        /*  MCS_Attach_User_request  */
FPAR                label           Natural,                        /*-----------------------------------------*/
                    localDomain     DomainSelector;
```

```
        DCL         p               PortalId,
                    dp              DataPriority,
                    ds              DomainSelector;
        START
        COMMENT     'Process an MCS.Attach.User.request input signal.
                    Allocate a portal, expecting to create an attachment,
                    and open the portal.
                    ';
        DECISION pFree = Empty;
         (True):    TASK    result := RT_congested;
                    JOIN 1f;
        ELSE:ENDDECISION;
        TASK        ds := localDomain;
        DECISION ds in dUsed;
         (False):   TASK    result := RT_no_such_domain;
                     1f :
                    OUTPUT  MCS.Attach.User.confirm
                            (label, result, Null, 0);
                    RETURN;
        ELSE:ENDDECISION;
        TASK        p := Pick(pFree),
                    pFree := Del(p, pFree),
                    pUsed := Incl(p, pUsed),
                    portal(p)!kind := Attached,
                    portal(p)!label := label,
                    portal(p)!domain := ds,
                    portal(p)!opened := False,
                    portal(p)!notify := False,
                    portal(p)!pids(0) := Null,
                    portal(p)!proc(0) := Engaged,
                    dp := 1;
         2b :       /* for dp = 1..3 */
        DECISION dp < 4;
         (True):    TASK    portal(p)!pids(dp) := Null,
                            portal(p)!proc(dp) := Nil,
                            dp := dp + 1;
                    JOIN 2b;
        ELSE:ENDDECISION;
        CALL        Open_portal(p);
        RETURN;
        ENDPROCEDURE;

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

        DCL         dp              DataPriority,
                    ds              DomainSelector,
                    numP            Natural,
                    parameters      DomainParameters;
        START
        COMMENT     'When all TCs have been established, or if this
                    is a user attachment, open the portal to a domain.
                    If the domain process is stopping, try again later.
                    Attachments must wait until domain parameters are set.
                    ';
        TASK        ds := portal(p)!domain,
                    dp := 0;
        DECISION portal(p)!kind = Attached;
         (True):    TASK    parameters := domain(ds)!minParms,
                            numP := 1;
         (False):   TASK    parameters := portal(p)!parameters,
                            numP := parameters!numPriorities;
        ENDDECISION;
         1b :       /* for dp = 0..? */
```

```
        DECISION dp < numP;
         (True):      DECISION portal(p)!proc(dp) = Engaged;
                       (False):RETURN;
                      ELSE:ENDDECISION;
                      TASK     dp := dp + 1;
                      JOIN 1b;
        ELSE:ENDDECISION;
        DECISION domain(ds)!pid = Null;
         (False):     DECISION domain(ds)!portals = 0;
                       (True):   RETURN;
                      ELSE:ENDDECISION;
         (True):      CREATE  Domain(parameters);
                      TASK     domain(ds)!pid := OFFSPRING,
                               domain(ds)!portals := 0,
                               domain(ds)!upward := 0,
                               domain(ds)!parameters := parameters;
        ENDDECISION;
        DECISION portal(p)!kind = Attached;
         (True):      CREATE  Attachment(portal(p)!label, domain(ds)!parameters);
                      TASK     portal(p)!pids(0) := OFFSPRING;
         (False):     DECISION domain(ds)!parameters = parameters;
                       (False):  CALL  Quit_portal(p, RN_parameters_unacceptable);
                                 RETURN;
                      ELSE:ENDDECISION;
                      DECISION portal(p)!kind = Uplink;
                       (True):   DECISION domain(ds)!upward = 0;
                                  (False):    CALL    Quit_portal(p, RN_domain_not_hierarchical);
                                              RETURN;
                                  (True):     TASK    domain(ds)!upward := p;
                                 ENDDECISION;
                      ELSE:ENDDECISION;
                      DECISION portal(p)!callSide = Called;
                       (False):  OUTPUT   MCS.Connect.Provider.confirm
                                          (portal(p)!label, RT_successful, portal(p)!mcId,
                                          portal(p)!parameters, portal(p)!userData);
                      ELSE:ENDDECISION;
        ENDDECISION;
        OUTPUT       Open.portal(p, portal(p)!kind, portal(p)!pids) TO domain(ds)!pid;
        TASK         domain(ds)!portals := domain(ds)!portals + 1,
                     portal(p)!opened := True;
        RETURN;
        ENDPROCEDURE;

                                                                          /*--------------------*/
PROCEDURE            Drop_portal;                                         /*  Drop_portal  */
FPAR                 p              PortalId,                             /*--------------------*/
                     reason         Reason;

        START
        COMMENT'     Process a Drop.portal input signal.
                     ';
        CALL         Quit_portal(p, reason);
        RETURN;
        ENDPROCEDURE;

                                                                          /*----------------------*/
PROCEDURE            Report_portal;                                       /*  Report_portal  */
FPAR                 p              PortalId,                             /*----------------------*/
                     diagnostic     Diagnostic;

        DCL          reason         Reason;
        START
        COMMENT      'Process a Report.portal input signal.
                     For testing, local diagnostic could be logged.
                     ';
        TASK         reason := RN_unspecified;
```

```
        DECISION diagnostic;
         (DC_throughput_inadequate):
                        TASK    reason := RN_provider_initiated;
         (DC_height_limit_exceeded):
                        TASK    reason := RN_domain_not_hierarchical;
        ELSE:ENDDECISION;
        CALL            Quit_portal(p, reason);
        RETURN;
        ENDPROCEDURE;

                                                                        /*-----------*/
PROCEDURE               RJum;                                           /*  RJum  */
FPAR                    pdu             PDUStruct;                      /*-----------*/

        DCL             p               PortalId,
                        dp              DataPriority;
        START
        COMMENT         'Process an RJum input signal.
                        For testing, remote pdu!diagnostic could be logged.
                        ';
        CALL            Identify_sender(p, dp);
        CALL            Quit_portal(p, RN_unspecified);
        RETURN;
        ENDPROCEDURE;

                                                                        /*--------------------*/
PROCEDURE               Quit_portal;                                    /*  Quit_portal  */
FPAR                    p               PortalId,                       /*--------------------*/
                        reason          Reason;

        DCL             result          Result,
                        dp              DataPriority;
        START
        COMMENT         'If necessary, notify the controlling user.
                        Quiesce the portal processes.
                        ';
        DECISION p in pUsed;
         (False):       RETURN;
        ELSE:ENDDECISION;
        DECISION portal(p)!notify;
         (True):        DECISION portal(p)!callSide = Called or portal(p)!opened;
                         (True):    OUTPUT  MCS.Disconnect.Provider.indication
                                            (portal(p)!mcId, reason);
                         (False):   DECISION reason;
                                     (RN_domain_not_hierarchical):
                                            TASK    result := RT_domain_not_hierarchical;
                                     (RN_parameters_unacceptable):
                                            TASK    result := RT_parameters_unacceptable;
                                    ELSE:
                                            TASK    result := RT_unspecified_failure;
                                    ENDDECISION;
                                    OUTPUT  MCS.Connect.Provider.confirm
                                            (portal(p)!label, result, 0, nullParms, NullString);
                        ENDDECISION;
        ELSE:ENDDECISION;
        TASK            portal(p)!notify := False,
                        dp := 0;
         1b :           /* for dp = 0..3 */
        DECISION dp < 4;
         (True):        DECISION portal(p)!proc(dp);
                        (Receiving, Responding, Engaged):
                                OUTPUT  Quit TO portal(p)!pids(dp);
                                TASK    portal(p)!proc(dp) := Quitting;
                        ELSE:ENDDECISION;
                        TASK    dp := dp + 1;
                        JOIN 1b;
```

```
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;
                                                                        /*---------*/
PROCEDURE       Quit;                                                   /*  Quit  */
                                                                        /*---------*/
        DCL     p               PortalId,
                pSet            PortalIdSet,
                dp              DataPriority,
                ds              DomainSelector;
        START
        COMMENT 'Process a Quit input signal.
                When all processes are quiesced, it is safe
                to shut this portal on the domain.
                If an upward portal, quiesce all others too.
                ';
        CALL    Identify_sender(p, dp);
        DECISION p in pUsed;
         (False):       RETURN;
        ELSE:ENDDECISION;
        TASK    portal(p)!proc(dp) := Quit;
        CALL    Quit_portal(p, RN_unspecified);
        TASK    dp := 0;
         1b :           /* for dp = 0..3 */
        DECISION dp < 4;
         (True):        DECISION portal(p)!proc(dp);
                        (Receiving, Responding, Engaged, Quitting):
                                RETURN;
                        ELSE:ENDDECISION;
                        TASK    dp := dp + 1;
                        JOIN 1b;
        ELSE:ENDDECISION;
        DECISION portal(p)!opened;
         (False):       CALL    Exit_portal(p);
                        RETURN;
        ELSE:ENDDECISION;
        TASK    ds := portal(p)!domain,
                domain(ds)!portals := domain(ds)!portals - 1;
        OUTPUT  Shut.portal(p) TO domain(ds)!pid;
        DECISION portal(p)!kind = Uplink;
         (True):        TASK domain(ds)!upward := 0,
                             pSet := pUsed;
              2b :      /* for p in pSet */
             DECISION pSet = Empty;
              (False):  TASK    p := Pick(pSet),
                                pSet := Del(p, pSet);
                        DECISION portal(p)!opened and portal(p)!domain = ds;
                         (True):    CALL    Quit_portal(p, RN_domain_disconnected);
                        ELSE:ENDDECISION;
                        JOIN 2b;
             ELSE:ENDDECISION;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;
                                                                        /*--------------------*/
PROCEDURE       Shut_portal;                                            /*  Shut_portal  */
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;
                              JOIN 1b;
                    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.
                  ';
        CALL      Initialize_resources;
        NEXTSTATE ~;

STATE ~;INPUT     MCS.Connect.Provider.request(label, localTSAP, localDomain,
                  remoteTSAP, remoteDomain, upward, targetParms, minParms, maxParms,
                  targetQOSByPri, minQOSByPri, userData);
        CALL      MCS_Connect_Provider_request(label, localTSAP, localDomain,
                  remoteTSAP, remoteDomain, upward, targetParms, minParms, maxParms,
                  targetQOSByPri, minQOSByPri, userData);
        NEXTSTATE -;

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

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

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

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

STATE ~;INPUT     Connect.Initial(remoteDomain, localDomain, upward,
                  targetParms, minParms, maxParms, userData);
        CALL      Connect_Initial(remoteDomain, localDomain, upward,
                  targetParms, minParms, maxParms, userData);
        NEXTSTATE -;

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

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

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

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

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

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

STATE ~;INPUT     RJum(pdu);
        CALL      RJum(pdu);
        NEXTSTATE -;
```

```
STATE ~;INPUT       Quit;
        CALL        Quit;
        NEXTSTATE -;

ENDPROCESS;
```

Appendix IV

SDL specification of the Domain process

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

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

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

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

          /* Type definitions */

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

NEWTYPE       Chan                  Array(ChannelId, ChannelStruct);
ENDNEWTYPE;

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

NEWTYPE       Token                 Array(TokenId, TokenStruct);
ENDNEWTYPE;

NEWTYPE       BooleanByPri          Array(DataPriority, Boolean);
ENDNEWTYPE;
NEWTYPE       NaturalByPri          Array(DataPriority, Natural);
ENDNEWTYPE;
NEWTYPE       BufferIdByPri         Array(DataPriority, BufferId);
ENDNEWTYPE;
NEWTYPE       BufferIdQueueByPri    Array(DataPriority, BufferIdQueue);
ENDNEWTYPE;
NEWTYPE       PortalIdSetByPri      Array(DataPriority, PortalIdSet);
ENDNEWTYPE;
```

```
NEWTYPE         NaturalByPriByKind          Array(PortalKind, NaturalByPri);
ENDNEWTYPE;

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

NEWTYPE     Portal                          Array(PortalId, PortalStruct);
ENDNEWTYPE;

NEWTYPE         PortalIdQueue                   Queue(PortalId);
ENDNEWTYPE;

SYNTYPE         BufferId                        = Integer CONSTANTS 0:maxBufferIds
ENDSYNTYPE;

NEWTYPE         BufferIdSet                     SetOf(BufferId);
ENDNEWTYPE;

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

NEWTYPE         Buffer                          Array(BufferId, BufferStruct);
ENDNEWTYPE;

NEWTYPE         BufferIdQueue                   Queue(BufferId);
ENDNEWTYPE;

GENERATOR       Queue       (TYPE ItemType)     /* a first-in first-out queue */
LITERALS
        EmptyQueue;
OPERATORS
        Push:   ItemType,  Queue    ->  Queue;          /* appends next item */
        Next:   Queue               ->  ItemType;       /* reveals first item */
        Pull:   Queue               ->  Queue;          /* deletes first item */
AXIOMS
        Next(EmptyQueue) == ERROR!;
        Pull(EmptyQueue) == ERROR!;
        FOR ALL q IN Queue (
        FOR ALL item IN ItemType
        (
                Next(Push(item,q)) == IF q = EmptyQueue THEN item
                                      ELSE Next(q) FI;
                Pull(Push(item,q)) == IF q = EmptyQueue THEN q
                                      ELSE Push(item,Pull(q)) FI;
        ));
DEFAULT
        EmptyQueue;
```

```
ENDGENERATOR;

        /* Data declarations */

DCL     control             PId;                /* the Control process */

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

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

DCL     uMerge              UserIdSet,          /* users still to be merged */
        uConfirm            UserIdSet,          /* users with merge confirmed */
        cMerge              ChannelIdSet,       /* channels still to be merged */
        cConfirm            ChannelIdSet,       /* channels with merge confirmed */
        tMerge              TokenIdSet,         /* tokens still to be merged */
        tConfirm            TokenIdSet,         /* tokens with merge confirmed */

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

DCL     pDrop               PortalIdSet,        /* dropped portals needing DPum */
        uDetach             UserIdSet,          /* disconnected users needing DUrq */
        uRevoke             UserIdSet,          /* revoked token users needing DUrq */
        cLeave              ChannelIdSet,       /* unjoined channels needing CLrq */
        cDisband            ChannelIdSet,       /* unmanaged channels needing CDin */
        tReject             TokenIdSet;         /* ungivable tokens needing TVcf */

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

DCL     chan                Chan,               /* resource arrays */
        token               Token,
        portal              Portal,
        buffer              Buffer;

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

DCL     cUsed               ChannelIdSet,       /* indexes used */
        tUsed               TokenIdSet,
        pUsed               PortalIdSet,
        bUsed               BufferIdSet;


DCL     cFree               ChannelIdSet,       /* indexes free */
        tFree               TokenIdSet,
        pFree               PortalIdSet,
        bFree               BufferIdSet;

DCL     uUsed               UserIdSet;          /* subset of cUsed */

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

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

DCL     inCredit            NaturalByPriByKind; /* initializer */
        /* Procedure decomposition */
```

```
/*      Initialize_resources
        Take_user           (c, diagnostic)
        Take_channel        (c, diagnostic)
        Take_token          (t, diagnostic)
        New_user            (u, result)
        New_channel         (c, result)
        Open_portal         (p, pKind, pids)
        Time_portal         (p)
        Drop_portal         (p, reason)
        Shut_portal         (p)
        Clean_queue         (p, queue)
        Erect_domain
        Identify_sender     (p, dp)
        PDU_ready               (dp)
        Input_PDU           (pdu)
        Allocate_buffer     (b)
        Cast_buffer         (b, p)
        Release_buffer      (b)
        Route_user          (u, p)
        Multicast_buffer    (b, uSet)
        Broadcast_buffer    (b)
        Crank_engine
        Drop_portals
        Merge_users
        Merge_channels
        Merge_tokens
        Detach_users
        Leave_channels
        Disband_channels
        Reject_tokens
        Process_PDU         (r, dp)
        Validate_input          (r, b, diagnostic)
        Top_provider        (r, b)
        Apply_PDU               (r, b, diagnostic)
        Token_status        (pdu)
        Token_route         (u, x, p)
        Delete_user         (u)
        Delete_channel      (c)
        Delete_token        (t)
        Purge_users         (uSet)
        Purge_channels      (cSet)
        Purge_tokens        (tSet)
        Output_buffer       (b, p)  */

                                                        /*--------------------------------*/
PROCEDURE   Initialize_resources;                       /*  Initialize_resources  */
                                                        /*--------------------------------*/
    DCL     c           ChannelId,
            t           TokenId,
            p           PortalId,
            b           BufferId,
            n           NaturalByPri;
    START
    COMMENT 'Initialize data structures during process start-up
            before accepting the first input signal.
            Note that each SetOf automatically defaults to Empty
            and each Queue to EmptyQueue.
            Fixed buffer credits are an example; other values could
            be computed from maxPortalIds and maxBufferIds.
            ';
    TASK    control := PARENT,
            upward := 0,
            merging := False,
            pdSend := False,
            edSend := False,
            numChannelIds := 0,
```

```
                numUserIds := 0,
                numTokenIds := 0,
                height := 1,
                interval := IF parameters!minThroughput = 0 THEN 0 ELSE oneSecond *
                    (Float(parameters!maxMCSPDUsize) / Float(parameters!minThroughput)) FI,
                maxInterval := 0,
                c := 65535,
                t := 65535,
                p := maxPortalIds,
                b := maxBufferIds;
 1b :           /* for c = ?..1 */
DECISION c > 0;
 (True):        TASK    cFree := Incl(c, cFree),
                        c := c - 1;
                JOIN 1b;
ELSE:ENDDECISION;
 2b :           /* for t = ?..1 */
TASK            tFree := Incl(t, tFree);
DECISION t > 1;
 (True):        TASK    t := t - 1;
                JOIN 2b;
ELSE:ENDDECISION;
 3b :           /* for p = ?..1 */
DECISION p > 0;
 (True):        TASK    pFree := Incl(p, pFree),
                        p := p - 1;
                JOIN 3b;
ELSE:ENDDECISION;
 4b :           /* for b = ?..1 */
DECISION b > 0;
 (True):        TASK    bFree := Incl(b, bFree),
                        b := b - 1;
                JOIN 4b;
ELSE:ENDDECISION;
TASK            n(0) := 2, n(1) := 1, n(2) := 1, n(3) := 1,
                inCredit(Attached) := n,
                n(0) := 3, n(1) := 2, n(2) := 1, n(3) := 1,
                inCredit(Downlink) := n,
                n(0) := 4, n(1) := 3, n(2) := 2, n(3) := 1,
                inCredit(Uplink) := n;
RETURN;
ENDPROCEDURE;

                                                                        /*------------------*/
PROCEDURE       Take_user;                                              /*  Take_user  */
FPAR            c               ChannelId,                              /*------------------*/
        IN/OUT  diagnostic      Diagnostic;

        DCL     u               UserId;
        START
        COMMENT 'Put the free channel id to use as a user id.
                ';
        DECISION numUserIds < parameters!maxUserIds;
         (False):       TASK    diagnostic := DC_too_many_users;
                        RETURN;
        ELSE:ENDDECISION;
        CALL            Take_channel(c, diagnostic);
        DECISION diagnostic = DC_OK;
         (True):        TASK    u := UserId(c),
                                numUserIds := numUserIds + 1,
                                uUsed := Incl(u, uUsed),
                                chan(c)!kind := UserId;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;
```

```
                                                                    /*-----------------------*/
PROCEDURE       Take_channel;                                       /*  Take_channel  */
FPAR            c               ChannelId,                          /*-----------------------*/
        IN/OUT  diagnostic      Diagnostic;

        START
        COMMENT 'Put the free channel id to unspecified use.
                ';
        DECISION c in cFree;
         (False):       TASK    diagnostic := DC_channel_id_conflict;
                        RETURN;
        ELSE:ENDDECISION;
        DECISION numChannelIds < parameters!maxChannelIds;
         (False):       TASK    diagnostic := DC_too_many_channels;
                        RETURN;
        ELSE:ENDDECISION;
        TASK            diagnostic := DC_OK,
                        numChannelIds := numChannelIds + 1,
                        cFree := Del(c, cFree),
                        cUsed := Incl(c, cUsed),
                        chan(c)!joined := Empty,
                        chan(c)!admitted := Empty;
        RETURN;
        ENDPROCEDURE;

                                                                    /*--------------------*/
PROCEDURE       Take_token;                                         /*  Take_token  */
FPAR            t               TokenId,                            /*--------------------*/
        IN/OUT  diagnostic      Diagnostic;

        START
        COMMENT 'Put the free token id to unspecified use.
                ';
        DECISION t in tFree;
         (False):       TASK    diagnostic := DC_token_id_conflict;
                        RETURN;
        ELSE:ENDDECISION;
        DECISION numTokenIds < parameters!maxTokenIds;
         (False):       TASK    diagnostic := DC_too_many_tokens;
                        RETURN;
        ELSE:ENDDECISION;
        TASK            diagnostic := DC_OK,
                        numTokenIds := numTokenIds + 1,
                        tFree := Del(t, tFree),
                        tUsed := Incl(t, tUsed),
                        token(t)!inhibitors := Empty;
        RETURN;
        ENDPROCEDURE;

                                                                    /*-------------------*/
PROCEDURE       New_user;                                           /*  New_user  */
FPAR    IN/OUT  u               UserId,                             /*-------------------*/
        IN/OUT  result          Result;

        DCL     c               ChannelId;
        START
        COMMENT 'Allocate a new user id or fail and return 0.
                ';
        TASK    u := 0;
        DECISION numUserIds < parameters!maxUserIds;
         (False):       TASK    result := RT_too_many_users;
                        RETURN;
        ELSE:ENDDECISION;
        CALL    New_channel(c, result);
        DECISION result = RT_successful;
         (True):        TASK    u := UserId(c),
                                numUserIds := numUserIds + 1,
```

```
                        uUsed := Incl(u, uUsed),
                        chan(c)!kind := UserId;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;

                                                                        /*----------------------*/
PROCEDURE       New_channel;                                            /*  New_channel  */
FPAR    IN/OUT  c               ChannelId,                              /*----------------------*/
        IN/OUT  result          Result;

        DCL     diagnostic      Diagnostic;
        START
        COMMENT 'Allocate a new channel id or fail and return 0.
                ';
        TASK    c := 0;
        DECISION numChannelIds < parameters!maxChannelIds;
         (False): TASK   result := RT_too_many_channels;
                RETURN;
        ELSE:ENDDECISION;
         1b :   /* keep trying */
        TASK    c := ANY(ChannelId);  /* randomize */
        DECISION c >= 1001 and c in cFree;
         (False): JOIN 1b;
        ELSE:ENDDECISION;
        CALL    Take_channel(c, diagnostic);
        TASK    result := RT_successful;
        RETURN;
        ENDPROCEDURE;

                                                                        /*----------------------*/
PROCEDURE       Open_portal;                                            /*  Open_portal  */
FPAR            p               PortalId,                               /*----------------------*/
                pKind           PortalKind,
                pids            PIdByPri;

        DCL     pid             PId,
                dp              DataPriority,
                c               ChannelId,
                cSet            ChannelIdSet,
                t               TokenId,
                tSet            TokenIdSet;
        START
        COMMENT 'Process an Open.portal input signal.
                Accept a new MCS connection or attachment to the domain.
                If this is an upward connection, prepare for merge.
                ';
        DECISION pKind = Attached;
         (True):  TASK  pid := pids(0),
                        pids(1) := pid,
                        pids(2) := pid,
                        pids(3) := pid;
        ELSE:ENDDECISION;
        TASK    pFree := Del(p, pFree),
                pUsed := Incl(p, pUsed),
                portal(p)!kind := pKind,
                portal(p)!pids := pids,
                portal(p)!inCredit := inCredit(pKind),
                portal(p)!outCount := 0,
                portal(p)!interval := IF pKind = Uplink THEN 0 ELSE interval FI,
                portal(p)!subHeight := 0,
                portal(p)!subInterval := 0,
                dp := 0;
         1b :   /* for dp = 0..? */
```

```
        DECISION dp < parameters!numPriorities;
         (True):        TASK    portal(p)!inBuffer(dp) := 0,
                                portal(p)!outReady(dp) := False,
                                portal(p)!outQueue(dp) := EmptyQueue,
                                pBufferWait(dp) := Incl(p, pBufferWait(dp)),
                                dp := dp + 1;
                        JOIN 1b;
        ELSE:ENDDECISION;
        DECISION portal(p)!kind = Uplink;
         (True):        TASK    upward := p,
                                merging := True,
                                pdSend := True,
                                edSend := True,
                                uMerge := uUsed,
                                uConfirm := Empty,
                                cMerge := cUsed,
                                cConfirm := Empty,
                                tMerge := tUsed,
                                tConfirm := Empty,
                                cLeave := Empty,
                                cDisband := Empty,
                                tReject := Empty,
                                cSet := cMerge;
                         2b :   /* for c in cSet */
                        DECISION cSet = Empty;
                         (False): TASK  c := Pick(cSet),
                                        cSet := Del(c, cSet),
                                        chan(c)!uMerge := chan(c)!admitted;
                                JOIN 2b;
                        ELSE:ENDDECISION;
                        TASK    tSet := tMerge;
                         3b :   /* for t in tSet */
                        DECISION tSet = Empty;
                         (False): TASK  t := Pick(tSet),
                                        tSet := Del(t, tSet),
                                        token(t)!uMerge := token(t)!inhibitors;
                                JOIN 3b;
                        ELSE:ENDDECISION;
        ELSE:ENDDECISION;
        CALL            Erect_domain;
        CALL            Crank_engine;
        RETURN;
        ENDPROCEDURE;


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


        START
        COMMENT 'Process a Time.portal input signal.
                 Ensure that minimum throughput is maintained.
                 Allow for the fact that outFlow takes integer steps.
                 ';
        DECISION portal(p)!elapsed > interval * (Float(portal(p)!outFlow) + 0.99);
         (True):        OUTPUT  Report.portal(p, DC_throughput_inadequate) TO control;
         (False):       TASK    portal(p)!outFlow := 0,
                                portal(p)!elapsed := portal(p)!interval;
                        SET(NOW + portal(p)!interval, Time.portal(p));
        ENDDECISION;
        CALL            Crank_engine;
        RETURN;
        ENDPROCEDURE;
```

```
                                                                    /*--------------------*/
PROCEDURE       Drop_portal;                                        /*  Drop_portal  */
FPAR            p           PortalId,                               /*--------------------*/
                reason      Reason;

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

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

        DCL         dp          DataPriority,
                    b           BufferId,
                    bSet        BufferIdSet,
                    c           ChannelId,
                    cSet        ChannelIdSet,
                    u           UserId;
        START
        COMMENT 'Process a Shut.portal input signal.
                Remove the corresponding MCS connection or attachment.
                When the last one is gone, stop the domain process.
                ';
        RESET(Time.portal(p));
        TASK        pUsed := Del(p, pUsed),
                    pFree := Incl(p, pFree),
                    pDrop := Del(p, pDrop),
                    dp := 0;
         1b :       /* for dp = 0..? */
        DECISION dp < parameters!numPriorities;
         (True):    2b :    /* for b in outQueue(dp) */
                    DECISION portal(p)!outQueue(dp) = EmptyQueue;
                     (False): TASK      b := Next(portal(p)!outQueue(dp)),
                                        portal(p)!outQueue(dp) := Pull(portal(p)!outQueue(dp)),
                                        buffer(b)!portals := buffer(b)!portals - 1;
                              CALL      Release_buffer(b);
                              JOIN 2b;
                    ELSE:ENDDECISION;
                    TASK    b := portal(p)!inBuffer(dp),
                            portal(p)!inBuffer(dp) := 0;
                    CALL    Release_buffer(b);
                    TASK    pBufferWait(dp) := Del(p, pBufferWait(dp)),
                            pInputWait(dp) := Del(p, pInputWait(dp)),
                            dp := dp + 1;
                    JOIN 1b;
        ELSE:ENDDECISION;
        TASK        bSet := bUsed;
         3b :       /* for b in bSet */
        DECISION bSet = Empty;
         (False):   TASK    b := Pick(bSet),
                            bSet := Del(b, bSet);
                    DECISION buffer(b)!receiver = p;
                     (True):    TASK    buffer(b)!receiver := 0;
                    ELSE:ENDDECISION;
                    JOIN 3b;
        ELSE:ENDDECISION;
        TASK        cSet := cUsed;
         4b :       /* for c in cSet */
```

```
        DECISION cSet = Empty;
        (False):    TASK    c := Pick(cSet),
                            cSet := Del(c, cSet);
                    DECISION p in chan(c)!joined;
                    (True):  TASK    chan(c)!joined := Del(p, chan(c)!joined);
                             DECISION chan(c)!joined = Empty;
                             (True):  TASK    cLeave := Incl(c, cLeave);
                             ELSE:ENDDECISION;
                    ELSE:ENDDECISION;
                    DECISION chan(c)!kind = UserId and chan(c)!portal = p;
                    (True):  TASK    chan(c)!portal := 0,
                                     u := UserId(c),
                                     uDetach := Incl(u, uDetach),
                                     cLeave := Del(c, cLeave);
                    ELSE:ENDDECISION;
                    JOIN 4b;
        ELSE:ENDDECISION;
        DECISION portal(p)!kind = Uplink;
        (True):     TASK    upward := 0,
                            merging := False,
                            mcrqQueue := EmptyQueue,
                            mtrqQueue := EmptyQueue,
                            aurqQueue := EmptyQueue;
        (False):    CALL    Clean_queue(p, mcrqQueue);
                    CALL    Clean_queue(p, mtrqQueue);
                    CALL    Clean_queue(p, aurqQueue);
        ENDDECISION;
        OUTPUT      Shut.portal(p) TO control;
        CALL        Erect_domain;
        CALL        Crank_engine;
        RETURN;
        ENDPROCEDURE;

                                                                    /*----------------------*/
PROCEDURE           Clean_queue;                                    /*  Clean_queue  */
FPAR                p           PortalId,                           /*----------------------*/
        IN/OUT      queue       PortalIdQueue;

        DCL         x           PortalId,
                    clean       PortalIdQueue;
        START
        COMMENT     'Remove a shut portal id from a queue.
                    ';
        TASK        clean := EmptyQueue;
        1b :        /* for x in queue */
        DECISION queue = EmptyQueue;
        (False):    TASK    x := Next(queue),
                            queue := Pull(queue),
                            x := IF x = p THEN 0 ELSE x FI,
                            clean := Push(x, clean);
                    JOIN 1b;
        ELSE:ENDDECISION;
        TASK        queue := clean;
        RETURN;
        ENDPROCEDURE;

                                                                    /*----------------------*/
PROCEDURE           Erect_domain;                                   /*  Erect_domain  */
                                                                    /*----------------------*/

        DCL         p           PortalId,
                    pSet        PortalIdSet,
                    h           Natural,
                    hmax        Natural,
                    i           Duration,
                    imax        Duration;
```

```
        START
        COMMENT   'Recalculate the provider height and maxInterval.
                  If either changed, communicate them upward.
                  ';
        TASK      hmax := 1,
                  imax := 0,
                  pSet := pUsed;
         1b :     /* for p in pSet */
        DECISION pSet = Empty;
         (False): TASK     p := Pick(pSet),
                           pSet := Del(p, pSet),
                           h := portal(p)!subHeight + 1,
                           hmax := IF h > hmax THEN h ELSE hmax FI,
                           i := portal(p)!interval,
                           imax := IF i > imax THEN i ELSE imax FI;
                  JOIN 1b;
        ELSE:ENDDECISION;
        DECISION height = hmax and maxInterval = imax;
         (False): TASK     height := hmax,
                           maxInterval := imax,
                           edSend := True;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;


                                                          /*-------------------------*/
PROCEDURE         Identify_sender;                        /*  Identify_sender  */
FPAR    IN/OUT    p              PortalId,                /*-------------------------*/
        IN/OUT    dp             DataPriority;


        DCL       pSet           PortalIdSet;
        START
        COMMENT   'An alternative would be to carry this
                  information explicitly in SDL signals.
                  ';
        TASK      pSet := pUsed;
         1b :     /* for p in pSet */
        DECISION pSet = Empty;
         (False): TASK     p := Pick(pSet),
                           pSet := Del(p, pSet),
                           dp := 0;
                  2b :     /* for dp = 0..? */
                  DECISION dp < parameters!numPriorities;
                   (True): DECISION portal(p)!pids(dp) = SENDER;
                            (True):   RETURN;
                           ELSE:ENDDECISION;
                           TASK      dp := dp + 1;
                           JOIN 2b;
                  ELSE:ENDDECISION;
                  JOIN 1b;
        ELSE:ENDDECISION;
        TASK      p := 0,
                  dp := 0;
        RETURN;
        ENDPROCEDURE;


                                                          /*--------------------*/
PROCEDURE         PDU_ready;                              /*  PDU_ready  */
FPAR              dp             DataPriority;            /*--------------------*/
        DCL       p              PortalId,
                  x              DataPriority,
                  b              BufferId;
```

```
        START
        COMMENT   'Process a PDU.ready input signal.
                  If a buffer is waiting, it can be output.
                  ';
        CALL      Identify_sender(p, x);
        DECISION p in pUsed;
         (False):    RETURN;
        ELSE:ENDDECISION;
        DECISION portal(p)!outQueue(dp) = EmptyQueue;
         (True):     TASK    portal(p)!outReady(dp) := True;
         (False):    TASK    b := Next(portal(p)!outQueue(dp)),
                             portal(p)!outQueue(dp) := Pull(portal(p)!outQueue(dp)),
                             buffer(b)!portals := buffer(b)!portals - 1;
                     CALL    Output_buffer(b, p);
                     TASK    portal(p)!outReady(dp) := False;
                     CALL    Release_buffer(b);
                     TASK    portal(p)!outFlow := portal(p)!outFlow + 1,
                             portal(p)!outCount := portal(p)!outCount - 1;
                     DECISION portal(p)!outCount = 0;
                      (True):  RESET(Time.portal(p));
                     ELSE:ENDDECISION;
        ENDDECISION;
        CALL      Crank_engine;
        RETURN;
        ENDPROCEDURE;

                                                                  /*------------------*/
PROCEDURE         Input_PDU;                                      /*  Input_PDU  */
FPAR              pdu           PDUStruct;                        /*------------------*/

        DCL       p             PortalId,
                  dp            DataPriority,
                  b             BufferId;
        START
        COMMENT   'Process an MCSPDU input signal.
                  If merging, requests and responses must wait.
                  ';
        CALL      Identify_sender(p, dp);
        DECISION p in pUsed;
         (False):    RETURN;
        ELSE:ENDDECISION;
        DECISION portal(p)!kind = Attached;
         (True):     DECISION pdu!kind;
                      (SDrq, SDin, USrq, USin):
                            TASK    dp := pdu!dataPriority,
                                    dp := IF dp < parameters!numPriorities THEN dp
                                                  ELSE parameters!numPriorities - 1 FI;
                     ELSE:ENDDECISION;
        ELSE:ENDDECISION;
        TASK      b := portal(p)!inBuffer(dp),
                  buffer(b)!pdu := pdu;
        DECISION p = upward or not merging;
         (True):     CALL    Process_PDU(p, dp);
         (False):    TASK    pInputWait(dp) := Incl(p, pInputWait(dp));
        ENDDECISION;
        CALL      Crank_engine;
        RETURN;
        ENDPROCEDURE;

                                                                  /*-------------------------*/
PROCEDURE         Allocate_buffer;                                /*  Allocate_buffer  */
FPAR    IN/OUT    b             BufferId;                         /*-------------------------*/

        START
        COMMENT   'Allocate a free buffer id or fail and return 0.
                  ';
```

```
        DECISION b /= 0 and buffer(b)!portals = 0;
         (True):        RETURN;
        ELSE:ENDDECISION;
        TASK            b := 0;
        DECISION bFree = Empty;
         (False):       TASK    b := Pick(bFree),
                                bFree := Del(b, bFree),
                                bUsed := Incl(b, bUsed),
                                buffer(b)!receiver := 0,
                                buffer(b)!dataPriority := 0,
                                buffer(b)!portals := 0;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;

                                                                        /*--------------------*/
PROCEDURE       Cast_buffer;                                            /*  Cast_buffer  */
FPAR            b               BufferId,                               /*--------------------*/
                p               PortalId;

        DCL     dp              DataPriority;
        START
        COMMENT 'Send buffer containing an MCSPDU to a single output.
                ';
        DECISION p = 0;
         (True):        RETURN;
        ELSE:ENDDECISION;
         TASK           dp := buffer(b)!dataPriority;
        DECISION portal(p)!outReady(dp);
         (True):        CALL    Output_buffer(b, p);
                        TASK    portal(p)!outReady(dp) := False;
         (False):       DECISION portal(p)!outCount = 0 and portal(p)!interval > 0;
                         (True):  TASK portal(p)!outFlow := 0,
                                        portal(p)!elapsed := portal(p)!interval;
                                  SET(NOW + portal(p)!interval, Time.portal(p));
                        ELSE:ENDDECISION;
                        TASK    portal(p)!outQueue(dp) := Push(b, portal(p)!outQueue(dp)),
                                buffer(b)!portals := buffer(b)!portals + 1,
                                portal(p)!outCount := portal(p)!outCount + 1;
        ENDDECISION;
        RETURN;
        ENDPROCEDURE;

                                                                        /*--------------------------*/
PROCEDURE       Release_buffer;                                         /*  Release_buffer  */
FPAR            b               BufferId;                               /*--------------------------*/

        DCL     p               PortalId,
                dp              DataPriority;
        START
        COMMENT 'Release a buffer that is no longer needed.
                ';
        DECISION b = 0 or buffer(b)!portals > 0;
         (True):        RETURN;
        ELSE:ENDDECISION;
        TASK            bUsed := Del(b, bUsed),
                        bFree := Incl(b, bFree),
                        p := buffer(b)!receiver,
                        dp := buffer(b)!dataPriority;
        DECISION p = 0;
         (False):       TASK    portal(p)!inCredit(dp) := portal(p)!inCredit(dp) + 1;
                        DECISION portal(p)!inBuffer(dp) = 0;
                         (True):  TASK pBufferWait(dp) := Incl(p, pBufferWait(dp));
                        ELSE:ENDDECISION;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;
```

```
                                                                    /*--------------------*/
PROCEDURE           Route_user;                                     /*  Route_user  */
FPAR                u               UserId,                         /*--------------------*/
        IN/OUT      p               PortalId;

        DCL         c               ChannelId;
        START
        COMMENT     'Return the portal id that leads toward a user.
                    ';
        TASK        p := 0;
        DECISION u in uUsed;
         (True):    TASK    c := ChannelId(u),
                            p := chan(c)!portal;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;

                                                                    /*--------------------------*/
PROCEDURE           Multicast_buffer;                               /*  Multicast_buffer  */
FPAR                b               BufferId,                       /*--------------------------*/
                    uSet            UserIdSet;

        DCL         u               UserId,
                    p               PortalId,
                    pSet            PortalIdSet;
        START
        COMMENT     'Send buffer containing an MCSPDU to multiple users.
                    ';
        TASK        pSet := Empty;
         1b :       /* for u in uSet */
        DECISION uSet = Empty;
         (False):   TASK    u := Pick(uSet),
                            uSet := Del(u, uSet);
                    CALL    Route_user(u, p);
                    TASK    pSet := Incl(p, pSet);
                    JOIN 1b;
        ELSE:ENDDECISION;
         2b :       /* for p in pSet */
        DECISION pSet = Empty;
         (False):   TASK    p := Pick(pSet),
                            pSet := Del(p, pSet);
                    CALL    Cast_buffer(b, p);
                    JOIN 2b;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;

                                                                    /*----------------------------*/
PROCEDURE           Broadcast_buffer;                               /*  Broadcast_buffer  */
FPAR                b               BufferId;                       /*----------------------------*/

        DCL         pdu             PDUStruct,
                    p               PortalId,
                    pSet            PortalIdSet;
        START
        COMMENT     'Send buffer containing an MCSPDU to the whole subtree.
                    ';
        TASK        pdu := buffer(b)!pdu;
        DECISION (pdu!kind = PCin and pdu!detachUserIds = Empty
                                and pdu!purgeChannelIds = Empty)
                    or (pdu!kind = PTin and pdu!purgeTokenIds = Empty)
                    or (pdu!kind = DUin and pdu!userIds = Empty);
         (True):    RETURN;
        ELSE:ENDDECISION;
        TASK        pSet := pUsed;
         1b :       /* for p in pSet */
```

```
        DECISION pSet = Empty;
         (False):       TASK      p := Pick(pSet),
                                  pSet := Del(p, pSet);
                        DECISION portal(p)!kind;
                         (Attached):
                                  DECISION pdu!kind = PDin;
                                   (False):CALL          Cast_buffer(b, p);
                                  ELSE:ENDDECISION;
                         (Downlink):
                                  CALL        Cast_buffer(b, p);
                        ELSE:ENDDECISION;
                        JOIN 1b;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;

                                                                          /*------------------------*/
PROCEDURE               Crank_engine;                                     /*  Crank_engine  */
                                                                          /*------------------------*/
        DCL             b               BufferId,
                        p               PortalId,
                        dp              DataPriority;
        START
        COMMENT         'After individual processing of each input signal,
                        look for the most deserving work to do next.
                        This advances incrementally through stages of a merge;
                        else it cleans up users, channels, tokens left behind.
                        Buffers are assigned to accept new inputs, with preference
                        to PDUs flowing downward and to higher priorities first.
                        ';
        TASK            b := 0;
        DECISION pdSend;
         (True):        CALL      Allocate_buffer(b);
                        DECISION b = 0;
                         (True):    RETURN;
                        ELSE:ENDDECISION;
                        TASK      pdSend := False,
                                  buffer(b)!pdu!kind := PDin,
                                  buffer(b)!pdu!heightLimit := parameters!maxHeight - 1;
                        CALL      Broadcast_buffer(b);
        ELSE:ENDDECISION;
        DECISION edSend;
         (True):        CALL      Allocate_buffer(b);
                        DECISION b = 0;
                         (True):    RETURN;
                        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,
                        portal(p)!inBuffer(dp) := b;
                OUTPUT  PDU.ready(dp) TO portal(p)!pids(dp);
                JOIN 3b;
ELSE:ENDDECISION;
RETURN;
ENDPROCEDURE;
```

```
                                                                    /*----------------------*/
PROCEDURE       Drop_portals;                                       /*  Drop_portals  */
                                                                    /*----------------------*/
        DCL     b               BufferId,
                p               PortalId,
                pdu             PDUStruct;
        START
        COMMENT 'Generate DPum MCSPDUs requested by controller.
                ';
        TASK    b := 0,
                pdu!kind := DPum,
                pdu!reason := RN_user_requested;
         1b :   /* for p in pDrop */
        DECISION pDrop = Empty;
         (False): CALL  Allocate_buffer(b);
                DECISION b = 0;
                 (True): RETURN;
                ELSE:ENDDECISION;
                TASK    p := Pick(pDrop),
                        pDrop := Del(p, pDrop),
                        buffer(b)!pdu := pdu;
                CALL    Cast_buffer(b, p);
                JOIN 1b;
        ELSE:ENDDECISION;
        CALL    Release_buffer(b);
        RETURN;
        ENDPROCEDURE;

                                                                    /*----------------------*/
PROCEDURE       Merge_users;                                        /*  Merge_users  */
                                                                    /*----------------------*/
        DCL     b               BufferId,
                u               UserId,
                c               ChannelId,
                pdu             PDUStruct,
                cAttr           ChannelAttributes;
        START
        COMMENT 'Generate MCrq MCSPDUs to merge user ids upward.
                Fill them to maximum size that encoding allows.
                ';
        TASK    b := 0,
                pdu!kind := MCrq,
                pdu!mergeChannels := Empty;
         1b :   /* for u in uMerge */
        DECISION uMerge = Empty;
         (False): CALL  Allocate_buffer(b);
                DECISION b = 0;
                 (True): RETURN;
                ELSE:ENDDECISION;
                TASK    u := Pick(uMerge),
                        uMerge := Del(u, uMerge),
                        c := ChannelId(u),
                        cMerge := Del(c, cMerge),
                        cAttr!channelId := c,
                        cAttr!kind := UserId,
                        cAttr!joined := (chan(c)!joined /= Empty),
                        pdu!mergeChannels := Incl(cAttr, pdu!mergeChannels);
                DECISION'encoding of pdu + 9 <= maxMCSPDUsize';
                 ('True'):  JOIN 1b;
                ELSE:ENDDECISION;
        ELSE:ENDDECISION;
        DECISION pdu!mergeChannels = Empty;
         (False): TASK  buffer(b)!pdu := pdu,
                        pdu!mergeChannels := Empty;
                CALL    Cast_buffer(b, upward);
                JOIN 1b;
```

```
        ELSE:ENDDECISION;
        CALL        Release_buffer(b);
        RETURN;
        ENDPROCEDURE;


                                                                    /*----------------------------*/
PROCEDURE           Merge_channels;                                 /*  Merge_channels  */
                                                                    /*----------------------------*/
        DCL         b               BufferId,
                    c               ChannelId,
                    u               UserId,
                    pdu             PDUStruct,
                    cAttr           ChannelAttributes;
        START
        COMMENT     'Generate MCrq MCSPDUs to merge channel ids upward.
                    Fill them to maximum size that encoding allows.
                    ';
        TASK        b := 0,
                    pdu!kind := MCrq,
                    pdu!mergeChannels := Empty,
                    pdu!purgeChannelIds := Empty;
         1b :       /* for c in cMerge */
        DECISION cMerge = Empty;
         (False):   CALL     Allocate_buffer(b);
                    DECISION b = 0;
                     (True):   RETURN;
                    ELSE:ENDDECISION;
                    TASK     c := Pick(cMerge),
                             cMerge := Del(c, cMerge),
                             cAttr!channelId := c,
                             cAttr!kind := chan(c)!kind,
                             cAttr!manager := chan(c)!manager,
                             cAttr!admitted := Empty,
                             cAttr!joined := (chan(c)!joined /= Empty);
                    DECISION (chan(c)!kind = Static or chan(c)!kind = Assigned)
                             and chan(c)!joined = Empty;
                     (True):  CALL       Delete_channel(c);
                             JOIN 1b;
                    ELSE:ENDDECISION;
                    DECISION chan(c)!kind = Private;
                     (True):  DECISION chan(c)!manager in uUsed;
                               (False):    TASK    pdu!purgeChannelIds :=
                                                       Incl(c, pdu!purgeChannelIds);
                                           JOIN 3f;
                              ELSE:ENDDECISION;
                               2b :        /* for u in chan(c)!uMerge */
                              DECISION chan(c)!uMerge = Empty;
                               (False):    TASK    u := Pick(chan(c)!uMerge),
                                                   chan(c)!uMerge := Del(u, chan(c)!uMerge),
                                                   cAttr!admitted := Incl(u, cAttr!admitted),
                                                   pdu!mergeChannels :=          /* try this many */
                                                       Incl(cAttr, pdu!mergeChannels);
                                           DECISION'encoding of pdu + 4 <= maxMCSPDUsize';
                                            ('True'): TASK     pdu!mergeChannels := /* try another */
                                                       Del(cAttr, pdu!mergeChannels);
                                                   JOIN 2b;
                                           ELSE:ENDDECISION;
                              ELSE:ENDDECISION;
                    ELSE:ENDDECISION;
                    TASK     pdu!mergeChannels := Incl(cAttr, pdu!mergeChannels);
                     3f :
                    DECISION'encoding of pdu + 23 <= maxMCSPDUsize';
                     ('True'):  JOIN 1b;
                    ELSE:ENDDECISION;
        ELSE:ENDDECISION;
```

```
        DECISION pdu!mergeChannels = Empty and pdu!purgeChannelIds = Empty;
         (False):    TASK    buffer(b)!pdu := pdu,
                             pdu!mergeChannels := Empty,
                             pdu!purgeChannelIds := Empty;
                     CALL    Cast_buffer(b, upward);
                     JOIN 1b;
        ELSE:ENDDECISION;
        CALL         Release_buffer(b);
        RETURN;
        ENDPROCEDURE;


                                                                    /*-------------------------*/
PROCEDURE            Merge_tokens;                                  /*  Merge_tokens  */
                                                                    /*-------------------------*/
        DCL          b              BufferId,
                     t              TokenId,
                     u              UserId,
                     pdu            PDUStruct,
                     tAttr          TokenAttributes;
        START
        COMMENT      'Generate MTrq MCSPDUs to merge token ids upward.
                     Fill them to maximum size that encoding allows.
                     ';
        TASK         b := 0,
                     pdu!kind := MTrq,
                     pdu!mergeTokens := Empty,
                     pdu!purgeTokenIds := Empty,
         1b :        /* for t in tMerge */
        DECISION tMerge = Empty;
         (False):    CALL    Allocate_buffer(b);
                     DECISION b = 0;
                      (True):  RETURN;
                     ELSE:ENDDECISION;
                     TASK    t := Pick(tMerge),
                             tMerge := Del(t, tMerge),
                             tAttr!tokenId := t,
                             tAttr!kind := token(t)!kind,
                             tAttr!grabber := token(t)!grabber,
                             tAttr!recipient := token(t)!recipient,
                             tAttr!inhibitors := Empty;
                     DECISION token(t)!kind = Inhibited;
                      (True):  2b :        /* for u in token(t)!uMerge */
                               DECISION token(t)!uMerge = Empty;
                                (False):   TASK    u := Pick(token(t)!uMerge),
                                                   token(t)!uMerge := Del(u, token(t)!uMerge),
                                                   tAttr!inhibitors := Incl(u, tAttr!inhibitors),
                                                   pdu!mergeTokens :=          /* try this many */
                                                                       Incl(tAttr, pdu!mergeTokens);
                                           DECISION'encoding of pdu + 4 <= maxMCSPDUsize';
                                            ('True'): TASK     pdu!mergeTokens := /* try another */
                                                                       Del(tAttr, pdu!mergeTokens);
                                                      JOIN 2b;
                                           ELSE:ENDDECISION;
                               ELSE:ENDDECISION;
                     ELSE:ENDDECISION;
                     TASK    pdu!mergeTokens := Incl(tAttr, pdu!mergeTokens);
                     DECISION'encoding of pdu + 16 <= maxMCSPDUsize';
                      ('True'):  JOIN 1b;
        ELSE:ENDDECISION;
        ELSE:ENDDECISION;
        DECISION pdu!mergeTokens = Empty;
         (False):    TASK    buffer(b)!pdu := pdu,
                             pdu!mergeTokens := Empty;
                     CALL    Cast_buffer(b, upward);
                     JOIN 1b;
```

```
        ELSE:ENDDECISION;
        CALL        Release_buffer(b);
        RETURN;
        ENDPROCEDURE;

                                                                    /*-----------------------*/
PROCEDURE           Detach_users;                                   /*  Detach_users  */
                                                                    /*-----------------------*/
        DCL         b               BufferId,
                    u               UserId,
                    pdu             PDUStruct,
                    diagnostic      Diagnostic;
        START
        COMMENT     'Generate DUrq MCSPDUs to detach users.
                    Fill them to maximum size that encoding allows.
                    ';
        TASK        b := 0,
                    pdu!kind := DUrq,
                    pdu!userIds := Empty;
         1b :       /* for u in uDetach */
        DECISION uDetach = Empty;
         (False):   CALL     Allocate_buffer(b);
                    DECISION b = 0;
                     (True):  RETURN;
                    ELSE:ENDDECISION;
                    TASK     u := Pick(uDetach),
                             uDetach := Del(u, uDetach),
                             uRevoke := Del(u, uRevoke),
                             pdu!reason := RN_domain_disconnected,
                             pdu!userIds := Incl(u, pdu!userIds);
                    DECISION'encoding of pdu + 4 <= maxMCSPDUsize';
                     ('True'):  JOIN 1b;
                    ELSE:ENDDECISION;
        ELSE:ENDDECISION;
        DECISION pdu!userIds = Empty;
         (False):   JOIN 3f;
        ELSE:ENDDECISION;
         2b :       /* for u in uRevoke */
        DECISION uRevoke = Empty;
         (False):   CALL     Allocate_buffer(b);
                    DECISION b = 0;
                     (True):  RETURN;
                    ELSE:ENDDECISION;
                    TASK     u := Pick(uRevoke),
                             uRevoke := Del(u, uRevoke),
                             pdu!reason := RN_channel_purged,
                             pdu!userIds := Incl(u, pdu!userIds);
                    DECISION'encoding of pdu + 4 <= maxMCSPDUsize';
                     ('True'):  JOIN 1b;
                    ELSE:ENDDECISION;
        ELSE:ENDDECISION;
         3f :
        DECISION pdu!userIds = Empty;
         (False):   TASK     buffer(b)!pdu := pdu,
                             pdu!userIds := Empty;
                    DECISION upward = 0;
                     (False):  CALL     Cast_buffer(b, upward);
                     (True):   TASK     buffer(b)!pdu!kind := DUin;
                               CALL     Apply_PDU(0, b, diagnostic);
                    ENDDECISION;
                    JOIN 1b;
        ELSE:ENDDECISION;
        CALL        Release_buffer(b);
        RETURN;
        ENDPROCEDURE;
```

```
                                                                    /*-------------------------*/
PROCEDURE          Leave_channels;                                  /*  Leave_channels  */
                                                                    /*-------------------------*/
        DCL        b                BufferId,
                   c                ChannelId,
                   pdu              PDUStruct;
        START
        COMMENT    'Generate CLrq MCSPDUs to leave channels.
                   Fill them to maximum size that encoding allows.
                   ';
        TASK       b := 0,
                   pdu!kind := CLrq,
                   pdu!channelIds := Empty;
         1b :      /* for c in cLeave */
        DECISION cLeave = Empty;
         (False):  CALL     Allocate_buffer(b);
                   DECISION b = 0;
                    (True):  RETURN;
                   ELSE:ENDDECISION;
                   TASK     c := Pick(cLeave),
                            cLeave := Del(c, cLeave);
                   DECISION chan(c)!kind;
                    (Static, Assigned):
                            CALL       Delete_channel(c);
                   ELSE:ENDDECISION;
                   TASK     pdu!channelIds := Incl(c, pdu!channelIds);
                   DECISION'encoding of pdu + 4 <= maxMCSPDUsize';
                    ('True'):  JOIN 1b;
                   ELSE:ENDDECISION;
        ELSE:ENDDECISION;
        DECISION pdu!channelIds = Empty;
         (False):  TASK     buffer(b)!pdu := pdu,
                            pdu!channelIds := Empty;
                   CALL     Cast_buffer(b, upward);
                   JOIN 1b;
        ELSE:ENDDECISION;
        CALL       Release_buffer(b);
        RETURN;
        ENDPROCEDURE;
```

```
                                                                    /*-----------------------------*/
PROCEDURE          Disband_channels;                                /*  Disband_channels  */
                                                                    /*-----------------------------*/
        DCL        b                BufferId,
                   c                ChannelId,
                   pdu              PDUStruct,
                   diagnostic       Diagnostic;
        START
        COMMENT    'Generate CDrq MCSPDUs to disband channels.
                   ';
        TASK       b := 0,
                   pdu!kind := CDrq;
         1b :      /* for c in cDisband */
        DECISION cDisband = Empty;
         (False):  CALL     Allocate_buffer(b);
                   DECISION b = 0;
                    (True):  RETURN;
                   ELSE:ENDDECISION;
                   TASK     c := Pick(cDisband),
                            cDisband := Del(c, cDisband),
                            pdu!channelId := c,
                            buffer(b)!pdu := pdu;
```

```
                DECISION upward = 0;
                 (False):  CALL      Cast_buffer(b, upward);
                 (True):   TASK      buffer(b)!pdu!kind := CDin;
                           CALL      Apply_PDU(0, b, diagnostic);
                ENDDECISION;
                JOIN 1b;
        ELSE:ENDDECISION;
        CALL    Release_buffer(b);
        RETURN;
        ENDPROCEDURE;


                                                                  /*------------------------*/
PROCEDURE       Reject_tokens;                                    /*  Reject_tokens  */
                                                                  /*------------------------*/
        DCL     b               BufferId,
                t               TokenId,
                u               UserId,
                p               PortalId,
                pdu             PDUStruct;
        START
        COMMENT 'Generate TVcf MCSPDUs to reject tokens.
                ';
        TASK    b := 0,
                pdu!kind := TVcf,
                pdu!result := RT_no_such_user;
         1b :   /* for t in tReject */
        DECISION tReject = Empty;
         (False):  CALL     Allocate_buffer(b);
                DECISION b = 0;
                 (True):   RETURN;
                ELSE:ENDDECISION;
                TASK    t := Pick(tReject),
                        tReject := Del(t, tReject),
                        token(t)!kind := Grabbed,
                        u := token(t)!grabber,
                        pdu!initiator := u,
                        pdu!tokenId := t;
                CALL    Token_status(pdu);
                TASK    buffer(b)!pdu := pdu;
                CALL    Route_user(u, p);
                CALL    Cast_buffer(b, p);
                JOIN 1b;
        ELSE:ENDDECISION;
        CALL    Release_buffer(b);
        RETURN;
        ENDPROCEDURE;


                                                                  /*------------------------*/
PROCEDURE       Process_PDU;                                      /*  Process_PDU  */
FPAR            r               PortalId,                         /*------------------------*/
                dp              DataPriority;


        DCL     b               BufferId,
                pdu             PDUStruct,
                diagnostic      Diagnostic;
        START
        COMMENT 'This is the main line of processing an MCSPDU.
                If its content is invalid, ignore or reject it.
                Else if this is the top provider, take the special
                actions of Top_provider.  Then whether or not this
                is the top provider, call Apply_PDU.
                ';
        TASK    b := portal(r)!inBuffer(dp),
                pdu := buffer(b)!pdu;
```

```
        DECISION pdu!kind = RJum;
         (True):        TASK      diagnostic := pdu!diagnostic;
         (False):       CALL      Validate_input(r, b, diagnostic);
                        DECISION diagnostic = DC_OK;
                         (True):   DECISION buffer(b)!pdu!kind;
                                    (MCrq):    TASK     mcrqQueue := Push(r, mcrqQueue);
                                    (MTrq):    TASK     mtrqQueue := Push(r, mtrqQueue);
                                    (AUrq):    TASK     aurqQueue := Push(r, aurqQueue);
                                   ELSE:ENDDECISION;
                                   DECISION upward = 0;
                                    (True):    CALL     Top_provider(r, b);
                                   ELSE:ENDDECISION;
                                   CALL       Apply_PDU(r, b, diagnostic);
                        ELSE:ENDDECISION;
        ENDDECISION;
        DECISION diagnostic;
         (DC_OK, DC_ignore):
                        /* no action */
        ELSE:
                        OUTPUT  Report.portal(r, diagnostic) TO control;
                        DECISION pdu!kind = RJum;
                         (False):  TASK'pdu!initialOctets := truncate pdu';
                        ELSE:ENDDECISION;
                        TASK      pdu!kind := RJum,
                                  pdu!diagnostic := diagnostic,
                                  buffer(b)!pdu := pdu,
                                  dp := buffer(b)!dataPriority,
                                  buffer(b)!dataPriority := 0;
                        CALL      Cast_buffer(b, r);
                        TASK      buffer(b)!dataPriority := dp;
        ENDDECISION;
        CALL            Release_buffer(b);
        TASK            portal(r)!inBuffer(dp) := 0;
        DECISION        portal(r)!inCredit(dp) > 0;
         (True):        TASK      pBufferWait(dp) := Incl(r, pBufferWait(dp));
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;

                                                                        /*-----------------------*/
PROCEDURE       Validate_input;                                         /*  Validate_input  */
FPAR            r               PortalId,                               /*-----------------------*/
                b               BufferId,
        IN/OUT  diagnostic      Diagnostic;

        DCL     pdu             PDUStruct,
                p               PortalId,
                dp              DataPriority,
                u               UserId,
                uSet            UserIdSet,
                c               ChannelId,
                cSet            ChannelIdSet,
                cAttr           ChannelAttributes,
                cAttrSet        ChannelAttributesSet,
                t               TokenId,
                tSet            TokenIdSet,
                tAttr           TokenAttributes,
                tAttrSet        TokenAttributesSet;
        START
        COMMENT 'Validate the direction and user ids of an MCSPDU
                and perform other checks, depending on its type.
                Top_provider and Apply_PDU trust this procedure
                to catch important anomalies and ease their logic.
                ';
        TASK    pdu := buffer(b)!pdu,
                diagnostic := DC_OK;
```

```
DECISION     portal(r)!kind;
 (Downlink):
                DECISION pdu!kind;
                 (EDrq, MCrq, MTrq, DPum, RJum,
                  AUrq, DUrq, CJrq, CLrq, CCrq,
                  CDrq, CArq, CErq, SDrq, USrq,
                  TGrq, TIrq, TVrq, TVrs, TPrq,
                  TRrq, TTrq):
                        /* OK */
                ELSE:
                        TASK        diagnostic := DC_forbidden_PDU_upward;
                        RETURN;
                ENDDECISION;
 (Uplink):
                DECISION pdu!kind;
                 (PDin, MCcf, PCin, MTcf, PTin,
                  DPum, RJum, AUcf, DUin, CJcf,
                  CCcf, CDin, CAin, CEin, SDin,
                  USin, TGcf, TIcf, TVin, TVcf,
                  TPin, TRcf, TTcf):
                        /* OK */
                ELSE:
                        TASK        diagnostic := DC_forbidden_PDU_downward;
                        RETURN;
                ENDDECISION;
ELSE:ENDDECISION;
TASK            dp := 0;
DECISION pdu!kind;
 (SDrq, SDin, USrq, USin):
                TASK        dp := pdu!dataPriority,
                            dp := IF dp < parameters!numPriorities THEN dp
                                                ELSE parameters!numPriorities - 1 FI;
ELSE:ENDDECISION;
DECISION buffer(b)!dataPriority = dp;
 (False):       TASK        diagnostic := DC_wrong_transport_priority;
                RETURN;
ELSE:ENDDECISION;
DECISION pdu!kind;
 (MCrq):        TASK        cAttrSet := pdu!mergeChannels,
                            cSet := pdu!purgeChannelIds;
                 1b :       /* for cAttr in cAttrSet */
                DECISION cAttrSet = Empty;
                 (False):   TASK        cAttr := Pick(cAttrSet),
                                        cAttrSet := Del(cAttr, cAttrSet),
                                        c := cAttr!channelId;
                            DECISION c in cSet;
                             (True):    TASK    diagnostic := DC_inconsistent_merge;
                                        RETURN;
                            ELSE:ENDDECISION;
                            TASK        cSet := Incl(c, cSet);
                            DECISION cAttr!kind = Private;
                             (False):   JOIN 1b;
                            ELSE:ENDDECISION;
                            TASK        pdu!mergeChannels := Del(cAttr, pdu!mergeChannels);
                            CALL        Route_user(cAttr!manager, p);
                            DECISION p = r;
                             (False):   TASK    pdu!purgeChannelIds :=
                                                                Incl(c, pdu!purgeChannelIds);
                                        JOIN 1b;
                            ELSE:ENDDECISION;
                            TASK        uSet := cAttr!admitted;
                             2b :       /* for u in uSet */
                            DECISION uSet = Empty;
                             (False):   TASK    u := Pick(uSet),
                                                uSet := Del(u, uSet);
                                        CALL    Route_user(u, p);
```

```
                        DECISION p = r;
                         (False): TASK     cAttr!admitted :=
                                                    Del(u, cAttr!admitted);
                        ELSE:ENDDECISION;
                        JOIN 2b;
                ELSE:ENDDECISION;
                TASK    pdu!mergeChannels := Incl(cAttr, pdu!mergeChannels);
                JOIN 1b;
        ELSE:ENDDECISION;
(MTrq):  TASK   tAttrSet := pdu!mergeTokens,
                tSet := pdu!purgeTokenIds;
         3b :   /* for tAttr in tAttrSet */
        DECISION tAttrSet = Empty;
         (False): TASK  tAttr := Pick(tAttrSet),
                        tAttrSet := Del(tAttr, tAttrSet),
                        t := tAttr!tokenId;
                DECISION t in tSet;
                 (True):    TASK    diagnostic := DC_inconsistent_merge;
                        RETURN;
                ELSE:ENDDECISION;
                TASK    tSet := Incl(t, tSet),
                        pdu!mergeTokens := Del(tAttr, pdu!mergeTokens);
                DECISION tAttr!kind;
                 (Grabbed, Ungivable):
                        CALL    Route_user(tAttr!grabber, p);
                        DECISION p = r;
                         (False): JOIN 4f;
                        ELSE:ENDDECISION;
                 (Given):
                        CALL    Route_user(tAttr!recipient, p);
                        DECISION p = r;
                         (False): JOIN 4f;
                        ELSE:ENDDECISION;
                 (Giving):
                        CALL    Route_user(tAttr!recipient, p);
                        DECISION p = r;
                         (True):  CALL  Route_user(tAttr!grabber, p);
                                DECISION p = r;
                                 (False): TASK          tAttr!kind := Given;
                                ELSE:ENDDECISION;
                         (False): TASK  tAttr!kind := Ungivable;
                                CALL    Route_user(tAttr!grabber, p);
                                DECISION p = r;
                                 (False): 4f :
                                        TASK        pdu!purgeTokenIds :=
                                                        Incl(t, pdu!purgeTokenIds);
                                        JOIN 3b;
                                ELSE:ENDDECISION;
                        ENDDECISION;
                 (Inhibited):
                        TASK    uSet := tAttr!inhibitors;
                         5b :   /* for u in uSet */
                        DECISION uSet = Empty;
                         (False): TASK  u := Pick(uSet),
                                        uSet := Del(u, uSet);
                                CALL    Route_user(u, p);
                                DECISION p = r;
                                 (False): TASK          tAttr!inhibitors :=
                                                        Del(u, tAttr!inhibitors);
                                ELSE:ENDDECISION;
                                JOIN 5b;
                        ELSE:ENDDECISION;
                ENDDECISION;
                TASK    pdu!mergeTokens := Incl(tAttr, pdu!mergeTokens);
                JOIN 3b;
        ELSE:ENDDECISION;
```

```
(DUrq):         TASK      uSet := pdu!userIds;
                 6b :     /* for u in uSet */
                DECISION uSet = Empty;
                 (False): TASK      u := Pick(uSet),
                                    uSet := Del(u, uSet);
                          CALL      Route_user(u, p);
                          DECISION p = r;
                           (False):   TASK    pdu!userIds := Del(u, pdu!userIds);
                          ELSE:ENDDECISION;
                          JOIN 6b;
                ELSE:ENDDECISION;
                DECISION pdu!userIds = Empty;
                 (True):  TASK      diagnostic := DC_ignore;
                          RETURN;
                ELSE:ENDDECISION;
(CJrq, CCrq, CDrq, CArq, CErq,
 SDrq, USrq, TGrq, TIrq, TVrq,
 TPrq, TRrq, TTrq):
                CALL      Route_user(pdu!initiator, p);
                DECISION p = r;
                 (False): TASK      diagnostic := DC_ignore;
                          RETURN;
                ELSE:ENDDECISION;
(TVin):         DECISION pdu!recipient in uUsed;
                 (False): TASK      diagnostic := DC_misrouted_user;
                          RETURN;
                ELSE:ENDDECISION;
(TVrs):         CALL      Route_user(pdu!recipient, p);
                DECISION p = r;
                 (False): TASK      diagnostic := DC_ignore;
                          RETURN;
                ELSE:ENDDECISION;
(CJcf, CCcf, TGcf, TIcf, TVcf,
 TRcf, TTcf):
                DECISION pdu!initiator in uUsed;
                 (False): TASK      diagnostic := DC_misrouted_user;
                          RETURN;
                ELSE:ENDDECISION;
ELSE:ENDDECISION;
DECISION pdu!kind;
(CDrq, CArq, CErq):
                TASK      c := pdu!channelId;
                DECISION c in cUsed and chan(c)!kind = Private
                          and pdu!initiator = chan(c)!manager;
                 (False): TASK      diagnostic := DC_ignore;
                          RETURN;
                ELSE:ENDDECISION;
(SDrq, USrq):
                TASK      c := pdu!channelId;
                DECISION c in cUsed and chan(c)!kind = Private;
                 (True):  DECISION pdu!initiator in chan(c)!admitted;
                           (False):   TASK    diagnostic := DC_ignore;
                                      RETURN;
                          ELSE:ENDDECISION;
                ELSE:ENDDECISION;
(MCcf):         DECISION not merging and mcrqQueue = EmptyQueue;
                 (True):  TASK      diagnostic := DC_unrequested_confirm;
                          RETURN;
                ELSE:ENDDECISION;
(MTcf):         DECISION not merging and mtrqQueue = EmptyQueue;
                 (True):  TASK      diagnostic := DC_unrequested_confirm;
                          RETURN;
                ELSE:ENDDECISION;
```

```
        (AUcf):     DECISION aurqQueue = EmptyQueue;
                    (True):  TASK     diagnostic := DC_unrequested_confirm;
                             RETURN;
                    ELSE:ENDDECISION;
        (CJcf, CCcf, TGcf, TIcf, TVcf,
         TRcf, TTcf):
                    DECISION merging;
                    (True):  TASK     diagnostic := DC_unrequested_confirm;
                             RETURN;
                    ELSE:ENDDECISION;
    ELSE:ENDDECISION;
    TASK        buffer(b)!pdu := pdu;
    RETURN
    COMMENT     'Additional tests might be applied to check
                that MCSPDUs flowing downward are consistent
                with user or channel states already recorded.
                But stronger assertions could contain flaws,
                and such defenses are not necessary for the
                continued functioning of this MCS provider.
                The value of an MCS domain involves trust in
                the correct operation of superior providers.
                ';
    ENDPROCEDURE;

                                                                    /*----------------------*/
PROCEDURE       Top_provider;                                       /*  Top_provider  */
FPAR            r           PortalId,                               /*----------------------*/
                b           BufferId;

    DCL         pdu         PDUStruct,
                new         PDUStruct,
                u           UserId,
                c           ChannelId,
                cAttr       ChannelAttributes,
                t           TokenId,
                tAttr       TokenAttributes,
                result      Result,
                diagnostic  Diagnostic;
    START
    COMMENT     'The buffer containing an MCSPDU will be processed next
                by Apply_PDU.  Its contents may first be modified here.
                Changing pdu!kind results in "turning the corner".
                ';
    TASK        pdu := buffer(b)!pdu;
    DECISION pdu!kind;
    (EDrq):     DECISION pdu!subHeight < parameters!maxHeight;
                (False): TASK     pdSend := True;
                ELSE:ENDDECISION;
    (MCrq):     TASK     new!kind := MCcf,
                         new!mergeChannels := Empty,
                         new!purgeChannelIds := pdu!purgeChannelIds;
                1b :     /* for cAttr in mergeChannels */
                DECISION pdu!mergeChannels = Empty;
                (False): TASK     cAttr := Pick(pdu!mergeChannels),
                                  pdu!mergeChannels := Del(cAttr, pdu!mergeChannels),
                                  c := cAttr!channelId;
                         DECISION c in cUsed;
                         (True):  DECISION chan(c)!kind;
                                  (Static):
                                           JOIN 2f;
                                  (Private):
                                           DECISION cAttr!kind = Private
                                                    and cAttr!manager = chan(c)!manager;
                                           (True):  JOIN 2f;
                                           ELSE:ENDDECISION;
                                  ELSE:ENDDECISION;
```

```
                    (False):    DECISION cAttr!kind = UserId;
                                 (True):   CALL      Take_user(c, diagnostic);
                                 (False):  CALL      Take_channel(c, diagnostic);
                                ENDDECISION;
                                DECISION diagnostic = DC_OK;
                                 (True):   TASK      chan(c)!kind := cAttr!kind;
                                            2f :
                                           TASK      new!mergeChannels :=
                                                                 Incl(cAttr, new!mergeChannels);
                                           JOIN 1b;
                                ELSE:ENDDECISION;
                    ENDDECISION;
                    TASK        new!purgeChannelIds := Incl(c, new!purgeChannelIds);
                    JOIN 1b;
          ELSE:ENDDECISION;
          TASK      pdu := new;
(MTrq):   TASK      new!kind := MTcf,
                    new!mergeTokens := Empty,
                    new!purgeTokenIds := pdu!purgeTokenIds;
           3b :     /* for tAttr in mergeTokens */
          DECISION pdu!mergeTokens = Empty;
           (False): TASK        tAttr := Pick(pdu!mergeTokens),
                                pdu!mergeTokens := Del(tAttr, pdu!mergeTokens),
                                t := tAttr!tokenId;
                    DECISION t in tUsed;
                     (True):    DECISION token(t)!kind;
                                 (Inhibited):
                                           DECISION tAttr!kind = Inhibited;
                                            (True):   JOIN 4f;
                                           ELSE:ENDDECISION;
                                ELSE:ENDDECISION;
                     (False):   CALL      Take_token(t, diagnostic);
                                DECISION diagnostic = DC_OK;
                                 (True):    4f :
                                           TASK      new!mergeTokens :=
                                                                 Incl(tAttr, new!mergeTokens);
                                           JOIN 3b;
                                ELSE:ENDDECISION;
                    ENDDECISION;
                    TASK        new!purgeTokenIds  :=  Incl(t, new!purgeTokenIds);
                    JOIN 3b;
          ELSE:ENDDECISION;
          TASK      pdu  :=  new;
(AUrq):   CALL      New_user(u, result);
          TASK      pdu!kind  :=  AUcf,
                    pdu!result  :=  result,
                    pdu!initiator  :=  u;
(DUrq):   TASK      pdu!kind  :=  DUin;
(CJrq):   TASK      pdu!kind  :=  CJcf,
                    c  :=  pdu!channelId,
                    pdu!requested  :=  c,
                    result  :=  RT_successful;
          DECISION c = 0;
           (True):  CALL        New_channel(c, result);
                    DECISION result = RT_successful;
                     (True):    TASK      chan(c)!kind  :=  Assigned;
                    ELSE:ENDDECISION;
                    TASK        pdu!channelId  :=  c;
           (False): DECISION c in cUsed;
                     (False):   DECISION c < 1001;
                                 (False):  TASK      result  :=  RT_no_such_channel;
                                 (True):   CALL      Take_chanel(c, diagnostic);
                                           DECISION diagnostic = DC_OK;
                                            (False):  TASK          result  :=  RT_too_many_channels;
                                            (True):   TASK          chan(c)!kind  :=  Static;
                                             ENDDECISION;
                                ENDDECISION;
```

```
                (True):  DECISION chan(c)!kind;
                         (UserId):
                                  TASK   u := UserId(c);
                                  DECISION pdu!initiator = u;
                                  (False):TASK   result := RT_other_user_id;
                                  ELSE:ENDDECISION;
                         (Private):
                                  DECISION pdu!initiator in chan(c)!admitted;
                                  (False):TASK   result := RT_not_admitted;
                                  ELSE:ENDDECISION;
                         ELSE:ENDDECISION;
                ENDDECISION;
                DECISION result = RT_successful;
                (False):TASK   pdu!channelId := 0;
                ELSE:ENDDECISION;
        ENDDECISION;
        TASK    pdu!result := result;
(CLrq): /* no special action */
(CCrq): CALL    New_channel(c, result);
        TASK    pdu!kind := CCcf,
                pdu!result := result,
                pdu!channelId := c;
(CDrq): TASK    pdu!kind := CDin;
(CArq): TASK    pdu!kind := CAin;
(CErq): TASK    pdu!kind := CEin;
(SDrq): TASK    pdu!kind := SDin;
(USrq): TASK    pdu!kind := USin;
(TGrq): TASK    pdu!kind := TGcf,
                pdu!result := RT_successful,
                t := pdu!tokenId,
                u := pdu!initiator;
        DECISION t in tUsed;
        (False): CALL   Take_token(t, diagnostic);
                DECISION diagnostic = DC_OK;
                (False): TASK pdu!result := RT_too_many_tokens;
                (True):  TASK token(t)!kind := Grabbed,
                              token(t)!grabber := u;
                ENDDECISION;
        (True):  DECISION token(t)!kind = Inhibited
                         and token(t)!inhibitors = Incl(u, Empty);
                (False):TASK  pdu!result := RT_token_not_available;
                (True): TASK  token(t)!kind := Grabbed,
                              token(t)!grabber := u;
                ENDDECISION;
        ENDDECISION;
        CALL    Token_status(pdu);
(TIrq): TASK            pdu!kind := TIcf,
                pdu!result := RT_successful,
                t := pdu!tokenId,
                u := pdu!initiator;
        DECISION t in tUsed;
        (False): CALL   Take_token(t, diagnostic);
                DECISION diagnostic = DC_OK;
                (False):TASK  pdu!result := RT_too_many_tokens;
                (True): TASK  token(t)!kind := Inhibited,
                              token(t)!inhibitors := Incl(u, Empty);
                ENDDECISION;
        (True):  DECISION token(t)!kind = Grabbed and token(t)!grabber = u;
                (True): TASK    token(t)!kind := Inhibited,
                              token(t)!inhibitors := Incl(u, Empty);
                ELSE:ENDDECISION;
                DECISION token(t)!kind = Inhibited;
                (False):TASK   pdu!result := RT_token_not_available;
                (True): TASK   token(t)!inhibitors :=
                                        Incl(u, token(t)!inhibitors);
                ENDDECISION;
```

```
                ENDDECISION;
                CALL    Token_status(pdu);
        (TVrq): TASK    pdu!kind := TVcf,
                        pdu!result := RT_token_not_possessed,
                        t := pdu!tokenId,
                        u := pdu!initiator;
                DECISION t in tUsed and token(t)!kind = Grabbed and token(t)!grabber = u;
                (True): DECISION pdu!recipient in uUsed;
                        (False):TASK    pdu!result := RT_no_such_user;
                        (True): TASK    pdu!kind := TVin,
                                        token(t)!kind := Giving,
                                        token(t)!recipient := pdu!recipient;
                        ENDDECISION;
                ELSE:ENDDECISION;
                CALL    Token_status(pdu);
        (TPrq): TASK    pdu!kind := TPin;
        (TRrq): TASK    pdu!kind := TRcf,
                        pdu!result := RT_token_not_possessed,
                        t := pdu!tokenId,
                        u := pdu!initiator;
                DECISION t in tUsed;
                (True): DECISION token(t)!kind;
                        (Grabbed, Ungivable):
                                DECISION token(t)!grabber = u;
                                (True): TASK    pdu!result := RT_successful;
                                        CALL    Delete_token(t);
                                ELSE:ENDDECISION;
                        (Giving):
                                DECISION token(t)!grabber = u;
                                (True): TASK    pdu!result := RT_successful,
                                                token(t)!kind := Given;
                                ELSE:ENDDECISION;
                        (Inhibited):
                                DECISION u in token(t)!inhibitors;
                                (True): TASK    pdu!result := RT_successful,
                                                token(t)!inhibitors :=
                                                        Del(u, token(t)!inhibitors);
                                ELSE:ENDDECISION;
                                DECISION token(t)!inhibitors = Empty;
                                (True): CALL    Delete_token(t);
                                ELSE:ENDDECISION;
                        ENDDECISION;
                ELSE:ENDDECISION;
                CALL    Token_status(pdu);
        (TTrq): TASK    pdu!kind := TTcf;
                CALL    Token_status(pdu);
        (TVrs): TASK    t := pdu!tokenId,
                        u := pdu!recipient;
                DECISION t in tUsed and token(t)!recipient = u;
                (True): DECISION token(t)!kind;
                        (Giving):
                                TASK    token(t)!kind := Grabbed,
                                        pdu!kind := TVcf,
                                        pdu!initiator := token(t)!grabber;
                                DECISION pdu!result = RT_successful;
                                (True): TASK    token(t)!grabber := u;
                                ELSE:ENDDECISION;
                                CALL    Token_status(pdu);
                        ELSE:ENDDECISION;
                ELSE:ENDDECISION;
        ELSE:ENDDECISION;
        TASK    buffer(b)!pdu := pdu;
        RETURN;
        ENDPROCEDURE;

                                                                        /*--------------------*/

PROCEDURE       Apply_PDU;                                              /*  Apply_PDU  */

FPAR            r               PortalId,                               /*--------------------*/
                b               BufferId,
```

```
IN/OUT    diagnostic        Diagnostic;

DCL       pdu               PDUStruct,
          new               PDUStruct,
          p                 PortalId,
          pSet              PortalIdSet,
          x                 PortalId,
          u                 UserId,
          uSet              UserIdSet,
          c                 ChannelId,
          cSet              ChannelIdSet,
          cAttr             ChannelAttributes,
          cAttrSet          ChannelAttributesSet,
          t                 TokenId,
          tSet              TokenIdSet,
          tAttr             TokenAttributes,
          tAttrSet          TokenAttributesSet,
          result            Result;
START
COMMENT'This is a large case selection based on MCSPDU kind.
          These actions and updates to the information base
          take place in both the top and subordinate providers.
          ';
TASK      pdu := buffer(b)!pdu;
DECISION pdu!kind;
(MCrq, MTrq, AUrq, DUrq, CCrq,
CDrq,  CArq,  CErq, USrq, TGrq,
TIrq,   TVrq,  TPrq, TRrq,  TTrq):
          CALL    Cast_buffer(b, upward);
(PDin):   DECISION pdu!heightLimit > 0;
          (True):TASK       buffer(b)!pdu!heightLimit := pdu!heightLimit - 1;
          (False):OUTPUT    Report.portal(r, DC_height_limit_exceeded) TO control;
          ENDDECISION;
          CALL    Broadcast_buffer(b);
(EDrq):   TASK    portal(r)!subHeight := pdu!subHeight,
                  portal(r)!subInterval := pdu!subInterval,
                  portal(r)!interval := IF pdu!subInterval = 0 THEN interval
                                        ELSE oneSecond + (pdu!subInterval * 3) FI;
          CALL    Erect_domain;
(MCcf):   DECISION merging;
          (False):TASK      p := Next(mcrqQueue),
                            mcrqQueue := Pull(mcrqQueue);
          ELSE:ENDDECISION;
          TASK    cAttrSet := pdu!mergeChannels;
          1b :    /* for cAttr in cAttrSet */
          DECISION cAttrSet = Empty;
          (False): TASK     cAttr := Pick(cAttrSet),
                            cAttrSet := Del(cAttr, cAttrSet),
                            c := cAttr!channelId;
                   DECISION c in cUsed and chan(c)!kind = cAttr!kind;
                   (False): DECISION cAttr!kind = UserId;
                            (True): CALL    Take_user(c, diagnostic);
                            (False): CALL   Take_channel(c, diagnostic);
                            ENDDECISION;
                            DECISION diagnostic = DC_OK;
                            (False):RETURN;
                            ELSE:ENDDECISION;
                            TASK  chan(c)!kind := cAttr!kind;
                   ELSE:ENDDECISION;
                   DECISION merging;
                   (True):  DECISION chan(c)!kind = UserId;
                            (True): TASK    u := UserId(c),
                                            uConfirm := Incl(u, uConfirm);
                            ELSE:ENDDECISION;
```

```
        DECISION chan(c)!kind = Private
                and  chan(c)!uMerge /= Empty;
        (True): TASK    cMerge  :=  Incl(c, cMerge);
        (False):TASK    cConfirm  :=  Incl(c, cConfirm);
        ENDDECISION;
        JOIN 1b;
ELSE:ENDDECISION;
        TASK    chan(c)!portal  :=  p;
        DECISION cAttr!kind = Static or cAttr!kind = Assigned
                or cAttr!joined;
        (True): DECISION p = 0;
                (False):TASK    chan(c)!joined :=
                                        Incl(p, chan(c)!joined),
                                cLeave  :=  Del(c, cLeave);
                (True): DECISION chan(c)!joined = Empty;
                        (True): TASK    cLeave  :=  Incl(c, cLeave);
                        ELSE:ENDDECISION;
                ENDDECISION;
        ELSE:ENDDECISION;
        DECISION cAttr!kind = UserId and p = 0;
        (True): TASK    u  :=  UserId(c),
                        uDetach  :=  Incl(u, uDetach),
                        cLeave  :=  Del(c, cLeave);
        ELSE:ENDDECISION;
        DECISION cAttr!kind = Private;
        (False):JOIN 1b;
        ELSE:ENDDECISION;
        CALL    Route_user(cAttr!manager, x);
        DECISION x = p;
        (False):TASK    chan(c)!manager  :=  0,
                        buffer(b)!pdu!mergeChannels :=
                                Del(cAttr, buffer(b)!pdu!mergeChannels),
                        buffer(b)!pdu!purgeChannelIds :=
                                Incl(c, buffer(b)!pdu!purgeChannelIds),
                        cDisband  :=  Incl(c, cDisband);
        (True): TASK    chan(c)!manager  :=  cAttr!manager,
                        uSet  :=  cAttr!admitted and uUsed;
                2b :    /* for u in uSet */
                DECISION uSet = Empty;
                (False):TASK    u  :=  Pick(uSet),
                                uSet  :=  Del(u, uSet);
                        CALL    Route_user(u, x);
                        DECISION x = p;
                        (True): TASK    chan(c)!admitted :=
                                                Incl(u, chan(c)!admitted);
                        ELSE:ENDDECISION;
                        JOIN 2b;
                ELSE:ENDDECISION;
        ENDDECISION;
        JOIN 1b;
ELSE:ENDDECISION;
DECISION merging;
(False): CALL  Cast_buffer(b, p);
(True):  TASK  new!kind  :=  PCin,
                new!detachUserIds  :=  Empty,
                new!purgeChannelIds  :=  Empty,
                cSet  :=  pdu!purgeChannelIds and cUsed;
        3b :    /* for c in cSet */
        DECISION cSet = Empty;
(False): TASK  c  :=  Pick(cSet),
                cSet  :=  Del(c, cSet);
        DECISION chan(c)!kind;
        (UserId):
                TASK    u  :=  UserId(c),
                        new!detachUserIds :=
                                Incl(u, new!detachUserIds);
```

```
                    (Static, Assigned, Private):
                        TASK    new!purgeChannelIds :=
                                        Incl(c, new!purgeChannelIds);
                    ENDDECISION;
                    JOIN 3b;
                ELSE:ENDDECISION;
                CALL    Purge_users(new!detachUserIds);
                CALL    Purge_channels(new!purgeChannelIds);
                TASK    buffer(b)!pdu  :=  new;
                CALL    Broadcast_buffer(b);
            ENDDECISION;
(PCin):     TASK    cSet  :=  pdu!purgeChannelIds and cUsed,
                    buffer(b)!pdu!purgeChannelIds  :=  cSet;
            DECISION merging;
            (True):  TASK    buffer(b)!pdu!detachUserIds  :=  pdu!detachUserIds
                                                        and uConfirm,
                    buffer(b)!pdu!purgeChannelIds  :=  cSet and cConfirm;
                4b :    /* for c in cSet */
                DECISION cSet = Empty;
                (False): TASK  c  :=  Pick(cSet),
                        cSet  :=  Del(c, cSet);
                    DECISION c in cConfirm;
                    (False):TASK    chan(c)!uMerge  :=  chan(c)!admitted;
                    ELSE:ENDDECISION;
                    JOIN 4b;
                ELSE:ENDDECISION;
            ELSE:ENDDECISION;
            CALL    Purge_users(buffer(b)!pdu!detachUserIds);
            CALL    Purge_channels(buffer(b)!pdu!purgeChannelIds);
            CALL    Broadcast_buffer(b);
(MTcf):     DECISION merging;
            (False):TASK    p  :=  Next(mtrqQueue),
                    mtrqQueue  :=  Pull(mtrqQueue);
            ELSE:ENDDECISION;
            TASK    tAttrSet  :=  pdu!mergeTokens;
              5b :  /* for tAttr in tAttrSet */
            DECISION tAttrSet = Empty;
            (False): TASK    tAttr  :=  Pick(tAttrSet),
                    tAttrSet  :=  Del(tAttr, tAttrSet),
                    t  :=  tAttr!tokenId;
                DECISION t in tUsed;
                (False): CALL  Take_token(t, diagnostic);
                    DECISION diagnostic = DC_OK;
                    (False):RETURN;
                    ELSE:ENDDECISION;
                ELSE:ENDDECISION;
                TASK    token(t)!kind  :=  tAttr!kind,
                    token(t)!grabber  :=  tAttr!grabber,
                    token(t)!recipient  :=  tAttr!recipient;
                DECISION merging;
                (True):  DECISION token(t)!kind = Inhibited
                            and token(t)!uMerge /= Empty;
                    (True): TASK    tMerge  :=  Incl(t, tMerge);
                    (False):TASK    tConfirm  :=  Incl(t, tConfirm);
                        DECISION token(t)!kind = Inhibited
                                and token(t)!inhibitors = Empty;
                        (True): CALL    Delete_token(t);
                        ELSE:ENDDECISION;
                    ENDDECISION;
                    JOIN 5b;
                ELSE:ENDDECISION;
                DECISION upward = 0 and token(t)!kind = Ungivable;
                (True): TASK    tReject  :=  Incl(t, tReject);
                ELSE:ENDDECISION;
                DECISION tAttr!kind;
```

```
                (Grabbed, Ungivable):
                        CALL  Token_route(tAttr!grabber, x, p);
                        DECISION x = p;
                        (False):JOIN 6f;
                        ELSE:ENDDECISION;
                (Given):
                        CALL  Token_route(tAttr!recipient, x, p);
                        DECISION x = p;
                        (False):JOIN 6f;
                        ELSE:ENDDECISION;
                (Giving):
                        CALL   Token_route(tAttr!recipient, x, p);
                        DECISION x = p;
                        (True): CALL   Token_route(tAttr!grabber, x, p);
                                DECISION x = p;
                                (False):TASK   token(t)!kind  :=  Given;
                                ELSE:ENDDECISION;
                        (False):CALL   Token_route(tAttr!grabber, x, p);
                                DECISION x = p;
                                (True): TASK   token(t)!kind  :=  Ungivable;
                                (False): 6f :
                                        TASK   buffer(b)!pdu!mergeTokens :=
                                             Del(tAttr, buffer(b)!pdu!mergeTokens),
                                                  buffer(b)!pdu!purgeTokenIds :=
                                             Incl(t, buffer(b)!pdu!purgeTokenIds);
                                        CALL   Delete_token(t);
                                ENDDECISION;
                        ENDDECISION;
                (Inhibited):
                        TASK   uSet  :=  tAttr!inhibitors and uUsed;
                          7b :   /* for u in uSet */
                        DECISION uSet = Empty;
                                (False):TASK   u  :=  Pick(uSet),
                                                  uSet  :=  Del(u, uSet);
                                        CALL   Token_route(u, x, p);
                                        DECISION x = p;
                                        (True): TASK   token(t)!inhibitors :=
                                                              Incl(u, token(t)!inhibitors);
                                        ELSE:ENDDECISION;
                                        JOIN 7b;
                                ELSE:ENDDECISION;
                                DECISION token(t)!inhibitors = Empty;
                                (True): CALL   Delete_token(t);
                                ELSE:ENDDECISION;
                        ENDDECISION;
                        JOIN 5b;
                ELSE:ENDDECISION;
                DECISION merging;
                (False): CALL   Cast_buffer(b, p);
                (True):  CALL   Purge_tokens(pdu!purgeTokenIds);
                        TASK   buffer(b)!pdu!kind  :=  PTin;
                        CALL   Broadcast_buffer(b);
                ENDDECISION;
        (PTin):  TASK   tSet  :=  pdu!purgeTokenIds and tUsed,
                        buffer(b)!pdu!purgeTokenIds  :=  tSet;
                DECISION merging;
                (True):  TASK   buffer(b)!pdu!purgeTokenIds  :=  tSet and tConfirm;
                          8b :   /* for t in tSet */
                        DECISION tSet = Empty;
                        (False):TASK   t  :=  Pick(tSet),
                                        tSet  :=  Del(t, tSet);
                                DECISION t in tConfirm;
                                (False):TASK   token(t)!uMerge  :=  token(t)!inhibitors;
                                ELSE:ENDDECISION;
                                JOIN 8b;
                        ELSE:ENDDECISION;
```

```
                ELSE:ENDDECISION;
                CALL    Purge_tokens(buffer(b)!pdu!purgeTokenIds);
                CALL    Broadcast_buffer(b);
        (DPum): OUTPUT  Drop.portal(r, pdu!reason) TO control;
        (AUcf): TASK    p  :=  Next(aurqQueue),
                        aurqQueue  :=  Pull(aurqQueue),
                        c  :=  ChannelId(u),
                        u  :=  pdu!initiator;
                DECISION pdu!result = RT_successful;
                (True):  DECISION upward = 0;
                        (False):CALL    Take_user(c, diagnostic);
                                DECISION diagnostic = DC_OK;
                                (False):RETURN;
                                ELSE:ENDDECISION;
                        ELSE:ENDDECISION;
                        TASK    chan(c)!portal  :=  p;
                        DECISION p = 0;
                        (True): TASK    uDetach  :=  Incl(u, uDetach),
                                        cLeave  :=  Del(c, cLeave);
                        ELSE:ENDDECISION;
                ELSE:ENDDECISION;
                CALL    Cast_buffer(b, p);
        (DUin): DECISION merging;
                (True):  TASK    buffer(b)!pdu!userIds  :=  pdu!userIds and uConfirm;
                ELSE:ENDDECISION;
                CALL    Purge_users(buffer(b)!pdu!userIds);
                CALL    Broadcast_buffer(b);
        (CJrq): TASK    c  :=  pdu!channelId,
                        result  :=  RT_successful,
                        p  :=  upward;
                DECISION c in cUsed;
                (True):  DECISION chan(c)!kind;
                        (UserId):
                                TASK    u  :=  UserId(c);
                                DECISION pdu!initiator = u;
                                (False):TASK    result  :=  RT_other_user_id;
                                ELSE:ENDDECISION;
                        (Private):
                                DECISION pdu!initiator in chan(c)!admitted;
                                (False):TASK    result  :=  RT_not_admitted;
                                ELSE:ENDDECISION;
                        ELSE:ENDDECISION;
                        DECISION result = RT_successful;
                        (False):TASK    buffer(b)!pdu!kind  :=  CJcf,
                                buffer(b)!pdu!requested  :=  c,
                                buffer(b)!pdu!result  :=  result,
                                buffer(b)!pdu!channelId  :=  0,
                                p  :=  r;
                (True):  DECISION chan(c)!joined /= Empty or c in cLeave;
                        (True): TASK    chan(c)!joined :=
                                                Incl(r, chan(c)!joined),
                                cLeave  :=  Del(c, cLeave),
                                buffer(b)!pdu!kind  :=  CJcf,
                                buffer(b)!pdu!requested  :=  c,
                                buffer(b)!pdu!result  :=  result,
                                p  :=  r;
                        ELSE:ENDDECISION;
                ENDDECISION;
        ELSE:ENDDECISION;
        CALL    Cast_buffer(b, p);
(CJcf):  TASK   c  :=  pdu!channelId,
                u  :=  pdu!initiator;
        CALL    Route_user(u, p);
        DECISION pdu!result = RT_successful;
        (True):  DECISION c in cUsed;
                (False): CALL    Take_channel(c, diagnostic);
```

```
                        DECISION diagnostic = DC_OK;
                        (False):RETURN;
                        ELSE:ENDDECISION;
                        TASK    chan(c)!kind  :=  IF c < 1001 THEN Static
                                                  ELSE Assigned FI;
                ELSE:ENDDECISION;
                DECISION p = 0;
                (False): TASK    chan(c)!joined  :=  Incl(p, chan(c)!joined),
                                 cLeave  :=  Del(c, cLeave);
                (True):  DECISION chan(c)!joined = Empty;
                         (True): TASK    cLeave  :=  Incl(c, cLeave);
                         ELSE:ENDDECISION;
                ENDDECISION;
        ELSE:ENDDECISION;
        CALL    Cast_buffer(b, p);
(CLrq): TASK    cSet  :=  pdu!channelIds and cUsed;
  9b :  /* for c in cSet */
        DECISION cSet = Empty;
        (False): TASK    c  :=  Pick(cSet),
                         cSet  :=  Del(c, cSet);
                DECISION r in chan(c)!joined;
                (True):  TASK    chan(c)!joined  :=  Del(r, chan(c)!joined);
                         DECISION chan(c)!joined = Empty;
                         (True): TASK    cLeave  :=  Incl(c, cLeave);
                         ELSE:ENDDECISION;
                ELSE:ENDDECISION;
                JOIN 9b;
        ELSE:ENDDECISION;
(CCcf): TASK    c  :=  pdu!channelId,
                u  :=  pdu!initiator;
        DECISION pdu!result = RT_successful;
        (True):  DECISION upward = 0;
                 (False): CALL    Take_channel(c, diagnostic);
                          DECISION diagnostic = DC_OK;
                          (False):RETURN;
                          ELSE:ENDDECISION;
                 ELSE:ENDDECISION;
                 TASK    chan(c)!kind  :=  Private,
                         chan(c)!manager  :=  u,
                         chan(c)!admitted  :=  Incl(u, Empty);
        ELSE:ENDDECISION;
        CALL    Route_user(u, p);
        CALL    Cast_buffer(b, p);
(CDin): 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  :=  Incl(chan(c)!manager, chan(c)!admitted);
        CALL    Delete_channel(c);
        CALL    Multicast_buffer(b, uSet);
(CAin): TASK    c  :=  pdu!channelId,
                uSet  :=  pdu!userIds and uUsed,
                buffer(b)!pdu!userIds  :=  uSet;
        DECISION merging;
        (True):  TASK     uSet  :=  uSet and uConfirm,
                          buffer(b)!pdu!userIds  :=  uSet;
                 10b :    /* for u in uSet */
                 DECISION uSet = Empty;
```

```
                (False): TASK   u  :=  Pick(uSet),
                                uSet  :=  Del(u, uSet),
                                c  :=  ChannelId(u),
                                chan(c)!portal = 0;
                         JOIN 10b;
                ELSE:ENDDECISION;
                TASK    buffer(b)!pdu!kind  :=  DUrq,
                        buffer(b)!pdu!reason  :=  RN_channel_purged;
                CALL    Cast_buffer(b, r);
                RETURN;
        ELSE:ENDDECISION;
        DECISION uSet = Empty;
        (False): DECISION c in cUsed and chan(c)!kind = Private;
                (False): CALL   Take_channel(c, diagnostic);
                        DECISION diagnostic = DC_OK;
                        (False):RETURN;
                        ELSE:ENDDECISION;
                ELSE:ENDDECISION;
                TASK    chan(c)!kind  :=  Private,
                        chan(c)!manager  :=  pdu!initiator,
                        chan(c)!admitted  :=  chan(c)!admitted or uSet;
                CALL    Multicast_buffer(b, uSet);
        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;
         11b :   /* 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 11b;
        ELSE:ENDDECISION;
        TASK    uSet  :=  pdu!userIds and chan(c)!admitted,
                buffer(b)!pdu!userIds  :=  uSet;
         12b :    /* 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 12b;
        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 13f;
(SDin):     TASK   pSet  :=  Empty;
            JOIN 13f;
(USin):     TASK   pSet  :=  Empty;
              13f :
            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;
              14b :   /* for p in pSet */
            DECISION pSet = Empty;
            (False): TASK    p  :=  Pick(pSet),
                             pSet  :=  Del(p, pSet);
                    CALL    Cast_buffer(b, p);
                    JOIN 14b;
            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;
                    ELSE:ENDDECISION;
                    TASK   token(t)!kind  :=  Grabbed,
                             token(t)!grabber  :=  u;
            (SelfInhibited):
                    DECISION t in tUsed;
                    (False): CALL   Take_token(t, diagnostic);
                             DECISION diagnostic = DC_OK;
                             (False):RETURN;
                             ELSE:ENDDECISION;
                    ELSE:ENDDECISION;
                    TASK   token(t)!kind  :=  Inhibited,
                             token(t)!inhibitors  :=  Incl(u, token(t)!inhibitors);
            (SelfRecipient):
                    TASK   token(t)!recipient  :=  u;
            (SelfGiving):
                    TASK   token(t)!grabber  :=  u;
            (NotInUse):
                    CALL   Delete_token(t);
            ELSE:
                    DECISION token(t)!kind;
                    (Grabbed, Ungivable):
                             DECISION token(t)!grabber = u;
                             (True): CALL   Delete_token(t);
                             ELSE:ENDDECISION;
                    (Giving, Given):
                             DECISION token(t)!grabber = u;
                             (True): TASK   token(t)!kind  :=  Given;
                             ELSE:ENDDECISION;
                             DECISION token(t)!recipient = u;
                             (True): CALL   Delete_token(t);
```

```
                    ELSE:ENDDECISION;
               (Inhibited):
                    TASK   token(t)!inhibitors :=
                                   Del(u, token(t)!inhibitors);
                    DECISION token(t)!inhibitors = Empty;
                    (True): CALL   Delete_token(t);
                    ELSE:ENDDECISION;
               ENDDECISION;
          ENDDECISION;
          CALL   Route_user(u, p);
          CALL   Cast_buffer(b, p);
(TVin):   TASK   t := pdu!tokenId,
               u := pdu!recipient;
          DECISION merging;
          (True): TASK   buffer(b)!pdu!kind := TVrs,
                    buffer(b)!pdu!result := RT_domain_merging;
               CALL   Cast_buffer(b, r);
               RETURN;
          ELSE:ENDDECISION;
          DECISION t in tUsed;
          (False): CALL   Take_token(t, diagnostic);
               DECISION diagnostic = DC_OK;
               (False):RETURN;
               ELSE:ENDDECISION;
          ELSE:ENDDECISION;
          TASK   token(t)!kind := Giving,
               token(t)!grabber := pdu!initiator,
               token(t)!recipient := u;
          CALL   Route_user(u, p);
          CALL   Cast_buffer(b, p);
(TVrs):   TASK   t := pdu!tokenId,
               u := pdu!recipient;
          DECISION t in tUsed and token(t)!recipient = u;
          (True): DECISION token(t)!kind;
               (Giving):
                    TASK   token(t)!kind := Grabbed;
                    DECISION pdu!result = RT_successful;
                    (True): TASK   token(t)!grabber := u;
                    (False):DECISION token(t)!grabber in uUsed;
                         (False):CALL   Delete_token(t);
                         ELSE:ENDDECISION;
                    ENDDECISION;
                    CALL   Cast_buffer(b, upward);
               (Given):
                    TASK   token(t)!kind := Grabbed,
                           token(t)!grabber := u;
                    DECISION pdu!result = RT_successful;
                    (False):CALL   Delete_token(t);
                    ELSE:ENDDECISION;
                    CALL   Cast_buffer(b, upward);
               ELSE:ENDDECISION;
          ELSE:ENDDECISION;
(TPin):   TASK   t := pdu!tokenId;
          DECISION t in tUsed and not merging;
          (True): DECISION token(t)!kind;
               (Grabbed, Ungivable):
                    CALL   Route_user(token(t)!grabber, p);
                    CALL   Cast_buffer(b, p);
               (Given):
                    CALL   Route_user(token(t)!recipient, p);
                    CALL   Cast_buffer(b, p);
               (Giving):
                    TASK   uSet := Incl(token(t)!grabber, Empty);
                    TASK   uSet := Incl(token(t)!recipient, uSet);
                    CALL   Multicast_buffer(b, uSet);
```

```
                        (Inhibited):
                                TASK    uSet  :=  token(t)!inhibitors;
                                CALL    Multicast_buffer(b, uSet);
                        ENDDECISION;
                ELSE:ENDDECISION;
        ENDDECISION;
        RETURN;
        ENDPROCEDURE;
                                                                        /*-----------------------*/
PROCEDURE       Token_status;                                           /*  Token_status  */
FPAR    IN/OUT  pdu             PDUStruct;                              /*-----------------------*/

        DCL     t       TokenId,
                u       UserId,
                status  TokenStatus;
        START
        COMMENT'Calculate for an MCSPDU the relationship between
                 the initiator user id and token id it contains.
                 ';
        TASK    t  :=  pdu!tokenId,
                u  :=  pdu!initiator;
        DECISION t in tUsed;
        (False):TASK    status  :=  NotInUse;
        (True): DECISION token(t)!kind;
                (Grabbed):
                        DECISION token(t)!grabber = u;
                        (True): TASK    status  :=  SelfGrabbed;
                        (False):TASK    status  :=  OtherGrabbed;
                        ENDDECISION;
                (Ungivable):
                        DECISION token(t)!grabber = u;
                        (True): TASK    status  :=  SelfGiving;
                        (False):TASK    status  :=  OtherGiving;
                        ENDDECISION;
                (Given):
                        DECISION token(t)!recipient = u;
                        (True): TASK    status  :=  SelfRecipient;
                        (False):TASK    status  :=  OtherGiving;
                        ENDDECISION;
                (Giving):
                        DECISION token(t)!recipient = u;
                        (True): TASK    status  :=  SelfRecipient;
                        (False):DECISION token(t)!grabber = u;
                                (True): TASK    status  :=  SelfGiving;
                                (False):TASK    status  :=  OtherGiving;
                                ENDDECISION;
                        ENDDECISION;
                (Inhibited):
                        DECISION u in token(t)!inhibitors;
                        (True): TASK    status  :=  SelfInhibited;
                        (False):TASK    status  :=  OtherInhibited;
                        ENDDECISION;
                ENDDECISION;
        ENDDECISION;
        TASK    pdu!tokenStatus  :=  status;
        RETURN;
        ENDPROCEDURE;
                                                                        /*-----------------------*/
PROCEDURE       Token_route;                                            /*  Token_route  */
FPAR            u               UserId,                                 /*-----------------------*/
        IN/OUT  x               PortalId,
                p               PortalId;
        START
        COMMENT'Revoke token user if its route x is no longer via p.
                ';
        CALL    Route_user(u, x);
```

```
        DECISION x = p;
        (False):  DECISION u in uUsed;
                  (True): TASK   uRevoke  :=  Incl(u, uRevoke);
                  ELSE:ENDDECISION;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;
                                                                        /*---------------------*/
PROCEDURE       Delete_user;                                            /*  Delete_user  */
FPAR            u               UserId;                                 /*---------------------*/

        DCL     p               PortalId,
                c               ChannelId,
                cSet            ChannelIdSet,
                a               UserId,
                aSet            UserIdSet,
                q               PortalId,
                t               TokenId,
                tSet            TokenIdSet;
        START
        COMMENT'Update the information base to delete a user id.
                 This has consequences for its channels and tokens.
                 Any data still in transit is left undisturbed.
                 ';
        DECISION u in uUsed;
        (False):RETURN;
        ELSE:ENDDECISION;
        TASK    c  :=  ChannelId(u),
                p  :=  chan(c)!portal,
                uUsed  :=  Del(u, uUsed),
                numUserIds  :=  numUserIds - 1,
                cUsed  :=  Del(c, cUsed),
                cFree  :=  Incl(c, cFree),
                numChannelIds  :=  numChannelIds - 1,
                uConfirm  :=  Del(u, uConfirm),
                cConfirm  :=  Del(c, cConfirm),
                uDetach  :=  Del(u, uDetach),
                uRevoke  :=  Del(u, uRevoke),
                cLeave  :=  Del(c, cLeave);
        TASK    cSet  :=  cUsed;
          1b :  /* for c in cSet */
        DECISION cSet = Empty;
        (False):  TASK   c  :=  Pick(cSet),
                         cSet  :=  Del(c, cSet);
                  DECISION portal(p)!kind = Attached and p in chan(c)!joined;
                  (True):  TASK   chan(c)!joined  :=  Del(p, chan(c)!joined);
                           DECISION chan(c)!joined = Empty;
                           (True): TASK   cLeave  :=  Incl(c, cLeave);
                           ELSE:ENDDECISION;
                  ELSE:ENDDECISION;
                  DECISION chan(c)!kind = Private;
                  (True):  DECISION chan(c)!manager = u;
                           (True):  TASK   chan(c)!manager  :=  0;
                                    DECISION upward = 0;
                                    (True): TASK   cDisband  :=  Incl(c, cDisband);
                                    ELSE:ENDDECISION;
                           ELSE:ENDDECISION;
                           TASK    chan(c)!admitted  :=  Del(u, chan(c)!admitted),
                                   chan(c)!uMerge  :=  Del(u, chan(c)!uMerge);
                           DECISION chan(c)!admitted /= Empty or chan(c)!manager in uUsed;
                           (False): DECISION not merging or c in cConfirm;
                                    (True): CALL   Delete_channel(c);
                                    ELSE:ENDDECISION;
                           (True):  DECISION p in chan(c)!joined;
                                    (True): TASK   aSet  :=  chan(c)!admitted;
                                              2b :  /* for a in aSet */
```

```
                                        DECISION aSet = Empty;
                                        (False):TASK    a  :=  Pick(aSet),
                                                        aSet  :=  Del(a, aSet);
                                                CALL    Route_user(a, q);
                                                DECISION q = p;
                                                (False):JOIN 2b;
                                                (True): JOIN 1b;
                                                ENDDECISION;
                                        ELSE:ENDDECISION;
                                        TASK    chan(c)!joined  :=  Del(p, chan(c)!joined);
                                        DECISION chan(c)!joined = Empty;
                                        (True): TASK    cLeave  :=  Incl(c, cLeave);
                                        ELSE:ENDDECISION;
                                ELSE:ENDDECISION;
                        ENDDECISION;
                ELSE:ENDDECISION;
                JOIN 1b;
        ELSE:ENDDECISION;
        TASK    tSet  :=  tUsed;
          3b :  /* for t in tSet */
        DECISION tSet = Empty;
        (False):  TASK    t  :=  Pick(tSet),
                        tSet  :=  Del(t, tSet);
                DECISION token(t)!kind;
                (Grabbed, Ungivable):
                        DECISION token(t)!grabber = u;
                        (True): JOIN 3f;
                        ELSE:ENDDECISION;
                (Given):
                        DECISION token(t)!recipient = u;
                        (True): JOIN 3f;
                        ELSE:ENDDECISION;
                (Giving):
                        DECISION token(t)!recipient = u;
                        (True):  TASK    token(t)!kind  :=  Ungivable;
                                DECISION token(t)!grabber in uUsed;
                                (False):JOIN 3f;
                                (True): DECISION upward = 0;
                                        (True): TASK    tReject  :=  Incl(t, tReject);
                                        ELSE:ENDDECISION;
                                ENDDECISION;
                        (False): DECISION token(t)!grabber = u;
                                (True): TASK    token(t)!kind  :=  Given;
                                ELSE:ENDDECISION;
                        ELSE:ENDDECISION;
                (Inhibited):
                        TASK    token(t)!inhibitors  :=  Del(u, token(t)!inhibitors),
                                token(t)!uMerge  :=  Del(u, token(t)!uMerge);
                        DECISION token(t)!inhibitors = Empty;
                        (True):   3f :
                                DECISION not merging or t in tConfirm;
                                (True): CALL    Delete_token(t);
                                (False):TASK    token(t)!kind  :=  Inhibited,
                                                token(t)!inhibitors  :=  Empty;
                                ENDDECISION;
                        ELSE:ENDDECISION;
                ENDDECISION;
                JOIN 3b;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;
                                                                        /*-------------------------*/
PROCEDURE       Delete_channel;                                         /*  Delete_channel  */
FPAR            c               ChannelId;                              /*-------------------------*/
```

```
        DCL     u               UserId;
        START
        COMMENT'Update the information base to delete a channel id.
                ';
        DECISION c in cUsed;
        (False):RETURN;
        ELSE:ENDDECISION;
        DECISION chan(c)!kind = UserId;
        (True):  TASK   u  :=  UserId(c);
                 CALL   Delete_user(u);
        (False): TASK   cUsed  :=  Del(c, cUsed),
                        cFree  :=  Incl(c, cFree),
                        numChannelIds  :=  numChannelIds - 1,
                        cConfirm  :=  Del(c, cConfirm),
                        cLeave  :=  Del(c, cLeave),
                        cDisband  :=  Del(c, cDisband);
        ENDDECISION;
        RETURN;
        ENDPROCEDURE;
                                                                /*----------------------*/
PROCEDURE       Delete_token;                                   /*  Delete_token  */
FPAR            t               TokenId;                        /*----------------------*/

        START
        COMMENT'Update the information base to delete a token id.
                ';
        DECISION t in tUsed;
        (False):RETURN;
        ELSE:ENDDECISION;
        TASK    tUsed  :=  Del(t, tUsed),
                tFree  :=  Incl(t, tFree),
                numTokenIds  :=  numTokenIds - 1,
                tConfirm  :=  Del(t, tConfirm),
                tReject  :=  Del(t, tReject);
        RETURN;
        ENDPROCEDURE;
                                                                /*----------------------*/
PROCEDURE       Purge_users;                                    /*  Purge_users  */
FPAR            uSet            UserIdSet;                      /*----------------------*/

        DCL     u               UserId;
        START
        COMMENT'Delete a set of user ids.
                ';
          1b :      /* for u in uSet */
        DECISION uSet = Empty;
        (False): TASK   u  :=  Pick(uSet),
                        uSet  :=  Del(u, uSet);
                 CALL   Delete_user(u);
                 JOIN 1b;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;
                                                                /*--------------------------*/
PROCEDURE       Purge_channels;                                 /*  Purge_channels  */
FPAR            cSet            ChannelIdSet;                   /*--------------------------*/

        DCL     c               ChannelId;
        START
        COMMENT'Delete a set of channel ids.
                ';
          1b :      /* for c in cSet */
        DECISION cSet = Empty;
        (False): TASK    c  :=  Pick(cSet),
                         cSet  :=  Del(c, cSet);
                 CALL   Delete_channel(c);
```

```
                JOIN 1b;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;
                                                                    /*-----------------------*/
PROCEDURE       Purge_tokens;                                       /*  Purge_tokens  */
FPAR            tSet            TokenIdSet;                         /*-----------------------*/

        DCL     t               TokenId;
        START
        COMMENT'Delete a set of token ids.
                ';
          1b :  /* for t in tSet */
        DECISION tSet = Empty;
        (False): TASK   t  :=  Pick(tSet),
                        tSet  :=  Del(t, tSet);
                CALL    Delete_token(t);
                JOIN 1b;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;
                                                                    /*-----------------------*/
PROCEDURE       Output_buffer;                                      /*  Output_buffer  */
FPAR            b               BufferId,                           /*-----------------------*/
                p               PortalId;

        DCL     dp              DataPriority,
                pdu             PDUStruct,
                pid             PId;
        START
        COMMENT'Send the output signal representing an MCSPDU.
                The next must wait until PDU.ready is received.
                ';
        TASK    dp  :=  buffer(b)!dataPriority,
                pdu  :=  buffer(b)!pdu,
                pid  :=  portal(p)!pids(dp);
        DECISION p = upward and pdu!kind = SDin;
        (True):   TASK   pdu!kind  :=  SDrq;
        ELSE:ENDDECISION;
        DECISION pdu!kind;
        (PDin)   : OUTPUT  PDin(pdu)  TO pid;
        (EDrq)   : OUTPUT  EDrq(pdu)  TO pid;
        (MCrq)   : OUTPUT  MCrq(pdu) TO pid;
        (MCcf)   : OUTPUT  MCcf(pdu)  TO pid;
        (PCin)   : OUTPUT  PCin(pdu)  TO pid;
        (MTrq)   : OUTPUT  MTrq(pdu)  TO pid;
        (MTcf)   : OUTPUT  MTcf(pdu)  TO pid;
        (PTin)   : OUTPUT  PTin(pdu)   TO pid;
        (DPum) : OUTPUT  DPum(pdu)TO pid;
        (RJum) : OUTPUT  RJum(pdu)TO pid;
        (AUrq)   : OUTPUT  AUrq(pdu)  TO pid;
        (AUcf)   : OUTPUT  AUcf(pdu)  TO pid;
        (DUrq)   : OUTPUT  DUrq(pdu)  TO pid;
        (DUin)   : OUTPUT  DUin(pdu)  TO pid;
        (CJrq)   : OUTPUT  CJrq(pdu)  TO pid;
        (CJcf)   : OUTPUT  CJcf(pdu)   TO pid;
        (CLrq)   : OUTPUT  CLrq(pdu)  TO pid;
        (CCrq)   : OUTPUT  CCrq(pdu)  TO pid;
        (CCcf)   : OUTPUT  CCcf(pdu)  TO pid;
        (CDrq)   : OUTPUT  CDrq(pdu)  TO pid;
        (CDin)   : OUTPUT  CDin(pdu)  TO pid;
        (CArq)   : OUTPUT  CArq(pdu)  TO pid;
        (CAin)   : OUTPUT  CAin(pdu)  TO pid;
        (CErq)   : OUTPUT  CErq(pdu)  TO pid;
        (CEin)   : OUTPUT  CEin(pdu)  TO pid;
        (SDrq)   : OUTPUT  SDrq(pdu)  TO pid;
```

```
        (SDin)   : OUTPUT  SDin(pdu)  TO pid;
        (USrq)   : OUTPUT  USrq(pdu)  TO pid;
        (USin)   : OUTPUT  USin(pdu)  TO pid;
        (TGrq)   : OUTPUT  TGrq(pdu)  TO pid;
        (TGcf)   : OUTPUT  TGcf(pdu)  TO pid;
        (TIrq)   : OUTPUT  TIrq(pdu)   TO pid;
        (TIcf)   : OUTPUT  TIcf(pdu)   TO pid;
        (TVrq)   : OUTPUT  TVrq(pdu)  TO pid;
        (TVin)   : OUTPUT  TVin(pdu)   TO pid;
        (TVrs)   : OUTPUT  TVrs(pdu)   TO pid;
        (TVcf)   : OUTPUT  TVcf(pdu)   TO pid;
        (TPrq)   : OUTPUT  TPrq(pdu)   TO pid;
        (TPin)   : OUTPUT  TPin(pdu)   TO pid;
        (TRrq)   : OUTPUT  TRrq(pdu)   TO pid;
        (TRcf)   : OUTPUT  TRcf(pdu)   TO pid;
        (TTrq)   : OUTPUT  TTrq(pdu)   TO pid;
        (TTcf)   : OUTPUT  TTcf(pdu)   TO pid;
        ENDDECISION;
        RETURN;
        ENDPROCEDURE;

        /* Input transitions */

        DCL       p         PortalId,
                  dp        DataPriority,
                  pdu       PDUStruct,
                  pKind     PortalKind,
                  pids      PIdByPri,
                  reason    Reason;
        START
        COMMENT'The state machine contains a single state.
                 ';
        CALL      Initialize_resources;
        NEXTSTATE ~;

STATE ~; INPUT    Open.portal(p, pKind, pids);
        CALL      Open_portal(p, pKind, pids);
        NEXTSTATE -;

STATE ~; INPUT    Time.portal(p);
        CALL      Time_portal(p);
        NEXTSTATE -;

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

STATE ~; INPUT    Shut.portal(p);
        CALL      Shut_portal(p);
        DECISION pUsed = Empty;
        (False):NEXTSTATE -;
        (True): STOP;
        ENDDECISION;

STATE ~; INPUT    PDU.ready(dp);
        CALL      PDU_ready(dp);
        NEXTSTATE -;

STATE ~; INPUT    PDin(pdu); TASK pdu!kind  :=  PDin;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    EDrq(pdu); TASK pdu!kind  :=  EDrq;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    MCrq(pdu); TASK pdu!kind  :=  MCrq;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    MCcf(pdu); TASK pdu!kind  :=  MCcf;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    PCin(pdu); TASK pdu!kind  :=  PCin;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    MTrq(pdu); TASK pdu!kind  :=  MTrq;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    MTcf(pdu); TASK pdu!kind  :=  MTcf;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    PTin(pdu);  TASK pdu!kind  :=  PTin;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    DPum(pdu); TASK pdu!kind   :=  DPum; CALL Input_PDU(pdu); NEXTSTATE –;
STATE ~; INPUT    RJum(pdu); TASK pdu!kind :=  RJum; CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    AUrq(pdu); TASK pdu!kind  :=  AUrq;  CALL Input_PDU(pdu);   NEXTSTATE –;
```

```
STATE ~; INPUT    AUcf(pdu); TASK pdu!kind  :=  AUcf;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    DUrq(pdu); TASK pdu!kind  :=  DUrq;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    DUin(pdu); TASK pdu!kind  :=  DUin;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    CJrq(pdu); TASK pdu!kind  :=  CJrq;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    CJcf(pdu); TASK pdu!kind  :=  CJcf;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    CLrq(pdu); TASK pdu!kind  :=  CLrq;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    CCrq(pdu); TASK pdu!kind  :=  CCrq;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    CCcf(pdu); TASK pdu!kind  :=  CCcf;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    CDrq(pdu); TASK pdu!kind  :=  CDrq;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    CDin(pdu); TASK pdu!kind  :=  CDin;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    CArq(pdu); TASK pdu!kind  :=  CArq;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    CAin(pdu); TASK pdu!kind  :=  CAin;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    CErq(pdu); TASK pdu!kind  :=  CErq;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    CEin(pdu); TASK pdu!kind  :=  CEin;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    SDrq(pdu); TASK pdu!kind  :=  SDrq;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    SDin(pdu); TASK pdu!kind  :=  SDin;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    USrq(pdu); TASK pdu!kind  :=  USrq;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    USin(pdu); TASK pdu!kind  :=  USin;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    TGrq(pdu); TASK pdu!kind  :=  TGrq;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    TGcf(pdu); TASK pdu!kind  :=  TGcf;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    TIrq(pdu);  TASK pdu!kind  :=  TIrq;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    TIcf(pdu);  TASK pdu!kind  :=  TIcf;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    TVrq(pdu); TASK pdu!kind  :=  TVrq;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    TVin(pdu); TASK pdu!kind  :=  TVin;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    TVrs(pdu); TASK pdu!kind  :=  TVrs;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    TVcf(pdu); TASK pdu!kind  :=  TVcf;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    TPrq(pdu); TASK pdu!kind  :=  TPrq;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    TPin(pdu); TASK pdu!kind  :=  TPin;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    TRrq(pdu); TASK pdu!kind  :=  TRrq;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    TRcf(pdu); TASK pdu!kind  :=  TRcf;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    TTrq(pdu); TASK pdu!kind  :=  TTrq;  CALL Input_PDU(pdu);  NEXTSTATE –;
STATE ~; INPUT    TTcf(pdu); TASK pdu!kind  :=  TTcf;  CALL Input_PDU(pdu);  NEXTSTATE –;

ENDPROCESS;
```

Appendix V

SDL specification of the Endpoint process

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

```
PROCESS Endpoint;
FPAR    tcId          TCEndpointId,          /* incoming TC if not Null */
        localTSAP     TSAPAddress,           /* own transport address */
        remoteTSAP    TSAPAddress,           /* other transport address */
        givenQOS      TransportQOS,          /* target or offered QOS */
        minQOS        TransportQOS,          /* minimum acceptable QOS */
        parameters    DomainParameters;      /* values established or null */

        /* Data declarations */

DCL     control       PId,                   /* single Control process */
        domain        PId;                   /* selected Domain process */

        /* Input transitions */

        DCL     localDomain     DomainSelector,
                remoteDomain    DomainSelector,
                upward          Boolean,
                targetParms     DomainParameters,
                minParms        DomainParameters,
                maxParms        DomainParameters,
                userData        UserData,
                result          Result,
                ccId            Natural,
                label           Natural,
                tsdu            TSDU,
                dp              DataPriority,
                pdu             PDUStruct;
```

```
        START
        COMMENT'State machine:                    NEXTSTATE
                        1  connecting          * . 2 . . . 6
                        2  connbusy            * . 2 3 . 5 6
                        3  connready             . 2 3 . 5 6
                        4  busy                  . . . 4 5 6
                        5  ready                 . . . 4 5 6
                        6  disconnected          . . . . . 6
                ';
        TASK    control  :=  PARENT,
                label  :=  0;
        DECISION tcId = Null;
        (True):  OUTPUT  T.Connect.request(label, localTSAP, remoteTSAP,
                                                  givenQOS, minQOS);
                NEXTSTATE connecting;
        (False): OUTPUT  T.Connect.response(tcId, givenQOS);
                OUTPUT  T.ready(tcId);
                NEXTSTATE connbusy;
        ENDDECISION;

STATE *;
        INPUT    Exit;
        STOP;

STATE *;
        INPUT    T.Disconnect.indication(label, tcId);
        OUTPUT   Quit TO control;
        NEXTSTATE disconnected;

STATE connecting;
        INPUT    T.Connect.confirm(label, tcId, givenQOS);
        OUTPUT   T.ready(tcId);
        NEXTSTATE connbusy;

STATE connecting;
        SAVE     *;  /* await outcome of TC */

STATE connbusy, connready, busy, ready;
        INPUT    *;
        OUTPUT   T.Disconnect.request(tcId);
        OUTPUT   Quit TO control;
        NEXTSTATE disconnected;

STATE connbusy;
        INPUT    T.ready(tcId);
        NEXTSTATE connready;

STATE connbusy;
        SAVE     Connect.Initial, Connect.Response, Connect.Additional, Connect.Result;

STATE connready;
        INPUT    Connect.Initial(localDomain, remoteDomain, upward,
                 targetParms, minParms, maxParms, userData);
        TASK'tsdu  :=  encode Connect-Initial using BER';
          1b :
        OUTPUT   T.Data.request(tcId, tsdu);
        NEXTSTATE connbusy;

STATE connready;
        INPUT    Connect.Response(result, ccId, parameters, userData);
        TASK'tsdu  :=  encode Connect-Response using BER';
        JOIN 1b;

STATE connready;
        INPUT    Connect.Additional(ccId, dp);
        TASK'tsdu  :=  encode Connect-Additional using BER';
        JOIN 1b;

STATE connready;
        INPUT    Connect.Result(result);
        TASK'tsdu  :=  encode Connect-Result using BER';
        JOIN 1b;
```

```
STATE connbusy, connready;
        INPUT     T.Data.indication(tcId, tsdu);
        TASK'connect MCSPDU  :=  decode tsdu using BER';
        DECISION'connect MCSPDU';
        ('Connect-Initial'):
                OUTPUT  Connect.Initial(localDomain, remoteDomain, upward,
                        targetParms, minParms, maxParms, userData) TO control;
                NEXTSTATE -;
        ('Connect-Response'):
                OUTPUT  Connect.Response(result, ccId, parameters, userData) TO control;
                NEXTSTATE -;
        ('Connect-Additional'):
                OUTPUT  Connect.Additional(ccId, dp) TO control;
                NEXTSTATE -;
        ('Connect-Result'):
                OUTPUT  Connect.Result(result) TO control;
                NEXTSTATE -;
        ELSE:
                OUTPUT  T.Disconnect.request(tcId);
                OUTPUT  Quit TO control;
                NEXTSTATE disconnected;
        ENDDECISION;

STATE connbusy;
        INPUT     PDU.ready(dp);
        TASK      domain  :=  SENDER;
        OUTPUT  T.ready(tcId);
        NEXTSTATE ready;

STATE connready;
        INPUT     PDU.ready(dp);
        TASK      domain  :=  SENDER;
        OUTPUT  PDU.ready(dp) TO domain;
        OUTPUT  T.ready(tcId);
        NEXTSTATE ready;

STATE busy;
        INPUT     PDU.ready(dp);
        OUTPUT  T.ready(tcId);
        NEXTSTATE ready;

STATE busy, ready;
        INPUT     T.ready(tcId);
        OUTPUT  PDU.ready(dp) TO domain;
        NEXTSTATE -;

STATE busy, ready;
        INPUT   PDin(pdu); TASK pdu!kind  :=  PDin;   JOIN  2f;
        INPUT   EDrq(pdu); TASK pdu!kind  :=  EDrq;   JOIN  2f;
        INPUT   MCrq(pdu); TASK pdu!kind  :=  MCrq;   JOIN  2f;
        INPUT   MCcf(pdu); TASK pdu!kind  :=  MCcf;   JOIN  2f;
        INPUT   PCin(pdu); TASK pdu!kind  :=  PCin;   JOIN  2f;
        INPUT   MTrq(pdu); TASK pdu!kind  :=  MTrq;   JOIN  2f;
        INPUT   MTcf(pdu); TASK pdu!kind  :=  MTcf;   JOIN  2f;
        INPUT   PTin(pdu);  TASK pdu!kind  :=  PTin;   JOIN  2f;
        INPUT   DPum(pdu); TASK pdu!kind  :=  DPum;  JOIN  2f;
        INPUT   RJum(pdu); TASK pdu!kind  :=  RJum;  JOIN  2f;
        INPUT   AUrq(pdu); TASK pdu!kind  :=  AUrq;  JOIN   2f;
        INPUT   AUcf(pdu); TASK pdu!kind  :=  AUcf;  JOIN   2f;
        INPUT   DUrq(pdu); TASK pdu!kind  :=  DUrq;  JOIN   2f;
        INPUT   DUin(pdu); TASK pdu!kind  :=  DUin;  JOIN   2f;
        INPUT   CJrq(pdu); TASK pdu!kind  :=  CJrq;  JOIN   2f;
        INPUT   CJcf(pdu);  TASK pdu!kind  :=  CJcf;  JOIN   2f;
        INPUT   CLrq(pdu); TASK pdu!kind  :=  CLrq;  JOIN   2f;
        INPUT   CCrq(pdu); TASK pdu!kind  :=  CCrq;  JOIN   2f;
        INPUT   CCcf(pdu); TASK pdu!kind  :=  CCcf;  JOIN   2f;
        INPUT   CDrq(pdu); TASK pdu!kind  :=  CDrq;  JOIN   2f;
        INPUT   CDin(pdu); TASK pdu!kind  :=  CDin;  JOIN   2f;
```

```
        INPUT   CArq(pdu); TASK pdu!kind  :=  CArq; JOIN  2f;
        INPUT   CAin(pdu); TASK pdu!kind  :=  CAin; JOIN  2f;
        INPUT   CErq(pdu); TASK pdu!kind  :=  CErq; JOIN  2f;
        INPUT   CEin(pdu); TASK pdu!kind  :=  CEin; JOIN  2f;
        INPUT   SDrq(pdu); TASK pdu!kind  :=  SDrq; JOIN  2f;
        INPUT   SDin(pdu); TASK pdu!kind  :=  SDin; JOIN  2f;
        INPUT   USrq(pdu); TASK pdu!kind  :=  USrq; JOIN  2f;
        INPUT   USin(pdu); TASK pdu!kind  :=  USin; JOIN  2f;
        INPUT   TGrq(pdu); TASK pdu!kind  :=  TGrq; JOIN  2f;
        INPUT   TGcf(pdu); TASK pdu!kind  :=  TGcf; JOIN  2f;
        INPUT   Tlrq(pdu);  TASK pdu!kind  :=  Tlrq;  JOIN  2f;
        INPUT   Tlcf(pdu);  TASK pdu!kind  :=  Tlcf;  JOIN  2f;
        INPUT   TVrq(pdu); TASK pdu!kind  :=  TVrq; JOIN  2f;
        INPUT   TVin(pdu); TASK pdu!kind  :=  TVin; JOIN  2f;
        INPUT   TVrs(pdu); TASK pdu!kind  :=  TVrs; JOIN  2f;
        INPUT   TVcf(pdu); TASK pdu!kind  :=  TVcf; JOIN  2f;
        INPUT   TPrq(pdu); TASK pdu!kind  :=  TPrq; JOIN  2f;
        INPUT   TPin(pdu); TASK pdu!kind  :=  TPin; JOIN  2f;
        INPUT   TRrq(pdu); TASK pdu!kind  :=  TRrq; JOIN  2f;
        INPUT   TRcf(pdu); TASK pdu!kind  :=  TRcf; JOIN  2f;
        INPUT   TTrq(pdu); TASK pdu!kind  :=  TTrq; JOIN  2f;
        INPUT   TTcf(pdu); TASK pdu!kind  :=  TTcf;
          2f :
        TASK'tsdu  :=  encode pdu using BER or PER,
              depending on parameters!protocolVersion';
        OUTPUT  T.Data.request(tcId, tsdu);
        NEXTSTATE -;

STATE ready;
        INPUT     T.Data.indication(tcId, tsdu);
        TASK'pdu  :=  decode tsdu using BER or PER,
              depending on parameters!protocolVersion';
        DECISION pdu!kind;
        (RJum):  OUTPUT  RJum(pdu) TO control;
                 OUTPUT  T.ready(tcId);
                 NEXTSTATE -;
        (PDin):   OUTPUT  PDin(pdu)  TO domain;  NEXTSTATE busy;
        (EDrq):   OUTPUT  EDrq(pdu)  TO domain;  NEXTSTATE busy;
        (MCrq):   OUTPUT  MCrq(pdu) TO domain;  NEXTSTATE busy;
        (MCcf):   OUTPUT  MCcf(pdu) TO domain;  NEXTSTATE busy;
        (PCin):   OUTPUT  PCin(pdu)  TO domain;  NEXTSTATE busy;
        (MTrq):   OUTPUT  MTrq(pdu) TO domain;  NEXTSTATE busy;
        (MTcf):   OUTPUT  MTcf(pdu)  TO domain;  NEXTSTATE busy;
        (PTin):   OUTPUT  PTin(pdu)  TO domain;  NEXTSTATE busy;
        (DPum):  OUTPUT  DPum(pdu) TO domain; NEXTSTATE busy;
        (AUrq):   OUTPUT  AUrq(pdu)  TO domain;  NEXTSTATE busy;
        (AUcf):   OUTPUT  AUcf(pdu)  TO domain;  NEXTSTATE busy;
        (DUrq):   OUTPUT  DUrq(pdu)  TO domain;  NEXTSTATE busy;
        (DUin):   OUTPUT  DUin(pdu)  TO domain;  NEXTSTATE busy;
        (CJrq):   OUTPUT  CJrq(pdu)  TO domain;  NEXTSTATE busy;
        (CJcf):   OUTPUT  CJcf(pdu)  TO domain;  NEXTSTATE busy;
        (CLrq):   OUTPUT  CLrq(pdu)  TO domain;  NEXTSTATE busy;
        (CCrq):   OUTPUT  CCrq(pdu)  TO domain;  NEXTSTATE busy;
        (CCcf):   OUTPUT  CCcf(pdu)  TO domain;  NEXTSTATE busy;
        (CDrq):   OUTPUT  CDrq(pdu)  TO domain;  NEXTSTATE busy;
        (CDin):   OUTPUT  CDin(pdu)  TO domain;  NEXTSTATE busy;
        (CArq):   OUTPUT  CArq(pdu)  TO domain;  NEXTSTATE busy;
        (CAin):   OUTPUT  CAin(pdu)  TO domain;  NEXTSTATE busy;
        (CErq):   OUTPUT  CErq(pdu)  TO domain;  NEXTSTATE busy;
        (CEin):   OUTPUT  CEin(pdu)  TO domain;  NEXTSTATE busy;
        (SDrq):   OUTPUT  SDrq(pdu)  TO domain;  NEXTSTATE busy;
        (SDin):   OUTPUT  SDin(pdu)  TO domain;  NEXTSTATE busy;
        (USrq):   OUTPUT  USrq(pdu)  TO domain;  NEXTSTATE busy;
        (USin):   OUTPUT  USin(pdu)  TO domain;  NEXTSTATE busy;
        (TGrq):   OUTPUT  TGrq(pdu)  TO domain;  NEXTSTATE busy;
        (TGcf):   OUTPUT  TGcf(pdu)  TO domain;  NEXTSTATE busy;
```

```
        (TIrq):   OUTPUT  TIrq(pdu)   TO domain;  NEXTSTATE busy;
        (TIcf):   OUTPUT  TIcf(pdu)   TO domain;  NEXTSTATE busy;
        (TVrq):   OUTPUT  TVrq(pdu)   TO domain;  NEXTSTATE busy;
        (TVin):   OUTPUT  TVin(pdu)   TO domain;  NEXTSTATE busy;
        (TVrs):   OUTPUT  TVrs(pdu)   TO domain;  NEXTSTATE busy;
        (TVcf):   OUTPUT  TVcf(pdu)   TO domain;  NEXTSTATE busy;
        (TPrq):   OUTPUT  TPrq(pdu)   TO domain;  NEXTSTATE busy;
        (TPin):   OUTPUT  TPin(pdu)   TO domain;  NEXTSTATE busy;
        (TRrq):   OUTPUT  TRrq(pdu)   TO domain;  NEXTSTATE busy;
        (TRcf):   OUTPUT  TRcf(pdu)   TO domain;  NEXTSTATE busy;
        (TTrq):   OUTPUT  TTrq(pdu)   TO domain;  NEXTSTATE busy;
        (TTcf):   OUTPUT  TTcf(pdu)   TO domain;  NEXTSTATE busy;
        ELSE:
                TASK'pdu!initialOctets  :=  truncate tsdu';
                TASK'pdu!diagnostic  :=  DC_invalid_?ER_encoding';
                OUTPUT  RJum(pdu) TO domain;
                NEXTSTATE busy;
        ENDDECISION;

STATE disconnected;
        INPUT   Quit;
        OUTPUT  Quit TO control;
        NEXTSTATE disconnected;

ENDPROCESS;
```

Appendix VI

SDL specification of the Attachment process

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

```
PROCESS Attachment;
FPAR    label           Natural,                        /* for MCS.Attach.User.confirm */
        parameters      DomainParameters;               /* values established in domain */

        /* Type definitions */

NEWTYPE         MCSRequest
STRUCT
        kind            PDUKind;                        /* SDrq, USrq, CArq, CErq */
        channelId       ChannelId;                      /* parameter of request */
        segmentation    Segmentation;                   /* parameter of request */
        userData        UserData;                       /* parameter of request */
        offset          Integer;                        /* octets sent so far */
        userIds         UserIdSet;                      /* remaining to affect */
ENDNEWTYPE;

NEWTYPE         PrioritySet             SetOf(DataPriority);
ENDNEWTYPE;

NEWTYPE         MCSRequestByPri         Array(DataPriority, MCSRequest);
ENDNEWTYPE;
NEWTYPE         PDUStructByPri          Array(DataPriority, PDUStruct);
ENDNEWTYPE;

        /* Data declarations */

DCL     maId            MCSAttachmentId,                /* this Attachment process */
        control         PId,                            /* single Control process */
        domain          PId;                            /* selected Domain process */

DCL     user            UserId;                         /* unique id of attached user */

DCL     cJoined         ChannelIdSet,                   /* channels the user has joined */
        cConvened       ChannelIdSet,                   /* channels the user has convened */
        cAdmitted       ChannelIdSet;                   /* channels user was admitted to */

DCL     tPossessed      TokenIdSet,                     /* tokens grabbed or inhibited */
        tRecipient      TokenIdSet;                     /* tokens given waiting response */

DCL     uPending        PrioritySet,                    /* request is pending at 0..3 */
```

```
        mcsreq          MCSRequestByPri,                /* content of segmented request */
        dReady          PrioritySet;                    /* domain MCSPDU allowed 0..? */

DCL     dPending        PrioritySet,                    /* SDin or USin pending at 0..3 */
        mcspdu          PDUStructByPri,                 /* content of the domain MCSPDU */
        uReady          PrioritySet;                    /* data indication allowed 0..3 */

        /* Procedure decomposition */

        /*      Segment_request             (dp)
                Indicate_data
                Track_token                 (t, status)     */
                                                        /*------------------------------*/
PROCEDURE       Segment_request;                        /*  Segment_request  */
FPAR            dp      DataPriority;                   /*------------------------------*/

        DCL     udp     DataPriority,
                req     MCSRequest,
                pdu     PDUStruct,
                b       Boolean,
                m       Integer,
                n       Integer,
                u       UserId;
        START
        COMMENT'Feed domain process the next segment of user data
                  or the next subset of private channel user ids,
                  if ready, for the specified transport priority.
                ';
        DECISION dp in dReady;
        (False):RETURN;
        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;
                 (False): TASK    dp  :=  dp + 1;
                          JOIN 1b;
                 ELSE:ENDDECISION;
        ENDDECISION;
        TASK    dPending  :=  Del(dp, dPending),
                pdu  :=  mcspdu(dp);
        DECISION pdu!kind;
        (SDin):  OUTPUT  MCS.Send.Data.indication(maId, pdu!channelId,
                          pdu!dataPriority, pdu!initiator, pdu!segmentation, pdu!userData);
        (USin):  OUTPUT  MCS.Uniform.Send.Data.indication(maId, pdu!channelId,
                          pdu!dataPriority, pdu!initiator, pdu!segmentation, pdu!userData);
        ENDDECISION;
        TASK    uReady  :=  Del(dp, uReady),
                dp  :=  IF dp < parameters!numPriorities THEN dp
                                 ELSE parameters!numPriorities - 1 FI;
        OUTPUT  PDU.ready(dp) TO domain;
        JOIN 1b;
        ENDPROCEDURE;

                                                                  /*----------------------*/
PROCEDURE               Track_token;                              /*  Track_token  */
FPAR                    t       TokenId,                          /*----------------------*/
                        status  TokenStatus;
```

```
        START
        COMMENT'Condense status into possessed or recipient.
                ';
        TASK    tPossessed  :=  Del(t, tPossessed),
                tRecipient  :=  Del(t, tRecipient);
        DECISION status;
        (SelfGrabbed, SelfInhibited, SelfGiving):
                TASK    tPossessed  :=  Incl(t, tPossessed);
        (SelfRecipient):
                TASK    tRecipient  :=  Incl(t, tRecipient);
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;

        /* Input transitions */

        DCL     dp              DataPriority,
                pdu             PDUStruct,
                c               ChannelId,
                cSet            ChannelIdSet,
                t               TokenId,
                u               UserId,
                uSet            UserIdSet,
                segmentation    Segmentation,
                userData        UserData,
                result          Result,
                kind            PDUKind,
                reason          Reason;
        START
        COMMENT'State machine:                                          NEXTSTATE
                        1  initial              * . 2 . . . 6
                        2  attaching              . 2 3 4 . 6
                        3  busy                   . . 3 4 5 6
                        4  ready                  . . 3 4 . 6
                        5  detaching               . . . . 5 6
                        6  detached               . . . . . 6
                ';
        TASK    mald  :=  SELF,
                control  :=  PARENT,
                user  :=  0;
        NEXTSTATE initial;

STATE *;
        INPUT   Exit;
        STOP;

STATE initial, attaching;
        INPUT   *;
        OUTPUT  MCS.Attach.User.confirm(label, RT_unspecified_failure, mald, user);
        OUTPUT  Quit TO control;
        NEXTSTATE detached;

STATE initial, attaching;
        INPUT   Quit;
        OUTPUT  MCS.Attach.User.confirm(label, RT_domain_disconnected, mald, user);
        OUTPUT  Quit TO control;
        NEXTSTATE detached;

STATE initial;
        INPUT   PDU.ready(dp);
        TASK    domain  :=  SENDER;
        DECISION dp = 0;
        (False):  TASK   dReady  :=  Incl(dp, dReady);
                  NEXTSTATE -;
        ELSE:ENDDECISION;
        OUTPUT  PDU.ready(0) TO domain;
        OUTPUT  AUrq(pdu) TO domain;
        NEXTSTATE attaching;
```

```
STATE attaching;
        INPUT     PDU.ready(dp);
        TASK      dReady  :=  Incl(dp, dReady);
        NEXTSTATE -;

STATE attaching;
        INPUT     PCin(pdu), PTin(pdu), DUin(pdu);
        /* no action */
        NEXTSTATE -;

STATE attaching;
        INPUT     AUcf(pdu);
        TASK      user  :=  pdu!initiator;
        OUTPUT  MCS.Attach.User.confirm(label, pdu!result, mald, user);
        DECISION pdu!result = RT_successful;
        (False):   OUTPUT  Quit TO control;
                   NEXTSTATE detached;
        (True):    TASK    dp  :=  0;
                     1b :    /* for dp = 0..3 */
                   DECISION dp < 4;
                   (True):  OUTPUT  MCS.ready(mald, dp);
                            DECISION dp < parameters!numPriorities;
                            (True):  OUTPUT  PDU.ready(dp) TO domain;
                            ELSE:ENDDECISION;
                            TASK    dp  :=  dp + 1;
                            JOIN 1b;
                   ELSE:ENDDECISION;
                   DECISION 0 in dReady;
                   (False): NEXTSTATE busy;
                   (True):  NEXTSTATE ready;
                   ENDDECISION;
        ENDDECISION;

STATE busy, ready;
        INPUT     Quit;
        OUTPUT  MCS.Detach.User.indication(mald, user, RN_provider_initiated);
        OUTPUT  Quit TO control;
        NEXTSTATE detached;

STATE busy, ready;
        INPUT     MCS.ready(mald, dp);
        TASK      uReady  :=  Incl(dp, uReady);
        CALL      Indicate_data;
        NEXTSTATE -;

STATE busy;
        INPUT     MCS.Detach.User.request(mald);
        TASK      reason  :=  RN_user_requested;
        NEXTSTATE detaching;

STATE busy;
        SAVE      *;  /* defer MCS other */

STATE ready;
        INPUT     MCS.Detach.User.request(mald);
        TASK      pdu!reason  :=  RN_user_requested,
                  pdu!userIds  :=  Incl(user, Empty);
        OUTPUT  DUrq(pdu) TO domain;
        NEXTSTATE detached;

STATE ready;
        INPUT     MCS.Channel.Join.request(mald, c);
        TASK      pdu!initiator  :=  user,
                  pdu!channelId  :=  c;
        OUTPUT  CJrq(pdu) TO domain;
          2b :
        TASK      dReady  :=  Del(0, dReady);
        NEXTSTATE busy;
```

```
STATE ready;
        INPUT    MCS.Channel.Leave.request(maId, c);
        TASK     cJoined := Del(c, cJoined),
                 pdu!channelIds := Incl(c, Empty);
        OUTPUT   CLrq(pdu) TO domain;
        JOIN 2b;
STATE ready;
        INPUT    MCS.Channel.Convene.request(maId);
        TASK     pdu!initiator := user;
        OUTPUT   CCrq(pdu) TO domain;
        JOIN 2b;
STATE ready;
        INPUT    MCS.Channel.Disband.request(maId, c);
        DECISION c in cConvened;
        (True):  TASK   cAdmitted := Del(c, cAdmitted),
                        cJoined := Del(c, cJoined);
        ELSE:ENDDECISION;
        TASK     cConvened := Del(c, cConvened),
                 pdu!initiator := user,
                 pdu!channelId := c;
        OUTPUT   CDrq(pdu) TO domain;
        JOIN 2b;
STATE ready;
        INPUT    MCS.Channel.Admit.request(maId, c, uSet);
        TASK     kind := CArq;
        JOIN 3f;
STATE ready;
        INPUT    MCS.Channel.Expel.request(maId, c, uSet);
        TASK     kind := CErq;
          3f :
        TASK     mcsreq(0)!kind := kind,
                 mcsreq(0)!channelId := c,
                 mcsreq(0)!userIds := uSet,
                 uPending := Incl(0, uPending);
        CALL     Segment_request(0);
        DECISION 0 in dReady;
        (False):NEXTSTATE busy;
        (True): NEXTSTATE ready;
        ENDDECISION;
STATE ready;
        INPUT    MCS.Send.Data.request(maId, c, dp, segmentation, userData);
        TASK     kind := SDrq;
        JOIN 4f;
STATE ready;
        INPUT    MCS.Uniform.Send.Data.request(maId, c, dp, segmentation, userData);
        TASK     kind := USrq;
          4f :
        DECISION dp >= 4 or dp in uPending;
        (True):  OUTPUT  MCS.Detach.User.indication(maId, user, RN_provider_initiated);
                 TASK    pdu!reason := RN_provider_initiated,
                         pdu!userIds := Incl(user, Empty);
                 OUTPUT  DUrq(pdu) TO domain;
                 NEXTSTATE detached;
        (False): TASK    mcsreq(dp)!kind := kind,
                         mcsreq(dp)!channelId := c,
                         mcsreq(dp)!segmentation := segmentation,
                         mcsreq(dp)!userData := userData,
                         mcsreq(dp)!offset := 0,
                         uPending := Incl(dp, uPending),
                         dp := IF dp < parameters!numPriorities THEN dp
                                       ELSE parameters!numPriorities - 1 FI;
                 CALL    Segment_request(dp);
                 DECISION 0 in dReady;
                 (False): NEXTSTATE busy;
```

```
                (True):  NEXTSTATE ready;
                ENDDECISION;
        ENDDECISION;

STATE ready;
        INPUT    MCS.Token.Grab.request(mald, t);
        TASK     pdu!initiator  :=  user,
                 pdu!tokenId  :=  t;
        OUTPUT   TGrq(pdu) TO domain;
        JOIN 2b;

STATE ready;
        INPUT    MCS.Token.Inhibit.request(mald, t);
        TASK     pdu!initiator  :=  user,
                 pdu!tokenId  :=  t;
        OUTPUT   TIrq(pdu) TO domain;
        JOIN 2b;

STATE ready;
        INPUT    MCS.Token.Give.request(mald, t, u);
        DECISION u >= 1001;
        (False):   OUTPUT  MCS.Token.Give.confirm(mald, t, RT_no_such_user);
                   NEXTSTATE -;
        (True):    TASK      pdu!initiator  :=  user,
                             pdu!tokenId  :=  t,
                             pdu!recipient  :=  u;
                   OUTPUT  TVrq(pdu) TO domain;
                   JOIN 2b;
        ENDDECISION;

STATE ready;
        INPUT    MCS.Token.Give.response(mald, t, result);
        DECISION result  = RT_successful;
        (False):TASK       result  :=  RT_user_rejected;
        ELSE:ENDDECISION;
        DECISION t in tRecipient and result = RT_successful;
        (True):    TASK    tPossessed  :=  Incl(t, tPossessed);
        ELSE:ENDDECISION;
        TASK     tRecipient  :=  Del(t, tRecipient),
                 pdu!result  :=  result,
                 pdu!recipient  :=  user,
                 pdu!tokenId  :=  t;
        OUTPUT   TVrs(pdu) TO domain;
        JOIN 2b;

STATE ready;
        INPUT    MCS.Token.Please.request(mald, t);
        TASK     pdu!initiator  :=  user,
                 pdu!tokenId  :=  t;
        OUTPUT   TPrq(pdu) TO domain;
        JOIN 2b;

STATE ready;
        INPUT    MCS.Token.Release.request(mald, t);
        TASK     tPossessed  :=  Del(t, tPossessed),
                 pdu!initiator  :=  user,
                 pdu!tokenId  :=  t;
        OUTPUT   TRrq(pdu) TO domain;
        JOIN 2b;

STATE ready;
        INPUT    MCS.Token.Test.request(mald, t);
        TASK     pdu!initiator  :=  user,
                 pdu!tokenId  :=  t;
        OUTPUT   TTrq(pdu) TO domain;
        JOIN 2b;

STATE busy, ready;
        INPUT    PDU.ready(dp);
        TASK     dReady  :=  Incl(dp, dReady);
        CALL     Segment_request(dp);
```

```
        DECISION 0 in dReady;
        (False):   NEXTSTATE busy;
        (True):    NEXTSTATE ready;
        ENDDECISION;

STATE busy, ready;
        INPUT      PCin(pdu);
        TASK       reason  :=  RN_channel_purged;
        DECISION user in pdu!detachUserIds;
        (True):    OUTPUT  MCS.Detach.User.indication(mald, user, reason);
                   OUTPUT  Quit TO control;
                   NEXTSTATE detached;
        (False):   TASK    uSet  :=  pdu!detachUserIds;
                     5b :  /* for u in uSet */
                   DECISION uSet = Empty;
                   (False): TASK    u  :=  Pick(uSet),
                                    uSet  :=  Del(u, uSet);
                            OUTPUT  MCS.Detach.User.indication(mald, u, reason);
                            JOIN 5b;
                   ELSE:ENDDECISION;
                   TASK    cSet  :=  pdu!purgeChannelIds;
                     6b :  /* for c in cSet */
                   DECISION cSet = Empty;
                   (False): TASK    c  :=  Pick(cSet),
                                    cSet  :=  Del(c, cSet);
                            DECISION c in cConvened;
                            (True):  TASK cConvened  :=  Del(c, cConvened),
                                          cAdmitted  :=  Del(c, cAdmitted),
                                          cJoined  :=  Del(c, cJoined);
                                     OUTPUT  MCS.Channel.Disband.indication(mald, c, reason);
                            ELSE:ENDDECISION;
                            DECISION c in cAdmitted;
                            (True):  TASK cAdmitted  :=  Del(c, cAdmitted),
                                          cJoined  :=  Del(c, cJoined);
                                     OUTPUT  MCS.Channel.Expel.indication(mald, c, reason);
                            ELSE:ENDDECISION;
                            DECISION c in cJoined;
                            (True):  TASK cJoined  :=  Del(c, cJoined);
                                     OUTPUT  MCS.Channel.Leave.indication(mald, c, reason);
                            ELSE:ENDDECISION;
                            JOIN 6b;
                   ELSE:ENDDECISION;
                   OUTPUT  PDU.ready(0) TO domain;
                   NEXTSTATE -;
        ENDDECISION;

STATE busy;
        INPUT      PTin(pdu);
        DECISION (pdu!purgeTokenIds and (tPossessed or tRecipient)) = Empty;
        (False):   OUTPUT  MCS.Detach.User.indication(mald, user, RN_token_purged);
                   TASK    reason  :=  RN_token_purged;
                   NEXTSTATE detaching;
        (True):    OUTPUT  PDU.ready(0) TO domain;
                   NEXTSTATE -;
        ENDDECISION;

STATE ready;
        INPUT      PTin(pdu);
        DECISION (pdu!purgeTokenIds and (tPossessed or tRecipient)) = Empty;
        (False):   OUTPUT  MCS.Detach.User.indication(mald, user, RN_token_purged);
                   TASK    pdu!reason  :=  RN_token_purged,
                           pdu!userIds  :=  Incl(user, Empty);
                   OUTPUT  DUrq(pdu) TO domain;
                   NEXTSTATE detached;
        (True):    OUTPUT  PDU.ready(0) TO domain;
                   NEXTSTATE -;
        ENDDECISION;
```

```
STATE busy, ready;
        INPUT     AUcf(pdu);
        OUTPUT  MCS.Detach.User.indication(mald, user, RN_unspecified);
        OUTPUT  Quit TO control;
        NEXTSTATE detached;

STATE busy, ready;
        INPUT     DUin(pdu);
        DECISION user in pdu!userIds;
        (True):    OUTPUT  MCS.Detach.User.indication(mald, user, pdu!reason);
                   OUTPUT  Quit TO control;
        NEXTSTATE detached;
        (False):   TASK    uSet  :=  pdu!userIds;
                     7b :    /* for u in uSet */
                   DECISION uSet = Empty;
                   (False): TASK    u  :=  Pick(uSet),
                                    uSet  :=  Del(u, uSet);
                            OUTPUT  MCS.Detach.User.indication(mald, u, pdu!reason);
                            JOIN 7b;
                   ELSE:ENDDECISION;
                   OUTPUT  PDU.ready(0) TO domain;
                   NEXTSTATE -;
        ENDDECISION;

STATE busy, ready;
        INPUT     CJcf(pdu);
        TASK      c  :=  pdu!channelId;
        DECISION pdu!result = RT_successful;
        (True):    TASK    cJoined  :=  Incl(c, cJoined);
        ELSE:ENDDECISION;
        OUTPUT  MCS.Channel.Join.confirm(mald, pdu!requested, pdu!result, c);
          8b :
        OUTPUT  PDU.ready(0) TO domain;
        NEXTSTATE -;

STATE busy, ready;
        INPUT     CCcf(pdu);
        TASK      c  :=  pdu!channelId;
        DECISION pdu!result = RT_successful;
        (True):    TASK    cConvened  :=  Incl(c, cConvened),
                           cAdmitted  :=  Incl(c, cAdmitted);
        ELSE:ENDDECISION;
        OUTPUT  MCS.Channel.Convene.confirm(mald, pdu!result, c);
        JOIN 8b;

STATE busy, ready;
        INPUT     CDin(pdu);
        TASK      c  :=  pdu!channelId,
                  reason  :=  RN_channel_disbanded;
        DECISION c in cConvened;
        (True):    TASK    cConvened  :=  Del(c, cConvened),
                           cAdmitted  :=  Del(c, cAdmitted),
                           cJoined  :=  Del(c, cJoined);
                   OUTPUT  MCS.Channel.Disband.indication(mald, c, reason);
        ELSE:ENDDECISION;
        DECISION c in cAdmitted;
        (True):    TASK    cAdmitted  :=  Del(c, cAdmitted),
                           cJoined  :=  Del(c, cJoined);
                   OUTPUT  MCS.Channel.Expel.indication(mald, c, reason);
        ELSE:ENDDECISION;
        JOIN 8b;

STATE busy, ready;
        INPUT     CAin(pdu);
        TASK      c  :=  pdu!channelId,
                  cAdmitted  :=  Incl(c, cAdmitted);
        OUTPUT  MCS.Channel.Admit.indication(mald, c, pdu!initiator);
        JOIN 8b;
```

```
STATE busy, ready;
        INPUT    CEin(pdu);
        TASK     c := pdu!channelId,
                 reason := RN_user_requested;
        DECISION c in cAdmitted;
        (True):  TASK   cAdmitted := Del(c, cAdmitted),
                        cJoined := Del(c, cJoined);
                 OUTPUT MCS.Channel.Expel.indication(maId, c, reason);
        ELSE:ENDDECISION;
        JOIN 8b;

STATE busy, ready;
        INPUT    SDin(pdu);
        TASK     pdu!kind := SDin;
        JOIN 9f;

STATE busy, ready;
        INPUT    USin(pdu);
        TASK     pdu!kind := USin;
          9f :
        TASK     c := pdu!channelId,
                 dp := pdu!dataPriority;
        DECISION c in cJoined;
        (True):  TASK   mcspdu(dp) := pdu,
                        dPending := Incl(dp, dPending);
                 CALL   Indicate_data;
        (False): TASK   dp := IF dp < parameters!numPriorities THEN dp
                                        ELSE parameters!numPriorities - 1 FI;
                 OUTPUT PDU.ready(dp) TO domain;
        ENDDECISION;
        NEXTSTATE -;

STATE busy, ready;
        INPUT    TGcf(pdu);
        CALL     Track_token(pdu!tokenId, pdu!tokenStatus);
        OUTPUT   MCS.Token.Grab.confirm(maId, pdu!tokenId, pdu!result);
        JOIN 8b;

STATE busy, ready;
        INPUT    TIcf(pdu);
        CALL     Track_token(pdu!tokenId, pdu!tokenStatus);
        OUTPUT   MCS.Token.Inhibit.confirm(maId, pdu!tokenId, pdu!result);
        JOIN 8b;

STATE busy, ready;
        INPUT    TVin(pdu);
        TASK     tRecipient := Incl(pdu!tokenId, tRecipient);
        OUTPUT   MCS.Token.Give.indication(maId, pdu!tokenId, pdu!initiator);
        JOIN 8b;

STATE busy, ready;
        INPUT    TVcf(pdu);
        CALL     Track_token(pdu!tokenId, pdu!tokenStatus);
        OUTPUT   MCS.Token.Give.confirm(maId, pdu!tokenId, pdu!result);
        JOIN 8b;

STATE busy, ready;
        INPUT    TPin(pdu);
        TASK     t := pdu!tokenId;
        DECISION t in tPossessed or t in tRecipient;
        (True):  OUTPUT MCS.Token.Please.indication(maId, t, pdu!initiator);
        ELSE:ENDDECISION;
        JOIN 8b;

STATE busy, ready;
        INPUT    TRcf(pdu);
        CALL     Track_token(pdu!tokenId, pdu!tokenStatus);
        OUTPUT   MCS.Token.Release.confirm(maId, pdu!tokenId, pdu!result);
        JOIN 8b;
```

```
STATE busy, ready;
        INPUT     TTcf(pdu);
        OUTPUT  MCS.Token.Test.confirm(mald, pdu!tokenId, pdu!tokenStatus);
        JOIN 8b;

STATE detaching, detached;
        INPUT     *;
        NEXTSTATE -;

STATE detaching, detached;
        INPUT     Quit;
        OUTPUT  Quit TO control;
        NEXTSTATE detached;

STATE detaching;
        INPUT     PDU.ready(dp);
        DECISION dp = 0;
        (False):   NEXTSTATE -;
        (True):    TASK    pdu!reason  :=  reason,
                           pdu!userIds  :=  Incl(user, Empty);
                   OUTPUT  DUrq(pdu) TO domain;
                   NEXTSTATE detached;
        ENDDECISION;

STATE detaching, detached;
        INPUT     PCin(pdu);
        DECISION user in pdu!detachUserIds;
        (False):   NEXTSTATE -;
        (True):    OUTPUT  Quit TO control;
                   NEXTSTATE detached;
        ENDDECISION;

STATE detaching, detached;
        INPUT     DUin(pdu);
        DECISION user in pdu!userIds;
        (False):   NEXTSTATE -;
        (True):    OUTPUT  Quit TO control;
                   NEXTSTATE detached;
        ENDDECISION;

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 Recommendation T.122 and CCITT Recommendation 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.

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 which 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, Uplink, or Downlink.

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 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 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 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.

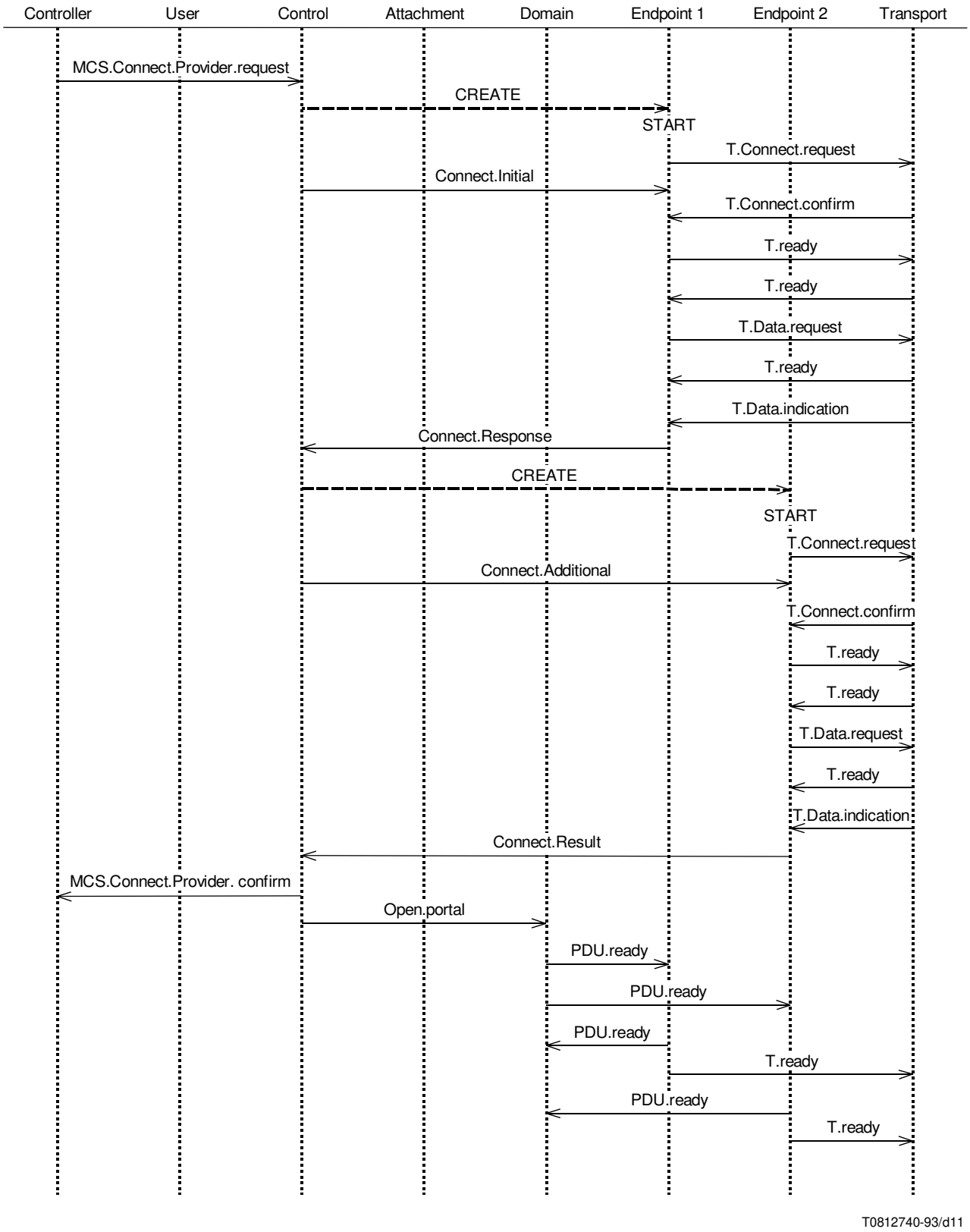


FIGURE VII.1/T.125

Calling MCS provider

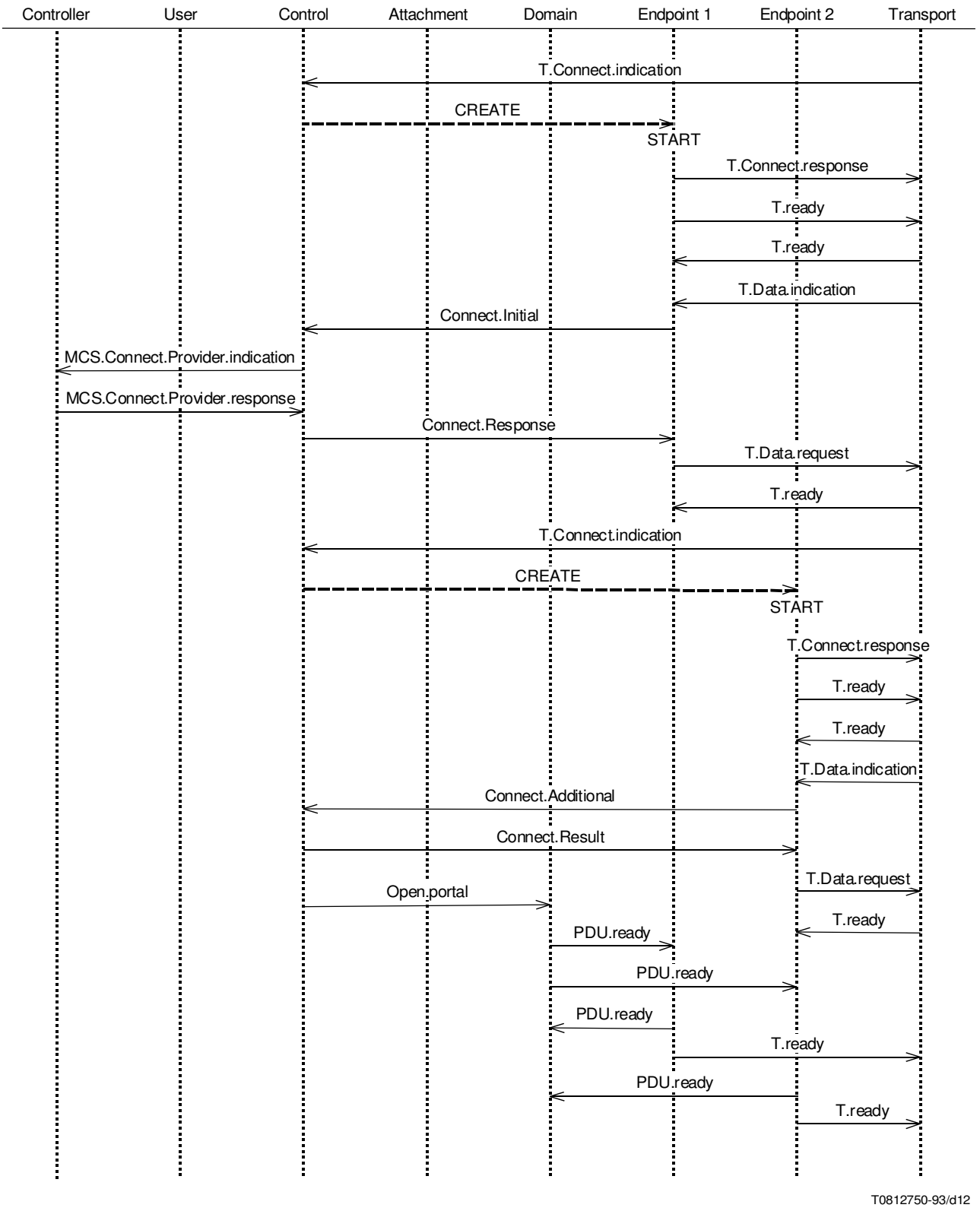


FIGURE VII.2/T.125

Called MCS provider

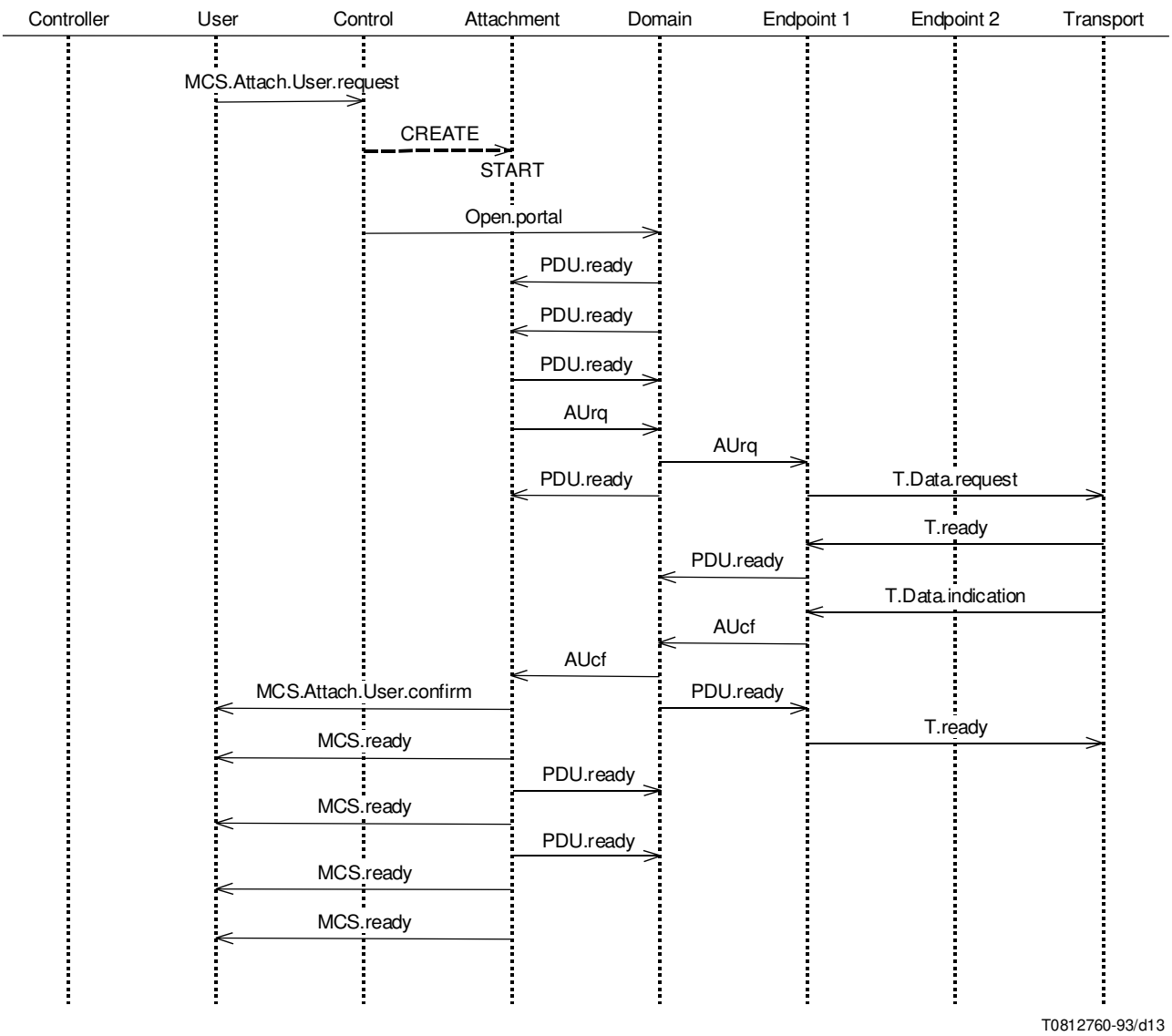


FIGURE VII.3/T.125

User attach

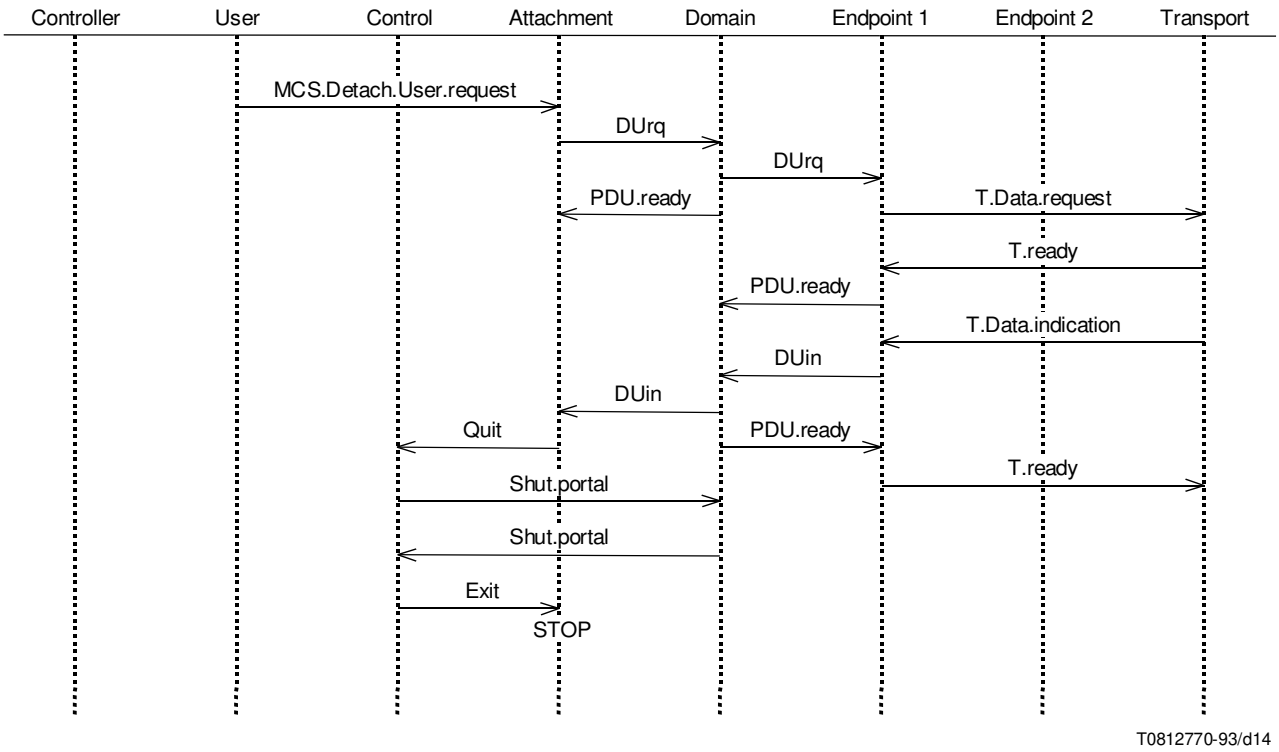


FIGURE VII.4/T.125

User-requested detach

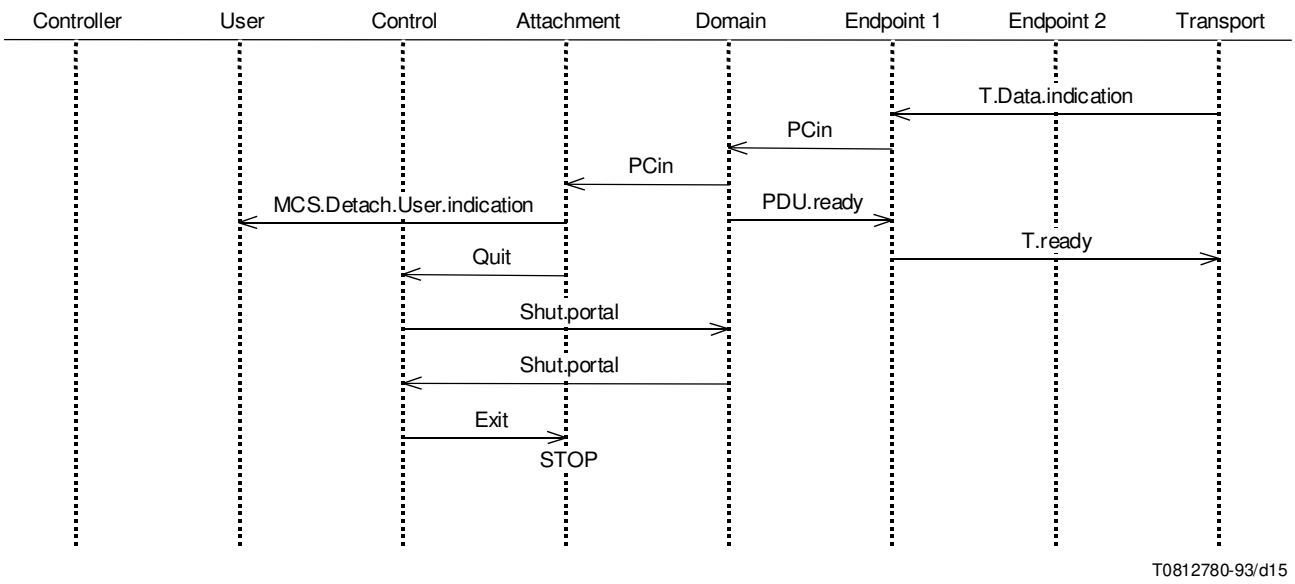


FIGURE VII.5/T.125

Provider-initiated detach

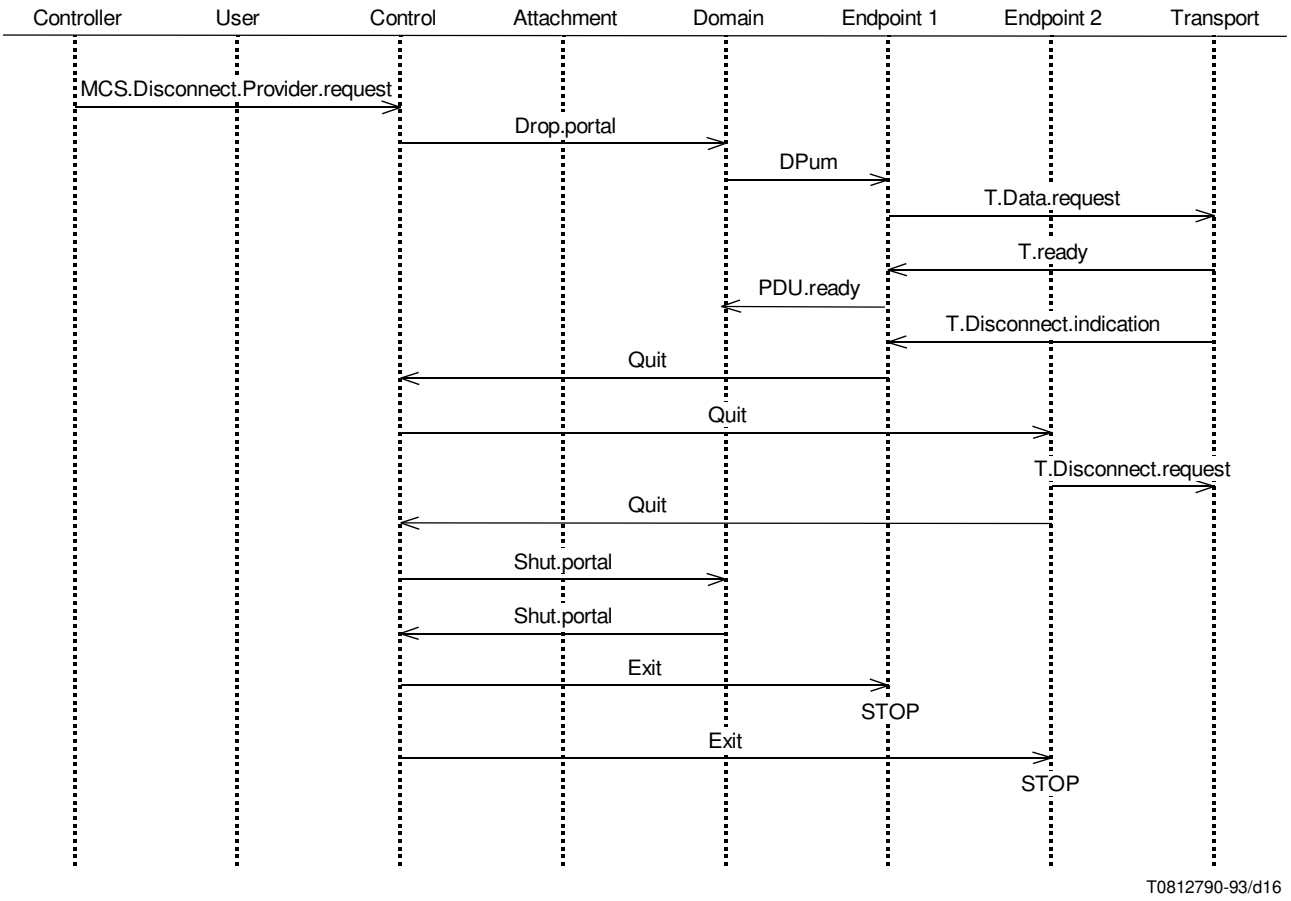


FIGURE VII.6/T.125

Local controller-requested disconnect

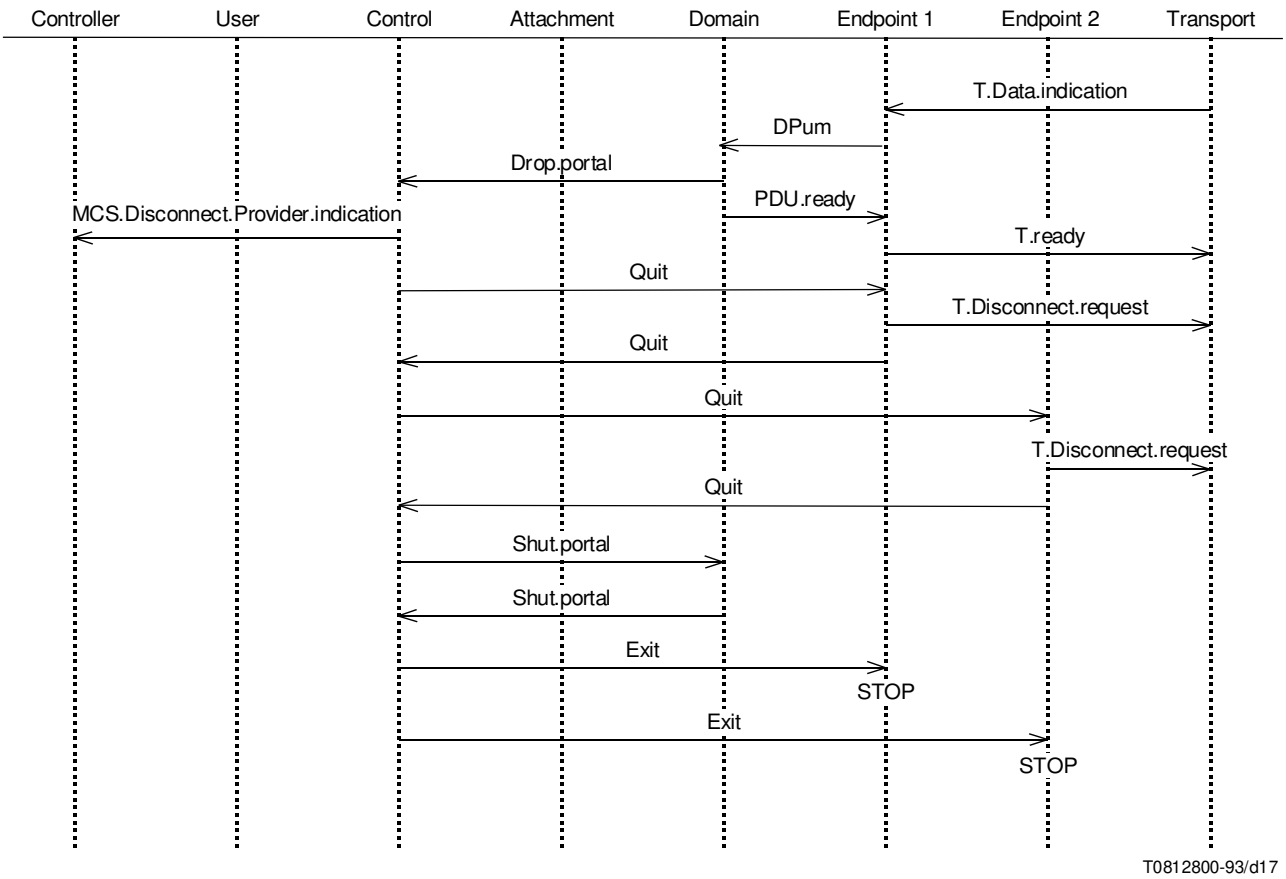


FIGURE VII.7/T.125

Remote controller-requested disconnect

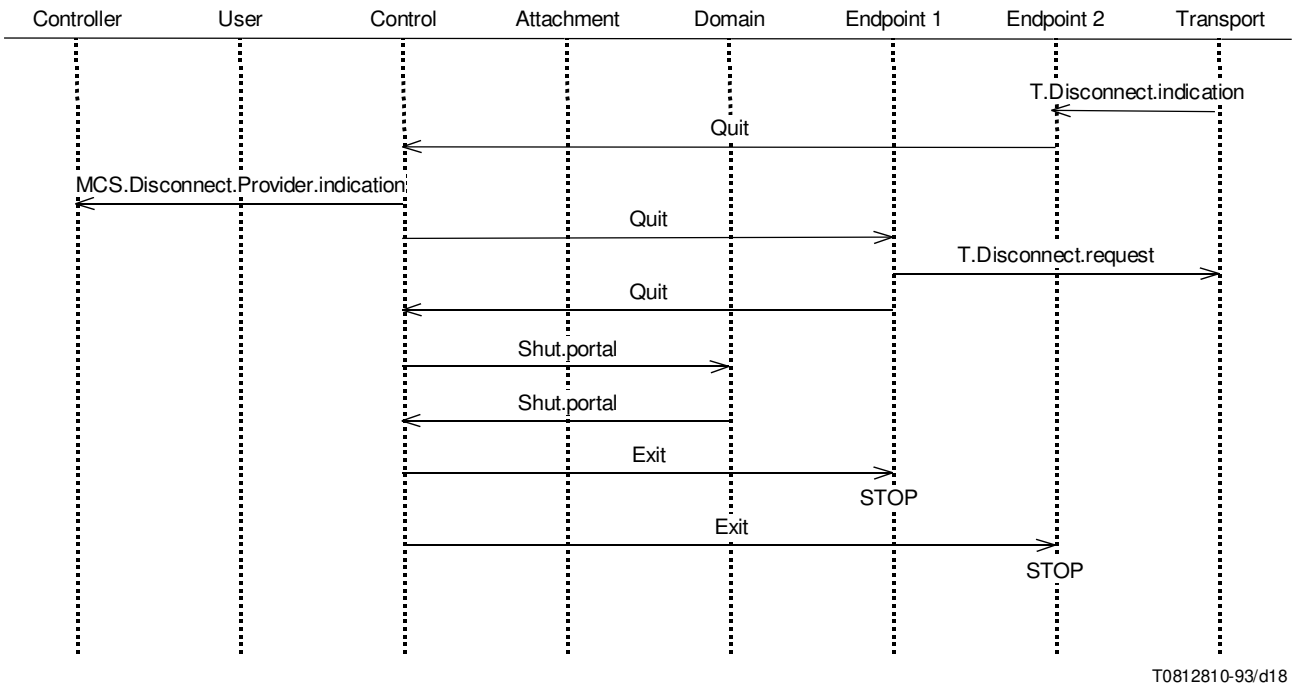


FIGURE VII.8/T.125

Provider-initiated disconnect