**Remplacée par une version plus récente**

UNION INTERNATIONALE DES TÉLÉCOMMUNICATIONS

# UIT-T T.125

SECTEUR DE LA NORMALISATION
DES TÉLÉCOMMUNICATIONS
DE L'UIT

(04/94)

## SERVICES TÉLÉMATIQUES

## ÉQUIPEMENTS TERMINAUX ET PROTOCOLES POUR LES SERVICES TÉLÉMATIQUES

---

# SPÉCIFICATION DE PROTOCOLE DU SERVICE DE COMMUNICATION MULTIPOINT

**Recommandation UIT-T T.125**
Remplacée par une version plus récente

(Antérieurement «Recommandation du CCITT»)

---

# Remplacée par une version plus récente

## AVANT-PROPOS

L'UIT-T (Secteur de la normalisation des télécommunications) est un organe permanent de l'Union internationale des télécommunications (UIT). Il est chargé de l'étude des questions techniques, d'exploitation et de tarification, et émet à ce sujet des Recommandations en vue de la normalisation des télécommunications à l'échelle mondiale.

La Conférence mondiale de normalisation des télécommunications (CMNT), qui se réunit tous les quatre ans, détermine les thèmes d'études à traiter par les Commissions d'études de l'UIT-T lesquelles élaborent en retour des Recommandations sur ces thèmes.

L'approbation des Recommandations par les Membres de l'UIT-T s'effectue selon la procédure définie dans la Résolution n° 1 de la CMNT (Helsinki, 1er-12 mars 1993).

La Recommandation T.125 de l'UIT-T, que l'on doit à la Commission d'études 8 (1993-1996) de l'UIT-T, a été approuvée le 7 avril 1994 selon la procédure définie dans la Résolution n° 1 de la CMNT.

---

NOTE

Dans la présente Recommandation, l'expression «Administration» est utilisée pour désigner de façon abrégée aussi bien une administration de télécommunications qu'une exploitation reconnue de télécommunications.

© UIT 1994

Droits de reproduction réservés. Aucune partie de cette publication ne peut être reproduite ni utilisée sous quelque forme que ce soit et par aucun procédé, électronique ou mécanique, y compris la photocopie et les microfilms, sans l'accord écrit de l'UIT.

# Remplacée par une version plus récente

## TABLE DES MATIÈRES

Remplacée par une version plus récente

## RÉSUMÉ

La présente Recommandation définit un protocole exploitant toute la hiérarchie d'un domaine de communication multipoint. Elle spécifie le format des messages de protocole et les procédures qui régissent l'échange de tels messages sur un ensemble de connexions de transport. L'objet de ce protocole est de mettre en œuvre le service de communication multipoint qui est défini dans la Recommandation T.122 de l'UIT-T.

# Remplacée par une version plus récente

**Recommandation T.125**

## SPÉCIFICATION DE PROTOCOLE DU SERVICE DE COMMUNICATION MULTIPOINT

*(Genève, 1994)*

## 1 Champ d'application

La présente Recommandation spécifie:

a) les procédures d'un unique protocole de transfert de données et d'informations de commande entre un fournisseur de service de communication multipoint (service MCS) et un fournisseur de service MCS homologue;

b) la structure et le codage des unités de données du protocole MCS qui sont utilisés pour le transfert des données et des informations de commande.

Les procédures sont définies en termes:

a) d'interactions entre fournisseurs de service MCS homologues par échange d'unités de données de protocole MCS;

b) d'interactions entre un fournisseur de service MCS et des utilisateurs de ce service par échange de primitives MCS;

c) d'interactions entre un fournisseur de service MCS et un fournisseur de service de transport par échange de primitives du service de transport.

Ces procédures sont applicables aux instances de communication multipoint entre des systèmes compatibles avec le service MCS et appelés à être interconnectés dans un environnement de systèmes ouverts.

## 2 Références

Les Recommandations et autres références suivantes contiennent des dispositions qui, par suite de la référence qui y est faite, constituent des dispositions valables pour la présente Recommandation. Au moment de la publication, les éditions indiquées étaient en vigueur. Toute Recommandation ou autre référence est sujette à révision; tous les utilisateurs de la présente Recommandation sont donc invités à rechercher la possibilité d'appliquer les éditions les plus récentes des Recommandations et autres références indiquées ci-après. Une liste des Recommandations UIT-T en vigueur est publiée régulièrement.

– Recommandation T.122 de l'UIT-T (1993), *Service de communication multipoint pour la définition des services de conférence audiographique et conférence audiovisuelle*.

– Recommandation T.123 de l'UIT-T (1993), *Piles de protocoles pour applications de téléconférence audiographique et audiovisuelle*.

– Recommandation X.200 du CCITT (1988), *Modèle de référence pour l'interconnexion des systèmes ouverts pour les applications du CCITT*.

– Recommandation X.214 du CCITT (1988), *Définition du service de transport pour l'interconnexion des systèmes ouverts (OSI) dans des applications du CCITT*.

– Recommandation X.208 du CCITT (1988), *Spécification de la syntaxe abstraite numéro un (ASN.1)*.

– Recommandation X.209 du CCITT (1988), *Spécification des règles de codage pour la notation de syntaxe abstraite numéro un (ASN.1)*.

– Projet de Recommandation X.691 de l'UIT-T | ISO/CEI DIS 8825-2, *Technologie de l'information – Interconnexion des systèmes ouverts – Spécification des règles de codage pour la notation de syntaxe abstraite numéro un (ASN.1) – Spécification des règles de métalangage condensé*.

## 3 Définitions

NOTE – Ces définitions font appel aux abréviations définies à l'article 4.

La présente Recommandation est fondée sur les concepts développés dans la Recommandation X.200 du CCITT. Elle fait appel aux termes suivants, qui y sont définis:

a) commande de flux;

b) réassemblage;

c) recombinaison;

d) segmentation;

e) maintien en séquence;

f) éclatement;

g) syntaxe de transfert;

h) connexion de transport;

i) identificateur d'extrémité de connexion de transport;

j) service de transport;

k) point d'accès au service de transport;

l) adresse de point d'accès au service de transport;

m) unité de données du service de transport.

La présente Recommandation est également fondée sur les concepts développés dans la Recommandation T.122 de l'UIT-T. Elle fait appel aux termes suivants, qui y sont définis:

a) point MCSAP de contrôle;

b) rattachement au point MCSAP;

c) canal MCS;

d) connexion MCS;

e) domaine MCS;

f) sélecteur de domaine MCS;

g) canal privé MCS;

h) gestionnaire de canal privé MCS;

i) fournisseur de service MCS;

j) point d'accès au service MCS;

k) utilisateur du service MCS;

l) identificateur d'utilisateur du service MCS;

m) fournisseur supérieur du service MCS.

Pour les besoins de la présente Recommandation, les définitions suivantes s'appliquent.

**3.1** **unité de données du service MCS**: partie des données d'utilisateur du service MCS dont l'identité est protégée pendant le transfert de l'émetteur aux récepteurs. Plus précisément, il s'agit du contenu d'une demande MCS-SEND-DATA ou d'une demande MCS-UNIFORM-SEND-DATA.

**3.2** **unité de données d'interface MCS**: unité d'information transférée via un point MCSAP entre un utilisateur du service MCS et un fournisseur de service MCS au cours d'une même interaction. Chaque unité de données d'interface MCS contient des informations de commande d'interface. Elle peut également contenir tout ou partie d'une unité de données du service MCS.

**3.3** **unité de données de protocole MCS**: unité d'information échangée au cours du protocole MCS, composée d'une part des informations transférées entre des fournisseurs de service MCS pour coordonner leur opération commune et d'autre part, le cas échéant, de données transférées pour le compte d'utilisateurs du service MCS auxquels ces fournisseurs sont en train de fournir ce service.

**3.4** **priorité de transfert de données MCS**: priorité affectée d'un des quatre niveaux suivants: absolue, élevée, moyenne, basse. La valeur est communiquée sans changement de l'émetteur aux récepteurs. Selon la valeur d'un paramètre de domaine MCS indiquant le nombre de priorités de transfert de données distinctes qui sont mises en œuvre, plusieurs priorités inférieures peuvent recevoir la même qualité de service.

**3.5 unité MCSPDU valide**: unité MCSPDU dont la structure et le codage sont conformes à la présente Recommandation.

**3.6 unité MCSPDU invalide**: unité MCSPDU qui n'est pas valide.

**3.7 erreur de protocole**: utilisation d'une unité MCSPDU d'une manière non compatible avec les procédures de la présente Recommandation.

**3.8 unité MCSPDU de type Connect**: unité MCSPDU d'un des types suivants: **Connect-Initial, Connect-Response, Connect-Additional, Connect-Result**.

**3.9 unité MCSPDU de domaine**: toute unité MCSPDU qui n'est pas une unité MCSPDU de type Connect.

**3.10 unité MCSPDU de données**: toute unité MCSPDU de type **SDrq, SDin, USrq, USin**.

**3.11 unité MCSPDU de contrôle**: toute unité MCSPDU de domaine qui n'est pas une unité MCSPDU de données.

**3.12 connexion TC initiale**: première connexion de transport d'une connexion de service MCS, utilisée pour échanger des unités MCSPDU de contrôle et de données ayant une priorité absolue.

**3.13 connexion TC additionnelle**: connexion de transport subséquente, appartenant à une connexion de service MCS, utilisée pour échanger des unités MCSPDU de données ayant une priorité inférieure.

**3.14 sous-arbre de fournisseur de service MCS**: dans le cadre d'un domaine MCS, arborescence constituée par le fournisseur de service MCS lui-même et de ses rattachements au service MCS, plus tous les fournisseurs de service MCS qui lui sont hiérarchiquement subordonnés, avec leurs propres rattachements au service MCS.

**3.15 hauteur d'un fournisseur de service MCS**: dans le cadre d'un domaine MCS, niveau situé immédiatement au-dessus de la position la plus élevée de tous les fournisseurs de service MCS hiérarchiquement subordonnés. Un fournisseur de service MCS sans subordonnés a la hauteur un.

## 4 Abréviations

Pour les besoins de la présente Recommandation, les abréviations suivantes sont utilisées.

| | |
|---|---|
| MCS | Service de communication multipoint (*multipoint communication service*) |
| MCSAP | Point d'accès au service MCS (*MCS service access point*) |
| MCSPDU | Unité de données de protocole du service MCS (*MCS protocol data unit*) |
| TC | Connexion de transport (*transport connection*) |
| TS | Service de transport (*transport service*) |
| TSAP | Point d'accès au service de transport (*transport service access point*) |
| TSDU | Unité de données du service de transport (*transport service data unit*) |

## 5 Vue d'ensemble du protocole MCS

### 5.1 Modèle de couche MCS

Un fournisseur de service MCS communique avec les utilisateurs de ce service au moyen de primitives MCS définies dans la Recommandation T.122 de l'UIT-T, passant par un point d'accès MCSAP. Ces primitives peuvent être la cause ou le résultat d'échanges d'unités MCSPDU entre fournisseurs MCS homologues utilisant une connexion MCS. Elles peuvent aussi être la cause ou le résultat d'actions prises dans le domaine d'un même fournisseur de service MCS. Les échanges d'unités MCSPDU ont lieu entre fournisseurs MCS qui couvrent le même domaine MCS.

Un fournisseur de service MCS peut avoir plusieurs homologues, à chacun desquels il est relié directement par une connexion MCS particulière ou indirectement par l'intermédiaire d'un fournisseur MCS homologue. Une connexion MCS comprend une ou plusieurs connexions de transport, selon le nombre de priorités de transfert de données mises en jeu dans un domaine MCS. Les échanges de la couche protocole s'effectuent au moyen des services de la couche transport en passant par une paire de points TSAP.

Ce modèle de couche MCS est illustré à la Figure 5-1.

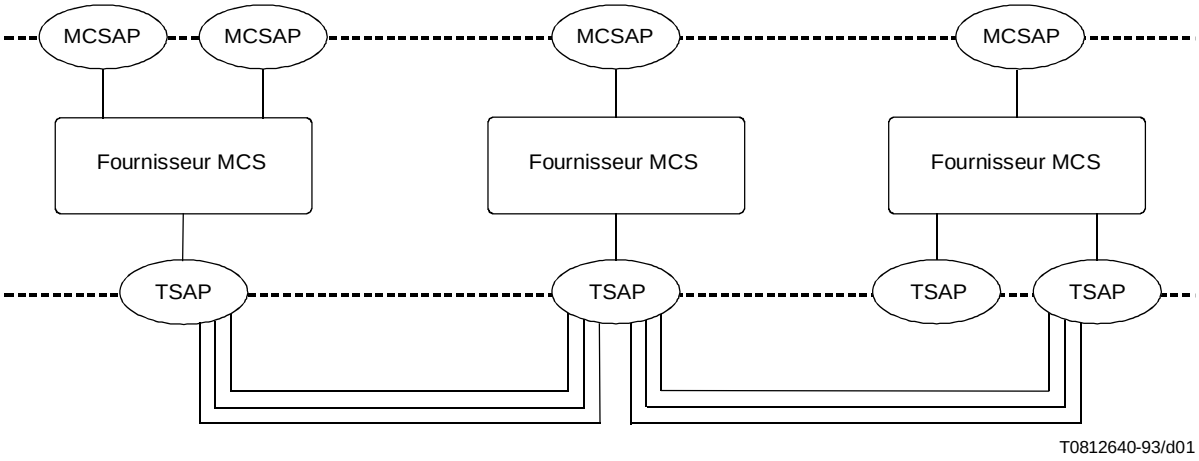



FIGURE 5-1/T.125

**Modèle de la couche MCS**

## 5.2 Services assurés par la couche MCS

Le protocole MCS prend en charge les services définis dans la Recommandation T.122 de l'UIT-T. Les informations sont transférées à destination et en provenance d'un utilisateur MCS au moyen des primitives MCS énumérées dans le Tableau 5-1.

## 5.3 Services assurés par la couche transport

Le protocole MCS suppose l'utilisation d'un sous-ensemble du service de transport en mode connexion qui est défini dans la Recommandation X.214 du CCITT. Les informations sont transférées en direction et en provenance d'un fournisseur de service de transport (TS) au moyen des primitives énumérées au Tableau 5-2.

## 5.4 Fonctions de la couche MCS

Le Tableau 5-1 énumère les unités fonctionnelles du service MCS et les unités MCSPDU associées à chaque primitive MCS. Les unités MCSPDU sont définies à l'article 7. La relation entre primitives et unités MCSPDU peut être aussi simple qu'un rapport de cause à effet bilatéral. Par exemple, la primitive de demande MCS-ATTACH-USER produit les unités **AUrq**, alors que l'unité **AUcf** produit la primitive de confirmation MCS-ATTACH-USER. D'autres cas peuvent être plus compliqués. La réalisation de la primitive MCS-CONNECT-PROVIDER exige par exemple l'échange d'unités MCSPDU additionnelles, à titre d'effet secondaire des quatre phases de cette primitive: l'une quelconque des cinq MCSPDU associées peut déclencher une indication MCS-DETACH-USER et l'une quelconque des quatre MCSPDU associées peut déclencher une indication MCS-CHANNEL-EXPEL.

### 5.4.1 Gestion de domaine

La couche de service MCS conserve l'intégrité des connexions MCS qui forment un domaine MCS. Une connexion MCS est orientée, c'est-à-dire qu'une de ses extrémités est hiérarchiquement supérieure à l'autre. Chaque domaine est supervisé par un seul fournisseur de service MCS.

L'établissement d'une connexion MCS consiste à fusionner deux domaines. La couche de service MCS fait en sorte qu'il ne reste qu'un seul fournisseur supérieur. Elle résout tous conflits d'identité unique ou de droits de propriété exclusifs pouvant se présenter.

La déconnexion d'une connexion MCS subdivise un domaine en deux parties. Celle qui contient le fournisseur supérieur survit. La partie inférieure se supprime automatiquement.

La couche MCS identifie de manière univoque les utilisateurs qui sont rattachés à un domaine donné. Les utilisateurs peuvent se percevoir les uns les autres par l'intermédiaire de leurs interactions au moyen de primitives MCS. La couche MCS notifie à tous les utilisateurs d'un domaine le moment où l'un d'entre eux se détache. La couche MCS récupère toutes les ressources éventuellement détenues par l'utilisateur détaché.

TABLEAU 5-1/T.125

**Primitives du service MCS**

| Unité fonctionnelle | Primitives | Unités MCSPDU associées |
|---|---|---|
| Gestion de domaine | Demande MCS-CONNECT-PROVIDER<br>Indication MCS-CONNECT-PROVIDER<br>Réponse MCS-CONNECT-PROVIDER<br>Confirmation MCS-CONNECT-PROVIDER<br>(effets secondaires) | **Connect-Initial**<br>**Connect-Initial**<br>**Connect-Response**<br>**Connect-Response**<br>**Connect-Additional**<br>**Connect-Result**<br>**PDin EDrq**<br>**MCrq MCcf PCin**<br>**MTrq MTcf PTin** |
| | Demande MCS-DISCONNECT-PROVIDER<br>Indication MCS-DISCONNECT-PROVIDER | **DPum**<br>**DPum RJum** |
| | Demande MCS-ATTACH-USER<br>Confirmation MCS-ATTACH-USER | **AUrq**<br>**AUcf** |
| | Demande MCS-DETACH-USER<br>Indication MCS-DETACH-USER | **DUrq**<br>**DUin**<br>**MCcf PCin**<br>**MTcf PTin** |
| Gestion de canal | Demande MCS-CHANNEL-JOIN<br>Confirmation MCS-CHANNEL-JOIN | **CJrq**<br>**CJcf** |
| | Demande MCS-CHANNEL-LEAVE<br>Indication MCS-CHANNEL-LEAVE | **CLrq**<br>**MCcf PCin** |
| | Demande MCS-CHANNEL-CONVENE<br>Confirmation MCS-CHANNEL-CONVENE | **CCrq**<br>**CCcf** |
| | Demande MCS-CHANNEL-DISBAND<br>Indication MS-CHANNEL-DISBAND | **CDrq**<br>**MCcf PCin** |
| | Demande MCS-CHANNEL-ADMIT<br>Indication MCS-CHANNEL-ADMIT | **CArq**<br>**CAin** |
| | Demande MCS-CHANNEL-EXPEL<br>Indication MCS-CHANNEL-EXPEL | **CErq**<br>**CEin CDin**<br>**MCcf PCin** |
| Transfert de données | Demande MCS-SEND-DATA<br>Indication MCS-SEND-DATA | **SDrq**<br>**SDin** |
| | Demande MCS-UNIFORM-SEND-DATA<br>Indication MCS-UNIFORM-SEND-DATA | **USrq**<br>**USin** |

TABLEAU 5-1/T.125 *(fin)*

**Primitives du service MCS**

| Unité fonctionnelle | Primitives | Unités MCSPDU associées |
|---|---|---|
| Gestion de jeton | Demande MCS-TOKEN-GRAB<br>Confirmation MCS-TOKEN-GRAB | **TGrq**<br>**TGcf** |
| | Demande MCS-TOKEN-INHIBIT<br>Confirmation MCS-TOKEN-INHIBIT | **TIrq**<br>**TIcf** |
| | Demande MCS-TOKEN-GIVE<br>Indication MCS-TOKEN-GIVE<br>Réponse MCS-TOKEN-GIVE<br>Confirmation MCS-TOKEN-GIVE | **TVrq**<br>**TVin**<br>**TVrs**<br>**TVcf** |
| | Demande MCS-TOKEN-PLEASE<br>Indication MCS-TOKEN-PLEASE | **TPrq**<br>**TPin** |
| | Demande MCS-TOKEN-RELEASE<br>Confirmation MCS-TOKEN-RELEASE | **TRrq**<br>**TRcf** |
| | Demande MCS-TOKEN-TEST<br>Confirmation MCS-TOKEN-TEST | **TTrq**<br>**TTcf** |

### 5.4.2 Gestion des canaux

La couche MCS enregistre les parties d'un domaine MCS qui contiennent un ou plusieurs utilisateurs ayant adhéré à un canal donné, de manière à pouvoir optimiser le transfert de données vers les destinations souhaitant les recevoir.

La couche MCS traite les identificateurs d'utilisateur comme des canaux pour membre unique, auxquels seuls des utilisateurs désignés sont autorisés à adhérer. Sur demande, la couche MCS peut créer des canaux privés auxquels seuls des utilisateurs habilités auront le droit d'accéder. Elle peut également attribuer des canaux publics, auxquels aucun autre utilisateur n'a encore adhéré.

### 5.4.3 Transfert de données

La couche du service MCS entretient un flux séquentiel de données à destination des utilisateurs qui ont adhéré à un canal. Un canal devient, en pratique, une liste de distribution multidestinataire avec une étendue comprise entre zéro destination et une diffusion générale.

Par défaut, la couche du service MCS aiguille les données vers chaque récepteur en passant par le plus court chemin offert par les connexions MCS. Sur option, elle achemine des unités de données spécifiées du service MCS en passant par le fournisseur MCS supérieur, garantissant ainsi leur réception uniforme par tous les récepteurs, émetteur éventuellement compris.

La couche du service MCS reconnaît une ou plusieurs priorités de transfert de données et leur applique un traitement préférentiel. Par opérations de segmentation et réassemblage, elle autorise des longueurs quelconques d'unités de données du service MCS.

La couche du service MCS règle le débit global des données à l'intérieur d'un domaine. L'incapacité d'un récepteur d'accepter des données à leur débit de présentation provoque une contre-pression qui elle-même provoque le blocage des émetteurs. Un utilisateur peut être détaché contre sa volonté s'il ne parvient pas à conserver un débit de réception minimal.

La couche du service MCS garantit une réception exempte d'erreurs des données émises, du moment que les utilisateurs d'origine et de destination restent rattachés au canal. Les données à priorité de niveau plus élevé doivent toutefois prendre le pas et une surabondance de telles données peut retarder indéfiniment l'acheminement de données à priorité plus faible.

TABLEAU 5-2/T.125

**Primitives du service de transport**

| Primitives | Utilisation | Paramètres | Utilisation |
|---|---|---|---|
| Demande T-CONNECT<br>Indication T-CONNECT | X<br>X | Adresse appelée<br>Adresse appelante<br>Option de données exprès<br>Qualité de service<br>Données d'utilisateur TS | X<br>X<br>–<br>X<br>– |
| Réponse T-CONNECT<br>Confirmation T-CONNECT | X<br>X | Adresse répondante<br>Option de données exprès<br>Qualité de service<br>Données d'utilisateur TS | –<br>–<br>X<br>– |
| Demande T-DATA<br>Indication T-DATA | X<br>X | Données d'utilisateur TS | X |
| Demande T-EXPEDITED-DATA<br>Indication T-EXPEDITED-DATA | –<br>– | Données d'utilisateur TS | – |
| Demande T-DISCONNECT | X | Données d'utilisateur TS | – |
| Indication T-DISCONNECT | X | Cause<br>Données d'utilisateur TS | –<br>– |
| X Indique que le protocole MCS part du principe que cette caractéristique est toujours disponible.<br>– Indique que le protocole MCS ne fait pas appel à cette caractéristique. | | | |

### 5.4.4 Gestion des jetons

La couche du service MCS met en œuvre des opérations de jeton au niveau du fournisseur MCS supérieur, assurant ainsi la cohérence et l'exclusion.

## 5.5 Traitement hiérarchique

Le traitement hiérarchique dans un domaine MCS est illustré par la Figure 5-2.

Les nœuds représentent, dans la figure ci-dessus, des fournisseurs de service MCS. Les flèches sont assorties de noms d'unités MCSPDU. Cet exemple porte sur une certaine période de temps après l'établissement du domaine au moyen de connexions entre fournisseurs MCS, lorsque l'utilisation du transfert de données commence à se développer. A l'étape 1, le fournisseur D demande, pour le compte d'un utilisateur, à adhérer à un canal sur lequel des données seront réparties. A l'étape 2, la demande est confirmée comme étant acceptée. A l'étape 3, un utilisateur rattaché au fournisseur A envoie des données et l'unité **SDrq** correspondante commence à remonter vers le sommet de la hiérarchie. En admettant que seuls des utilisateurs rattachés aux fournisseurs A, C et D ont adhéré au canal sur lequel les données sont actuellement envoyées, l'unité MCSPDU de demande est répercutée en aval aux étapes 6 et 7 sous la forme **SDin**. Le fournisseur E, percevant qu'aucun autre subordonné n'a besoin de recevoir ces données, expédie simplement l'unité **SDrq** en amont (étape 4). Le fournisseur F expédie l'unité **SDrq** en amont à l'étape 5 mais la répercute également en aval à l'étape 6, sachant que le fournisseur C a fait part de son intérêt pour ce canal.

Les fournisseurs de service MCS ne sont pas particulièrement concernés par leur hauteur dans la hiérarchie, sauf pour ce qui est de leur rôle dans le maintien d'une limite globale de hauteur du domaine et du fait que, d'une manière générale, ils sont ou non le fournisseur supérieur. Celui-ci ne possède pas de connexion en amont. Tous les autres fournisseurs en ont exactement une seule.

Un fournisseur de service MCS enregistre des informations sur les canaux et jetons utilisés dans son sous-arbre de domaine MCS.

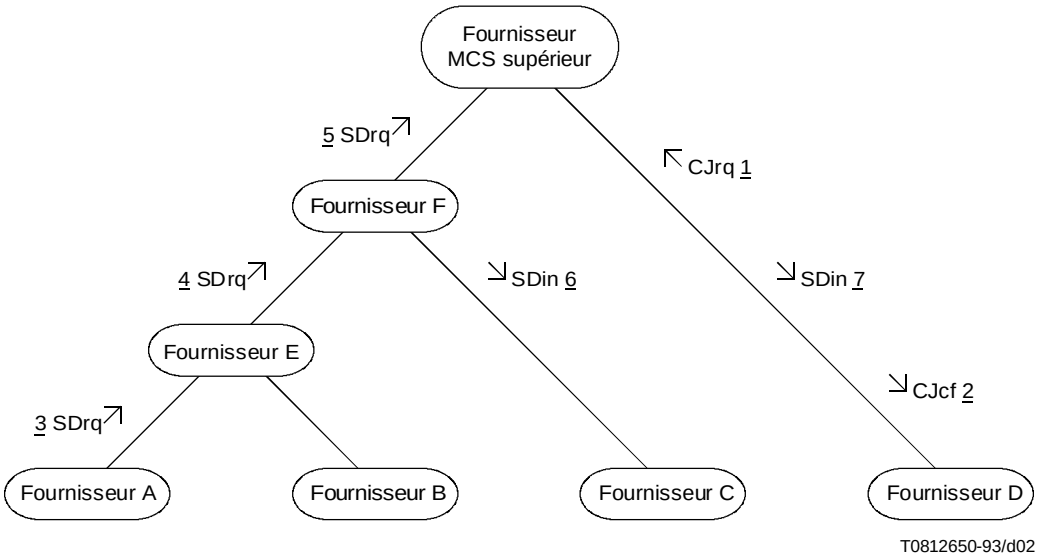



FIGURE 5-2/T.125

**Traitement hiérarchique dans un domaine MCS**

Un fournisseur de service MCS enregistre les canaux auxquels des utilisateurs ont adhéré à l'intérieur du sous-arbre. Pour chacun de ces canaux, il enregistre également l'origine de l'adhésion, c'est-à-dire les rattachements et les connexions MCS en aval dont ils proviennent. Il enregistre les identificateurs d'utilisateur qui sont attribués dans le sous-arbre et leur origine. Il enregistre les canaux privés qui possèdent soit un gestionnaire ou un utilisateur admis dans le sous-arbre. Il enregistre les identificateurs d'utilisateur correspondants.

Un fournisseur de service MCS enregistre les jetons qui sont saisis ou inhibés par des utilisateurs dans le sous-arbre et il enregistre les identificateurs d'utilisateur correspondants.

Un fournisseur de service MCS examine les demandes issues de son sous-arbre pour vérifier que l'identificateur de l'utilisateur initiateur est légitimement attribué au rattachement ou à la connexion MCS descendante d'origine. Cela crée des anneaux de protection autour du fournisseur MCS supérieur et limite le nombre d'interruptions qu'un participant malveillant peut provoquer dans un domaine par ailleurs coopératif.

En termes généraux, le fonctionnement de la couche de service MCS peut, sauf plusieurs exceptions, être décrit comme suit.

a) Une primitive de demande de service MCS, invoquée à un point de rattachement MCS, produit une unité MCSPDU au niveau du fournisseur MCS correspondant et la remonte vers le fournisseur MCS supérieur, où elle sera suivie d'effet si toutes les informations sur le domaine MCS sont présentes.

b) Une unité MCSPDU de confirmation peut être produite au niveau du fournisseur MCS supérieur afin de retourner des résultats vers le rattachement demandeur. Les fournisseurs MCS qui la transmettent mettent à jour leurs enregistrements en fonction des effets de l'opération sur leur sous-arbre. Un message de confirmation est acheminé vers l'utilisateur initiateur identifié après consultation des enregistrements locaux à chaque bond descendant successif.

c) Une unité MCSPDU d'indication au lieu de confirmation peut être produite pour informer d'autres rattachements de l'action entreprise. Ces messages d'indication peuvent être copiés et descendus vers plusieurs connexions aboutissant à des utilisateurs affectés. Les fournisseurs de service MCS peuvent également mettre à jour leurs enregistrements en fonction de l'effet de l'opération, dans le cadre du traitement d'un flux d'indication.

La description ci-dessus donne une vue d'ensemble destinée à créer un cadre théorique. Les articles suivants spécifient en plus amples détails les informations enregistrées chez un fournisseur de service MCS et la façon dont des unités MCSPDU spécifiques sont traitées.

Les exceptions sont en particulier les suivantes: certaines requêtes, comme **CJrq** et **CLrq**, peuvent ne pas remonter jusqu'au fournisseur supérieur et certaines indications, comme **SDin** peuvent être produites à un niveau inférieur au fournisseur supérieur. Une seule unité MCSPDU, **TVrs**, appartient à la catégorie des réponses. Et certaines unités MCSPDU peuvent être produites par un fournisseur de service MCS au titre de la poursuite d'un traitement, contrairement à des unités telles que **CLrq** faisant suite à **DUin**.

### 5.6 Paramètres de domaine

Les fournisseurs de service MCS qui couvrent un seul domaine MCS attribuent les ressources et exécutent les procédures conformément aux paramètres ci-après, dont les valeurs sont identiques dans un domaine donné.

a) Nombre maximal de canaux MCS pouvant être utilisés simultanément. Ce paramètre comprend les canaux auxquels un utilisateur quelconque adhère, les identificateurs d'utilisateur qui ont été attribués et les canaux privés qui ont été créés.

b) Nombre maximal d'identificateurs d'utilisateur MCS pouvant être attribués simultanément. Ce paramètre indique une limite inférieure à l'intérieur de la contrainte représentée par le paramètre précédent.

c) Nombre maximal d'identificateurs de jeton pouvant être saisis ou inhibés simultanément.

d) Nombre de priorités de transfert de données mises en œuvre. Ce nombre est égal à celui des connexions de transport (TC) d'une connexion MCS. Un utilisateur du service MCS peut toujours émettre et recevoir des données avec des priorités extérieures à la limite. Mais toutes ces priorités peuvent être traitées de la même manière que la plus basse priorité mise en œuvre.

e) Débit renforcé. Bien que la commande de flux globale impose une limite au transfert de données à l'intérieur d'un domaine en fonction de la vitesse du récepteur le plus lent, les récepteurs ne doivent jamais être autorisés à adopter une vitesse arbitrairement basse. Sinon, un des participants à une conférence peut bloquer tous les autres. Ce paramètre invite les fournisseurs de service MCS à appliquer un débit de réception minimal à chaque rattachement MCS et sur chaque connexion MCS descendante. Les transgresseurs courent le risque d'être détachés ou déconnectés, selon le cas, contre leur volonté.

f) Hauteur maximale. Ce paramètre limite la hauteur de tous les fournisseurs MCS, en particulier le fournisseur MCS supérieur.

g) Longueur maximale des unités MCSPDU de domaine. La commande de flux globale est fondée sur la mise en mémoire tampon d'unités MCSPDU de domaine (mais non des unités MCSPDU de type connect), chez un fournisseur MCS. Par souci de simplicité, on suppose que les mémoires tampons ont une capacité fixe. Un fournisseur MCS ne doit pas émettre d'unités MCSPDU plus longues. Cela limite la quantité d'informations qui peuvent être assemblées pour former une même unité MCSPDU de commande. Cela donne également une indication du point où un flux continu de données d'utilisateur devrait être segmenté pour former des unités MCSPDU de données.

h) Version du protocole. Ce paramètre peut prendre l'une des deux valeurs qui définissent des codages différents pour les unités MCSPDU de domaine.

NOTE – Une certaine instance d'un fournisseur de service MCS peut fonctionner avec des contraintes de ressources locales qui sont également paramétrées. Il peut s'agir de la capacité de mémoire disponible pour tamponner les unités MCSPDU en attente de transport, du nombre maximal de rattachements MCS et du nombre maximal de connexions MCS vers d'autres fournisseurs. De tels paramètres relèvent de décisions locales et ne sont pas communiqués dans un domaine MCS.

## 6 Utilisation du service de transport

### 6.1 Modèle du service de transport

La présente description paraphrase les parties applicables de la Recommandation X.214 du CCITT, étant supposé qu'il n'est jamais fait appel à des données exprès.

Le service de transport offre les possibilités suivantes à un utilisateur TS:

a) le moyen d'établir une connexion de transport avec un autre utilisateur du service de transport afin d'échanger des unités TSDU. Plusieurs connexions de transport peuvent exister entre un même couple d'utilisateurs du service de transport;

b) la possibilité de demander, de négocier et de faire agréer par le fournisseur du service de transport, pour chaque connexion de transport au moment de son établissement, une certaine qualité de service spécifiée par les paramètres de qualité de service;

c) le moyen de transférer des unités TSDU sur une connexion de transport. Les unités TSDU, qui comprennent un nombre entier d'octets, sont transférées en transparence, en ce sens que les limites et le contenu des unités TSDU sont préservés sans changement par le fournisseur TS;

d) le moyen, pour l'utilisateur destinataire du service de transport, de commander la vitesse à laquelle l'utilisateur expéditeur peut transmettre les données;

e) la libération inconditionnelle et donc éventuellement destructive d'une connexion de transport.

Le fonctionnement d'une connexion de transport est modélisé sous forme abstraite par une paire de files d'attente reliant deux points TSAP, chaque file correspondant à un sens de transmission du flux d'informations. Chaque connexion de transport est modélisée par une paire distincte de files d'attente.

Le modèle des files d'attente sert à exprimer la fonction de commande de flux. Une file d'attente a une capacité limitée mais celle-ci n'est pas forcément fixe ni déterminable. L'introduction et l'extraction d'objets d'unités de type connect, d'unités TSDU et d'unités de type disconnect résultent des interactions au niveau des deux points TSAP. La capacité d'un utilisateur du service de transport (TS) à ajouter des objets à une file d'attente dépend du comportement de l'utilisateur TS qui retire des objets de cette file ainsi que de l'état de celle-ci. Les seuls objets qui peuvent être placés dans une file d'attente par le fournisseur du service TS sont des objets de type disconnect. Des objets sont ajoutés à une file d'attente sous le contrôle du fournisseur TS. Les objets sont normalement retirés dans l'ordre où ils ont été introduits. La seule exception à leur retrait normal est le fait qu'un objet peut être supprimé par le fournisseur TS si, et seulement si, l'objet suivant est du type disconnect.

Un mécanisme d'identification d'extrémité de connexion de transport doit être prévu au niveau local si l'utilisateur TS et le fournisseur TS ont besoin de distinguer entre elles plusieurs connexions de transport au niveau d'un même point d'accès au service de transport (TSAP). Toutes les primitives doivent alors utiliser ce mécanisme d'identification pour identifier la connexion de transport à laquelle elles s'appliquent. Cette identification implicite n'est pas représentée sous la forme d'un paramètre des primitives du service TS et ne doit pas être confondue avec les paramètres d'adresse des primitives de type T-CONNECT.

### 6.2 Utilisation de connexions multiples

Une connexion MCS se compose d'une ou de plusieurs connexions de transport (TC) entre la même paire de fournisseurs MCS. La première connexion TC établie est dite TC initiale; les connexions TC établies ultérieurement sont dites TC additionnelles. Toutes les connexions TC qui appartiennent à une même connexion MCS sont établies par le même fournisseur de service MCS, en réaction à une primitive de demande MCS-CONNECT-PROVIDER. Cette demande contiendra les paramètres d'adresse, c'est-à-dire les adresses de point TSAP appelant et appelé. Ces paramètres seront utilisés sans modification dans les demandes T-CONNECT qui en résulteront.

Le nombre de connexions TC par connexion MCS est uniforme dans un même domaine MCS. Ce paramètre de domaine est égal au nombre de niveaux de priorité de transfert de données mis en œuvre. Des connexions TC distinctes sont requises parce que chacune véhicule une commande de flux. Des blocages affectant des données de priorité inférieure ne doivent normalement pas provoquer de contre-pression sur des données de priorité supérieure. Pour pouvoir être mises en œuvre totalement, les données de priorité inférieure et de priorité supérieure doivent être acheminées sur des connexions TC différentes.

La qualité de service demandée pour une connexion TC peut varier selon la priorité de transfert de données pour laquelle cette connexion a été établie. Ces objectifs de qualité de service peuvent ne pas être uniformes à l'intérieur d'un domaine MCS.

Les aspects de qualité de service à prendre en compte sont le débit maximal ou moyen et le temps de transit. Des données à priorité élevée peuvent favoriser un temps de transit court pour une réponse en temps réel mais ne peuvent pas exiger un débit utile élevé. Des données à priorité basse, par ailleurs, peuvent favoriser un débit utile élevé pour des transferts de masse mais ne peuvent pas exiger un temps de transit court.

La priorité des connexions de transport constitue un autre aspect de la qualité de service bien que celui-ci ne recouvre pas exactement le concept de priorité de transfert des données du service MCS. Il s'agit plutôt de l'ordre relatif dans lequel les connexions TC verront leur qualité de service se dégrader, le cas échéant. Une priorité TC élevée peut être requise en même temps que d'autres caractéristiques comme un faible temps de transit, afin de garantir que la priorité MCS des données recevra le traitement préférentiel qu'elle mérite.

Une unité MCSPDU de type connect n'apparaît qu'en tant que première unité TSDU acheminée dans un sens ou dans l'autre d'une connexion TC. Les unités de type **Connect-Initial** et **Connect-Response** traversent la connexion de transport initiale d'une connexion MCS. Les unités de type **Connect-Additional** et **Connect-Result** traversent des connexions de transport additionnelles, si elles existent.

Un fournisseur de service MCS appelant, qui émet des primitives de demande T-CONNECT, détermine par ses propres actions les connexions TC qui feront partie de la même connexion MCS ainsi que les priorités de transfert de données que ces connexions achemineront.

Un fournisseur de service MCS appelé, qui reçoit des primitives d'indication T-CONNECT, doit en général accepter une connexion TC et lire sa première unité TSDU avant de prendre connaissance de sa signification. Une unité de type **Connect-Initial** annonce une connexion TC entrante qui formera le début d'une nouvelle connexion MCS. Une unité de type **Connect-Additional** annonce une connexion TC qui fera partie d'une connexion MCS en cours.

L'unité **Connect-Additional** contient une valeur qui est attribuée par le fournisseur MCS appelé et qui est transmise au fournisseur MCS appelant dans une unité **Connect-Response** acheminée sur la connexion TC initiale qui désignera la connexion TC additionnelle comme appartenant à la même connexion MCS. Une unité **Connect-Additional** indique explicitement la priorité des données qui est acheminée par la connexion de transport.

Les unités MCSPDU de connexion sont échangées immédiatement après l'établissement de la connexion TC. Une fois qu'une connexion MCS a été confirmée, elle est intégrée dans le domaine MCS hiérarchique. C'est alors que la connexion MCS achemine les unités MCSPDU du domaine.

A l'exception des unités MCSPDU de données, les unités MCSPDU de domaine traversent la connexion TC initiale d'une connexion MCS. Les unités MCSPDU de données traversent la connexion TC qui correspond à leur niveau de priorité de données. Si la priorité spécifiée est supérieure au nombre mis en œuvre dans un domaine MCS, l'unité MCSPDU de données traversera la connexion TC ayant la plus basse priorité mise en œuvre.

Si une seule priorité est mise en œuvre dans un domaine MCS, chacune des connexions MCS de ce domaine comportera une seule connexion de transport, il ne sera pas fait appel aux unités **Connect-Additional** ou **Connect-Result** et toutes les unités MCSPDU seront acheminées en séquence entre les fournisseurs.

### 6.3 Libération d'une connexion de transport

Un fournisseur de service de transport (TS) augmente la fiabilité de ses connexions de bout en bout en exécutant un protocole suffisant pour compenser toute faiblesse éventuelle se manifestant dans les services des couches réseau sous-jacentes. Un fournisseur de service MCS ne duplique pas cette fonction. En cas de panne de transport, il ne tente pas d'effectuer un autre processus de rétablissement automatique.

Les erreurs irrémédiables sont annoncées par une primitive d'indication T-DISCONNECT. Si l'une quelconque des connexions de transport faisant partie d'une connexion MCS est déconnectée, les autres sont immédiatement déconnectées aussi. Une primitive d'indication MCS-DISCONNECT-PROVIDER est alors émise avec la cause engendré par le fournisseur, sauf si cette déconnexion a été demandée par l'utilisateur.

Par ailleurs, une primitive de demande MCS-DISCONNECT-PROVIDER doit normalement apparaître sous forme d'indication à l'autre extrémité avec la cause «demandé par l'utilisateur». Malgré les indications données dans la Recommandation X.214, le protocole de transport de la classe la plus simple n'autorise pas le passage de données d'utilisateur dans les primitives T-DISCONNECT. Le code de cause de déconnexion est transféré dans une unité MCSPDU spéciale, qui obligera le fournisseur MCS qui la recevra à déconnecter la connexion MCS qui aura acheminé ce code.

## 7 Structure des unités MCSPDU

La structure des unités MCSPDU est spécifiée au moyen de la notation ASN.1 de la Recommandation X.208 du CCITT. Les articles 9 et 10 décriront plus en détail l'utilisation et la signification de ces unités MCSPDU.

```
MCS-PROTOCOL DEFINITIONS ::=

BEGIN

-- Partie 1: Types MCS fondamentaux

ChannelId ::= INTEGER (0..65535)                -- domaine de 16 bits

StaticChannelId ::= ChannelId (1..1000)         -- pour les canaux connus en permanence

DynamicChannelId ::= ChannelId (1001..65535)    -- canaux créés et supprimés

UserId ::= DynamicChannelId                     -- canal créé par utilisateur de rattachement
                                                -- canal supprimé par utilisateur de détachement

PrivateChannelId ::= DynamicChannelId           -- canal créé par constitution de canal
                                                -- canal supprimé par dissolution de canal

AssignedChannelId ::= DynamicChannelId          -- canal créé par adhésion à un canal zéro
                                                -- canal supprimé par sortie du dernier canal

TokenId ::= INTEGER (1..65535)                  -- tous ces entiers sont connus en permanence

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),
                                -- limite sur les identificateurs de canal utilisés,
                                -- canal statique + ident. utilisateur + privé + attribué
        maxUserIds              INTEGER (0..MAX),
                                -- sous-limite sur les seuls canaux d'utilisateur identifié
        maxTokenIds             INTEGER (0..MAX),
                                -- limite sur les identificateurs de jeton utilisés
                                -- jeton saisi + inhibé + donnant + incessible + donné
        numPriorities           INTEGER (0..MAX),
                                -- nombre de connexions TC dans une connexion MCS
        minThroughput           INTEGER (0..MAX),
                                -- nombre réel d'octets par seconde
        maxHeight               INTEGER (0..MAX),
                                -- limite quant à la hauteur d'un fournisseur
        maxMCSPDUsize           INTEGER (0..MAX),
                                -- limite quant au nombre d'octets des unités MCSPDU du domaine
        protocolVersion         INTEGER (0..MAX)
}

-- Partie 2: Fournisseur de connexions

Connect-Initial ::= [APPLICATION 101] IMPLICIT SEQUENCE
{
        callingDomainSelector   OCTET STRING,
        calledDomainSelector    OCTET STRING,
        upwardFlag              BOOLEAN,
                                -- prend la valeur TRUE si le fournisseur appelé est supérieur
        targetParameters        DomainParameters,
        minimumParameters       DomainParameters,
        maximumParameters       DomainParameters,
        userData                OCTET STRING
}

Connect-Response ::= [APPLICATION 102] IMPLICIT SEQUENCE
{
        result                  Result,
        calledConnectId         INTEGER (0..MAX),
                                -- cet entier est attribué par le fournisseur appelé afin d'identifier
                                -- les connexions de transport additionnelles appartenant à la
                                -- même connexion MCS
        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
}

-- Partie 3: Fusion de domaines

PDin ::= [APPLICATION 0] IMPLICIT SEQUENCE -- indication de plombage de domaines
{
        heightLimit             INTEGER (0..MAX)
                                -- restriction relative au récepteur d'unités MCSPDU
}
```

```
EDrq  ::=  [APPLICATION 1] IMPLICIT SEQUENCE  --  demande d'érection de domaine
{
        subHeight                   INTEGER (0..MAX),
                                    --  hauteur dans le domaine de l'émetteur d'unités MCSPDU
        subInterval                 INTEGER (0..MAX)
                                    --  son intervalle d'application de débit, en millisecondes
}

ChannelAttributes  ::=  CHOICE
{
        static                      [0]   IMPLICIT SEQUENCE
        {
            channelId                     StaticChannelId
                                          --  la valeur d'adhésion est implicitement TRUE
        },
        userId                      [1]   IMPLICIT SEQUENCE
        {
            joined                        BOOLEAN,
            userId                        UserId
                                          --  valeur TRUE si l'utilisateur a adhéré à son canal
                                          --  d'identification d'utilisateur
        },
        private                     [2]   IMPLICIT SEQUENCE
        {
            joined                        BOOLEAN,
                                          --  valeur TRUE si l'adhésion à l'identificateur de canal
                                          --  est effectuée à un niveau inférieur
            channelId                     PrivateChannelId,
            manager                       UserId,
            admitted                      SET OF UserId
                                          --  cet ensemble peut couvrir plusieurs demandes de fusion MCrq
        },
        assigned                    [3]   IMPLICIT SEQUENCE
        {
            channelId                     AssignedChannelId
                                          --  la valeur d'adhésion est implicitement à TRUE
        }
}

MCrq  ::=  [APPLICATION 2] IMPLICIT SEQUENCE  --  demande de fusion de canaux
{
        mergeChannels               SET OF ChannelAttributes,
        purgeChannelIds             SET OF ChannelId
}

MCcf  ::=  [APPLICATION 3] IMPLICIT SEQUENCE  --  confirmation de fusion de canaux
{
        mergeChannels               SET OF ChannelAttributes,
        purgeChannelIds             SET OF ChannelId
}

PCin  ::=  [APPLICATION 4] IMPLICIT SEQUENCE  --  indication de purge de canaux
{
        detachUserIds               SET OF UserId,
                                    --  purge de canaux d'identification d'utilisateur
        purgeChannelIds             SET OF ChannelId
                                    --  purge d'autres canaux
}
```

```
TokenAttributes ::= CHOICE
{
        grabbed                 [0]   IMPLICIT SEQUENCE
        {
            tokenId                       TokenId,
            grabber                       UserId
        },
        inhibited               [1]   IMPLICIT SEQUENCE
        {
            tokenId                       TokenId,
            inhibitors                    SET OF UserId
                                          -- cet ensemble peut couvrir plusieurs demandes de fusion MTrq
        },
        giving                  [2]   IMPLICIT SEQUENCE
        {
            tokenId                       TokenId,
            grabber                       UserId,
            recipient                     UserId
        },
        ungivable               [3]   IMPLICIT SEQUENCE
        {
            tokenId                       TokenId,
            grabber                       UserId
                                          -- le destinataire s'est détaché entre-temps
        },
        given                   [4]   IMPLICIT SEQUENCE
        {
            tokenId                       TokenId,
            recipient                     UserId
                                          -- le saisisseur a été libéré ou détaché
        }
}

MTrq ::= [APPLICATION 5] IMPLICIT SEQUENCE -- demande de fusion de jetons
{
        mergeTokens             SET OF TokenAttributes,
        purgeTokenIds           SET OF TokenId
}

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

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

-- Partie 4: Fournisseur du service de déconnexion

DPum ::= [APPLICATION 8] IMPLICIT SEQUENCE -- ultimatum du fournisseur du service de déconnexion
{
        reason                  Reason
}

RJum ::= [APPLICATION 9] IMPLICIT SEQUENCE -- ultimatum de rejet d'unité MCSPDU
{
        diagnostic              Diagnostic,
        initialOctets           OCTET STRING
}

-- Partie 5: Rattachement/détachement d'utilisateur

AUrq ::= [APPLICATION 10] IMPLICIT SEQUENCE -- demande de rattachement d'utilisateur
{
}
```

```
AUcf  ::=  [APPLICATION 11] IMPLICIT SEQUENCE  --  confirmation de rattachement d'utilisateur
{
        result                  Result,
        initiator               UserId OPTIONAL
}

DUrq  ::=  [APPLICATION 12] IMPLICIT SEQUENCE  --  demande de détachement d'utilisateur
{
        reason                  Reason,
        userIds                 SET OF UserId
}

DUin  ::=  [APPLICATION 13] IMPLICIT SEQUENCE  --  indication de détachement d'utilisateur
{
        reason                  Reason,
        userIds                 SET OF UserId
}

-- Partie 6: Gestion de canal

CJrq  ::=  [APPLICATION 14] IMPLICIT SEQUENCE  --  demande d'adhésion à un canal
{
        initiator               UserId,
        channelId               ChannelId
                                -- l'identificateur peut prendre la valeur zéro
}

CJcf  ::=  [APPLICATION 15] IMPLICIT SEQUENCE  --  confirmation d'adhésion à un canal
{
        result                  Result,
        initiator               UserId,
        requested               ChannelId,
                                -- peut avoir la valeur zéro
        channelId               ChannelId OPTIONAL
}

CLrq  ::=  [APPLICATION 16] IMPLICIT SEQUENCE  --  demande de sortie de canal
{
        channelIds              SET OF ChannelId
}

CCrq  ::=  [APPLICATION 17] IMPLICIT SEQUENCE  --  demande de constitution de canal
{
        initiator               UserId
}

CCcf  ::=  [APPLICATION 18] IMPLICIT SEQUENCE  --  confirmation de constitution de canal
{
        result                  Result,
        initiator               UserId,
        channelId               PrivateChannelId OPTIONAL
}

CDrq  ::=  [APPLICATION 19] IMPLICIT SEQUENCE  --  demande de dissolution de canal
{
        initiator               UserId,
        channelId               PrivateChannelId
}

CDin  ::=  [APPLICATION 20] IMPLICIT SEQUENCE  --  indication de dissolution de canal
{
        channelId               PrivateChannelId
}

CArq  ::=  [APPLICATION 21] IMPLICIT SEQUENCE  --  demande d'admission de canal
{
        initiator               UserId,
        channelId               PrivateChannelId,
        userIds                 SET OF UserId
}
```

```
CAin  ::=  [APPLICATION 22] IMPLICIT SEQUENCE  --  indication d'admission de canal
{
        initiator                   UserId,
        channelId                   PrivateChannelId,
        userIds                     SET OF UserId
}

CErq  ::=  [APPLICATION 23] IMPLICIT SEQUENCE  --  demande d'expulsion de canal
{
        initiator                   UserId,
        channelId                   PrivateChannelId,
        userIds                     SET OF UserId
}

CEin  ::=  [APPLICATION 24] IMPLICIT SEQUENCE  --  indication d'expulsion de canal
{
        channelId                   PrivateChannelId,
        userIds                     SET OF UserId
}

-- Partie 7: Transfert de données

SDrq  ::=  [APPLICATION 25] IMPLICIT SEQUENCE  --  demande d'envoi de données
{
        initiator                   UserId,
        channelId                   ChannelId,
        dataPriority                DataPriority,
        segmentation                Segmentation,
        userData                    OCTET STRING
}

SDin  ::=  [APPLICATION 26] IMPLICIT SEQUENCE  --  indication d'envoi de données
{
        initiator                   UserId,
        channelId                   ChannelId,
        dataPriority                DataPriority,
        segmentation                Segmentation,
        userData                    OCTET STRING
}

USrq  ::=  [APPLICATION 27] IMPLICIT SEQUENCE  --  demande d'envoi de données en séquences uniformes
{
        initiator                   UserId,
        channelId                   ChannelId,
        dataPriority                DataPriority,
        segmentation                Segmentation,
        userData                    OCTET STRING
}

USin  ::=  [APPLICATION 28] IMPLICIT SEQUENCE  --  indication d'envoi de données en séquences uniformes
{
        initiator                   UserId,
        channelId                   ChannelId,
        dataPriority                DataPriority,
        segmentation                Segmentation,
        userData                    OCTET STRING
}

-- Partie 8: Gestion de jeton

TGrq  ::=  [APPLICATION 29] IMPLICIT SEQUENCE  --  demande de saisie de jeton
{
        initiator                   UserId,
        tokenId                     TokenId
}
```

```
TGcf  ::=  [APPLICATION 30] IMPLICIT SEQUENCE  --  confirmation de saisie de jeton
{
        result                  Result,
        initiator               UserId,
        tokenId                 TokenId,
        tokenStatus             TokenStatus
}

TIrq  ::=  [APPLICATION 31] IMPLICIT SEQUENCE  --  demande d'inhibition de jeton
{
        initiator               UserId,
        tokenId                 TokenId
}

TIcf  ::=  [APPLICATION 32] IMPLICIT SEQUENCE  --  confirmation d'inhibition de jeton
{
        result                  Result,
        initiator               UserId,
        tokenId                 TokenId,
        tokenStatus             TokenStatus
}

TVrq  ::=  [APPLICATION 33] IMPLICIT SEQUENCE  --  demande de cession de jeton
{
        initiator               UserId,
        tokenId                 TokenId,
        recipient               UserId
}

TVin  ::=  [APPLICATION 34] IMPLICIT SEQUENCE  --  indication de cession de jeton
{
        initiator               UserId,
        tokenId                 TokenId,
        recipient               UserId
}

TVrs  ::=  [APPLICATION 35] IMPLICIT SEQUENCE  --  réponse de cession de jeton
{
        result                  Result,
        recipient               UserId,
        tokenId                 TokenId
}

TVcf  ::=  [APPLICATION 36] IMPLICIT SEQUENCE  --  confirmation de cession de jeton
{
        result                  Result,
        initiator               UserId,
        tokenId                 TokenId,
        tokenStatus             TokenStatus
}

TPrq  ::=  [APPLICATION 37] IMPLICIT SEQUENCE  --  requête de demande de jeton
{
        initiator               UserId,
        tokenId                 TokenId
}

TPin  ::=  [APPLICATION 38] IMPLICIT SEQUENCE  --  indication de demande de jeton
{
        initiator               UserId,
        tokenId                 TokenId
}

TRrq  ::=  [APPLICATION 39] IMPLICIT SEQUENCE  --  demande de libération de jeton
{
        initiator               UserId,
        tokenId                 TokenId
}
```

```
TRcf  ::=  [APPLICATION 40] IMPLICIT SEQUENCE  --  confirmation de libération de jeton
{
        result                  Result,
        initiator               UserId,
        tokenId                 TokenId,
        tokenStatus             TokenStatus
}

TTrq  ::=  [APPLICATION 41] IMPLICIT SEQUENCE  --  demande de test de jeton
{
        initiator               UserId,
        tokenId                 TokenId
}

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

-- Partie 9: Codes d'états

Reason  ::=  ENUMERATED         --  codes contenus dans des unités 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         --  codes contenus dans des unités de type Connect, réponse, confirmation
{
                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     --  codes contenus dans des unités de type 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)
}

-- Partie 10: Répertoire des unités MCSPDU

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 Codage des unités MCSPDU

Chaque unité MCSPDU est transportée sous forme d'une unité TSDU sur une connexion de transport (TC) appartenant à une connexion du service MCS. Les unités MCSPDU de type connect sont de longueur illimitée. Les unités MCSPDU de domaine sont limitées en longueur par un paramètre du domaine MCS.

Un codage normalisé des valeurs de données ASN.1 est utilisé pour transférer des unités MCSPDU entre des fournisseurs de service MCS homologues. Deux versions du présent protocole sont définies, ne différant que par la spécification des règles de codage:

– *version 1* – Utilise les règles de codage de base de la Recommandation X.209 du CCITT pour toutes les unités MCSPDU;

– *version 2* – Utilise les règles de codage de base pour les unités MCSPDU de type connect et les règles de codage en métalangage condensé (PER) de l'ISO/CEI 8825-2 pour toutes les unités MCSPDU de domaine suivantes. Plus précisément, la variante ALIGNED de l'élément BASIC-PER doit être appliquée aux unités de type ASN.1 **DomainMCSPDU**. La chaîne d'éléments binaires ainsi produite doit être acheminée sous forme d'un nombre entier d'octets. Le bit initial de cette chaîne doit coïncider avec le bit de poids le plus fort du premier octet.

L'Appendice I donne un exemple d'unité MCSPDU pour transfert de données, selon les variantes de codage de la version 1 et de la version 2.

La négociation de la version du protocole implique l'échange d'une unité MCSPDU de type **Connect-Initial** et d'une unité MCSPDU de type **Connect-Response** sur la connexion de transport initiale. Ces deux unités seront toujours codées selon les règles de codage de base, de même que toutes unités MCSPDU de type **Connect-Additional** et **Connect-Result** pouvant venir ensuite, avant l'apparition d'unités MCSPDU de domaine. Celles-ci commencent avec la deuxième TSDU transmise sur une connexion de transport.

La version 2 du présent protocole ne doit pas être utilisée avant que les règles de codage en métalangage condensé aient été adoptées en tant que Recommandation UIT-T | Norme internationale ISO/CEI.

NOTES

1 Les règles de codage en métalangage condensé produisent des en-têtes plus compacts pour les unités MCSPDU.

2 Les règles de codage de base (BER) et condensées (PER) sont auto-délimitatrices, en ce sens qu'elles contiennent assez de renseignements pour localiser la fin de chaque unité MCSPDU codée. On pourrait en tirer la conclusion que cela rend inutile l'utilisation d'unités TSDU et que ce protocole pourrait être mis en œuvre au moyen de services de transport non normalisés, acheminant des trains d'octets sans tenir compte des frontières entre unités TSDU. Une telle conception prête cependant plus le flanc à des erreurs de mise en œuvre car si jamais les frontières entre unités MCSPDU étaient perdues, le rétablissement serait très difficile.

# 9 Acheminement des unités MCSPDU

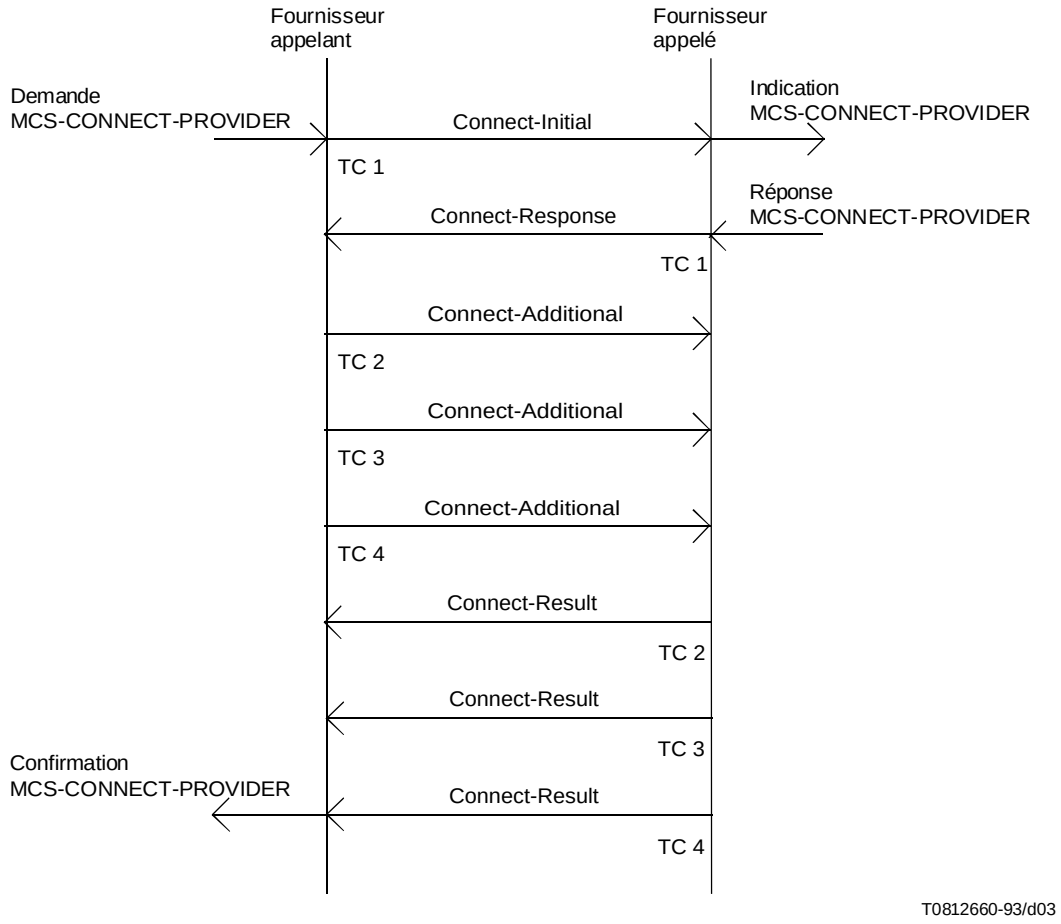
## 9.1 Unités MCSPDU de type connect

La Figure 9-1 spécifie l'échange d'unités MCSPDU de type connect.

Dès qu'il reçoit une unité de type **Connect-Response**, le fournisseur MCS appelant prend connaissance de la valeur négociée du nombre de priorités de transfert de données mises en œuvre dans le domaine. A titre d'exemple, on a représenté des unités MCSPDU de type connect sur des connexions de transport additionnelles suivant strictement la séquence 2, 3, 4 au niveau du fournisseur MCS appelé. En réalité, on ne peut pas établir de connexions de transport dans le même ordre que celui de leur demande: une unité de type **Connect-Additional** peut arriver dans un ordre différent de celui où elle a été envoyée, ce qui entraînera le retour d'une unité **Connect-Result** dans un ordre également différent. Ou encore, cette dernière unité peut, même si elle a été envoyée en séquence, être réordonnancée lors de son transit. Un fournisseur MCS appelant n'a pas besoin d'attendre que toutes les connexions de transport additionnelles soient établies avant d'envoyer la première unité **Connect-Additionnal**. Le fournisseur MCS appelé n'a pas besoin d'attendre l'arrivée d'une série complète d'unités MCSPDU de type **Connect-Additional** pour retourner une unité de type **Connect-Result**. La réception d'une série complète d'unités de résultats favorables au niveau du fournisseur MCS appelant, dans un ordre ou dans un autre, engendre une primitive de confirmation MCS-CONNECT-PROVIDER contenant une valeur favorable.

Une unité de type **Connect-Response** ou **Connect-Result** contenant une valeur défavorable, ou une primitive d'indication T-DISCONNECT, reçue à un point intermédiaire quelconque, provoquera la déconnexion de toutes les connexions de transport appartenant jusque là à la connexion MCS tout en engendrant une primitive de confirmation MCS-CONNECT-PROVIDER contenant une valeur défavorable.

Une primitive de demande MCS-CONNECT-PROVIDER spécifiera lequel des deux fournisseurs de service MCS a le niveau hiérarchique le plus élevé. Cette relation hiérarchique détermine l'acheminement ultérieur des unités MCSPDU de domaine et la distinction entre fournisseur MCS appelant et fournisseur MCS appelé ne sera donc plus applicable. Par exemple, l'étape suivante consistera, pour la couche du service MCS, à fusionner les ressources des deux domaines jusque là indépendants. Des unités de type **MCrq** et **MTrq** seront produites par le fournisseur MCS inférieur et seront transmises sur la nouvelle connexion MCS à destination du fournisseur MCS supérieur. Le sens dans lequel ces unités MCSPDU sont envoyées peut être d'appelant à appelé ou d'appelé à appelant, selon la façon dont le fanion de montée a été positionné.




NOTE – Le numéro et l'ordre relatif aux TC additionnelles peuvent varier.

FIGURE 9-1/T.125

**Flux de messages des unités MCSPDU de type connect**

## 9.2 Unités MCSPDU de domaine

Le Tableau 9-1 spécifie l'acheminement des unités MCSPDU de domaine.

Si un fournisseur MCS produit ou envoie une unité MCSPDU de la catégorie des demandes (*request*), cette unité transitera en amont sur l'unique connexion MCS. Les demandes de type **EDrq, CJrq** et **CLrq** peuvent être absorbées au niveau d'un quelconque fournisseur MCS intermédiaire. Les autres demandes remontent vers le fournisseur MCS supérieur pour y être traitées, à moins que le contenu d'une de ces demandes ne soit jugé invalide, auquel cas l'unité MCSPDU correspondante peut être ignorée, sans confirmation.

TABLEAU 9-1/T.125

**Acheminement des unités MCSPDU de domaine**

| Catégorie | Unités MCSPDU | | | TC | Sens |
|---|---|---|---|---|---|
| Demande | **EDrq** | **MCrq** | **MTrq** | I | Ascendant |
| | **AUrq** | **DUrq** | | | |
| | **CJrq** | **CLrq** | **CCrq** | | |
| | **CDrq** | **CArq** | **CErq** | | |
| | **TGrq** | **TIrq** | **TVrq** | | |
| | **TPrq** | **TRrq** | **TTrq** | | |
| | **SDrq** | **USrq** | | A | |
| Indication | **PDin** | **PCin** | **PTin** | I | Descendant |
| | | **DUin** | | | |
| | **CDin** | **CAin** | **CEin** | | |
| | | | **TVin** | | |
| | **TPin** | | | | |
| | **SDin** | **USin** | | A | |
| Réponse | | | **TVrs** | I | Ascendant |
| Confirmation | | **MCcf** | **MTcf** | I | Descendant |
| | **AUcf** | | | | |
| | **CJcf** | | **CCcf** | | |
| | **TGcf** | **TIcf** | **TVcf** | | |
| | | **TRcf** | **TTCF** | | |
| Ultimatum | **DPum** | **RJum** | | I | Asc. ou Desc. |

| | |
|---|---|
| I | L'unité MCSPDU traverse la connexion de transport initiale. |
| A | L'unité MCSPDU peut traverser une connexion de transport additionnelle, selon sa priorité de transfert de données. |
| Ascendant | L'unité MCSPDU remonte vers le fournisseur MCS supérieur. |
| Descendant | L'unité MCSPDU redescend du fournisseur MCS supérieur. |

Si un fournisseur MCS produit ou envoie une unité MCSPDU de la catégorie des indications (*indication*), des copies de cette unité, de contenu éventuellement modifié, descendront sur zéro ou un autre nombre de connexions MCS, conformément aux règles suivantes:

a) une unité d'indication de type **PDin** est envoyée en aval sur toutes les connexions MCS. La hauteur limite qu'elle contient est diminuée de 1. Un fournisseur MCS recevant cette unité MCSPDU avec une hauteur limite égale à zéro doit déconnecter;

b) une unité de type **PCin** est envoyée en aval sur toutes les connexions MCS. L'ensemble des identificateurs d'utilisateurs envoyé n'est pas modifié, de manière que tous les utilisateurs détachés soient annoncés à ceux qui restent attachés. L'ensemble des autres identificateurs de canal expédiés peut être limité à ceux qui sont en cours d'utilisation dans un sous-arbre. Chez un ancien fournisseur supérieur qui est encore en cours de fusion avec un domaine supérieur, aussi bien les identificateurs d'utilisateurs que les identificateurs d'autres canaux sont limités à ceux dont l'acceptation dans le domaine supérieur a été confirmée;

c) une unité de type **PTin** est envoyée en aval sur toutes les connexions MCS. L'ensemble d'identificateurs de jeton expédiés peut être limité à ceux qui sont en cours d'utilisation dans un sous-arbre. Chez un ancien fournisseur supérieur qui est encore en cours de fusion avec un domaine supérieur, les identificateurs d'utilisateurs sont limités à ceux dont l'acceptation dans le domaine supérieur a été confirmée;

d) une unité de type **DUin** est envoyée en aval sur toute les connexions MCS. L'ensemble d'identificateurs d'utilisateur expédié reste inchangé, de sorte que tous les utilisateurs détachés seront annoncés à ceux qui restent. Chez un ancien fournisseur supérieur qui est encore en cours de fusion avec un domaine supérieur, les identificateurs d'utilisateurs sont limités à ceux dont l'acceptation dans le domaine supérieur a été confirmée;

e) une unité de type **CDin** est envoyée en aval sur toutes les connexions MCS qui contiennent dans leur sous-arbre le gestionnaire du canal privé ou tout utilisateur admis;

f) des unités de type **CAin** et **CEin** sont envoyées en aval sur toutes les connexions MCS qui contiennent dans leur sous-arbre un ou plusieurs des utilisateurs affectés. L'ensemble d'identificateurs d'utilisateur expédié peut être limité à ceux qui résident dans un sous-arbre;

g) une unité de type **TVin** est envoyée en aval sur une seule connexion MCS qui contient le destinataire désigné dans son sous-arbre;

h) une unité de type **TPin** est envoyée en aval sur toutes les connexions MCS qui contiennent dans leur sous-arbre un utilisateur qui a saisi, inhibé ou va recevoir le jeton;

i) des unités de type **SDin** et **USin** sont envoyées en aval sur toutes les connexions MCS qui adhèrent au canal spécifié, sauf qu'une unité **SDin**, lorsqu'elle est produite, n'est pas réexpédiée sur la connexion par laquelle l'unité **SDrq** est arrivée.

Les unités d'indication de type **PCin, PTin, DUin, CAin** et **CEin** n'ont pas besoin d'être envoyées si les ensembles d'identificateurs qu'elles contiennent sont vides.

Si un fournisseur de service MCS produit ou envoie une unité MCSPDU de la catégorie des réponses (*response*), celle-ci transitera en amont sur l'unique connexion MCS. Elle remontera vers le fournisseur MCS supérieur pour y être traitée, à moins que son contenu ne soit jugé invalide.

Si un fournisseur de service MCS produit ou envoie une unité MCSPDU de la catégorie des confirmations (*confirm*), celle-ci transitera en aval sur une seule connexion MCS, conformément aux règles suivantes:

a) une unité de confirmation de type **MCcf** refait, dans le sens opposé, le chemin parcouru par la plus récente demande **MCrq** qui n'a pas encore fait l'objet d'une confirmation en réponse. Cela implique que chaque fournisseur MCS maintienne une file d'attente des requêtes en instance, du type premier entré-premier sorti;

b) une unité de type **MTcf** refait, dans le sens opposé, le chemin parcouru par la plus récente demande **MTrq** qui n'a pas encore fait l'objet d'une confirmation en réponse. Cela implique que chaque fournisseur MCS maintienne une file d'attente des requêtes en instance, du type premier entré-premier sorti;

c) une unité de type **AUcf** refait, dans le sens opposé, le chemin parcouru par une demande **AUrq** antérieure – peu importe laquelle si plusieurs sont en instance – n'ayant pas encore fait l'objet d'une confirmation en réponse. Pour la bonne règle, il convient cependant que chaque fournisseur de service MCS maintienne une file d'attente de ces unités, du type premier entré-premier sorti. Lorsqu'il expédie une confirmation **AUcf**, un fournisseur MCS doit enregistrer à quel sous-arbre l'identificateur d'utilisateur contenu dans cette unité va être ainsi assigné;

d) d'autres unités MCSPDU de la catégorie des confirmations contiennent un identificateur d'utilisateur initiateur qui avait déjà été attribué dans le cadre de l'action de confirmation par **AUcf** qui vient d'être exposée. Ces unités MCSPDU sont envoyées en aval sur la connexion MCS qui aboutit au sous-arbre auquel l'identificateur d'utilisateur a été attribué. Poursuivant leur chemin, les unités finissent par revenir au fournisseur qui couvre le rattachement MCS demandeur.

Les unités de confirmation sont produites au cours du traitement, comme les demandes. Toutes ces unités, sauf **CJcf**, sont produites par le fournisseur MCS supérieur.

Si un fournisseur MCS produit une unité MCSPDU du type *ultimatum*, cette unité transitera sur une seule connexion MCS, soit en amont ou en aval. L'unité **DPum** donne au fournisseur MCS récepteur l'ordre de déconnecter la connexion MCS qui l'achemine. L'unité **RJum** rejette une unité MCSPDU erronée avec un code de diagnostic. Elle invite également le fournisseur qui l'a transmise à opérer la déconnexion. Les unités d'ultimatum ne sont pas expédiées.

# 10 Signification des unités MCSPDU

Les Tableaux 10-1 à 10-47 reprennent le contenu de chacune des unités MCSPDU qui ont été définies dans l'article 8.

## 10.1 Connect-Initial

L'unité de type **Connect-Initial** est produite par une primitive de demande MCS-CONNECT-PROVIDER. Elle est envoyée en tant que première unité TSDU sur la connexion de transport initiale d'une nouvelle connexion MCS. Elle produit, au niveau du récepteur, une primitive d'indication MCS-CONNECT-PROVIDER.

TABLEAU 10-1/T.125

**Unité MCSPDU de type Connect-Initial**

| Contenu | Source | Collecteur |
|---|---|---|
| Sélecteur de domaine appelant | Demande | Indication |
| Sélecteur de domaine appelé | Demande | Indication |
| Fanion ascendant | Demande | Indication |
| Paramètres de domaine cibles | Demande | Indication |
| Paramètres de domaine minimaux | Demande | Indication |
| Paramètres de domaine maximaux | Demande | Indication |
| Données d'utilisateur | Demande | Indication |

L'adresse de transport appelante et l'adresse de transport appelée sont des paramètres additionnels de la demande et de l'indication MCS-CONNECT-PROVIDER qui deviennent des paramètres de la primitive T-CONNECT et qui ne sont pas explicitement insérés dans une unité MCSPDU. La même paire d'adresses de transport doit être utilisée pour demander toutes les connexions de transport appartenant à la même connexion de couche MCS.

La qualité du service de transport est un paramètre additionnel de la demande – mais non de l'indication – MCS-CONNECT-PROVIDER. La qualité de service peut varier d'une connexion de transport à une autre et la qualité disponible n'est révélée que lors de l'établissement de chaque connexion de transport. Comme le nombre de connexions de transport additionnelles n'est pas connu avant que la primitive MCS-CONNECT-PROVIDER ait négocié, le paramètre de domaine relatif au nombre de priorités de données mises en œuvre, cette primitive ne peut pas en même temps négocier complètement la qualité du service de transport de ces connexions. Un fournisseur MCS appelé doit donc accepter automatiquement les connexions de transport entrantes au niveau de qualité de service offert, du moment que ce niveau est conforme à toute valeur minimale spécifiée à titre d'option par le fournisseur MCS appelant et indiquée par chaque primitive T-CONNECT.

L'interprétation des valeurs du sélecteur de domaine relève d'une décision locale pour chaque fournisseur de service MCS. Ces valeurs sont des chaînes d'octets qui ont les caractéristiques d'une adresse. Les valeurs acceptables peuvent être déterminées au cours du processus de configuration d'un fournisseur MCS. Un même domaine peut être sélectionné par plusieurs valeurs. Un sélecteur de domaine non spécifié est représenté par une chaîne d'octets de longueur nulle. Ce conflit peut être résolu, conformément à des accords locaux, en donnant à cette chaîne une valeur explicite quelconque.

Le fanion ascendant spécifie le sens d'une nouvelle connexion MCS; il prend la valeur *vrai* si le fournisseur appelé est à considérer comme ayant un niveau supérieur à celui du fournisseur appelant et la valeur *faux* dans le cas contraire. Un fournisseur de service MCS joue un rôle dans la hiérarchie d'un domaine selon le sens des connexions MCS auxquelles il participe. Aucun fournisseur ne doit autoriser deux connexions vers des fournisseurs de niveau supérieur. Un fournisseur sans connexion vers un fournisseur de niveau supérieur doit remplir le rôle de fournisseur MCS supérieur.

Les paramètres de domaine cibles de l'unité **Connect-Initial** doivent chacun avoir une valeur comprise entre les valeurs maximale et minimale spécifiées. Un fournisseur MCS doit réviser les paramètres de domaine demandés pour tenir compte des limites de sa mise en œuvre ou pour imposer des valeurs déjà convenues par les membres existants d'un domaine. Le fournisseur peut augmenter les valeurs minimales et diminuer les valeurs maximales. Il ne doit modifier les cibles que pour les faire cadrer avec l'intervalle. Il y a lieu qu'un fournisseur de service MCS soit disposé à donner suite à toute réponse satisfaisant à la gamme de valeurs qu'il propose.

Les données d'utilisateur forment une chaîne d'octets arbitraire, qui peut avoir une longueur nulle.

Un fournisseur de service MCS accepte automatiquement chaque connexion de transport, selon ses limites de capacité. Les données d'utilisateur contenues dans la primitive T-CONNECT ne sont pas utilisées. La première unité TSDU reçue lors du transfert de données, qui sera une MCSPDU de type **Connect-Initial** ou **Connect-Additional**, déterminera la nature de la connexion de transport. Si le contenu n'est pas acceptable, un fournisseur MCS appelé peut déconnecter immédiatement la connexion de transport. La réaction préférée consiste à retourner une unité **Connect-Response** ou **Connect-Result**, selon le cas, expliquant pourquoi la connexion MCS n'a pu être établie. Le fournisseur MCS appelant doit ensuite déconnecter.

## 10.2 Connect-Response

L'unité **Connect-Response** est produite par une primitive de réponse MCS-CONNECT-PROVIDER. C'est la première unité TSDU qui est renvoyée dans le sens inverse sur la connexion de transport initiale d'une nouvelle connexion MCS. Cette unité signale au fournisseur MCS appelant l'acceptation d'une connexion de couche MCS. Ce fournisseur procède ensuite à l'établissement de toutes connexions de transport requises.

TABLEAU 10-2/T.125

**Unité MCSPDU de type Connect-Response**

| Contenu | Source | Collecteur |
|---|---|---|
| Résultat | Réponse | Confirmation |
| Paramètres de domaine | Réponse | Confirmation |
| Identificateur de connexion chez l'appelé | Fournisseur appelé | Fournisseur appelant |
| Données d'utilisateur | Réponse | Confirmation |

Si le résultat est favorable, cette unité MCSPDU fixe les paramètres de domaine effectifs, dont le nombre de priorités de transfert de données MCS mises en œuvre, égal au nombre de connexions de transport appartenant à une connexion de couche MCS. Si ce nombre est supérieur à un, des connexions de transport additionnelles doivent être créées et associées à la connexion MCS par échange d'unités MCSPDU de type **Connect-Additional** et **Connect-Result**.

L'identificateur de connexion pour l'appelé sert d'outil pour associer des connexions de transport additionnelles entrantes à la connexion de transport initiale, au niveau du fournisseur MCS appelé. Sa valeur est choisie à cette seule fin. Il doit identifier uniquement une connexion MCS en cours au niveau du fournisseur appelé. La signification de cet identificateur ne dure pas au-delà de l'exécution de la primitive MCS-CONNECT-PROVIDER.

La plupart des paramètres de la primitive de confirmation MCS-CONNECT-PROVIDER sont acheminés dans l'unité **Connect-Response**. Si le résultat est défavorable ou si aucune connexion de transport additionnelle n'est requise, la confirmation est émise immédiatement. Sinon, elle est retardée jusqu'au moment où les résultats sont connus pour l'affectation de connexions de transport additionnelles à la connexion de couche MCS.

### 10.3 Connect-Additional

L'unité de type **Connect-Additional** est produite dès réception de l'unité **Connect-Response**. Elle est envoyée en tant que première unité TSDU sur une connexion de transport additionnelle d'une nouvelle connexion de couche MCS.

Le paramètre priorité des données prend les valeurs haute, moyenne, basse, en séquence, jusqu'au nombre de connexions de transport additionnelles requises.

TABLEAU 10-3/T.125

**Unité MCSPDU de type Connect-Additional**

| Contenu | Source | Collecteur |
|---|---|---|
| Identificateur de connexion chez l'appelé | Fournisseur appelant | Fournisseur appelé |
| Priorité des données | Fournisseur appelant | Fournisseur appelé |

### 10.4 Connect-Result

L'unité **Connect-Result** est produite dès réception d'une unité **Connect-Additional**. Il s'agit de la première unité TSDU renvoyée dans le sens inverse sur une connexion de transport additionnelle d'une nouvelle connexion de couche MCS.

TABLEAU 10-4/T.125

**Unité MCSPDU de type Connect-Result**

| Contenu | Source | Collecteur |
|---|---|---|
| Résultat | Fournisseur appelé | Confirmation |

Si un résultat quelconque est défavorable, la confirmation MCS-CONNECT-PROVIDER doit être immédiatement produite. Toutes les connexions de transport associées à la connexion MCS doivent être déconnectées et toutes unités MCSPDU qu'elles acheminent doivent être ignorées.

Dans le cas contraire, on attendra des résultats favorables pour chaque connexion de transport additionnelle. Ces résultats pourront être retournés hors séquence. Lorsqu'ils auront tous été collectés, une primitive de confirmation MCS-CONNECT-PROVIDER sera émise.

Après la confirmation MCS-CONNECT-PROVIDER, des unités MCSPDU de domaine pourront transiter par une connexion MCS. Chaque connexion MCS appartient à un seul domaine. Dans les configurations où un fournisseur MCS couvre plusieurs domaines, la connexion MCS qui achemine les MCSPDU détermine le domaine auquel celles-ci sont applicables. Dans le reste du présent article, la description de chaque unité MCSPDU est placée dans le contexte d'un seul domaine.

## 10.5 PDin

L'unité d'indication **PDin** est produite dès que la primitive MCS-CONNECT-PROVIDER a été suivie d'effet. Cette unité raccorde par «plombage» la hiérarchie des fournisseurs MCS au-dessous d'une nouvelle connexion MCS pour faire en sorte qu'aucune boucle n'ait été créée. L'unité **PDin** est également produite par le fournisseur MCS supérieur pour mettre en application la hauteur maximale d'un domaine.

TABLEAU 10-5/T.125

**Unité MCSPDU de type PDin**

| Contenu | Source | Collecteur |
|---|---|---|
| Limite de hauteur | Fournisseur supérieur antérieur ou présent | Subordonnés |

L'unité **PDin** est produite à l'extrémité inférieure d'une nouvelle connexion MCS, par le fournisseur qui a cessé d'être le fournisseur supérieur d'un domaine. Son contenu est initialisé conformément au paramètre de domaine relatif à la hauteur maximale du domaine. L'unité **PDin** est transmise en aval sur toutes les connexions MCS.

Le cas échéant, l'unité **PDin** est produite de la même façon au niveau du fournisseur MCS supérieur.

Chaque fois qu'une unité **PDin** est reçue, la hauteur limite qu'elle contient est contrôlée. Si elle est supérieure à zéro, cette limite est diminuée de 1 et **PDin** est envoyée en aval sur toutes les connexions MCS. Une valeur égale à zéro signifie par ailleurs que le récepteur est trop éloigné du fournisseur supérieur. Il doit réagir en déconnectant la connexion MCS ascendante, ce qui supprime un sous-arbre entier et contribue à reconstituer la hauteur du domaine.

En présence d'une boucle, la hauteur limite contenue dans l'unité **PDin** doit diminuer jusqu'à atteindre zéro. Le fournisseur qui détecte la boucle doit l'interrompre, quitte à retrancher du domaine tous les fournisseurs de la boucle ainsi que leurs subordonnés.

NOTE – Un fournisseur MCS, n'ayant qu'une connaissance purement locale des connexions MCS, ne peut pas empêcher la formation de boucles. Il peut veiller à ce qu'il n'y ait, à tout moment, qu'une seule connexion ascendante. Mais il ne peut pas garantir que cette connexion ascendante ne se reboucle pas avec un certain fournisseur situé plus bas dans la hiérarchie. Lorsqu'une boucle se forme, la cause immédiate est la création d'une connexion ascendante erronée à partir du fournisseur MCS supérieur. Les applications de régulation, qui spécifient les connexions MCS à créer, doivent faire en sorte que de telles erreurs ne se produisent pas.

## 10.6 EDrq

L'unité de demande **EDrq** est produite dès que la primitive MCS-CONNECT-PROVIDER a été suivie d'effet. Elle communique dans le sens ascendant des modifications de hauteur de fournisseurs ainsi que les intervalles d'application de débit correspondants. L'unité **EDrq** est produite par un fournisseur MCS chaque fois que la hauteur ou l'intervalle de celui-ci change.

La hauteur d'un fournisseur MCS peut changer lorsqu'une connexion MCS est ajoutée ou supprimée ou lorsqu'un fournisseur subordonné signale une modification au moyen d'une unité **EDrq**. Son intervalle de supervision pour mettre en application le débit minimal spécifié sous forme d'un paramètre de domaine peut se modifier par adaptation aux intervalles signalés par des subordonnés ou pour d'autres raisons. Si la valeur de hauteur ou d'intervalle change, le fournisseur MCS doit transmettre une unité de demande **EDrq** à son supérieur immédiat.

TABLEAU 10-6/T.125

**Unité MCSPDU de type EDrq**

| Contenu | Source | Collecteur |
|---|---|---|
| Hauteur dans le domaine | Subordonné | Fournisseur supérieur |

## 10.7 MCrq

L'unité de demande **MCrq** est produite dès que la primitive MCS-CONNECT-PROVIDER est suivie d'effet. Elle communique dans le sens ascendant les attributs des canaux dépendant d'un précédent fournisseur supérieur de manière qu'ils puissent être intégrés dans le domaine fusionné.

TABLEAU 10-7/T.125

**Unité MCSPDU de type MCrq**

| Contenu | Source | Collecteur |
|---|---|---|
| Canaux de fusion | Précédent fournisseur supérieur | Fournisseur supérieur |
| Identificateurs de canal de purge | Fournisseurs intermédiaires | Fournisseur supérieur |

On peut insérer dans l'unité **MCrq** les attributs de plusieurs canaux, dans les limites fixées par le domaine quant à la longueur des unités MCSPDU. Comme indiqué dans les définitions ASN.1 de l'article 7, chacun des quatre types de canaux utilisés (statiques, d'identificateurs d'utilisateur, privés, attribués) possède son propre ensemble d'attributs. Ceux-ci sont conservés dans la base de données du fournisseur MCS supérieur et sont partiellement copiés dans les sous-arbres où un canal est utilisé. Lorsque deux domaines sont fusionnés par l'action d'une primitive MCS-CONNECT-PROVIDER, les canaux qui sont en service dans le domaine inférieur doivent soit être intégrés dans la base de données du domaine supérieur ou être purgés du domaine inférieur. Cette décision relève du fournisseur supérieur du domaine fusionné.

Chaque canal doit être pris en compte individuellement. Si les limites de domaine concernant les canaux utilisés le permettent et si l'identificateur de canal n'a pas déjà été mis en œuvre de manière conflictuelle, le domaine supérieur doit s'agrandir pour intégrer ce canal. L'utilisation d'un identificateur de canal statique ne provoque jamais de conflit. L'utilisation d'un identificateur de canal privé dans le domaine inférieur ne provoque pas de conflit si cet identificateur est également utilisé pour identifier un canal privé dans le domaine supérieur et s'il possède le même identificateur d'utilisateur que le gestionnaire. Toutes les autres combinaisons d'utilisations simultanées sont interdites car elles imposent que l'identificateur de canal soit purgé du domaine inférieur.

Si un canal privé possède un ensemble de nombreux utilisateurs admis, ses attributs peuvent ne pas s'intégrer dans une même unité de demande **Mcrq**: ces attributs doivent être envoyés vers le haut de la hiérarchie dans plusieurs unités MCSPDU. La deuxième demande de fusion du même canal privé, ainsi que les demandes subséquentes, doivent être tenues en attente de la réception d'une confirmation **MCcf** en réponse à la première demande. Ce n'est qu'à ce moment que l'on sait si les limites de domaine ont permis que le canal soit mis en œuvre dans le domaine supérieur. Si la première demande n'est pas suivie d'effet, elle ne doit pas être renouvelée avec un sous-ensemble restant d'utilisateurs admis.

Chaque unité **MCrq** suscite une réponse de confirmation de type **MCcf** de la part du fournisseur MCS supérieur, dans la même séquence. Une unité **MCcf** ne contient rien qui permette d'identifier expressément l'unité **MCrq** précédente. Les réponses ne doivent être acheminées que dans l'ordre où ces unités ont été reçues. Les fournisseurs MCS situés au-dessus du précédent supérieur doivent effectuer un enregistrement de chaque unité **MCrq** non suivie de réponse et du point dont elle provient, de manière que la confirmation **MCcf** correspondante puisse être renvoyée via la même connexion MCS.

Les fournisseurs MCS intermédiaires doivent valider les identificateurs d'utilisateur annoncés dans les attributs de canal privé, pour faire en sorte qu'ils soient légitimement attribués au sous-arbre dont l'unité **MCrq** provient. Les identificateurs d'utilisateur invalides doivent être supprimés. Si le gestionnaire d'un canal privé est supprimé, tous les attributs de ce canal doivent être retranchés de la demande de fusion et seul l'identificateur du canal doit être inséré dans l'ensemble à purger. A l'exception de cette validation des identificateurs d'utilisateur, les fournisseurs MCS intermédiaires ne doivent pas modifier le contenu d'une unité **MCrq**.

Un précédent fournisseur supérieur doit attendre un message individuel de confirmation que tous les identificateurs d'utilisateur et tous les identificateurs de jeton ont été intégrés dans un domaine fusionné ou ont été purgés avant que ce fournisseur commence à soumettre des attributs de canal statique, attribué ou privé en vue d'une fusion.

## 10.8 MCcf

L'unité de confirmation **MCcf** répond à une précédente unité **MCrq**. Elle correspond au même ensemble d'identificateurs de canal et à un sous-ensemble de leurs attributs. Les attributs de canal non intégrés dans le domaine fusionné sont signalés comme étant des identificateurs de canal à purger.

TABLEAU 10-8/T.125

**Unité MCSPDU de type MCcf**

| Contenu | Source | Collecteur |
|---|---|---|
| Canaux de fusion | Fournisseur supérieur | Intermédiaires |
| Identificateurs de canal de purge | Fournisseur supérieur | Supérieur précédent |

Les identificateurs de canal acceptés sont repris avec les attributs introduits dans la base de données du fournisseur MCS supérieur. Les fournisseurs intermédiaires doivent mettre à jour leurs informations en conséquence.

Les canaux à expulser du domaine inférieur ne sont énumérés que par leurs identificateurs. Si les mêmes identificateurs de canal sont utilisés dans le domaine supérieur, ils sont laissés tels quels. Les fournisseurs intermédiaires doivent expédier les identificateurs de canal purgés sans les traiter.

Les fournisseurs de service MCS doivent acheminer l'unité **MCcf** vers la source de l'unité **MCrq** précédente, en appliquant le principe qu'il existe une seule réponse à chaque demande. La confirmation **MCcf** retourne vers le précédent supérieur qui avait émis la demande **MCrq**. Ensuite, les canaux fusionnés peuvent être ignorés, car ils sont restés dans la base de données en attendant la réponse. Les identificateurs de canal purgés doivent être supprimés car ils doivent faire l'objet d'une indication **PCin**.

Les fournisseurs MCS de niveau intermédiaire doivent confirmer que les identificateurs d'utilisateurs annoncés dans des attributs de canal privé sont attribués au sous-arbre vers lequel l'unité **MCcf** est acheminée. Si un gestionnaire de canal privé a été détaché et réaffecté ailleurs dans l'intervalle qui s'est écoulé depuis la validation de l'unité **MCrq** antécédente, un fournisseur de niveau intermédiaire doit émettre une demande **CDrq** faisant du canal privé un corollaire de la fusion de domaines, puis doit transférer l'identificateur de ce canal dans l'ensemble à purger. Si des utilisateurs autorisés ont été réaffectés ailleurs, ce fournisseur doit les exclure du canal.

## 10.9 PCin

L'unité d'indication **PCin** est produite chez un précédent fournisseur supérieur dès réception d'une confirmation **MCcf**. Elle fait l'objet d'une communication multipoint en aval et supprime l'utilisation d'identificateurs de canal spécifiés par des fournisseurs subordonnés.

TABLEAU 10-9/T.125

**Unité MCSPDU de type PCin**

| Contenu | Source | Collecteur |
|---|---|---|
| Identificateurs de détachement d'utilisateur | Supérieur précédent | Subordonnés |
| Identificateurs de purge de canal | Supérieur précédent | Subordonnés |

Selon l'utilisation en cours d'un identificateur de canal, l'effet de sa purge est l'envoi: d'une indication MCS-DETACH-USER à tous les utilisateurs d'un canal d'identification d'utilisateur; d'une indication MCS-CHANNEL-LEAVE aux utilisateurs qui ont adhéré à un identificateur de canal statique ou attribué; d'une indication MCS-CHANNEL-DISBAND au gestionnaire et d'une indication MCS-CHANNEL-EXPEL aux utilisateurs admis d'un identificateur de canal privé.

Un précédent fournisseur supérieur, connaissant l'utilisation de tous les canaux dans son domaine inférieur, peut produire les indications appropriées à partir des seuls identificateurs de canal. Ses subordonnés peuvent cependant n'en avoir qu'une connaissance partielle. Ils doivent toujours être informés des identificateurs de canaux qui représentent des utilisateurs détachés, pour lesquels une indication est toujours émise, et qui représentent d'autres types de canaux, pour lesquels une indication n'est émise que si le canal en cause est en cours d'utilisation chez le fournisseur subordonné. L'unité **PCin** subdivisera donc les identificateurs de canal à purger selon ces deux catégories.

La purge d'un identificateur d'utilisateur par l'intermédiaire d'une unité **PCin** doit avoir les mêmes conséquences que leur suppression par l'intermédiaire d'une unité **DUin**, sauf que celle-ci n'a pas besoin de produire, à titre d'effet secondaire, une unité **CDin** ou **TVcf**.

NOTE – Un fournisseur peut recevoir, insérés dans une indication PCin, des identificateurs de canal statique et de canal attribué auxquels il n'a pas adhéré. Il peut également recevoir des identificateurs de canal privé pour lesquels ses rattachements ne sont ni des gestionnaires ni des utilisateurs admis. Les enregistrements d'utilisation contenus dans la base de données permettront à un fournisseur de supprimer les primitives d'indication correspondant à de tels identificateurs de canal.

### 10.10 MTrq

L'unité de demande **MTrq** est produite dès que la primitive MCS-CONNECT-PROVIDER est suivie d'effet. Elle transmet en amont de la hiérarchie les attributs des jetons détenus par un précédent fournisseur supérieur, de manière que ces attributs puissent être intégrés dans le domaine fusionné.

TABLEAU 10-10/T.125

**Unité MCSPDU de type MTrq**

| Contenu | Source | Collecteur |
|---|---|---|
| Jetons de fusion | Supérieur précédent | Fournisseur supérieur |
| Identificateurs de jeton de purge | Intermédiaires | Fournisseur supérieur |

On peut insérer dans l'unité **MTrq** les attributs de plusieurs jetons, dans les limites fixées par le domaine quant à la longueur des unités MCSPDU. Comme indiqué dans les définitions ASN.1 de l'article 7, chaque état de jeton utilisé (saisi, inhibé, donnant, incessible, donné) possède son propre ensemble d'attributs. Ceux-ci sont conservés dans la base de données du fournisseur MCS supérieur et sont partiellement copiés dans les sous-arbres où un jeton est utilisé. Lorsque deux domaines sont fusionnés par l'action d'une primitive MCS-CONNECT-PROVIDER, les jetons qui sont en service dans le domaine inférieur doivent soit être intégrés dans la base de données du domaine supérieur ou être purgés du domaine inférieur. Cette décision relève du fournisseur supérieur du domaine fusionné.

Chaque jeton doit être pris en compte individuellement. Si les limites de domaine concernant les jetons utilisés le permettent et si l'identificateur de jeton n'a pas déjà été mis en œuvre de manière conflictuelle, le domaine supérieur doit s'agrandir pour intégrer ce jeton. L'inhibition d'un identificateur de jeton dans le domaine inférieur ne provoque pas de conflit s'il est également inhibé dans le domaine supérieur. Toutes les autres combinaisons pour usage simultané sont interdites, ce qui nécessite que l'identificateur de jeton soit expulsé du domaine inférieur.

Si un jeton possède un grand ensemble d'utilisateurs qui l'inhibent, il se peut que ses attributs ne puissent pas s'insérer dans une seule unité **MTrq** et qu'il soit nécessaire de les remonter dans plusieurs unités MCSPDU. Cependant, la deuxième demande suivie d'effet concernant le même jeton inhibé doit être tenue en attente de la réception d'une confirmation **MTcf** en réponse à la première demande. Ce n'est qu'à ce moment que l'on saura si les limites du domaine ont permis de mettre en œuvre ce jeton dans le domaine supérieur. Si la première demande n'est pas suivie d'effet, il ne faut pas la répéter avec un sous-ensemble restant d'inhibiteurs.

Chaque unité **MTrq** suscite une réponse de confirmation de type **MTcf** de la part du fournisseur MCS supérieur, dans la même séquence. Une unité **MTcf** ne contient rien qui permette d'identifier expressément l'unité **MTrq** précédente. Les réponses ne doivent être acheminées que dans l'ordre où ces unités ont été reçues. Les fournisseurs MCS situés au-dessus du précédent supérieur doivent effectuer un enregistrement de chaque unité **MTrq** non suivie de réponse et du point dont elle provient, de manière que la confirmation **MTcf** correspondante puisse être renvoyée via la même connexion MCS.

Les fournisseurs MCS intermédiaires doivent valider les identificateurs d'utilisateur annoncés dans les attributs de jeton, pour faire en sorte qu'ils soient légitimement attribués au sous-arbre dont l'unité **MTrq** provient. Les identificateurs d'utilisateur invalides doivent être supprimés. Un jeton en cours de cession doit rester saisi si l'utilisateur saisisseur ou l'utilisateur destinataire, mais non les deux, est supprimé. Si un jeton est libéré à la suite de cette suppression d'identificateurs d'utilisateur, tous les attributs de ce jeton doivent être retranchés de la demande de fusion et seul l'identificateur du jeton doit être inséré dans l'ensemble à purger. Un jeton inhibé doit resté inhibé dans l'unité **MTrq** même si tous les inhibiteurs sont supprimés, ce qui laissera un ensemble vide dans les attributs, parce que certains inhibiteurs peuvent continuer à exister après la purge d'autres unités MCSPDU. A l'exception de cette validation des identificateurs d'utilisateur, les fournisseurs MCS intermédiaires ne doivent pas modifier le contenu d'une unité **MTrq**.

Un précédent fournisseur supérieur doit attendre un message individuel de confirmation que tous les identificateurs d'utilisateur ont été intégrés dans un domaine fusionné ou ont été purgés avant que ce fournisseur commence à soumettre des attributs de jeton en vue d'une fusion.

## 10.11 MTcf

Une unité de confirmation **MTcf** répond à une précédente demande **MTrq**. Elle correspond au même ensemble d'identificateurs de jeton et à un sous-ensemble de ses attributs. Les attributs de jeton non intégrés dans le domaine fusionné sont signalés comme étant des identificateurs de jeton à purger.

TABLEAU 10-11/T.125

**Unité MCSPDU de type MTcf**

| Contenu | Source | Collecteur |
|---|---|---|
| Jetons de fusion | Fournisseur supérieur | Intermédiaires |
| Identificateurs de jeton de purge | Fournisseur supérieur | Précédent supérieur |

Les identificateurs de jeton acceptés correspondent aux attributs qui ont été insérés dans la base de données au niveau du fournisseur MCS supérieur. Les fournisseurs intermédiaires doivent mettre à jour leur base de données en conséquence.

Les jetons à purger du domaine inférieur ne sont énumérés que par leurs identificateurs. Si les mêmes identificateurs de jeton sont utilisés dans le domaine supérieur, ils sont laissés inchangés. Les fournisseurs intermédiaires doivent expédier les identificateurs de jeton purgés sans les modifier.

Les fournisseurs de service MCS doivent acheminer les confirmations **MTcf** vers la source de la demande **MTrq** précédente, en appliquant le principe qu'il existe une seule réponse à chaque demande. La confirmation **MTcf** retourne vers le précédent supérieur qui avait émis la demande **MTrq**. Ensuite, les jetons fusionnés peuvent être ignorés, car ils sont restés dans la base de données en attendant la réponse. Les identificateurs de jeton purgés doivent être supprimés car ils doivent faire l'objet d'une indication **PTin**.

Les fournisseurs MCS intermédiaires doivent confirmer que les identificateurs d'utilisateurs annoncés dans des attributs de jeton sont attribués au sous-arbre vers lequel la confirmation **MTcf** est acheminée. Si certains identificateurs d'utilisateurs ont été détachés et réattribués à un autre moment depuis la validation de la demande **MTrq** précédente, un fournisseur intermédiaire doit émettre, pour ces identificateurs, une demande **DUrq** avec le code de cause *canal purgé*, faisant de ces identificateurs des corollaires de la fusion de domaines. Si un identificateur de jeton non inhibé se trouve libéré à cause de cette suppression d'identificateurs d'utilisateurs, il doit être transféré dans l'ensemble purgé.

### 10.12 PTin

L'unité d'indication **PTin** est produite chez un précédent fournisseur supérieur dès réception d'une confirmation **MTcf**. Elle est fait l'objet d'une communication multipoint en aval et supprime l'utilisation d'identificateurs de jeton spécifiés par des fournisseurs subordonnés.

TABLEAU 10-12/T.125

**Unité MCSPDU de type PTin**

| Contenu | Source | Collecteur |
|---|---|---|
| Identificateurs de jeton de purge | Supérieur précédent | Subordonnés |

L'effet de la purge d'un jeton est grave: elle provoque l'envoi d'une primitive d'indication MCS-DETACH-USER à tout utilisateur ayant saisi, inhibé ou recevant un des identificateurs de jeton. Un fournisseur doit mettre en œuvre cette procédure en émettant une demande **DUrq** pour le compte des utilisateurs affectés, avec la cause *jeton purgé*.

NOTE – On prévoit que la Recommandation T.122 de l'UIT-T puisse être révisée pour permettre une indication MCS-TOKEN-RELEASE (libération de jeton MCS) dans cette situation. Cela permettrait à l'utilisateur affecté de rester rattaché bien que son droit d'utilisation du jeton ait été retiré.

### 10.13 DPum

L'unité d'ultimatum **DPum** est produite par une primitive de demande MCS-DISCONNECT-PROVIDER. A son tour, cette unité produit une primitive d'indication MCS-DISCONNECT-PROVIDER à l'autre extrémité d'une connexion MCS. L'unité **DPum** oblige le récepteur à déconnecter la connexion MCS qui l'a acheminée.

L'unité **DPum** peut également être produite par un fournisseur MCS qui détecte un état d'erreur comme l'existence d'une boucle dans la hiérarchie d'un domaine. En ces occurrences, la cause est autre que *sur demande de l'utilisateur*.

TABLEAU 10-13/T.125

**Unité MCSPDU de type DPum**

| Contenu | Source | Collecteur |
|---|---|---|
| Cause | Fournisseur demandeur | Indication |

### 10.14 RJum

L'unité d'ultimatum **RJum** est produite lorsqu'un fournisseur MCS reçoit une unité MCSPDU invalide ou détecte une erreur de protocole de communication MCS. Cette unité invite le fournisseur homologue, à l'autre extrémité d'une connexion MCS, à déconnecter parce qu'il n'est pas certain que l'on puisse opérer un rétablissement à partir d'une situation qui ne devrait pas se produire.

L'unité **RJum** diagnostique l'erreur et renvoie une portion initiale de l'unité TSDU fautive, normalement autant d'octets que pourra en recevoir une unité MCSPDU de longueur maximale. Le fournisseur récepteur a la possibilité de déconnecter ou de poursuivre.

TABLEAU 10-14/T.125

**Unité MCSPDU de type RJum**

| Contenu | Source | Collecteur |
|---|---|---|
| Diagnostic | Fournisseur rejetant | Fournisseur rejeté |
| Octets initiaux | Fournisseur rejetant | Fournisseur rejeté |

## 10.15 AUrq

L'unité de demande **AUrq** est produite par une primitive de demande MCS-ATTACH-USER. Elle remonte vers le fournisseur MCS supérieur, qui retourne une réponse de type **AUcf**. Si la limite de domaine concernant le nombre d'identificateurs d'utilisateur le permet, un nouvel identificateur d'utilisateur est produit.

TABLEAU 10-15/T.125

**Unité MCSPDU de type AUrq**

| Contenu | Source | Collecteur |
|---|---|---|
| (Néant) | – | – |

L'unité **AUrq** ne contient pas d'autres informations que son type d'unité MCSDPU. Le domaine auquel l'utilisateur se rattache est déterminé par la connexion MCS qui achemine cette MCSPDU. La seule caractéristique initiale de l'identificateur d'utilisateur produit est le fait qu'elle est unique.

Un fournisseur MCS doit tenir un registre de toutes les unités **AUrq** reçues sans réponse et des connexions MCS par lesquelles ces unités sont arrivées, de manière qu'une confirmation **AUcf** puisse être réexpédiée en réponse à la même source. Pour répartir équitablement les réponses, il convient que chaque fournisseur constitue à cette fin une file d'attente de type premier entré-premier sorti.

## 10.16 AUcf

L'unité de confirmation **AUcf** est produite au niveau du fournisseur MCS supérieur dès réception d'une demande **AUrq**. Renvoyée au fournisseur demandeur, cette confirmation engendre une primitive de confirmation MCS-ATTACH-USER.

TABLEAU 10-16/T.125

**Unité MCSPDU de type AUcf**

| Contenu | Source | Collecteur |
|---|---|---|
| Résultat | Fournisseur supérieur | Confirmation |
| Initiateur (sur option) | Fournisseur supérieur | Confirmation |

L'unité **AUcf** contient un identificateur d'utilisateur si et seulement si le résultat est favorable. Les fournisseurs qui reçoivent une confirmation **AUcf** favorable doivent introduire l'identificateur d'utilisateur dans leur base de données.

Les fournisseurs MCS doivent acheminer l'unité **AUcf** vers la source d'une demande **AUrq** antérieure, en appliquant le principe qu'il existe une seule réponse par demande. Un fournisseur qui envoie une unité **AUcf** doit enregistrer la connexion MCS descendante à laquelle le nouvel identificateur d'utilisateur est ainsi attribué, de manière qu'il puisse valider cet identificateur lorsqu'il réapparaîtra dans d'autres demandes.

### 10.17 DUrq

L'unité de demande de détachement **DUrq** est produite par une primitive de demande MCS-DETACH-USER. Si elle est valide, elle remonte jusqu'au fournisseur MCS supérieur, qui supprime l'utilisateur de sa base de données et diffuse une indication **DUin** pour aviser les autres fournisseurs du changement.

TABLEAU 10-17/T.125

**Unité MCSPDU de type DUrq**

| Contenu | Source | Collecteur |
|---|---|---|
| Cause | Fournisseur demandeur | Fournisseur supérieur |
| Identificateurs d'utilisateur | Fournisseur demandeur | Fournisseur supérieur |

Une primitive de demande MCS-DETACH-USER produit une unité **DUrq** contenant la cause *demandé par l'utilisateur* et un seul identificateur d'utilisateur.

L'unité **DUrq** peut aussi être produite par un fournisseur MCS lorsqu'une connexion MCS descendante est déconnectée. A ce point, tous les utilisateurs contenus dans le sous-arbre en question sont perdus et doivent être signalés comme étant détachés avec la cause *domaine déconnecté*. Si des attributions d'identificateur d'utilisateur sont en instance, on peut laisser dans le sous-arbre déconnecté une réponse ultérieure, de type **MCcf** ou **AUcf**, sans adresse de réacheminement à sa source. Un fournisseur confronté à cette situation doit également produire une demande **DUrq** pour supprimer les identificateurs d'utilisateur dont l'attribution n'est pas possible.

Les fournisseurs qui reçoivent des demandes **DUrq** doivent valider les identificateurs d'utilisateur contenus dans ces unités pour faire en sorte que ces identificateurs soient légitimement attribués au sous-arbre d'origine. Les identificateurs d'utilisateur invalides doivent être supprimés. S'il ne reste plus d'identificateurs d'utilisateur, une demande **DUrq** doit être ignorée.

Les identificateurs d'utilisateur contenus dans l'unité **DUrq** ne doivent pas être supprimés de la base de données avant qu'un fournisseur reçoive une indication **DUin**. Cela assure la cohérence avec le fournisseur MCS supérieur.

NOTE – Si plusieurs priorités de données sont mises en œuvre dans un domaine MCS, l'indication DUin peut arriver chez un fournisseur donné avant les données déjà envoyées par le même utilisateur, mais avec une priorité plus basse. Ce protocole n'empêche pas les données d'être acheminées jusqu'à un point de rattachement, même si cela a lieu après qu'une indication DUin a signalé que l'expéditeur a été détaché.

### 10.18 DUin

Une unité d'indication **DUin** est produite au niveau du fournisseur MCS supérieur dès réception d'une primitive **DUrq**. Cette unité fait l'objet d'une communication multipoint en aval à tous les autres fournisseurs. Elle produit des primitives d'indication MCS-DETACH-USER à tous les rattachements.

A un point de rattachement survivant, l'unité **DUin** produit une indication MCS-DETACH-USER pour chaque identificateur d'utilisateur qu'elle contient, que l'utilisateur notifié ait ou non déjà appris l'existence d'un utilisateur détaché.

Dès réception d'une indication **DUin** contenant son propre identificateur d'utilisateur, un rattachement MCS cesse d'exister. Tous les canaux auxquels il n'est plus adhéré à la suite du détachement d'un utilisateur doivent être quittés via l'unité **CLrq**, sauf le canal d'identification d'utilisateur lui-même.

TABLEAU 10-18/T.125

**Unité MCSPDU de type DUin**

| Contenu | Source | Collecteur |
|---|---|---|
| Cause | Fournisseur supérieur | Indication |
| Identificateurs d'utilisateur | Fournisseur supérieur | Indication |

Les fournisseurs qui reçoivent une indication **DUin** doivent supprimer de leur base de données les identificateurs d'utilisateur spécifiés. Les canaux privés qui étaient gérés par un utilisateur détaché doivent être supprimés s'il n'y a pas d'autres utilisateurs admis. Si d'autres utilisateurs restent attachés, la suppression du gestionnaire aura pour conséquence que le fournisseur supérieur leur diffusera une indication **CDin**. L'état de tous les jetons saisis, en cession ou inhibés par un utilisateur détaché doit être ajusté en conséquence. La suppression d'un destinataire de jeton prévu doit avoir pour conséquence que le fournisseur supérieur produise une confirmation **TVcf** défavorable à destination du donneur de ce jeton, à moins qu'il n'ait libéré le jeton ou qu'il se soit lui-même détaché.

### 10.19 CJrq

L'unité de demande **CJrq** est produite par une primitive de demande MCS-CHANNEL-JOIN. Si elle est valide, elle remonte jusqu'à atteindre un fournisseur MCS possédant assez d'informations pour émettre une réponse de confirmation **CJcf**. Ce fournisseur peut être le fournisseur MCS supérieur.

TABLEAU 10-19/T.125

**Unité MCSPDU de type CJrq**

| Contenu | Source | Collecteur |
|---|---|---|
| Initiateur | Fournisseur demandeur | Fournisseur plus élevé |
| Identificateur de canal | Demande | Fournisseur plus élevé |

Le fournisseur MCS qui reçoit la primitive de demande fournit l'identificateur d'utilisateur du rattachement MCS initiateur. Les fournisseurs qui recevront la demande **CJrq** par la suite devront valider cet identificateur d'utilisateur pour s'assurer qu'il est légitimement attribué au sous-arbre d'origine. Si l'utilisateur est invalide, cette unité MCSPDU doit être ignorée.

NOTE – Cette procédure tient compte de la possibilité qu'une unité **CJrq** montante croise une unité descendante de purge d'identificateur de l'utilisateur initiateur. Un fournisseur qui reçoit d'abord une indication **PCin** peut recevoir peu après une demande **CJrq** contenant un identificateur d'utilisateur invalide. Il s'agit d'une occurrence normale, qui ne justifie pas le rejet de l'unité MCSPDU.

La demande **CJrq** peut remonter jusqu'à un fournisseur MCS qui possède, dans sa base de données, l'identificateur de canal demandé. Un tel fournisseur, pour rester en accord avec le fournisseur MCS supérieur, décidera si cette demande doit être suivie d'effet. S'il convient que la demande échoue, le fournisseur doit émettre une confirmation **CJcf** d'échec. S'il convient que la demande soit suivie d'effet et que le fournisseur ait déjà adhéré au même canal, le fournisseur doit émettre une confirmation **CJcf** avec la valeur «succès». Dans ces deux cas, la primitive MCS-CHANNEL-JOIN (adhésion à un canal MCS) se réalisera sans nécessairement passer par le fournisseur MCS supérieur. Dans le cas contraire, s'il y a lieu que la demande soit suivie d'effet mais que le canal n'ait pas encore fait l'objet de l'adhésion correspondante, un fournisseur doit envoyer la demande **CJrq** plus haut.

Si la demande **CJrq** remonte jusqu'au fournisseur MCS supérieur, l'identificateur de canal demandé peut être zéro: cette information ne se trouve dans aucune base de données car il s'agit d'un identificateur invalide. Si la limite de domaine imposée au nombre de canaux en usage le permet, un nouvel identificateur de canal attribué doit être émis et renvoyé dans une confirmation **CJcf** favorable. Si l'identificateur de canal demandé se trouve dans l'ensemble des canaux statiques et que la limite de domaine sur le nombre de canaux utilisés le permette, cet identificateur de canal doit être introduit dans la base de données et être également renvoyé dans une confirmation **CJcf** favorable.

Si la demande ne remonte pas jusqu'au fournisseur MCS supérieur, elle ne sera suivie d'effet que si l'identificateur de canal se trouve déjà dans la base de données du fournisseur MCS supérieur. Un canal d'identificateur d'utilisateur ne peut recevoir d'adhésion que de cet utilisateur. Seuls des utilisateurs préalablement admis par le gestionnaire d'un identificateur de canal privé peuvent adhérer à celui-ci. Tout utilisateur peut adhérer à un identificateur de canal attribué.

### 10.20 CJcf

L'unité de confirmation **CJcf** est produite chez un fournisseur MCS de niveau plus élevé dès sa réception. Renvoyée au fournisseur demandeur, elle produit une primitive de confirmation d'adhésion à un canal: MCS-CHANNEL-JOIN.

TABLEAU 10-20/T.125

**Unité MCSPDU de type CJcf**

| Contenu | Source | Collecteur |
|---|---|---|
| Résultat | Fournisseur plus élevé | Confirmation |
| Initiateur | Fournisseur plus élevé | Acheminement d'unité MCSPDU |
| Demandé | Fournisseur plus élevé | Confirmation |
| Identificateur de canal | Fournisseur plus élevé | Confirmation |

L'unité **CJcf** ne contient un identificateur de canal d'adhésion que si et seulement si le résultat est favorable.

La demande d'identificateur de canal est la même que dans l'unité **CJrq**. Cela aide le rattachement initiateur à établir une relation entre une requête antécédente et une confirmation MCS-CHANNEL-JOIN. Etant donné que l'unité **CJrq** n'a pas besoin de remonter jusqu'au fournisseur supérieur, les confirmations peuvent se produire dans le désordre.

Si le résultat est favorable, la confirmation **CJcf** opère l'adhésion du fournisseur MCS récepteur au canal spécifié. Ensuite, les fournisseurs plus élevés doivent acheminer vers ce récepteur toutes données envoyées par des utilisateurs sur ce canal. Un fournisseur doit maintenir son adhésion à un canal tant qu'un de ses rattachements ou de ses fournisseurs subordonnés la conserve. Pour sortir du canal, un fournisseur doit émettre une demande **CLrq**.

Les fournisseurs qui reçoivent une confirmation **CJcf** favorable doivent introduire dans leur base de données l'identificateur du canal. Si celui-là ne s'y trouve pas déjà, il doit être affecté au type statique ou attribué, selon sa catégorie.

L'unité **CJcf** doit être envoyée en direction de l'identificateur d'utilisateur initiateur. Si cet identificateur ne peut pas être atteint parce qu'il n'existe plus de connexion MCS, le fournisseur doit décider s'il se justifie que cet identificateur conserve son adhésion au canal. Si ce n'est pas le cas, il doit émettre une demande de sortie de canal, **CLrq**.

### 10.21 CLrq

L'unité de demande **CLrq** est produite par un fournisseur MCS qui souhaite se retirer d'un faisceau de canaux. La cause peut être une demande de sortie de canal MCS-CHANNEL-LEAVE issue du dernier rattachement ayant adhéré à un canal. La demande **CLrq** continue à remonter dans la hiérarchie si des fournisseurs plus élevés perdent aussi, par voie de conséquence, leur raison de faire l'objet d'une adhésion.

Les fournisseurs qui reçoivent l'unité **CLrq** doivent arrêter d'acheminer, vers la connexion MCS qui l'a transportée, toutes données envoyées par des utilisateurs sur les canaux spécifiés. Lorsque le dernier rattachement ou le dernier fournisseur subordonné sort d'un canal, un fournisseur MCS doit émettre une demande **CLrq** correspondante.

TABLEAU 10-21/T.125

**Unité MCSPDU de type CLrq**

| Contenu | Source | Collecteur |
|---|---|---|
| Identificateurs de canal | Fournisseur demandeur | Fournisseur plus élevé |

## 10.22 CCrq

L'unité de demande **CCrq** est produite par une primitive de demande de constitution de canal MCS-CHANNEL-CONVENE. Si elle est valide, elle remonte jusqu'au fournisseur MCS supérieur, qui retourne une réponse **CCcf**. Si la limite de domaine imposée au nombre d'identificateurs de canal le permet, un nouvel identificateur de canal privé est produit.

L'unité **CCrq** contient l'identificateur d'utilisateur initiateur, qui doit être validé comme expliqué pour la demande **CJrq**.

Le demandeur devient gestionnaire du canal privé. Initialement, aucun utilisateur n'a adhéré à ce canal et son gestionnaire y est le seul utilisateur admis.

TABLEAU 10-22/T.125

**Unité MCSPDU de type CCrq**

| Contenu | Source | Collecteur |
|---|---|---|
| Initiateur | Fournisseur demandeur | Fournisseur supérieur |

## 10.23 CCcf

L'unité de confirmation **CCcf** est produite au niveau du fournisseur MCS supérieur dès réception de la demande **CCrq**. Renvoyée vers le fournisseur demandeur, cette unité produit une primitive de confirmation de constitution de canal MCS-CHANNEL-CONVENE.

TABLEAU 10-23/T.125

**Unité MCSPDU de type CCcf**

| Contenu | Source | Collecteur |
|---|---|---|
| Résultat | Fournisseur supérieur | Confirmation |
| Initiateur | Fournisseur supérieur | Acheminement d'unité MCSPDU |
| Identificateur de canal (sur option) | Fournisseur supérieur | Confirmation |

L'unité **CCcf** contient un identificateur de canal privé si et seulement si le résultat confirmé est favorable.

Les fournisseurs qui reçoivent une confirmation **CCcf** favorable doivent introduire l'identificateur de canal dans leur base de données en tant que canal privé ayant comme gestionnaire l'identificateur de l'utilisateur initiateur.

L'unité **CCcf** doit être envoyée en direction de l'identificateur d'utilisateur initiateur. Si cet identificateur ne peut pas être atteint parce qu'il n'existe plus de connexion MCS, il n'est besoin d'aucune action particulière car l'unité **DUin** doit toujours arriver ultérieurement pour signaler que l'initiateur s'est détaché. Comme celui-ci est le gestionnaire du canal, l'identificateur de ce dernier sera supprimé de la base de données.

### 10.24 CDrq

L'unité de demande **CDrq** est produite par une primitive de demande de dissolution de canal, MCS-CHANNEL-DISBAND. Si elle est valide, elle remonte jusqu'au fournisseur MCS supérieur, qui supprime l'identificateur du canal privé et produit une indication **CDin**.

TABLEAU 10-24/T.125

**Unité MCSPDU de type CDrq**

| Contenu | Source | Collecteur |
|---|---|---|
| Initiateur | Fournisseur demandeur | Fournisseur supérieur |
| Identificateur de canal | Demande | Fournisseur supérieur |

L'unité **CDrq** peut également être produite à la propre initiative d'un fournisseur MCS pour dissoudre un canal.

L'unité **CDrq** contient l'identificateur d'utilisateur initiateur, qui doit être validé pour veiller à ce qu'il soit légitimement attribué au sous-arbre d'origine. Si l'initiateur ne correspond pas au gestionnaire du canal privé, tel qu'enregistré dans la base de données, l'unité MCSPDU doit être ignorée.

### 10.25 CDin

L'unité d'indication **CDin** est produite au niveau du fournisseur MCS supérieur dès réception de la demande **CDrq**. Elle fait l'objet d'une communication multipoint en aval aux fournisseurs qui contiennent dans leur sous-arbre le gestionnaire ou un utilisateur admis. Elle produit des primitives d'indication d'expulsion de canal MCS-CHANNEL-EXPEL qui sont envoyées aux utilisateurs admis avec la cause *canal dissous*. A l'initiative du fournisseur, elle envoie également au gestionnaire une indication MCS-CHANNEL-DISBAND.

TABLEAU 10-25/T.125

**Unité MCSPDU de type CDin**

| Contenu | Source | Collecteur |
|---|---|---|
| Identificateur de canal | Fournisseur supérieur | Indication |

L'unité **CDin** doit aussi être produite par le fournisseur MCS supérieur lorsque le gestionnaire d'un canal privé est détaché.

Les fournisseurs qui reçoivent l'indication **CDin** doivent supprimer le canal de leur base de données.

### 10.26 CArq

L'unité de demande **CArq** est produite par une primitive de demande d'admission de canal MCS-CHANNEL-ADMIT. Si elle est valide, elle remonte jusqu'au fournisseur MCS supérieur, qui admettra les utilisateurs spécifiés dans le canal privé et multidestinera l'indication **CAin** pour aviser les fournisseurs dans un sous-arbre desquels ces utilisateurs résident.

L'unité **CArq** contient l'identificateur d'utilisateur initiateur, qui doit être validé comme expliqué pour l'unité **CDrq**.

Les autres identificateurs d'utilisateur contenus dans l'unité **CArq**, représentant des utilisateurs à admettre, doivent être validés au niveau du fournisseur MCS supérieur qui est le seul à avoir connaissance de l'entière population des utilisateurs. Les identificateurs invalides doivent être omis de l'indication **CAin** résultante.

TABLEAU 10-26/T.125

**Unité MCSPDU de type CArq**

| Contenu | Source | Collecteur |
|---|---|---|
| Initiateur | Fournisseur demandeur | Fournisseur supérieur |
| Identificateur de canal | Demande | Fournisseur supérieur |
| Identificateurs d'utilisateur | Demande | Fournisseur supérieur |

Les identificateurs d'utilisateur contenus dans l'unité **CArq** ne doivent pas être admis dans le canal privé avant qu'un fournisseur ait reçu l'indication **CAin**: cela assure la cohérence avec le fournisseur MCS supérieur.

### 10.27 CAin

L'unité d'indication **CAin** est produite au niveau du fournisseur MCS supérieur dès réception de la demande **CArq**. Elle fait l'objet d'une communication multipoint en aval vers les fournisseurs qui contiennent dans leur sous-arbre un utilisateur nouvellement admis. Elle produit des primitives d'indication d'admission de canal MCS-CHANNEL-ADMIT aux rattachements affectés.

Les fournisseurs qui reçoivent l'indication **CAin** doivent normalement mettre à jour leur base de données pour admettre les utilisateurs spécifiés qui résident dans leur sous-arbre. Si toutefois un fournisseur est le précédent supérieur d'un domaine inférieur qui est en cours de fusion à la suite d'une primitive MCS-CONNECT-PROVIDER, ce fournisseur peut refuser l'admission en émettant la demande **DUrq** pour les identificateurs d'utilisateurs affectés, avec la raison *canal purgé*.

TABLEAU 10-27/T.125

**Unité MCSPDU de type CAin**

| Contenu | Source | Collecteur |
|---|---|---|
| Initiateur | Fournisseur supérieur | Indication |
| Identificateur de canal | Fournisseur supérieur | Indication |
| Identificateurs d'utilisateur | Fournisseur supérieur | Acheminement d'unité MCSPDU |

### 10.28 CErq

L'unité de demande **CErq** est produite par une primitive de demande d'expulsion de canal MCS-CHANNEL-EXPEL. Si elle est valide, elle remonte jusqu'au fournisseur MCS supérieur, qui expulsera les utilisateurs spécifiés du canal privé et multidestinera une indication **CEin** pour aviser les fournisseurs dans le sous-arbre desquels ces utilisateurs résident.

L'unité **CErq** contient l'identificateur d'utilisateur initiateur, qui doit être validé comme expliqué pour l'unité **CDrq**.

Les autres identificateurs d'utilisateur contenus dans l'unité **CErq**, représentant des utilisateurs à expulser, doivent être validés au niveau du fournisseur MCS supérieur, qui est le seul à connaître l'ensemble complet des utilisateurs admis. Ceux qui n'ont pas été admis ne sont pas insérés dans l'indication **CEin** résultante.

Les identificateurs d'utilisateur contenus dans l'unité **CErq** ne doivent pas être expulsés du canal privé avant qu'un fournisseur ait reçu une indication **CEin**: cela assure la cohérence avec le fournisseur MCS supérieur.

TABLEAU 10-28/T.125

**Unité MCSPDU de type CErq**

| Contenu | Source | Collecteur |
|---|---|---|
| Initiateur | Fournisseur demandeur | Fournisseur supérieur |
| Identificateurs de canal | Demande | Fournisseur supérieur |
| Identificateurs d'utilisateur | Demande | Fournisseur supérieur |

### 10.29 CEin

L'unité d'indication **CEin** est produite au niveau du fournisseur MCS supérieur dès réception d'une unité **CErq**. Elle fait l'objet d'une communication multipoint en aval vers les fournisseurs dont le sous-arbre contient un utilisateur expulsé. Elle produit des primitives d'indication d'expulsion de canal MCS-CHANNEL-EXPEL aux rattachements affectés, avec la cause *demandé par l'utilisateur*.

TABLEAU 10-29/T.125

**Unité MCSPDU de type CEin**

| Contenu | Source | Collecteur |
|---|---|---|
| Identificateur de canal | Fournisseur supérieur | Indication |
| Identificateurs d'utilisateur | Fournisseur supérieur | Acheminement d'unité MCSPDU |

Les fournisseurs qui reçoivent l'indication **CEin** doivent mettre à jour le canal enregistré dans leur base de données en supprimant les utilisateurs spécifiés de l'ensemble des utilisateurs admis dans ce canal. Si l'ensemble des utilisateurs admis dans un canal privé devient vide et que le gestionnaire ne réside pas dans le sous-arbre, l'identificateur de canal doit être supprimé de la base de données. Sinon, si le canal ne fait plus l'objet d'adhésions à la suite d'expulsions, un fournisseur doit émettre une demande **CLrq** correspondante.

Un fournisseur qui expédie une unité **CEin** doit calculer, pour chaque sous-arbre de destination, si celui-ci contiendra, après l'opération, des rattachements admis dans le canal privé. S'il n'y en a pas, le fournisseur en conclut que le fournisseur subordonné correspondant n'adhère plus au canal privé. Il doit mettre immédiatement à jour sa base de données en conséquence, sans attendre de demande **CLrq**.

### 10.30 SDrq

L'unité de demande **SDrq** est produite par une primitive de demande d'envoi de données MCS-SEND-DATA. Si elle est valide, elle remonte jusqu'au fournisseur MCS supérieur. En cours de route, des fournisseurs peuvent en tirer une indication **SDin** de contenu identique, qu'ils multidestineront en aval.

L'unité **SDrq** contient l'identificateur d'utilisateur initiateur, qui doit être validé comme expliqué pour l'unité **CJrq**.

Si l'identificateur de canal est répertorié dans la base de données du fournisseur MCS récepteur en tant que canal privé et que l'initiateur de l'unité **SDrq** ne soit pas un utilisateur admis, l'unité MCSPDU doit être ignorée.

La connexion de transport initiale ou additionnelle qui achemine la demande **SDrq** doit correspondre à sa priorité de transfert de données, compte tenu du nombre de priorités mis en œuvre dans le domaine. Les unités MCSPDU arrivant sur une connexion MCS par une connexion de transport erronée doivent être rejetées.

TABLEAU 10-30/T.125

**Unité MCSPDU de type SDrq**

| Contenu | Source | Collecteur |
|---|---|---|
| Initiateur | Fournisseur demandeur | Fournisseur plus élevé |
| Identificateur de canal | Demande | Fournisseur plus élevé |
| Priorité des données | Demande | Fournisseur plus élevé |
| Segmentation | Fournisseur demandeur | Fournisseur plus élevé |
| Données d'utilisateur | Demande | Fournisseur plus élevé |

Les fanions de segmentation *début* et *fin* doivent être insérés par un fournisseur pour montrer la relation entre les données contenues dans l'unité **SDrq** et les frontières d'une unité de données du service MCS. Les fournisseurs ont la possibilité de segmenter et de réassembler des unités MCSPDU qui font partie de la même unité de données du service MCS, du moment que ces opérations n'ont pas d'incidence sur l'intégrité des données d'utilisateur. Une telle manipulation ne devrait cependant apporter que peu d'avantages car la longueur maximale d'une unité MCSPDU est constante dans l'ensemble d'un domaine.

Un fournisseur doit tirer d'une demande **SDrq** une indication **SDin** ayant le même contenu, puis la transmettre à tous les fournisseurs qui ont adhéré au canal spécifié, à l'exception du fournisseur subordonné qui a transmis la demande **SDrq** en amont. Il doit aussi transmettre la demande **SDrq** en amont, à moins que le canal ne soit répertorié dans sa base de données comme étant un identificateur d'utilisateur résidant dans son sous-arbre.

### 10.31 SDin

L'unité d'indication **SDin** est produite au niveau d'un fournisseur MCS dès réception d'une unité **SDrq**. Elle fait l'objet d'une communication multipoint en aval et produit des primitives d'indication d'envoi de données MCS-SEND-DATA à tous les rattachements qui ont adhéré au canal.

TABLEAU 10-31/T.125

**Unité MCSPDU de type SDin**

| Contenu | Source | Collecteur |
|---|---|---|
| Initiateur | Fournisseur plus élevé | Indication |
| Identificateur d'utilisateur | Fournisseur plus élevé | Indication |
| Priorité des données | Fournisseur plus élevé | Indication |
| Segmentation | Fournisseur plus élevé | Fournisseur indicateur |
| Données d'utilisateur | Fournisseur plus élevé | Indication |

La connexion de transport initiale ou additionnelle qui achemine l'indication **SDin** doit correspondre à sa priorité de données, compte tenu du nombre de priorités mises en œuvre dans un domaine. Les unités MCSPDU arrivant sur une connexion MCS par une connexion de transport erronée doivent être rejetées.

Les fanions de segmentation *début* et *fin* permettent de réassembler les données d'utilisateur pour constituer une unité de données du service MCS complète. Ces fanions doivent être interprétés dans le contexte d'unités MCSPDU d'indication **SDin** provenant du même utilisateur sur le même canal et avec la même priorité. Un train de fragments à réassembler peut être entrelacé avec d'autres unités MCSPDU et avec des données issues d'autres utilisateurs sur des canaux ayant d'autres priorités.

La manière dont les unités de données du service sont indiquées à des utilisateurs MCS rattachés relève d'une décision de mise en œuvre locale. Une possibilité consiste à remettre chaque unité MCSPDU sous forme d'une unité de données d'interface distincte, fanions de segmentation inclus. D'autres solutions, visant à effectuer le réassemblage au niveau du fournisseur récepteur, doivent normalement tenir compte de grandes unités de données du service et de l'ordre relatif dans lequel ces unités commencent à arriver.

Les fournisseurs qui reçoivent l'unité **SDin** doivent l'expédier à tous les subordonnés qui ont adhéré au canal.

### 10.32 USrq

L'unité de demande **USrq** est produite par une primitive de demande d'envoi de données en séquences uniformes MCS-UNIFORM-SEND-DATA. Si elle est valide, elle remonte jusqu'au fournisseur MCS, qui en tirera une unité **USin** de contenu identique et qui la multidestinera en aval.

TABLEAU 10-32/T.125

**Unité MCSPDU de type USrq**

| Contenu | Source | Collecteur |
|---|---|---|
| Initiateur | Fournisseur demandeur | Fournisseur supérieur |
| Identificateur de canal | Demande | Fournisseur supérieur |
| Priorité des données | Demande | Fournisseur supérieur |
| Segmentation | Fournisseur demandeur | Fournisseur supérieur |
| Données d'utilisateur | Demande | Fournisseur supérieur |

L'unité **USrq** contient l'identificateur d'utilisateur initiateur, qui doit être validé comme expliqué pour l'unité **CJrq**.

Si l'identificateur de canal est répertorié dans la base de données du fournisseur MCS récepteur en tant que canal privé et que l'initiateur de la demande **USrq** ne soit pas un utilisateur admis, l'unité MCSPDU doit être ignorée.

La connexion de transport initiale ou additionnelle qui achemine **USrq** doit être adaptée à sa priorité de données, compte tenu du nombre de priorités mises en œuvre dans le domaine. Les unités MCSPDU arrivant sur une connexion MCS par une connexion de transport erronée doivent être rejetées.

Les fanions de segmentation *début* et *fin* doivent être insérés par un fournisseur pour montrer la relation entre les données contenues dans l'unité **USrq** et les frontières d'une unité de données du service MCS. Les fournisseurs ont la possibilité de segmenter et de réassembler des unités MCSPDU qui font partie de la même unité de données du service MCS, du moment que ces opérations n'ont pas d'incidence sur l'intégrité des données d'utilisateur. Une telle manipulation ne devrait cependant apporter que peu d'avantages car la longueur maximale d'une unité MCSPDU est constante dans l'ensemble d'un domaine.

Le fournisseur MCS supérieur doit tirer de l'unité **USrq** une unité **USin** ayant le même contenu.

### 10.33 USin

L'unité d'indication **USin** est produite au niveau du fournisseur MCS supérieur dès réception d'une demande **USrq**. Elle fait l'objet d'une communication multipoint en aval et produit des primitives d'indication d'envoi de données en séquences uniformes MCS-UNIFORM-SEND-DATA à tous les rattachements qui ont adhéré au canal.

TABLEAU 10-33/T.125

**Unité MCSPDU de type USin**

| Contenu | Source | Collecteur |
|---|---|---|
| Initiateur | Fournisseur supérieur | Indication |
| Identificateur de canal | Fournisseur supérieur | Indication |
| Priorité des données | Fournisseur supérieur | Indication |
| Segmentation | Fournisseur supérieur | Fournisseur indicateur |
| Données d'utilisateur | Fournisseur supérieur | Indication |

La connexion de transport initiale ou additionnelle qui achemine l'indication **USin** doit correspondre à sa priorité de données, compte tenu du nombre de priorités mises en œuvre dans un domaine. Les unités MCSPDU arrivant sur une connexion MCS par une connexion de transport erronée doivent être rejetées.

Les fanions de segmentation *début* et *fin* permettent de réassembler les données d'utilisateur pour constituer une unité de données du service MCS complète. Ces fanions doivent être interprétés dans le contexte d'unités MCSPDU d'indication **USin** provenant du même utilisateur sur le même canal et avec la même priorité. Un train de fragments à réassembler peut être entrelacé avec d'autres unités MCSPDU et avec des données issues d'autres utilisateurs sur des canaux ayant d'autres priorités.

La manière dont les unités de données du service sont indiquées aux utilisateurs MCS rattachés relève d'une décision de mise en œuvre locale.

Les fournisseurs qui reçoivent l'unité **USin** doivent l'expédier à tous les subordonnés qui ont adhéré au canal.

## 10.34 TGrq

L'unité de demande **TGrq** est produite par une primitive de demande de saisie de jeton MCS-TOKEN-GRAB. Si elle est valide, elle remonte jusqu'au fournisseur MCS supérieur qui retourne une réponse de confirmation **TGcf**.

TABLEAU 10-34/T.125

**Unité MCSPDU de type TGrq**

| Contenu | Source | Collecteur |
|---|---|---|
| Initiateur | Fournisseur demandeur | Fournisseur supérieur |
| Identificateur de jeton | Demande | Fournisseur supérieur |

L'unité **TGrq** contient l'identificateur d'utilisateur initiateur, qui doit être validé comme expliqué pour l'unité **CJrq**.

Si le jeton est libre et que la limite de domaine quant au nombre de jetons en usage le permet, ce jeton doit passer à l'état saisi. Si le jeton n'est inhibé que par l'utilisateur demandeur, il doit passer à l'état saisi. Sinon, l'état du jeton ne doit pas changer.

## 10.35 TGcf

L'unité de confirmation **TGcf** est produite au niveau du fournisseur MCS supérieur dès réception de l'unité **TGrq**. Réacheminée vers le fournisseur demandeur, elle produit une primitive de confirmation de saisie de jeton MCS-TOKEN-GRAB.

TABLEAU 10-35/T.125

**Unité MCSPDU de type TGcf**

| Contenu | Source | Collecteur |
|---|---|---|
| Résultat | Fournisseur supérieur | Confirmation |
| Initiateur | Fournisseur supérieur | Acheminement d'unité MCSPDU |
| Identificateur de jeton | Fournisseur supérieur | Confirmation |
| Etat du jeton | Fournisseur supérieur | Confirmation |

Le résultat doit être *favorable* si le jeton était préalablement libre ou s'il était le résultat d'une conversion de l'état inhibé à l'état saisi par le même utilisateur. Les autres résultats auront les valeurs *jetons trop nombreux* et *jeton non disponible*. Ce dernier cas s'applique à un jeton déjà saisi par le demandeur, ce qui peut être détecté par examen de l'état du jeton.

Les fournisseurs qui reçoivent l'unité **TGcf** doivent mettre à jour l'état du jeton dans leur base de données en fonction de l'état signalé.

L'unité **TGcf** doit être envoyée en direction de l'identificateur d'utilisateur initiateur. Si cet identificateur ne peut pas être atteint parce qu'il n'existe plus de connexion MCS, il n'est besoin d'aucune action particulière car l'unité **DUin** doit toujours arriver ultérieurement pour signaler que l'initiateur s'est détaché. Cette opération libérera le jeton dans la base de données.

### 10.36 TIrq

L'unité de demande **TIrq** est produite par une primitive de demande d'inhibition de jeton MCS-TOKEN-INHIBIT. Si elle est valide, elle remonte jusqu'au fournisseur MCS supérieur qui retourne une réponse de confirmation **TIcf**.

TABLEAU 10-36/T.125

**Unité MCSPDU de type TIrq**

| Contenu | Source | Collecteur |
|---|---|---|
| Initiateur | Fournisseur supérieur | Fournisseur supérieur |
| Identificateur de jeton | Demande | Fournisseur supérieur |

L'unité **TIrq** contient l'identificateur d'utilisateur initiateur, qui doit être validé comme expliqué pour l'unité **CJrq**.

Si le jeton est libre et que la limite de domaine quant au nombre de jetons en usage le permet, ce jeton doit passer à l'état saisi. Si le jeton n'est inhibé que par l'utilisateur demandeur, il doit passer à l'état saisi. Sinon, l'état du jeton ne doit pas changer.

### 10.37 TIcf

L'unité de confirmation **TIcf** est produite au niveau du fournisseur MCS supérieur dès réception de l'unité **TIrq**. Réacheminée vers le fournisseur demandeur, elle produit une primitive de confirmation d'inhibition de jeton MCS-TOKEN-INHIBIT.

Le résultat doit être *favorable* si le jeton était préalablement libre ou inhibé ou s'il était le résultat d'une conversion de l'état saisi à l'état inhibé par le même utilisateur. Les autres résultats auront les valeurs *jetons trop nombreux* et *jeton non disponible*.

TABLEAU 10-37/T.125

**Unité MCSPDU de type TIcf**

| Contenu | Source | Collecteur |
|---|---|---|
| Résultat | Fournisseur supérieur | Confirmation |
| Initiateur | Fournisseur supérieur | Acheminement d'unité MCSPDU |
| Identificateur de jeton | Fournisseur supérieur | Confirmation |
| Etat du jeton | Fournisseur supérieur | Confirmation |

Les fournisseurs qui reçoivent l'unité **TIcf** doivent mettre à jour l'état du jeton dans leur base de données en fonction de l'état signalé.

Cette unité MCSPDU est acheminée par les mêmes voies que l'unité **TGcf**.

### 10.38 TVrq

L'unité de demande **TVrq** est produite par une primitive de demande de cession de jeton MCS-TOKEN-GIVE. Si elle est valide, elle remonte jusqu'au fournisseur MCS supérieur, qui retourne une indication **TVin** ou une confirmation **TVcf** défavorable.

TABLEAU 10-38/T.125

**Unité MCSPDU de type TVrq**

| Contenu | Source | Collecteur |
|---|---|---|
| Initiateur | Fournisseur demandeur | Fournisseur supérieur |
| Identificateur de jeton | Demande | Fournisseur supérieur |
| Destinataire | Demande | Fournisseur supérieur |

L'unité **TVrq** contient l'identificateur d'utilisateur initiateur, qui doit être validé comme expliqué pour l'unité **CJrq**.

Si le jeton est saisi par le demandeur et que le destinataire prévu existe, l'indication **TVin** doit être transmise au destinataire. Sinon, la demande échoue, l'état du jeton ne change pas et une confirmation **TVcf** est envoyée au demandeur avec le résultat *jeton non détenu* ou *utilisateur inconnu*.

### 10.39 TVin

L'unité d'indication **TVin** est produite au niveau du fournisseur MCS supérieur dès réception d'une unité **TVrq**. Acheminée jusqu'au destinataire prévu, elle produit une primitive d'indication de cession de jeton MCS-TOKEN-GIVE.

Les fournisseurs qui reçoivent l'unité **TVin** doivent normalement mettre à jour l'identificateur de jeton dans leur base de données pour indiquer qu'il est en phase de cession d'initiateur à destinataire. Mais si un fournisseur est le précédent supérieur d'un domaine inférieur en cours de fusion à la suite d'une primitive MCS-CONNECT-PROVIDER, ce fournisseur peut rejeter le jeton offert en émettant une réponse **TVrs** avec la cause *fusion de domaines*.

TABLEAU 10-39/T.125

**Unité MCSPDU de type TVin**

| Contenu | Source | Collecteur |
|---|---|---|
| Initiateur | Fournisseur supérieur | Indication |
| Identificateur de jeton | Fournisseur supérieur | Indication |
| Destinataire | Fournisseur supérieur | Acheminement d'unité MCSPDU |

L'unité **TVin** doit être envoyée en direction de l'identificateur d'utilisateur destinataire. Si cet identificateur ne peut pas être atteint parce qu'il n'existe plus de connexion MCS, il n'est besoin d'aucune action particulière car l'unité **DUin** doit toujours arriver ultérieurement pour signaler que le destinataire s'est détaché. Cette opération libérera le jeton dans la base de données.

### 10.40 TVrs

L'unité de réponse **TVrs** est produite par une primitive de réponse de cession de jeton MCS-TOKEN-GIVE. Si elle est valide, elle remonte jusqu'au fournisseur MCS supérieur, qui retourne une réponse de confirmation **TVcf**.

TABLEAU 10-40/T.125

**Unité MCSPDU de type TVrs**

| Contenu | Source | Collecteur |
|---|---|---|
| Résultat | Réponse | Fournisseur supérieur |
| Destinataire | Fournisseur répondeur | Fournisseur supérieur |
| Identificateur de jeton | Fournisseur répondeur | Fournisseur supérieur |

Un résultat de type *favorable* signifie que le destinataire accepte le jeton proposé.

L'identificateur d'utilisateur du rattachement MCS répondeur est communiqué par le fournisseur MCS qui reçoit la primitive de réponse. Les fournisseurs qui recevront des réponses **TVrs** ultérieures devront valider l'identificateur d'utilisateur pour faire en sorte qu'il soit légitimement attribué au sous-arbre de l'origine. Si l'identificateur d'utilisateur n'est pas répertorié dans la base de données du fournisseur comme étant attribué au destinataire, l'unité MCSPDU doit être ignorée.

Si l'identificateur de jeton est encore saisi par le donneur, son état doit être mis à jour pour indiquer qu'il est saisi par le destinataire si le résultat est favorable; sinon, l'état doit revenir à *saisi par le donneur* ou être supprimé de la base de données, selon que le donneur réside ou non dans le sous-arbre du fournisseur. Si l'identificateur de jeton a entre-temps été libéré par le donneur et que le résultat ne soit pas favorable, ce jeton doit être supprimé de la base de données du fournisseur.

Si l'unité MCSPDU n'est ni invalide ni ignorée, elle doit être expédiée en amont. Le fournisseur MCS supérieur doit donner suite à la réponse **TVrs** comme spécifié ci-dessus. De plus, si le donneur n'a pas déjà libéré le jeton, le fournisseur supérieur doit émettre une confirmation **TVcf** contenant le même résultat que l'unité **TVrs**.

### 10.41 TVcf

L'unité de confirmation **TVcf** est produite au niveau du fournisseur MCS supérieur dès réception de l'unité **TVrs**. Réacheminée vers le fournisseur demandeur, elle produit une primitive de confirmation de cession de jeton MCS-TOKEN-GIVE.

TABLEAU 10-41/T.125

**Unité MCSPDU de type TVcf**

| Contenu | Source | Collecteur |
|---|---|---|
| Résultat | Fournisseur supérieur | Confirmation |
| Initiateur | Fournisseur supérieur | Acheminement d'unité MCSPDU |
| Identificateur de jeton | Fournisseur supérieur | Confirmation |
| Etat du jeton | Fournisseur supérieur | Confirmation |

L'unité **TVcf** doit également être produite par le fournisseur MCS supérieur dès réception d'une unité **TVrq** si un jeton ne peut pas être offert au destinataire prévu. Cette opération remplace celle de production d'une unité **TVin**. Une unité **TVcf** doit également être produite avec le résultat *utilisateur inconnu* si le destinataire est détaché avant d'avoir reçu **TVrs**.

Les fournisseurs qui reçoivent l'unité **TVcf** doivent mettre à jour l'état du jeton dans leur base de données en fonction de l'état signalé.

Cette unité MCSPDU est acheminée par les mêmes voies que l'unité **TGcf**.

### 10.42 TPrq

L'unité de demande **TPrq** est produite par une primitive de demande de jeton souhaité MCS-TOKEN-PLEASE. Si elle est valide, elle remonte jusqu'au fournisseur MCS supérieur, qui multidestine une indication **TPin** afin d'alerter les utilisateurs actuels du jeton.

L'unité **TPrq** contient l'identificateur d'utilisateur initiateur, qui doit être validé comme expliqué pour l'unité **CJrq**.

TABLEAU 10-42/T.125

**Unité MCSPDU de type TPrq**

| Contenu | Source | Collecteur |
|---|---|---|
| Initiateur | Fournisseur demandeur | Fournisseur supérieur |
| Identificateur de jeton | Demande | Fournisseur supérieur |

### 10.43 TPin

L'unité d'indication **TPin** est produite au niveau du fournisseur MCS supérieur dès réception d'une unité **TPrq**. Elle fait l'objet d'une communication multipoint en aval et produit une primitive d'indication de demande de jeton souhaité MCS-TOKEN-PLEASE.

Les fournisseurs qui reçoivent l'unité **TPin** doivent l'envoyer à tous les subordonnés dont le sous-arbre contient un utilisateur qui a saisi, inhibé ou qui va recevoir le jeton spécifié.

TABLEAU 10-43/T.125

**Unité MCSPDU de type TPin**

| Contenu | Source | Collecteur |
|---|---|---|
| Initiateur | Fournisseur supérieur | Indication |
| Identificateur de jeton | Fournisseur supérieur | Indication |

## 10.44 TRrq

L'unité de demande **TRrq** est produite par une primitive de demande de libération de jeton MCS-TOKEN-RELEASE. Si elle est valide, elle remonte jusqu'au fournisseur MCS supérieur, qui retourne en réponse une confirmation **TRcf**.

TABLEAU 10-44/T.125

**Unité MCSPDU de type TRrq**

| Contenu | Source | Collecteur |
|---|---|---|
| Initiateur | Fournisseur demandeur | Fournisseur supérieur |
| Identificateurs de jeton | Demande | Fournisseur supérieur |

L'unité **TRrq** contient l'identificateur d'utilisateur initiateur, qui doit être validé comme expliqué pour l'unité **CJrq**.

Si le jeton est saisi par le demandeur, il doit devenir libre. S'il est inhibé, le demandeur doit être supprimé de l'ensemble des inhibiteurs; si cet ensemble devient vide, le jeton devient libre. Si le jeton est en cours de cession par le demandeur, il doit prendre un état intermédiaire distinct: *donné au destinataire prévu*, en attendant la réception d'une réponse **TVrs**. Sinon, l'état du jeton ne doit pas changer.

## 10.45 TRcf

L'unité de confirmation **TRcf** est produite au niveau du fournisseur MCS supérieur dès réception de l'unité **TRrq**. Réacheminée vers le fournisseur demandeur, elle produit une primitive de confirmation de libération de jeton MCS-TOKEN-RELEASE.

TABLEAU 10-45/T.125

**Unité MCSPDU de type TRcf**

| Contenu | Source | Collecteur |
|---|---|---|
| Résultat | Fournisseur supérieur | Confirmation |
| Initiateur | Fournisseur supérieur | Acheminement d'unité MCSPDU |
| Identificateur de jeton | Fournisseur supérieur | Confirmation |
| Etat du jeton | Fournisseur supérieur | Confirmation |

Le résultat doit être *favorable* si le jeton a été saisi ou inhibé par le demandeur ou si celui-ci était en train de le donner. L'autre résultat possible est *jeton non détenu*.

Les fournisseurs qui reçoivent l'unité **TRcf** doivent mettre à jour l'état du jeton dans leur base de données en fonction de l'état signalé.

Cette unité MCSPDU est acheminée par les mêmes voies que l'unité **TGcf**.

### 10.46 TTrq

L'unité de demande **TTrq** est produite par une primitive de demande de test de jeton MCS-TOKEN-TEST. Si elle est valide, elle remonte jusqu'au fournisseur MCS supérieur, qui retourne en réponse une confirmation **TTcf**.

L'unité **TTrq** contient l'identificateur d'utilisateur initiateur, qui doit être validé comme expliqué pour l'unité **CJrq**.

TABLEAU 10-46/T.125

**Unité MCSPDU de type TTrq**

| Contenu | Source | Collecteur |
|---|---|---|
| Initiateur | Fournisseur demandeur | Fournisseur supérieur |
| Identificateur de jeton | Demande | Fournisseur supérieur |

### 10.47 TTcf

L'unité de confirmation **TTcf** est produite au niveau du fournisseur MCS supérieur dès réception de l'unité **TTrq**. Réacheminée vers le fournisseur demandeur, elle produit une primitive de confirmation de test de jeton MCS-TOKEN-TEST.

Les fournisseurs qui reçoivent l'unité **TTcf** doivent constater que l'état du jeton dans leur base de données est conforme à l'état signalé.

Cette unité MCSPDU est acheminée par les mêmes voies que l'unité **TGcf**.

TABLEAU 10-47/T.125

**Unité MCSPDU de type TTcf**

| Contenu | Source | Collecteur |
|---|---|---|
| Résultat | Fournisseur supérieur | Acheminement d'unité MCSPDU |
| Identificateur de jeton | Fournisseur supérieur | Confirmation |
| Etat du jeton | Fournisseur supérieur | Confirmation |

## 11 Base de données de fournisseur de service MCS

### 11.1 Copie hiérarchique

Bien qu'un fournisseur MCS puisse couvrir plusieurs domaines, il dessert chacun d'entre eux indépendamment. Il tient logiquement à jour une base de données distincte pour chaque domaine afin d'y enregistrer l'état de canal et les ressources de jeton utilisées. La description ci-après est conçue dans le cadre d'un domaine unique.

Les ressources du service MCS qu'il y a lieu de gérer dans un domaine sont les identificateurs de canal et les identificateurs de jeton. Les identificateurs d'utilisateur sont un sous-ensemble des identificateurs de canal. Les paramètres de domaine imposent une limite au nombre d'identificateurs de chaque catégorie que l'on peut utiliser simultanément. Cela permettra à un fournisseur de calculer la quantité de mémoire nécessaire pour sa base de données dans le cas le moins favorable d'un domaine complètement utilisé.

Dans la hiérarchie d'un domaine, les identificateurs utilisés au niveau de tout fournisseur MCS donné se réduisent au sous-ensemble de ceux qui sont utilisés au niveau de son fournisseur immédiatement supérieur. Les renseignements relatifs à un identificateur sont enregistrés au point où ils peuvent être utilisés pour prendre en charge des services MCS mettant en œuvre cet identificateur. Un enregistrement plus étendu d'informations entraînerait des coûts supplémentaires dans le trafic des unités MCSPDU afin de tenir les renseignements à jour. Comme les informations enregistrées au niveau d'un fournisseur sont cohérentes avec celles qui sont enregistrées au niveau de fournisseurs plus élevés, dans les limites du temps de propagation des unités MCSPDU, on peut considérer que la base de données du fournisseur est partiellement copiée dans la hiérarchie du domaine.

Les paramètres de domaine deviennent fixes et immuables lors de l'établissement de la première connexion MCS de ce domaine. Un fournisseur qui n'a pas la capacité correspondant au nombre maximal d'identificateurs spécifié dans chaque catégorie peut négocier son adhésion à un domaine sous un faux prétexte. Il peut faire l'hypothèse que, dans sa position inférieure de la hiérarchie, il ne sera pas appelé à conserver plus qu'une fraction de la base de données totale. Un tel fournisseur peut prendre en charge des rattachements et des subordonnés sans la gamme complète de services MCS que ces rattachements et subordonnés peuvent escompter. Néanmoins, tant que sa capacité n'est pas effectivement dépassée, un tel fournisseur peut apparaître comme un membre à part entière du domaine. Cette stratégie peut convenir à des nœuds terminaux à besoins limités.

Les identificateurs sont d'abord mis en application au niveau du fournisseur MCS supérieur. Ils sont mis en application au niveau de fournisseurs subordonnés au moyen d'un flux descendant et sélectif d'unités MCSPDU. La plupart de ces identificateurs sont supprimés de la même manière, de haut en bas. Il y a nécessairement des intervalles pendant lesquels un fournisseur enregistre comme utilisé un identificateur que ses supérieurs n'utilisent plus, parce que l'unité MCSPDU qui supprime cet identificateur est encore en transit. Ce n'est cependant pas une situation durable. Les unités MCSPDU de commande sont reçues et traitées dans l'ordre où elles ont été envoyées. Les conséquences du traitement d'une unité MCSPDU, y compris sa création ou sa suppression d'identificateurs de canal ou de jeton, prennent effet avant que l'attention passe à l'événement d'entrée suivant.

Une exception à la règle du paragraphe précédent est la suppression d'identificateurs de canal statique et de canal attribué. Bien que mis en usage par un flux descendant d'unités de confirmation **CJcf**, ces identificateurs de canal sont supprimés dans l'ordre contraire: de bas en haut. Plus précisément, ils sont supprimés lorsqu'une accumulation de demandes MCS-CHANNEL-LEAVE issues de rattachements se combine avec des unités MCSPDU de type **CLrq** issues de fournisseurs subordonnés pour effectuer une sortie de canal sans adhésion. De telles transitions justifient l'envoi plus en amont d'unités **CLrq**. Dans ces deux cas, les identificateurs de canal enregistrés comme étant en usage formeront donc un sous-ensemble exact des identificateurs de canal en usage au niveau du fournisseur plus élevé. Il s'agit d'un corollaire obligé des optimisations conçues pour accélérer la gestion des canaux avant un transfert de données.

Les identificateurs de canal sont mis en usage par les unités **MCcf, AUcf, CJcf, CCcf** et **CAin**; ils sont supprimés par les unités **MCcf, PCin, DUin, CLrq, CDin** et **CEin**. Les identificateurs de jeton sont mis en usage par les unités **MTcf, TGcf, TIcf** et **TVin**; ils sont supprimés par les unités **MTcf, PTin, TRcf, TVrs** et **TVcf**. Lorsqu'un identificateur est mis en usage au niveau d'un fournisseur donné, l'unité MCSPDU qui en est la cause peut être envoyée à zéro, un, plusieurs ou tous les fournisseurs subordonnés. L'utilisation d'un identificateur peut augmenter ou diminuer progressivement, par exemple si des utilisateurs individuels sont admis dans un canal privé et en sont expulsés. Lorsqu'un identificateur est supprimé d'un fournisseur donné, l'unité MCSPDU qui effectue cette opération est envoyée à tous les subordonnés qui peuvent encore être en train d'enregistrer cet identificateur comme étant en usage.

L'utilisation d'un identificateur est finalement liée aux actions sur un canal ou sur un jeton par un utilisateur rattaché au domaine (malgré un certain retard dû, comme cela a été expliqué, à la communication des changements par l'envoi d'unités MCSPDU). Les identificateurs enregistrés de manière stable comme étant en usage au niveau d'un fournisseur MCS donné sont ceux qui sont activement employés par un utilisateur se trouvant dans le sous-arbre de ce fournisseur. Il en découle que ces identificateurs forment un sous-ensemble de ceux qui ont été enregistrés de manière stable au niveau d'un fournisseur plus élevé quelconque.

La suppression d'un identificateur d'utilisateur a pour effet secondaire de supprimer les identificateurs de canal et les identificateurs de jeton dont il est l'unique utilisateur dans un sous-arbre.

Les paragraphes suivants spécifient les critères permettant de considérer que des identificateurs de canal et des identificateurs de jeton sont en usage.

## 11.2 Informations relatives aux canaux

Les quatre types de canal possèdent des critères correspondants pour déterminer si un point de rattachement donné est considéré comme utilisant l'identificateur de canal et donc s'il doit être représenté dans une base de données de fournisseur:

a) un identificateur de canal statique (dans l'étendue 1 ... 1000) est en usage si l'utilisateur a adhéré à ce canal avec une primitive de confirmation MCS-CHANNEL-JOIN contenant une valeur favorable et s'il n'est pas sorti de ce canal par une primitive de demande ou d'indication MCS-CHANNEL-LEAVE;

b) un canal d'identification d'utilisateur est en usage s'il a été attribué à l'utilisateur par une primitive de confirmation MCS-ATTACH-USER contenant une valeur favorable et si cet utilisateur n'a pas été détaché par une demande ou indication MCS-DETACH-USER;

c) un identificateur de canal privé est en usage si l'utilisateur a créé le canal avec une confirmation MCS-CHANNEL-CONVENE favorable ou s'il y a été admis avec une indication MCS-CHANNEL-ADMIT et n'en a pas été expulsé par une indication MCS-CHANNEL-EXPEL; le canal ne doit pas non plus avoir été dissous par une demande ou indication MCS-CHANNEL-DISBAND;

d) un identificateur de canal attribué est en usage si l'utilisateur a adhéré au canal avec une confirmation MCS-CHANNEL-JOIN favorable et qu'il n'en soit pas sorti par une demande ou indication MCS-CHANNEL-LEAVE.

Les informations suivantes doivent être enregistrées pour un identificateur de canal en usage:

a) le type de canal qu'il représente (statique, d'identification d'utilisateur, privé ou attribué);

b) les rattachements et connexions MCS avec des fournisseurs subordonnés qui ont adhéré au canal;

c) dans le cas d'un canal d'identification d'utilisateur, le sens d'acheminement correspondant: soit celui du rattachement MCS local auquel l'identificateur d'utilisateur est attribué ou la connexion MCS descendante vers un fournisseur subordonné dans le sous-arbre duquel l'utilisateur réside;

d) dans le cas d'un identificateur de canal privé, l'identificateur d'utilisateur du gestionnaire qui l'a constitué (que ce gestionnaire se trouve ou non dans le sous-arbre du fournisseur) et l'ensemble de tous les identificateurs d'utilisateur contenus dans le sous-arbre du fournisseur et admis dans ce canal.

Les informations enregistrées pour les identificateurs de canal sont employées comme expliqué à l'article 10 afin de valider les unités MCSPDU de demande et pour acheminer des unités MCSPDU d'indication et de confirmation.

## 11.3 Informations relatives aux jetons

Les transitions d'état d'un identificateur de jeton sont représentées à la Figure 11-1.

Un identificateur de jeton individuel peut être *saisi* par un utilisateur unique ou être *inhibé* par un ou plusieurs utilisateurs. L'action de l'unité **TVin** fait passer l'état du jeton à *donnant* sur la branche d'une hiérarchie de domaine conduisant du fournisseur MCS supérieur au destinataire prévu. Cet état se dégrade en *incessible* si le destinataire se détache avant que son fournisseur ait envoyé une unité **TVrs** en réponse. Il passe à *donné* si, au contraire, le donneur libère explicitement le jeton ou se détache. Au cours de la cession d'un jeton, la branche d'une hiérarchie de domaine qui part du donneur coupe la branche menant vers le destinataire au moins au niveau du fournisseur MCS supérieur. L'état du jeton passe de *saisi* à *donnant* puis, le cas échéant, à *non donnable* ou à *donné*, mais seulement à partir de cette intersection.

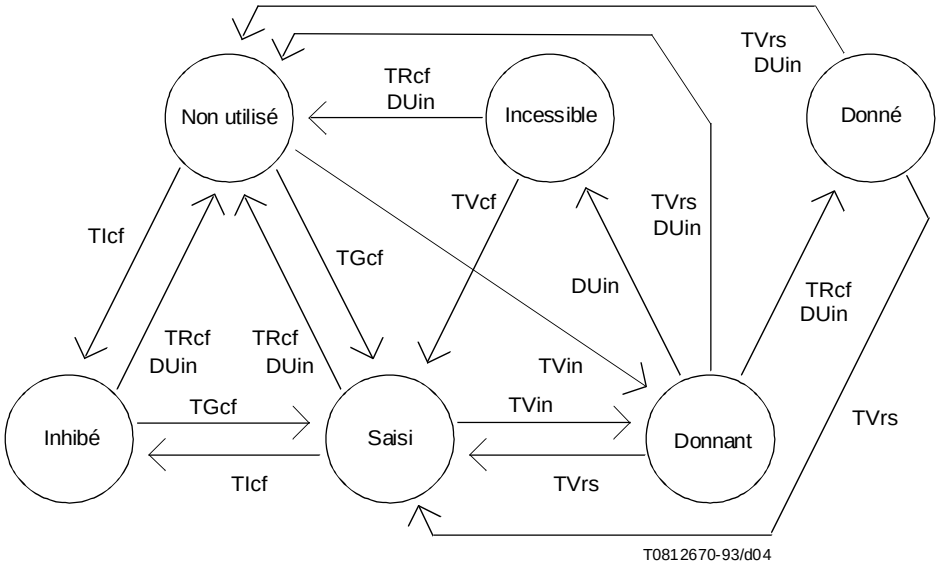



FIGURE 11-1/T.125

**Transitions d'état d'un identificateur de jeton**

L'utilisateur d'un jeton a avec celui-ci une relation de saisisseur, inhibiteur, destinataire ou à la fois de saisisseur et de destinataire (lorsqu'il se donne à lui-même un jeton):

a) l'utilisateur est un saisisseur s'il a saisi un jeton avec une primitive de confirmation MCS-TOKEN-GRAB contenant une valeur favorable et qu'il ne l'ait pas libéré par une demande MCS-TOKEN-RELEASE ou par une confirmation MCS-TOKEN-GIVE favorable ni transformé avec une confirmation MCS-TOKEN-INHIBIT favorable ou avec une réponse MCS-TOKEN-GIVE s'il a accepté un jeton proposé;

b) l'utilisateur est un inhibiteur s'il a saisi un jeton avec une confirmation MCS-TOKEN-INHIBIT favorable et qu'il ne l'ait pas libéré avec une demande MCS-TOKEN-RELEASE ni transformé avec une confirmation MCS-TOKEN-GRAB favorable;

c) l'utilisateur est un destinataire si un jeton lui a été proposé par une indication MCS-TOKEN-GIVE et qu'il n'ait pas libéré ce jeton avec une réponse MCS-TOKEN-GIVE défavorable.

Ces informations doivent être enregistrées comme suit pour un identificateur de jeton en usage:

a) l'état de l'identificateur de jeton au niveau du fournisseur MCS (cet état n'est pas nécessairement le même qu'au niveau du fournisseur supérieur);

b) si le jeton est saisi ou incessible, l'identificateur d'utilisateur du saisisseur dans le sous-arbre du fournisseur;

c) si le jeton est donnant, l'identificateur d'utilisateur du saisisseur (que celui-ci soit ou non dans le sous-arbre du fournisseur);

d) si le jeton est donnant ou donné, l'identificateur d'utilisateur du destinataire dans le sous-arbre du fournisseur;

e) si le jeton est inhibé, l'ensemble de tous les identificateurs d'utilisateur se trouvant dans le sous-arbre du fournisseur, qui ont inhibé le jeton.

Les informations enregistrées pour les identificateurs de jeton en usage sont employées comme expliqué à l'article 10 pour valider les unités MCSPDU de réponse et pour acheminer les unités MCSPDU d'indication.

L'état d'un identificateur de jeton au niveau d'un fournisseur subordonné n'a pas besoin d'être identique à ce qu'il serait au niveau du fournisseur MCS supérieur. Cela est dû au fait qu'un donneur de jeton ne traite généralement pas les unités **TVin** ou **TVrs** et qu'un destinataire ne traite généralement pas une confirmation **TRcf** de donneur. La Figure 11-2 montre les états qui peuvent apparaître lors d'une interaction complexe entre jetons.

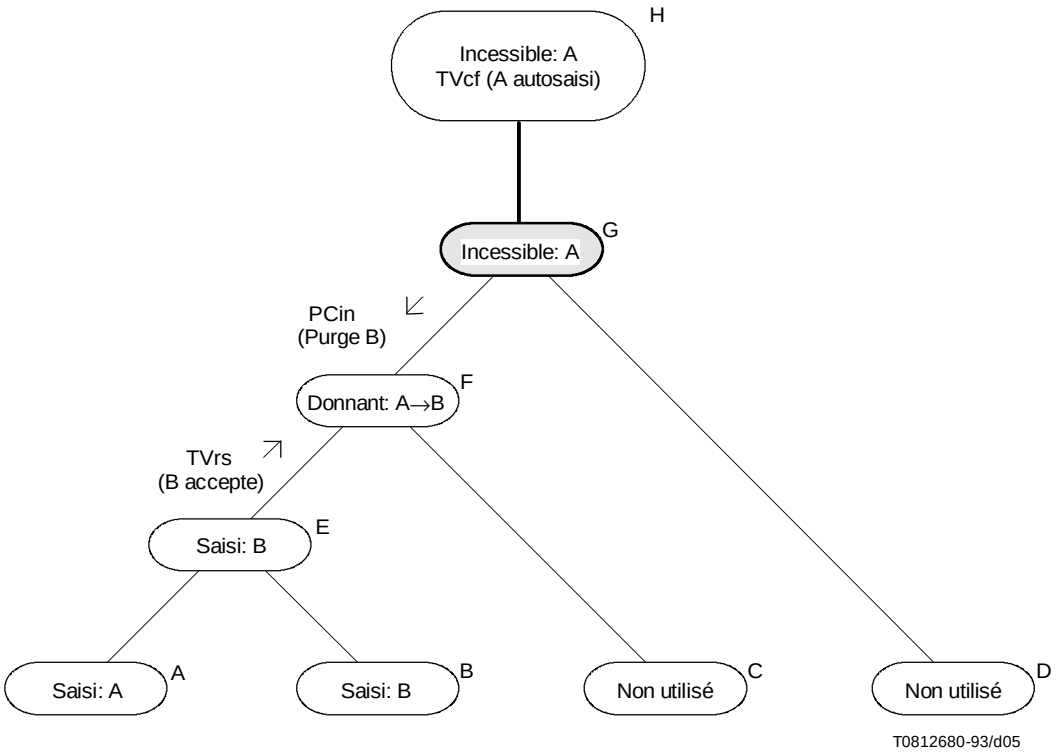



FIGURE 11-2/T.125

**Etats d'un jeton pouvant apparaître lors d'une interaction complexe**

La figure concerne un seul identificateur de jeton dans la base de données des fournisseurs A à H. Un cas de figure plausible est celui où le jeton a été donné par l'utilisateur A, lui-même rattaché au fournisseur A, à l'utilisateur B rattaché au fournisseur B. Avant que l'utilisateur B puisse répondre, le fournisseur G a toutefois connecté le domaine à un nouveau fournisseur supérieur H et a commencé une fusion. Le nouveau domaine, incapable d'accepter l'utilisateur B en raison d'un conflit entre identificateurs de canal, a commencé sa purge par l'intermédiaire de l'unité **PCin**. Cette unité MCSPDU est représentée après avoir effectué une partie de son itinéraire passant par les fournisseurs G et D. Le fournisseur G, par conséquent, a adapté l'état de son jeton de *donnant* à incessi*ble*. Puis il a fusionné le jeton, avec cet état, dans le nouveau domaine. Le nouveau fournisseur supérieur H, auquel a été présenté un jeton avec cet état, a mis en file d'attente de transmission une confirmation **TVcf** de valeur défavorable afin de renvoyer le jeton à l'utilisateur A. A l'instant de la description, le jeton a aussi été finalement accepté par l'utilisateur B et une unité de réponse **TVrs** est en train de remonter la branche destinataire de la hiérarchie domaniale: elle fait passer l'état du jeton, à chaque fournisseur, de *donnant* à *saisi* par B. Deux unités MCSPDU convergent donc vers le fournisseur F et celle qui arrivera la première déterminera l'état suivant du jeton à ce point: soit un retour à l'état *saisi* par A ou le maintien transitoire à *saisi* par B puis de nouveau l'état *non utilisé* une fois que l'utilisateur B se sera détaché. Dans un cas comme dans l'autre, l'état du jeton se stabilisera à *saisi* par A lorsque le nouveau fournisseur supérieur enverra la confirmation **TVcf** qu'il tient en instance.

## 12 Eléments de procédure

### 12.1 Ordonnancement des unités MCSPDU

Les unités MCSPDU de commande restent en séquence entre toute paire de fournisseurs MCS parce qu'elles se déplacent sur une même connexion de transport: la connexion de transport initiale d'une connexion MCS. Les fournisseurs MCS doivent traiter les unités MCSPDU reçues et envoyer toutes unités MCSPDU de sortie résultantes dans le même ordre. Cela s'applique aux unités MCSPDU qui sont simplement expédiées en amont ou en aval de la

hiérarchie du domaine ainsi qu'aux unités MCSPDU qui sont transformées, comme des demandes ou des réponses devenant des indications ou des confirmations. L'ordonnancement des unités MCSPDU doit être conservé dans le cadre d'un fournisseur de service MCS s'il est nécessaire de mettre des informations de sortie en file d'attente pour transmission ultérieure en raison de la contre-pression de la commande de flux sur une connexion de transport.

Les unités MCSPDU de données ayant des priorités différentes n'ont pas besoin de rester en séquence. Au contraire, l'avantage de priorités relatives n'est obtenu que lorsque les données de priorité supérieure sont mises devant les données de priorité inférieure. C'est-à-dire qu'il y a lieu de les transmettre sur des connexions de transport distinctes et de les mettre en files d'attente séparément chez chaque fournisseur MCS. Si le nombre de priorités de données mises en œuvre dans un domaine est inférieur à la valeur maximale, un moins grand nombre de connexions de transport sera disponible. Mais les fournisseurs qui décident de faire ainsi peuvent toujours gérer des files d'attente distinctes en interne. Cela tirera parti de certains avantages de la priorité relative, mais pas de tous.

Un fournisseur MCS doit respecter une certaine priorité pour la séquence des unités MCSPDU de données transmises. Il s'agit là d'une restriction plus sévère que celle qui est imposée par la Recommandation T.122 de l'UIT-T, qui ne garantit que l'ordonnancement des unités de données du service qui ont été transmises avec une priorité donnée sur le même canal de destination.

Les unités MCSPDU de commande et de données qui ont une priorité absolue sont transmises sur la même connexion de transport initiale. Elles doivent faire l'objet d'une attention égale de la part d'un fournisseur MCS. Les données de priorité inférieure peuvent venir après. Des indications de commande progressant en tête peuvent arriver avant les données de priorité inférieure qui ont été transmises effectivement les premières.

## 12.2 Commande du flux d'entrée

Un fournisseur MCS se trouve parfois devant des objectifs contradictoires: continuer à faire avancer rapidement des données malgré des blocages transitoires affectant certains récepteurs, donner aux émetteurs un accès équitable à la largeur de bande disponible et empêcher un quelconque correspondant de se faire largement distancer par des homologues qui reçoivent les mêmes données faisant l'objet d'une communication point à multipoint. Le service de communication multipoint (MCS) est un service fiable qui préserve l'intégrité des données d'utilisateur. Comme un fournisseur est limité dans sa capacité d'enregistrer des unités MCSPDU lorsque celles-ci ne peuvent pas être transmises immédiatement, ce fournisseur doit avoir la possibilité de se protéger en refusant de continuer à recevoir des données. Bien que les détails de l'interface avec les services de transport relèvent de décisions locales, l'effet abstrait de ce refus doit être que des unités TSDU entrantes sont conservées dans des pipelines de connexions de transport, intactes et en séquence, pour réception ultérieure dès que la commande de flux le permettra. Lorsque ces pipelines de connexion de transport se remplissent, des fournisseurs MCS distants peuvent se trouver devant une contre-pression bloquant leurs émissions de nouvelles MCSPDU. Ils peuvent dans ce cas faire appel à un système de protection analogue.

La commande de flux n'est pas décrite explicitement dans le protocole MCS. Il s'agit d'une fonction des couches inférieures qu'il serait dispendieux de dédoubler. Il en découle qu'il est difficile de percevoir, par l'intermédiaire d'une connexion de transport mise à contribution, si un fournisseur distant se protège contre de nouvelles entrées. Pour répondre au mieux à des objectifs contradictoires, on peut recommander les politiques décrites ci-après.

Un fournisseur MCS peut accorder à chaque connexion de transport entrante un quota fixe de tampons qu'il peut remplir avec des unités MCSPDU avant que la contre-pression soit appliquée. Chaque tampon est traité comme spécifié dans le présent protocole, puis affecté à un accès de sortie vers zéro, une ou plusieurs connexions de transport sortantes. La sortie peut se produire immédiatement ou être retardée parce qu'un pipeline de transport est plein. Avant que son contenu soit envoyé en sortie vers la dernière connexion de transport, un tampon se charge en fonction du quota d'entrée de la connexion de transport qui a transporté celui-ci. Une fois le tampon vidé vers toutes les connexions de transport requises, il est réinitialisé pour se recharger en fonction de ce quota. Lorsqu'un quota est épuisé, les nouvelles entrées sur la connexion de transport correspondante sont arrêtées. Les quotas d'entrée peuvent être déterminés compte tenu du fait qu'une connexion de transport fait ou non partie d'une connexion MCS ascendante ou descendante et du niveau de priorité de données qu'elle représente.

La mise en mémoire-tampon peut atténuer une disparité de débits entre émetteurs et récepteurs. Un quota imposé aux entrées peut empêcher une certaine connexion de transport de monopoliser les ressources. Elle peut également imposer une limite à la mesure dans laquelle deux récepteurs des mêmes données faisant l'objet d'une communication point à multipoint peuvent être asynchrones. Le procédé recommandé n'est cependant pas assez élaboré pour prévoir tous les cas d'utilisation. Il peut parfois ralentir le débit de transfert des données à travers un domaine lorsqu'il existe d'autres solutions acceptables. La découverte de meilleures politiques de commande de flux (mises en œuvre localement et ne nécessitant pas de communication additionnelle d'unités MCSPDU) pourra être un moyen de différencier des produits.

### 12.3 Application du débit

Contrairement à la commande du débit d'entrée, le protocole MCS développe quelques moyens d'appliquer le débit utile. Tout d'abord, le débit qui est appliqué est un paramètre de domaine qui est négocié par l'intermédiaire de la primitive MCS-CONNECT-PROVIDER. En deuxième lieu, l'intervalle de temps pendant lequel le débit est surveillé avant de prendre une mesure contraire est communiqué par chaque fournisseur MCS à son supérieur par l'intermédiaire de l'unité **EDrq**. La description de l'intervalle d'application du débit nécessite que les fournisseurs aient en commun certains principes de comportement. Il n'en reste pas moins que l'application reste une technique heuristique qui laisse place à l'invention.

L'application d'un débit d'entrée minimal à chaque récepteur est une option disponible pour commander des applications qui établissent les connexions MCS d'un domaine. Cette option est choisie par l'intermédiaire du paramètre de domaine relatif à l'application du débit, qui est indiqué en octets par seconde. Bien qu'il paraisse évident qu'il n'y a pas lieu d'autoriser un correspondant à fonctionner à une vitesse arbitrairement lente et à faire ainsi obstacle au transfert de données avec d'autres correspondants, le risque inverse est de chercher à imposer trop strictement un débit.

Les configurations complexes à émetteurs multiples posent un problème. Les types de contre-pression auxquels une politique d'application de débit réagit peuvent ne pas provenir d'un seul récepteur de vitesse anormalement lente. Tout d'abord, les unités MCSPDU descendantes sont en concurrence de traitement avec les unités MCSPDU qui remontent mais dont une image redescend, comme l'indication **SDin**. Le flux descendant rencontré par l'intermédiaire d'un fournisseur homologue peut donc n'atteindre qu'une fraction de la largeur de bande nominale et peut varier dynamiquement selon le nombre d'autres connexions et rattachements à cet homologue. En deuxième lieu, seules les unités MCSPDU de priorité absolue sont censées transiter en continu sur une connexion MCS. Des unités de moindre priorité peuvent tout à fait être bloquées par un fournisseur homologue pendant de longues périodes à cause de l'intensité du trafic provenant d'autres sources avec des priorités plus élevées. Finalement, une grande variation de débit instantané peut se produire si la mesure tient compte du temps pendant lequel une unité MCSPDU bloquée doit attendre avant d'être acceptée sur une connexion MCS. Ce temps peut dépendre, entre autres variables, du nombre et de l'ordre des unités MCSPDU issues d'autres sources et mises en file d'attente chez le fournisseur homologue.

Néanmoins, l'application du débit est une option intéressante dans les cas réels où l'on sait que les configurations de transfert de données sont plus uniformes. Il faut l'interpréter comme exigeant un envoi minimal d'unités MCSPDU vers chaque rattachement MCS direct et vers chaque connexion MCS descendante dans un intervalle de temps qui sera spécifié par le fournisseur MCS imposant son débit. Chaque unité MCSPDU sortante, de commande aussi bien que de données, doit compter pour le débit comme si elle avait la longueur maximale autorisée par les paramètres de domaine. La sortie d'une MCSPDU vers un rattachement MCS implique l'acheminement de la primitive d'indication ou de confirmation correspondante. La sortie d'une MCSPDU vers une connexion MCS descendante implique l'absence de contre-pression à l'interface avec le service de transport et l'acceptation de cette unité dans le pipeline de la connexion de transport correspondante.

La sortie doit être contrôlée tant qu'au moins une unité MCSPDU est, quelle que soit la priorité des données, en file d'attente de transmission vers un rattachement donné ou une connexion descendante donnée. Chaque fois que les files sont vides, le contrôle doit s'arrêter et aucune mesure d'application de débit ne doit être prise. En même temps, aucune accumulation de droits n'est ouverte par un bon comportement afin de compenser de futures défaillances. Le contrôle doit reprendre dès qu'une contre-pression empêche une unité MCSPDU d'être envoyée en sortie et oblige au contraire à la mettre en file d'attente. Lorsqu'au moins une unité MCSPDU reste en file d'attente, le nombre d'unités effectivement envoyées en sortie doit être compté sur un intervalle de temps déterminé.

Chaque fournisseur MCS doit choisir un intervalle d'application de débit. Cet intervalle doit être assez long pour qu'au moins une unité MCSPDU de longueur maximale puisse être envoyée en sortie au débit minimal. Un fournisseur MCS doit signaler à son supérieur l'intervalle choisi et tous changements ultérieurement apportés à cette durée par un envoi d'unité **EDrq** en amont. Un fournisseur doit se protéger contre le rattachement MCS ou la connexion MCS descendante fautifs à la fin de tout intervalle pendant lequel le débit contrôlé n'arrive pas à atteindre les valeurs prévues. Ce fournisseur doit alors détacher l'utilisateur ou déconnecter la connexion.

Des fournisseurs de niveau élevé dans la hiérarchie peuvent régler leur intervalle d'application de débit de manière qu'il soit plus long que celui de chacun de leurs subordonnés, plus une certaine marge pour le temps de réaction. Le but est d'inciter à ce que la mesure d'application du débit soit prise à partir du fournisseur de niveau le moins élevé qui a la possibilité de détecter un problème. Lorsqu'un contrevenant est éliminé à la fin d'un intervalle correspondant à un niveau inférieur, les fournisseurs de niveau plus élevé doivent normalement avoir assez de temps pour percevoir le rétablissement du débit à des niveaux adéquats. S'ils agissent trop rapidement, ils peuvent pénaliser un plus grand sous-arbre que nécessaire et interrompre des utilisateurs non fautifs dans le domaine.

Il y a lieu que les applications de régulation qui déterminent les paramètres de domaine soient prudentes dans leurs exigences et qu'elles s'attendent que le débit pourra occasionnellement chuter en périodes d'intenses stimulations applicatives. Si leur objet est simplement d'assurer une protection à l'encontre des récepteurs qui arrêtent d'accepter

toute entrée que ce soit, ces applications peuvent fixer à une valeur très basse le débit minimal. Connaissant la longueur maximale des unités MCSPDU et le débit utile appliqué, les régulateurs peuvent calculer l'intervalle minimal qui devra toujours s'écouler avant qu'un blocage quelconque soit détecté et résolu.

## 12.4 Configuration de domaine

La Recommandation T.122 de l'UIT-T n'indique aucun mécanisme permettant de configurer l'ensemble des domaines gérés par un fournisseur MCS. Il faut considérer cela comme relevant d'une décision locale, dont la normalisation pourra faire l'objet d'un complément d'étude. Le présent protocole part du principe qu'un fournisseur MCS reconnaîtra certains sélecteurs de domaine comme étant valides et d'autres comme invalides. Il prévoit que les sélecteurs de domaine soient communiqués dans le cadre de l'établissement d'une connexion MCS.

Un fournisseur MCS participe implicitement à la négociation des paramètres de domaine. Qu'il soit du côté appelant ou du côté appelé, il contraint l'étendue des valeurs paramétriques autorisées en fonction des limites de sa configuration. Un fournisseur MCS doit suspendre la possibilité de négocier les paramètres d'un domaine une fois qu'un utilisateur quelconque s'est rattaché à ce domaine ou que la première connexion MCS a été établie.

## 12.5 Fusion de domaines

Les domaines sont fusionnés à la suite d'une primitive MCS-CONNECT-PROVIDER. Si l'on souhaite faire en sorte qu'un domaine ou un autre soit vide en l'occurrence, cela ne complique pas beaucoup une fusion. Dans le cas le plus général, il faut toutefois tenir compte d'une mise à jour de la base de données au niveau du fournisseur supérieur restant de manière qu'elle reprenne le contenu de la base de données du précédent fournisseur supérieur, et prévoir la résolution de tous conflits pouvant apparaître. L'article 10 développe cela plus en détail.

Pour faciliter la compréhension, la séquence des Figures 12-1 à 12-4 montre un exemple de fusion de domaines. Le fournisseur E y représente un ancien supérieur qui a adhéré à un nouveau domaine au moyen de la connexion MCS indiquée, qui le relie à un fournisseur intermédiaire F. La connexion MCS peut avoir été établie par le fournisseur E ou par le fournisseur F.

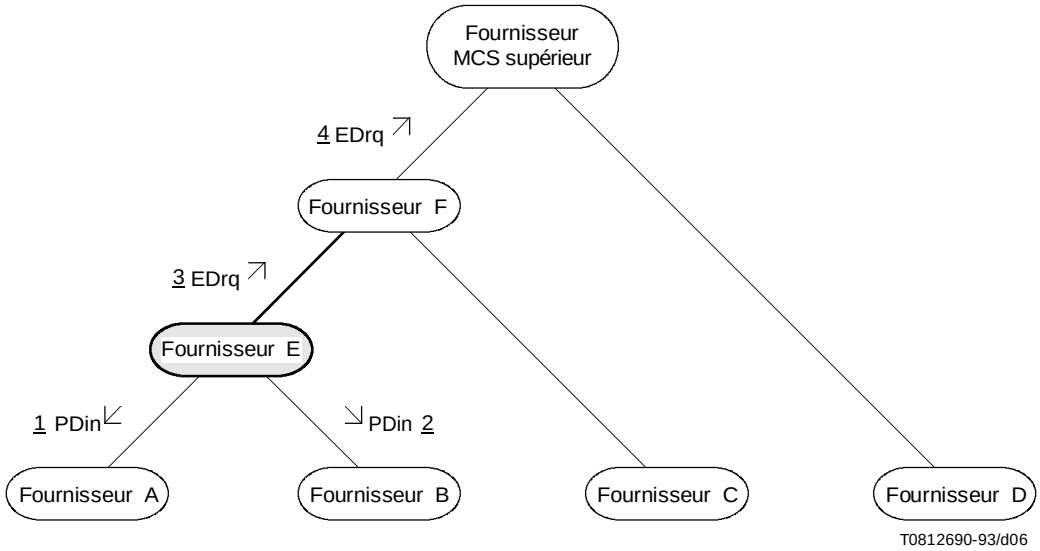



FIGURE 12-1/T.125

**Première phase de la fusion de domaines – Etablissement de la hiérarchie**

Le fournisseur E, qui se trouve à l'extrémité inférieure d'une connexion ascendante, prend la responsabilité d'exécuter la fusion. Comme il est risqué d'entreprendre une autre activité pendant que la base de données est en consultation, le fournisseur E arrête d'accepter des entrées issues de son sous-arbre. Toute unité MCSPDU déjà en transit depuis les fournisseurs A et B – ou de nouveaux fournisseurs créés avant l'achèvement de la fusion – sera sauvegardée et traitée ultérieurement. Le flux descendant d'unités MCSPDU n'est toutefois pas gêné, qu'il ait été produit par le fournisseur E ou expédié de plus haut.

En attendant que la fusion soit effectuée, les seules unités MCSPDU de confirmation que le fournisseur E recevra seront des unités **MCcf** et **MTcf**, étant donné que les demandes d'utilisateur ne sont pas autorisées à remonter dans la hiérarchie. Ces unités confirmeront ou purgeront les identificateurs de canaux et de jetons déjà en usage. Des unités d'indication de type **PCin, PTin, DUin** et **CDin** peuvent également arriver du domaine de niveau plus élevé et peuvent supprimer des identificateurs de canaux et de jetons issus du domaine de niveau moins élevé dont la fusion a été confirmée à titre individuel. Les identificateurs non confirmés du domaine moins élevé sont protégés contre la suppression car ils ont encore une signification différente de leurs homologues du domaine plus élevé. L'unité **PDin** doit toujours être suivie d'effet afin d'appliquer la limite de hauteur de domaine qu'elle indique. Les autres unités d'indication peuvent ne pas être appliquées par le fournisseur E pendant la réalisation de la fusion. Il n'est en particulier pas obligatoire que des transferts de données s'effectuent entre domaines antérieurement séparés, en attendant que la fusion soit terminée; ces données ne peuvent pas passer avant que les canaux qui les acheminent aient au moins été réorganisés. Il peut être gênant que deux unités d'indication puissent mettre en usage de nouveaux identificateurs. Afin que la base d'informations reste cohérente, le fournisseur E doit réagir comme spécifié à l'article 10 s'il refuse une indication **CAin** ou **TVin**.

Les premières actions du fournisseur E sont d'envoyer en aval une unité **PDin** afin de s'assurer que la plus récente connexion MCS n'a pas créé de boucle qui contreviendrait au principe que chaque hiérarchie de domaine ne possède qu'un seul fournisseur supérieur et d'envoyer en amont une unité **EDrq** pour signaler sa hauteur existante et l'intervalle d'application de débit. Le fournisseur F, passant ainsi de la hauteur 2 à la hauteur 3, relaie la demande **EDrq** vers le fournisseur supérieur, qui passe alors à la hauteur 4.

Au cours de la deuxième étape de la fusion de domaines, le fournisseur E envoie en amont autant d'instances de l'unité **MCrq** qu'il en faut pour insérer les identificateurs d'utilisateur dans sa base de données. Les identificateurs d'utilisateur qui n'entrent pas en conflit avec le domaine supérieur sont confirmés. Les autres sont purgés dans un nombre égal d'instances de l'unité **MCcf**. Le fournisseur E produit l'indication **PCin** à partir de la confirmation **MCcf** afin de signaler toutes purges dans l'ensemble de son sous-arbre. Cette phase se termine lorsque tous les identificateurs d'utilisateur ont soit été explicitement confirmés ou ont été purgés.

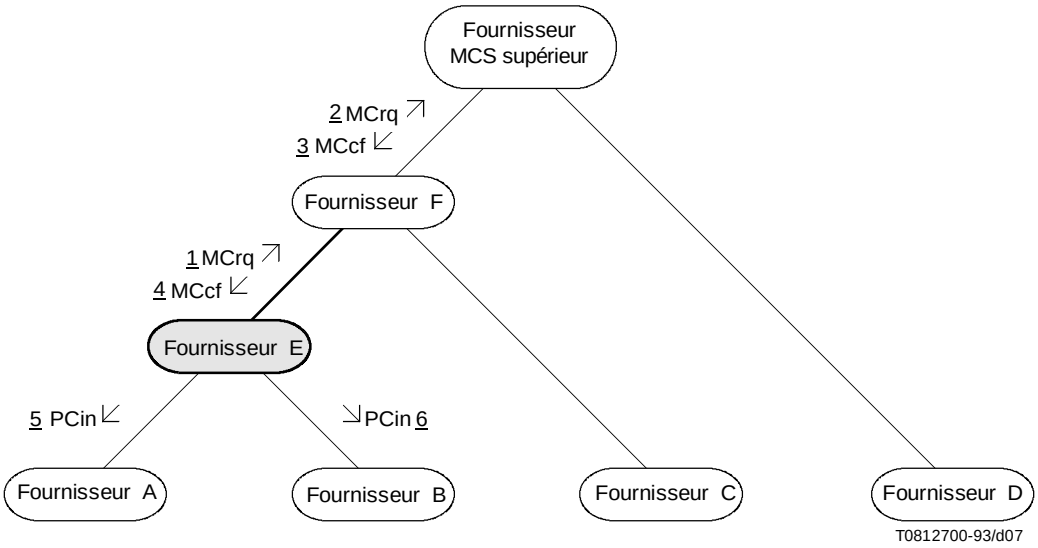



FIGURE 12-2/T.125

**Deuxième phase de la fusion de domaines – Fusion des canaux d'identification d'utilisateur**

La troisième phase ressemble à la deuxième, mais concerne les identificateurs de jeton plutôt que les canaux d'identification d'utilisateur, au moyen d'un ensemble parallèle d'unités MCSPDU. Si les identificateurs d'utilisateur n'ont pas été fusionnés d'abord, des portions d'une ultérieure unité **MTrq** pourront être refusées parce qu'invalides et les identificateurs de jeton affectés seront purgés sans nécessité. Dans le cas d'un jeton inhibé, l'ensemble entier des utilisateurs inhibeurs pourra ne pas s'intégrer dans une seule unité MCSPDU. Le fournisseur E attend confirmation du premier sous-ensemble qu'il envoie en amont, avant d'envoyer de nouveau le jeton inhibé avec les identificateurs d'utilisateur restants. Cela assure une protection contre le refus du premier ensemble en raison de trop nombreux identificateurs de jeton en usage alors que les autres unités seraient acceptées ultérieurement, ce qui dénaturerait la base de données. Cette phase se termine lorsque tous les identificateurs de jeton ont soit été explicitement confirmés ou ont été purgés.

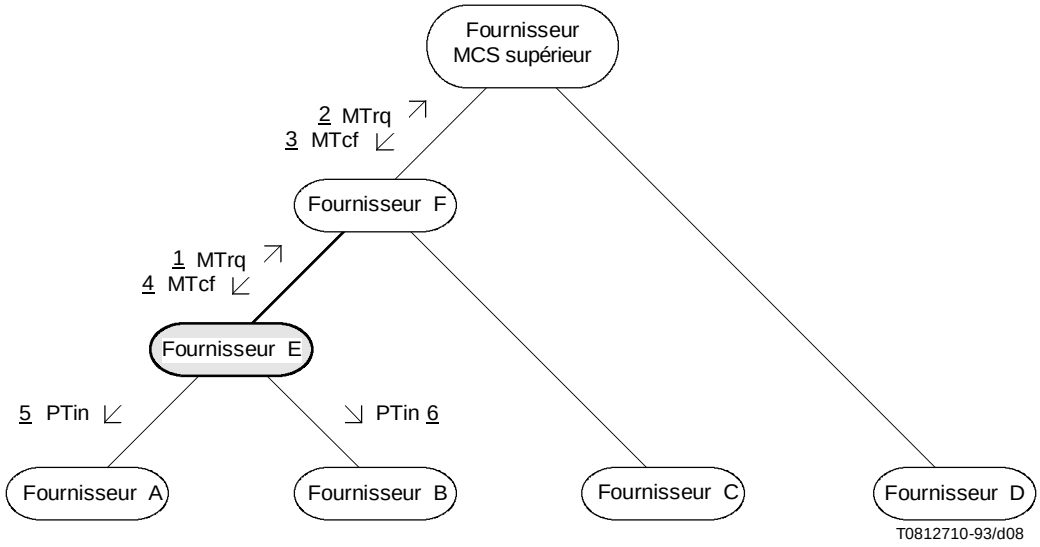



FIGURE 12-3/T.125

**Troisième phase de la fusion de domaines – Fusion des identificateurs de jeton**

La quatrième phase fait intervenir les mêmes unités MCSPDU que dans la deuxième phase, mais avec des identificateurs de canal différents. Les canaux d'identification d'utilisateur ayant été pris en compte, il ne reste que les identificateurs de canaux statiques, privés et attribués. Si les identificateurs d'utilisateur n'ont pas été fusionnés en premier, des portions d'unités **MCrq** ultérieures pourront être refusées parce que invalides et les identificateurs de canal affectés seront purgés sans nécessité. Dans le cas d'un canal privé, l'ensemble entier des utilisateurs admis par le gestionnaire de canaux pourra ne pas s'intégrer dans une seule unité MCSPDU. Le fournisseur E attend confirmation du premier sous-ensemble qu'il envoie en amont, avant d'envoyer de nouveau le canal privé avec les identificateurs d'utilisateur restants. Cela assure une protection contre le refus du premier ensemble en raison de trop nombreux identificateurs de canal en usage alors que les autres unités seraient acceptées ultérieurement, ce qui dénaturerait la base de données. Cette phase se termine, de même que la fusion de domaines, lorsque tous les identificateurs de canal restants ont soit été explicitement confirmés ou ont été purgés.

NOTE – La fusion des identificateurs de jeton prend leur possession exclusive plus intéressante dans la mesure où cela supprime les conflits entre flux de données. D'autre part, des données peuvent se perdre entre des domaines, lors de la confirmation des identificateurs de canal, avant que le conflit soit révélé dans le cadre d'une purge de jetons.

## 12.6 Déconnexion de domaine

Lorsqu'une connexion MCS ascendante est déconnectée, un fournisseur MCS doit éliminer son sous-arbre du domaine en détachant tous ses rattachements MCS directs et en déconnectant toutes ses autres connexions MCS. Le fournisseur affecté ne peut en général pas établir de domaine résiduel dans son propre sous-arbre, car il ne possède pas d'enregistrement d'unités MCSPDU de demande qui auraient été envoyées en amont, pour lesquelles il ne recevra jamais d'unité MCSPDU de confirmation correspondante.

Lorsqu'une connexion MCS descendante est déconnectée, un fournisseur MCS doit produire des unités MCSPDU de type **DUrq** pour tous les utilisateurs résidant dans cette portion de son sous-arbre, en donnant comme cause: *domaine déconnecté*.

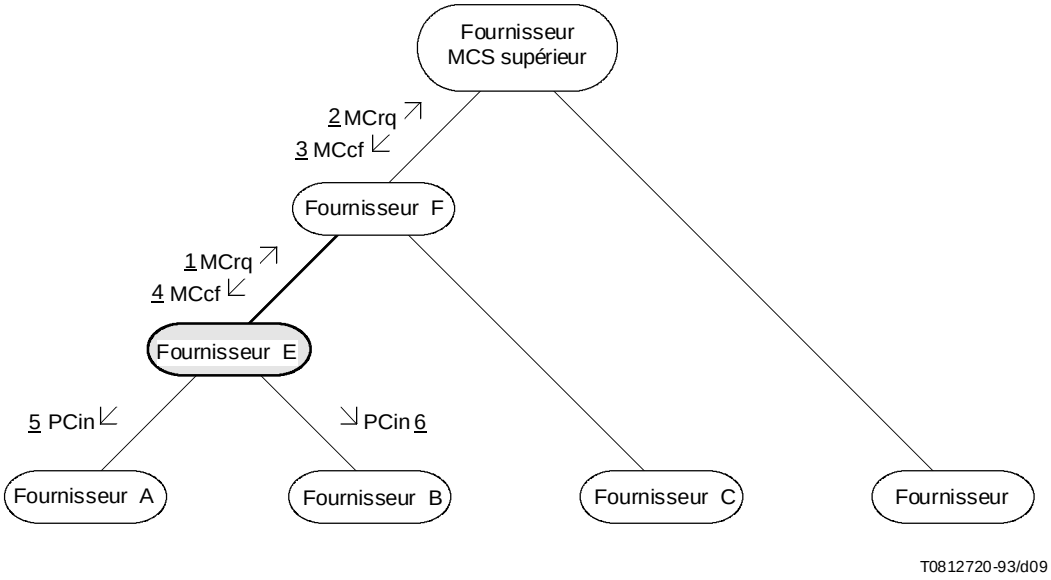



FIGURE 12-4/T.125

**Quatrième phase de la fusion de domaines – Fusion des identificateurs de canal restants**

## 12.7 Attribution des identificateurs de canal

Les identificateurs de canal se trouvant dans l'étendue de 1001 et plus sont attribués dynamiquement au niveau du fournisseur MCS supérieur pendant le traitement des primitives de demande MCS-ATTACH-USER, MCS-CHANNEL-JOIN de valeur zéro et MCS-CHANNEL-CONVENE. Il n'est pas prescrit que les valeurs attribuées correspondent à une quelconque configuration particulière. Il est en revanche souhaitable que les valeurs soient dispersées aléatoirement dans l'étendue autorisée. Cela augmentera la probabilité que deux domaines fonctionnant indépendamment l'un de l'autre pendant une période notable puissent être ultérieurement fusionnés sans conflits quant aux attributions de leurs identificateurs de canal respectifs. Cette précaution protégera également contre une remise en circuit trop précoce des identificateurs à l'intérieur d'un même domaine lorsqu'ils sont libérés d'un usage puis réattribués à un autre. Les applications faisant appel au service MCS devront avoir le temps de s'adapter à la disparition d'un identificateur de canal ou d'utilisateur avant qu'il revienne sous une autre forme «d'incarnation».

Dans les situations où la fusion cohérente de domaines actifs est requise, on peut éviter des conflits en sélectionnant des identificateurs de canaux à l'intérieur d'un sous-ensemble unique à chaque fournisseur. On peut créer de tels sous-ensembles en subdivisant en plusieurs bandes les identificateurs de canaux affectés dynamiquement de 1001 à 65535. En outre, à l'intérieur d'un de ces sous-ensembles, on peut affecter des identificateurs en séquence dans un sens, à partir du sommet ou de la racine. Les fournisseurs qui font appel à l'affectation par sous-ensembles doivent toujours respecter tous les aspects du présent protocole MCS, y compris la procédure de fusion de domaines. Ils doivent ainsi tenir compte des fournisseurs homologues qui n'ont peut-être pas de sous-ensembles particuliers.

NOTE – La manière dont un fournisseur choisit un sous-ensemble à partir duquel il effectuera des affectations relève d'une décision locale. Des conférences préorganisées peuvent être réunies par un système de gestion réticulé.

Au contraire des identificateurs de canal, les identificateurs de jeton ne sont pas attribués et la ligne de démarcation de valeur 1001 n'a pas de signification pour eux. Un identificateur de jeton libre, possédant une valeur donnée, est statiquement disponible à tout moment pour être saisi ou inhibé, sous la seule réserve d'une limite de domaine quant au nombre total de jetons utilisés en même temps.

### 12.8 Etat des jetons

L'état des jetons est défini formellement dans l'article 7. Il est signalé en tant que composant des unités MCSPDU de confirmation relatives aux jetons. Il sert à mettre à jour l'enregistrement concernant un identificateur de jeton dans la base de données des fournisseurs subordonnés. L'état des jetons n'est pas nécessairement signalé directement par l'intermédiaire d'une primitive de confirmation à l'utilisateur initiateur. Mais il peut être signalé indirectement au moyen d'une valeur de résultat.

Lorsque plus d'une seule valeur de statut de jeton décrit la relation entre un utilisateur donné et un identificateur de jeton spécifié, l'ordre de préférence doit être le suivant. La valeur d'état *autodestinataire* doit être signalée la première, si elle peut être insérée, afin de rappeler aux utilisateurs qu'ils doivent toujours répondre à une indication de cession de jeton MCS-TOKEN-GIVE. La valeur préférée ensuite est *autodonnant*, afin de rappeler aux utilisateurs qu'ils sont engagés dans une opération non terminée, puis *autosaisi* et *auto-inhibé*. Les valeurs d'état restantes viennent à la fin: elles rendent compte de l'état actuel du jeton en conséquence de son seul usage par d'autres correspondants.

Un fournisseur MCS se fonde sur la préférence spécifiée des valeurs d'état de jeton pour mettre à jour correctement ces états dans sa base de données.

## 13 Mise en œuvre de référence

Les Appendices II à VII ci-après décrivent une mise en œuvre de fournisseur MCS en langage de description et de spécification fonctionnelle (SDL) (*specification and description language*). Ils démontrent que l'on peut, au prix d'un effort raisonnable, réaliser le protocole spécifié. Les dispositions de cet exemple devraient épargner aux réalisateurs le travail de réinventer des relations logiques équivalentes et devraient accélérer l'introduction de systèmes compatibles.

Le langage SDL est une technique de description formelle plus puissante que les tables d'états conventionnelles. Il est mieux adapté à la complexité du service MCS. Il admet deux représentations équivalentes: la représentation textuelle et la représentation graphique. La mise en œuvre de référence est présentée sous la première forme, qui est plus concise et plus facile à traduire en langage de programmation conventionnel. Elle fait un large usage de types de données abstraits comme le générateur de mode ensembliste. On trouvera de plus amples renseignements sur le langage SDL dans les références suivantes:

- Recommandation Z.100 du CCITT (1988), *Langage de description et de spécification.*
- 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

L'Appendice II commence par une figure qui résume les relations codifiées dans les définitions de système et de bloc qui suivent. Les processus représentés sur cette figure – commande, domaine, extrémité et rattachement – sont définis individuellement dans les Appendices III à VI suivants. L'Appendice VII explique les hypothèses du modèle et illustre les principaux flux de signaux.

Ces appendices ont été revus par des experts techniques et on estime qu'ils constituent, sans pour autant avoir valeur normative, une application correcte du protocole de service MCS défini dans la présente spécification. En cas de divergence, les descriptions contenues dans le corps de la présente spécification font foi.

Le protocole du service MCS peut également être réalisé selon d'autres modèles, qui peuvent ne pas être fondés sur la mise en œuvre de référence.

La mise en œuvre de référence décrite dans les Appendices II à VII est paramétrisée de manière à développer un fournisseur MCS tout à fait générique. Les autres mises en œuvre, limitées à des cas particuliers, pourront être plus compactes et plus efficaces. Un cas particulièrement intéressant est celui d'un fournisseur MCS limité à une seule connexion MCS, condition appropriée à un nœud terminal. On pourrait spécialiser encore ce cas pour simplifier les opérations de fusion en purgeant délibérément tous les identificateurs employés dans un domaine inférieur. La sélection de cas spéciaux importants et leur mise en œuvre en langage SDL reste un sujet d'étude complémentaire.

# Remplacée par une version plus récente

## Appendice I

## Variantes de codage d'une unité MCSPDU

(Cet appendice ne fait pas partie intégrante de la présente Recommandation)

### I.1 Demande d'envoi de données

Les définitions applicables, extraites de l'article 7, sont les suivantes:

```
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))
```

Une valeur échantillon de ce type est:

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

### I.2 Règles de codage de base (BER) (*basic encoding rules*)

Les règles BER appliquent de manière récursive aux types de composant un schéma *identifier-length-contents* (identificateur-longueur-contenu) comme suit:

| 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 |

Octets d'en-tête acheminant exclusivement des données d'utilisateur: 18 à 24, selon l'étiquette de l'unité MCSPDU, l'identificateur de canal et la longueur de ces données d'utilisateur.

### I.3 Règles de codage condensé (PER) (*packed encoding rules*)

Les règles de codage PER ne peuvent être décodées que si l'on sait au préalable quel type ASN.1 est contenu dans le codage. Les étiquettes ne sont en effet pas acheminées. C'est là une raison pour définir un type combiné d'unités MCSPDU de domaine. Les étiquettes applicatives des unités MCSPDU de domaine progressent en séquence à partir de zéro. Cela est pratique parce que les règles PER codent les valeurs sous forme de décalage par rapport à la base de leur étendue. Selon ce principe, la valeur 25 identifie une unité de demande **SDrq** conformément aux règles BER comme aux règles PER:

| | |
|---|---|
| **CHOICE** | -- *6 bits + justification* |
| **64** | |
| **INTEGER (1001..65535)** | -- *décalage 1001* |
| **02 BC** | |
| **INTEGER (0..65535)** | -- *décalage zéro* |
| **00 05** | |
| **ENUMERATED + BIT STRING (SIZE (2))** | -- *2 bits + 2 bits + justification* |
| **60** | |
| **CTET STRING** | -- *longueur + contenu* |
| **03 4D 43 53** | |

Octets d'en-tête acheminant exclusivement des données d'utilisateur: 7 et 8, selon la longueur de ces données.

Les règles PER codent en un seul octet une chaîne d'octets non contrainte d'une longueur inférieure ou égale à 127 et une chaîne en deux octets d'une longueur inférieure ou égale à 16 384.

# Remplacée par une version plus récente

## Appendice II

## Décomposition en SDL d'un fournisseur MCS

(Cet appendice ne fait pas partie intégrante de la présente Recommandation.)

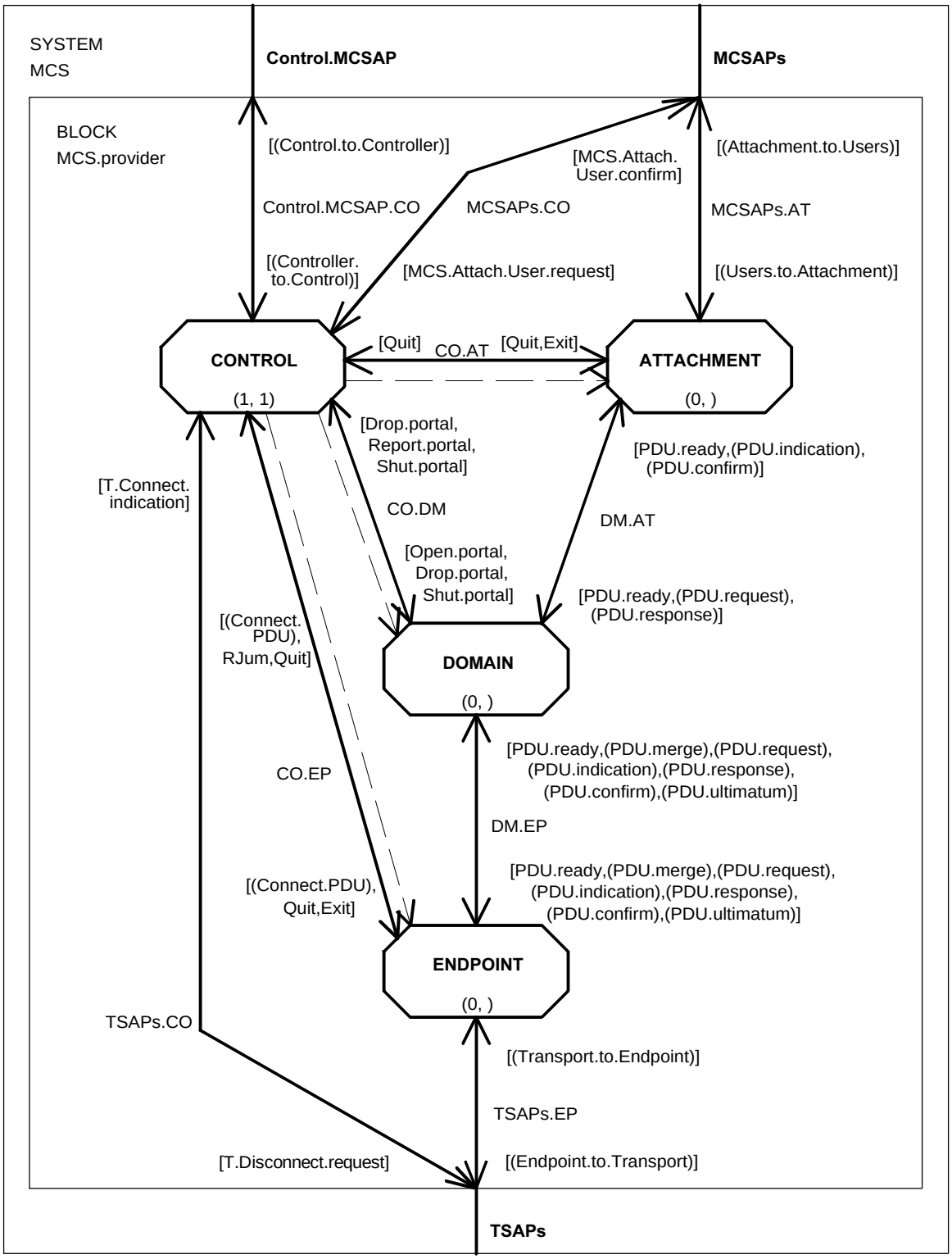



FIGURE II.1/T.125

**Décomposition en SDL d'un fournisseur MCS**

```
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,
            PIdByPri
        );
SIGNAL  Report.portal
        (
            PortalId,
            Diagnostic
        );
SIGNAL  Drop.p7ortal
        (
                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  Tlrq     (PDUStruct);     /* token inhibit request */
SIGNAL  Tlcf     (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;
```

## Appendice III

### Spécification en SDL du processus de contrôle

(Cet appendice ne fait pas partie intégrante de la présente Recommandation)

```
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;
```

## Appendice IV

## Spécification en SDL du processus de domaine

(Cet appendice ne fait pas partie intégrante de la présente Recommandation)

```
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 := Userld(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, Tlrq, 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, Tlcf, 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, Tlcf, 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;
```

# Remplacée par une version plus récente

```
            (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;
```

## Appendice V
## Spécification en SDL du processus d'extrémité

(Cet appendice ne fait pas partie intégrante de la présente Recommandation)

```
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   TIrq(pdu);  TASK pdu!kind  :=  TIrq;  JOIN   2f;
        INPUT   TIcf(pdu);  TASK pdu!kind  :=  TIcf;  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;
```

## Appendice VI

## Spécification en SDL du processus de rattachement

(Cet appendice ne fait pas partie intégrante de la présente Recommandation)

```
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(mald, 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(mald, pdu!channelId,
                         pdu!dataPriority, pdu!initiator, pdu!segmentation, pdu!userData);
        (USin):  OUTPUT  MCS.Uniform.Send.Data.indication(mald, 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(mald, 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(mald);
        TASK     pdu!initiator  :=  user;
        OUTPUT   CCrq(pdu) TO domain;
        JOIN 2b;

STATE ready;
        INPUT    MCS.Channel.Disband.request(mald, 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(mald, c, uSet);
        TASK     kind  :=  CArq;
        JOIN 3f;

STATE ready;
        INPUT    MCS.Channel.Expel.request(mald, 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(mald, c, dp, segmentation, userData);
        TASK     kind  :=  SDrq;
        JOIN 4f;

STATE ready;
        INPUT    MCS.Uniform.Send.Data.request(mald, c, dp, segmentation, userData);
        TASK     kind  :=  USrq;
          4f :
        DECISION dp >= 4 or dp in uPending;
        (True):  OUTPUT  MCS.Detach.User.indication(mald, 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(mald, 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(mald, pdu!tokenId, pdu!result);
        JOIN 8b;

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

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

STATE busy, ready;
        INPUT    TVcf(pdu);
        CALL     Track_token(pdu!tokenId, pdu!tokenStatus);
        OUTPUT   MCS.Token.Give.confirm(mald, 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(mald, 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(mald, 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;
```

# Appendice VII

## Caractéristiques de la mise en œuvre de référence

(Cet appendice ne fait pas partie intégrante de la présente Recommandation)

### VII.1 Décomposition en langage SDL

La Figure II.1 décrit un fournisseur MCS dans le cadre d'un système avec trois canaux qui le relient à son environnement: le canal Control.MCSAP qui le relie à une application de contrôleur unique, le canal des points MCSAP vers zéro ou plus utilisateurs rattachés et le canal des points TSAP vers zéro ou plus fournisseurs du service de transport. Lors d'une décomposition de l'entité fournisseur, ces canaux externes se raccordent à des itinéraires de signaux en direction et en provenance de processus composants.

Chaque élément représenté dans un cartouche octogonal sur la Figure II.1 représente un type de processus et indique des limites (minimum, maximum) quant au nombre de ses instances. Un seul processus commande existe en permanence. Les traits interrompus montrent que ce processus crée des instances des trois autres processus. Les processus rattachement, domaine et extrémité n'existent pas initialement, mais leur nombre possible est illimité.

Les itinéraires de signaux à destination et en provenance de processus composants acheminent des signaux en langage SDL. L'ensemble des signaux transmis dans un sens donné est indiqué près de la tête de flèche correspondante. Les identificateurs entre parenthèses désignent des listes de signaux associés. Le développement de ces listes aurait été trop volumineux pour pouvoir être inclus dans la figure. Les détails sont repris dans le texte de l'Appendice II.

## VII.2 Définitions de service

Les signaux passant par les canaux externes représentent des utilisations du service MCS et des services de transport dont la définition abstraite est spécifiée dans la Recommandation T.122 de l'UIT-T et dans la Recommandation X.214 du CCITT. La modélisation de ces signaux en langage SDL nécessite la formulation d'autres hypothèses quant aux détails des interactions. Deux aspects méritent une mention spéciale: l'utilisation d'identificateurs d'extrémité pour spécifier le contexte d'une primitive de service et l'utilisation de signaux de disponibilité à la commande de flux afin de solliciter le transfert de données.

Les trois identificateurs en question sont les suivants: MCSConnectionId, MCSAttachmentId et TCEndpointId. Ces paramètres sont implicites (non décrits) dans les spécifications de définition de service. Le modèle examiné ici part du principe que ces paramètres sont attribués par le fournisseur responsable. Par exemple, l'identificateur TCEndpointId apparaît dans les primitives T.Connect.indication et T.Connect.confirm. De façon que l'utilisateur d'un service soit en mesure d'associer une confirmation à une demande antérieure. Cet utilisateur peut spécifier qu'une étiquette arbitraire lui soit renvoyée en écho.

La commande de flux est modélisée ici par un mécanisme de fenêtrage exclusif. Un signal de disponibilité doit toujours être envoyé dans un sens donné avant qu'un signal de transfert de données puisse être envoyé dans le sens inverse. Avant le transfert suivant, un autre signal de disponibilité doit être reçu. Dans une mise en œuvre réelle, l'indication de disponibilité peut se traduire par le déplacement d'un pointeur ou par la progression d'un compteur. L'indication de disponibilité sous forme de signal n'implique pas que celui-ci doive être aussi complexe que pour indiquer un transfert de données. Il existe trois signaux de disponibilité: T.ready, PDU.ready et MCS.ready. Ils permettent, respectivement, le transfert d'une seule unité TSDU, d'une seule unité MCSPDU de domaine et d'une seule unité de données du service MCS. La commande de flux est applicable indépendamment des deux sens d'un itinéraire de signalisation.

La qualité du service (QOS) de transport sur une connexion de transport est modélisée ici par des paramètres de débit utile, de temps de transit et de priorité des données. Il s'agit d'un amalgame de normes qui ne forment pas un ensemble homogène. En pratique, il y aura certainement des différences de détail. La spécification de la QOS représente une tâche qui doit être prise en charge par les applications du contrôleur.

## VII.3 Portails ouvrant sur un domaine

Le processus de domaine est au centre de cette décomposition d'une entité de fournisseur MCS. Différentes instances représentent différents domaines couverts par le même fournisseur. Un processus de domaine est créé avec un ensemble de paramètres qui restent immuables pendant sa durée de vie. Ces paramètres de domaine sont fournis par le processus de commande. Celui-ci détermine les sélecteurs de domaine qui sont valides et la façon dont les processus de domaine correspondants doivent être configurés. S'il y a possibilité de négociation des paramètres au cours de la réalisation d'une primitive MCS.Connect.Provider, le processus de commande supervisera cette interaction.

Pour que le processus de domaine reste aussi simple que possible, les rattachements MCS et les connexions MCS avec lesquels il a une interface sont transposés métaphoriquement en portails possédant un certain nombre de caractéristiques communes. Un portail est soit un simple processus de rattachement ou un ensemble de processus d'extrémité, un pour chaque connexion de transport constituant une connexion MCS. Le processus de commande attribue des identificateurs de portail puis ouvre et ferme leurs associations avec un processus de domaine. Il occulte les détails de construction et de démolition des processus ainsi que l'échange d'unités MCSPDU de connexion.

Vu à partir du processus de domaine, un portail ressemble à un ensemble d'itinéraires de signaux, un pour chaque priorité de données mise en œuvre. Chaque itinéraire achemine des unités MCSPDU sous réserve de la commande de flux par le signal PDU.ready. S'il n'est pas possible d'éviter d'opérer des distinctions lors du traitement, on invoquera celles-ci en classant un portail dans l'une des trois sortes suivantes: attaché, ascendant ou descendant.

Les Figures VII.1 à VII.8 montrent les flux de signaux pour des opérations types. Elles supposent qu'un seul utilisateur est rattaché localement à un domaine avec une seule connexion MCS vers un autre fournisseur. Les processus extrémité 1 et extrémité 2 découlent de l'hypothèse que deux priorités de données sont mises en œuvre dans ce domaine.

La signalisation de fermeture d'un portail est assez longue. Elle protège contre l'éventualité qu'un des processus en cause n'envoie un signal à un autre processus qui s'est déjà arrêté, ce qui provoquerait une erreur d'exécution en SDL. Les processus de rattachement et d'extrémité donnent au dernier signal qu'ils envoient la forme Quit et ils s'arrêtent dès qu'ils reçoivent un signal de la forme Exit. Le processus de domaine n'envoie plus d'autres signaux à un portail une fois qu'il a envoyé en réponse, au processus de commande, un signal Shut.portal. Il s'arrête après avoir indiqué ainsi que le dernier portail est fermé.

Parmi les signaux divers qui restent, le signal Drop.portal est bilatéral et correspond à l'unité d'ultimatum **DPum**. Le signal Report.portal indique au processus de commande qu'un diagnostic a été envoyé en direction d'un autre fournisseur MCS. Un ultimatum **RJum** issu du processus d'extrémité est envoyé par le processus de domaine pour parvenir à la connexion de transport initiale. Une unité **RJum** reçue de l'extérieur est envoyée directement par le processus d'extrémité au processus de commande.

Il est à noter que les primitives MCS.Attach.User.request et T.Connect.indication sont dirigées vers le processus de commande, car ce sont des stimuli pour la création des portails correspondants. Le processus de commande peut s'effacer avec une primitive MCS.Attach.User.confirm défavorable ou avec une primitive T.Disconnect.request.

Le présent modèle ne limite pas le nombre de rattachements MCS ou de connexions MCS par domaine, sauf pour ce qui est d'une limite locale (maxPortalIds) quant au nombre de portails répartis dans tous les domaines couverts par le fournisseur. De nombreux aspects de configuration de domaine doivent être gérés localement, dans l'attente d'une normalisation complémentaire.

### VII.4 Alignement des unités MCSPDU

Par souci de clarté, il faut que les unités MCSPDU définies à l'article 7 soient représentées individuellement par des signaux SDL. Mais la simplicité du processus de domaine dépend d'un traitement plus homogène. La solution actuelle consiste à définir un type de données PDUStruct dont les composants peuvent être choisis de façon à s'adapter à toute unité MCSPDU du domaine. Les unités MCSPDU de connexion, qui sont traitées à l'extérieur du processus de domaine, continuent à avoir chacune leur propre structure.

Un élément clé du type de données PDUStruct est sa sorte, qui identifie l'unité MCSPDU de domaine prévue. Sur la base de la sorte, un sous-ensemble d'autres champs peut devenir applicable, comme ceux qui sont énumérés dans la notation ASN.1 comme étant des composants de l'unité MCSPDU. Les types de données définis en langage SDL correspondent autant que possible aux types de données définis en notation ASN.1 à l'article 7. Il convient de bien préciser l'idée qu'un type PDUStruct est la représentation interne et décodée d'une unité MCSPDU de domaine.

Trois unités MCSPDU comportent des composants facultatifs. Il s'agit des unités de confirmation **AUcf , CJcf** et **CCcf**. En langage SDL, on indique l'absence de ces composants en codant zéro pour le champ correspondant. Il s'agit d'une valeur interdite pour un identificateur de canal statique ou dynamique.

Les signaux envoyés entre les processus de rattachement, de domaine et d'extrémité prennent comme paramètre unique un type PDUStruct, sauf le signal PDU.ready. Dans une mise en œuvre réelle, ce type de communication peut se réduire à déplacer un pointeur de mémoire tampon.

### VII.5 Méthode d'utilisation du langage SDL

La mise en œuvre de référence fait appel à la représentation textuelle du langage SDL, en tant que langage de programmation dont la puissance est augmentée par la possibilité de définir des types de données abstraits. Dans les processus de commande et de domaine, les objets de gestion sont trop complexes et trop nombreux pour tirer parti du type de machine à états finis que le langage SDL permet d'exploiter. Ces processus restent sous la forme d'états uniques traitant des signaux d'entrée. Les processus de rattachement et d'extrémité, dont la portée est plus réduite, peuvent tirer plus d'avantages d'un ensemble d'états.

L'Appendice II contient une définition du générateur d'ensembles SetOf, dont le champ d'application recouvre toute l'entité de fournisseur. Cet item s'applique à tout autre type défini, comme les identificateurs de canal, et formalise le concept de sous-ensembles de valeurs de ce type. L'item SetOf élargit le générateur de mode ensembliste en ajoutant un nouvel opérateur qui choisit un élément arbitraire dans un sous-ensemble non vide. La signification de cet opérateur de prélèvement (Pick) sera définie au moyen d'axiomes simples.

Les ensembles sont utilisés couramment dans toute cette mise en œuvre. Une partie de la base de données, par exemple, servira à enregistrer les sous-ensembles d'identificateurs de canal qui sont en cours d'utilisation et le sous-ensemble des identificateurs de portail qui ont adhéré à un canal donné.

En langage SDL, on déploie des tableaux dans toute l'étendue du type d'indexation de données. Les structures de canal, par exemple, existent pour chaque identificateur de canal compris entre 0 et 65535. Une mise en œuvre réelle doit gérer des tableaux beaucoup moins denses. A cette fin, des ensembles distincts d'identificateurs sont conservés afin de savoir quels canaux et quelles autres ressources sont en cours d'utilisation à un moment donné. Un principe de base est qu'il n'est pas nécessaire de conserver en base de données des tableaux de valeurs pour les identificateurs non explicitement repérés comme étant en usage.

De nombreuses itérations suivent un schéma familier: créer un ensemble d'éléments candidats à un objectif donné, puis prélever et supprimer un seul membre de cet ensemble à la fois, jusqu'à ce qu'il n'y en ait plus. De telles itérations sont formées de décisions et d'adhésions. Des commentaires indiquent quelle est la structure de commande de haut niveau qui est visée. Les étiquettes sont numérotées en séquence dans une même procédure, avec une seule lettre accolée pour préciser si les branches aboutissant à cette étiquette sont ascendantes ou descendantes.

Une table des matières, montrant les séquences d'appel, est établie dans chaque processus avant la première définition de procédure. Les procédures SDL peuvent rendre leurs résultats par l'intermédiaire de paramètres formels dont on déclare s'ils sont entrants ou sortants.

Quelques constantes apparaissent en ordre dispersé dans l'ensemble du code, par exemple le nombre maximal de priorités de données (4) et la charnière entre identificateurs de canal statique et de canal dynamique (1001). Il ne s'agit pas de paramètres dont la valeur est redéfinissable.

Dans le service de couche MCS, un point important est le fait que les identificateurs d'utilisateur forment un sous-ensemble des identificateurs de canal. Mais les contextes d'utilisation de chacun de ces ensembles sont différents, de telle sorte qu'il convient de distinguer deux types distincts. L'Appendice II formalise une paire d'opérateurs pour les communications entre ces deux types d'identificateur.

Les processus des Appendices III à VI coopèrent dans le cadre général qui est spécifié dans l'Appendice II. Les signaux sont liés aux itinéraires spécifiés et aucun écart de comportement n'est admis de part et d'autre des interfaces internes. Le codage de protection est axé sur les applications de contrôleur et d'utilisateur résidant dans l'environnement à l'extérieur du fournisseur ainsi que sur les unités MCSPDU reçues de fournisseurs homologues externes.

La mise en œuvre de référence permet d'ordonnancer des signaux d'entrée selon une séquence arbitraire, à condition que plusieurs d'entre eux soient en instance. Aucune hypothèse n'y est faite quant à la priorité relative des processus. Mais elle dépend en réalité des transitions de chaque unité d'entrée pour leur exécution jusqu'à leur achèvement, sans préemption.

### VII.6 Remarques sur le processus de domaine

On fait un usage important du générateur de files d'attente en plus du générateur d'ensembles. Les tampons sont mis en file d'attente dans leur ordre de transmission par les utilisateurs, en direction d'un portail de sortie. Les identificateurs de portail sont mis en file d'attente dans l'ordre des unités MCSPDU de demande **MCrq, MTrq** et **AUrq** n'ayant pas encore fait l'objet d'une réponse.

Afin d'éviter de recopier des données d'utilisateur, les unités MCSPDU sont prises en charge par des mémoires tampons qui peuvent alimenter plusieurs ports de sortie. Le nombre de tampons mis à la disposition d'un processus de domaine est modélisé sous la forme d'un paramètre externe de valeur fixe dans ce domaine. On peut le changer facilement pour modifier la configuration, d'un domaine à un autre. L'implémentation se dégrade progressivement, par l'intermédiaire d'une commande de flux globale, si le nombre de tampons disponibles est trop bas. Elle attribue les tampons, au fur et à mesure qu'ils se libèrent, au flux de données ayant la priorité la plus élevée.

Lorsqu'une unité MCSPDU se présente à une entrée, le processus de domaine a besoin de connaître son portail d'origine et sa priorité de données. La procédure Identify_sender effectue une recherche exhaustive des portails ouverts, sur la base des identificateurs de processus signalés par les unités SDL. Les informations recherchées seront ensuite occultées. Dans une mise en œuvre réelle, cette modélisation ne représente pas un point critique.

Le noyau du processus de domaine est constitué par la procédure Process_PDU, qui appelle en séquence les unités Validate_input, Top_provider et Apply_PDU. Chacune d'elles est une vaste sélection de cas fondée sur la sorte d'unité PDU en cause. Les détails de chacune de ces sortes sont en général évidents. Une bonne méthode pour saisir ces procédures consiste à opérer des coupes transversales dans leur déroulement en suivant une à une des unités MCSPDU particulièrement importantes, comme **SDrq**.

Le codage d'unités MCSPDU, au moyen des règles BER ou PER selon la version du protocole, est un détail qui est principalement pris en charge par le processus d'extrémité. Tout ce qu'un processus de domaine a besoin de savoir est la longueur qu'il peut donner aux unités MCSPDU de contenu variable, comme les demandes **DUrq** avec leurs multiples identificateurs d'utilisateur. De telles considérations sont exprimées au moyen de déclarations décisionnelles, en style non formel, avec des constantes de cas le moins favorable pour le codage de base BER. En pratique, on peut précalculer certaines limites de contenu variable, en termes accessibles par le processus de domaine, comme le nombre maximal d'identificateurs. D'autres limites peuvent exiger une connaissance plus précise du codage.

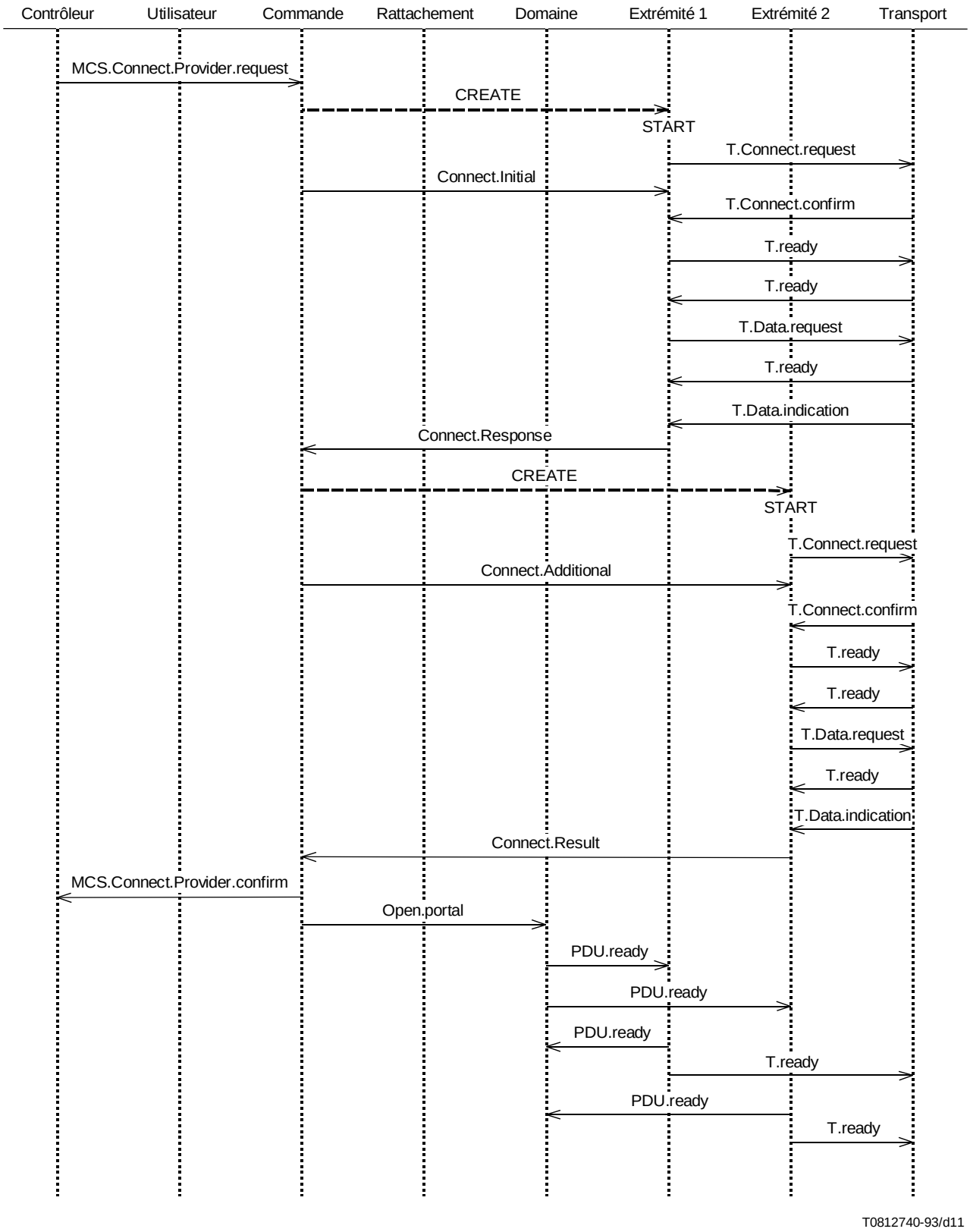



FIGURE VII.1/T.125

**Fournisseur MCS appelant**

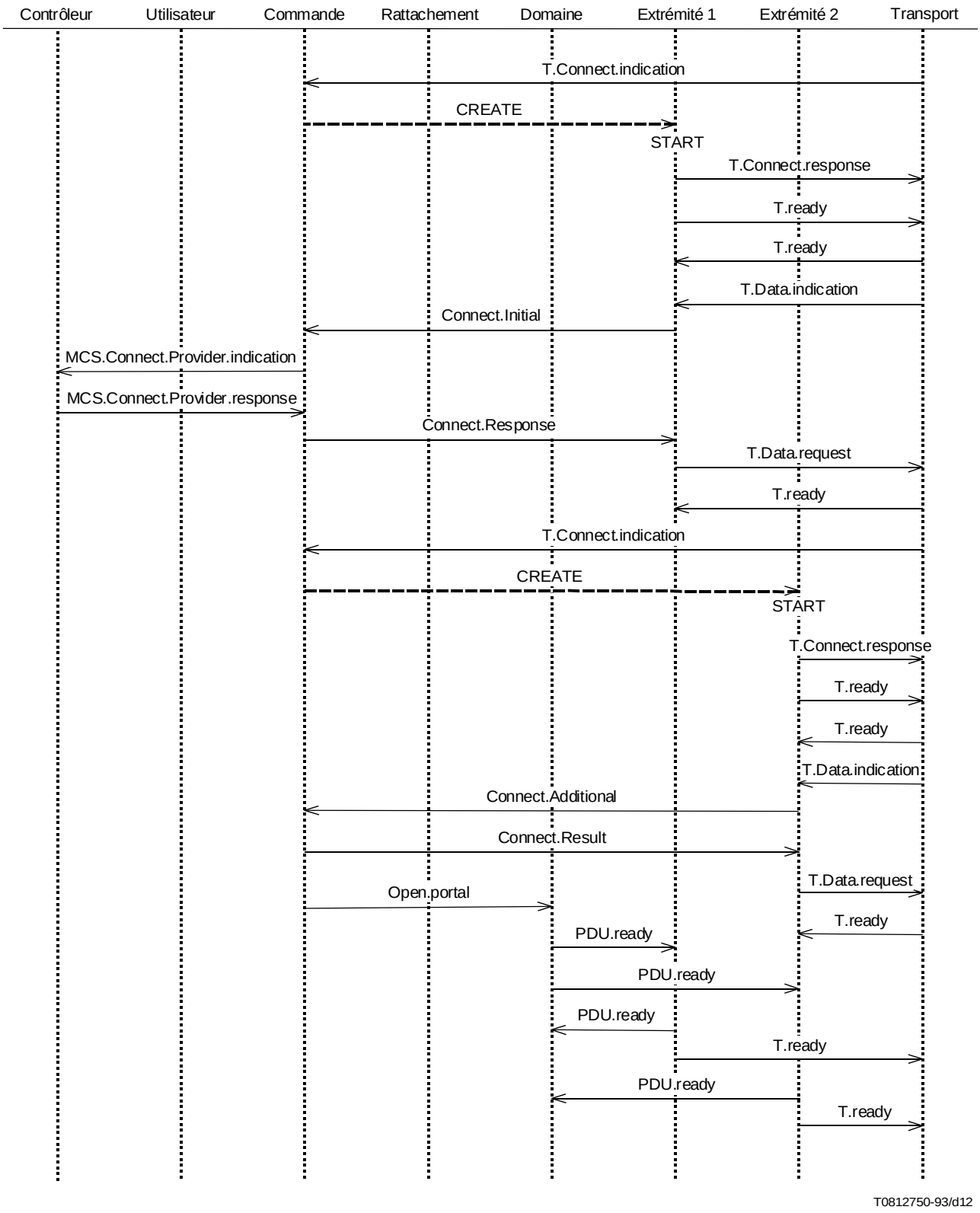



FIGURE VII.2/T.125

**Fournisseur MCS appelé**

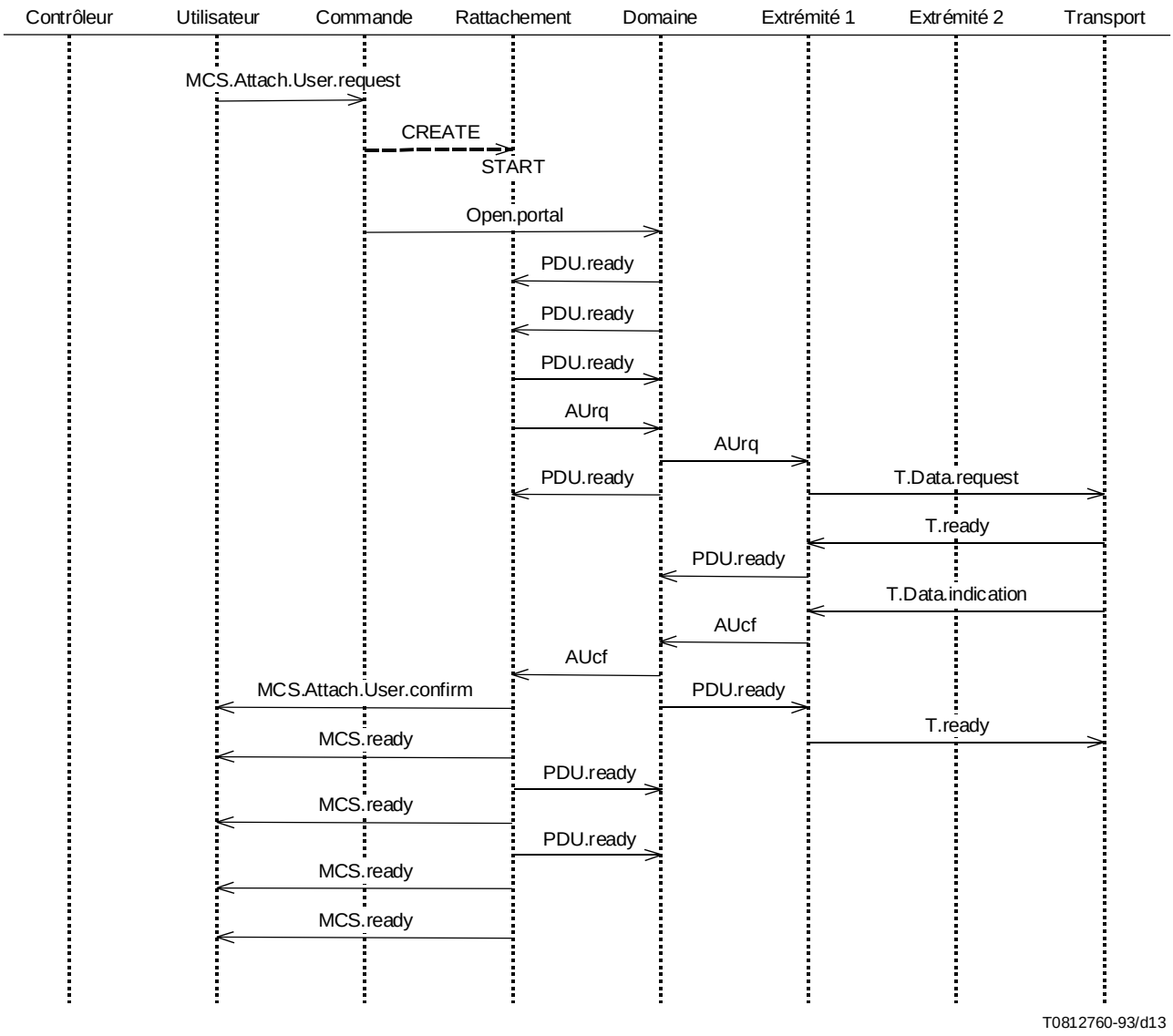



FIGURE VII.3/T.125

**Rattachement d'utilisateur**

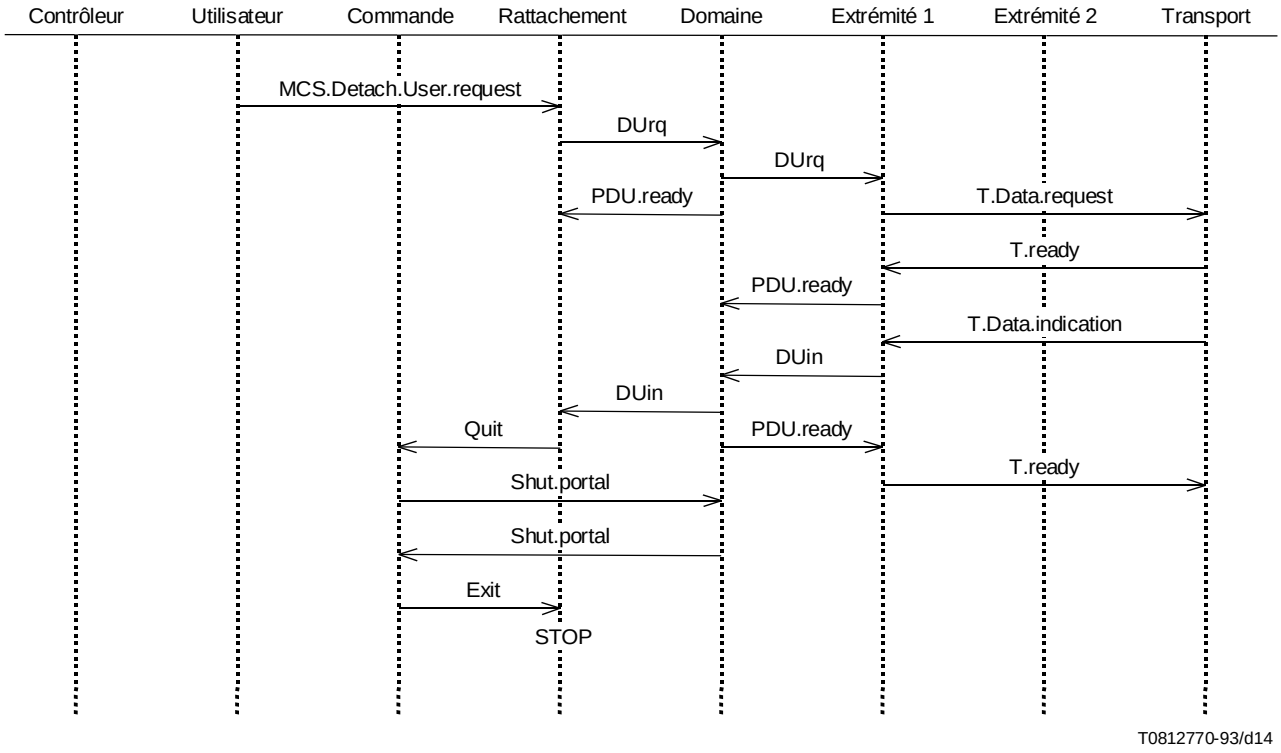



FIGURE VII.4/T.125

**Détachement demandé par l'utilisateur**

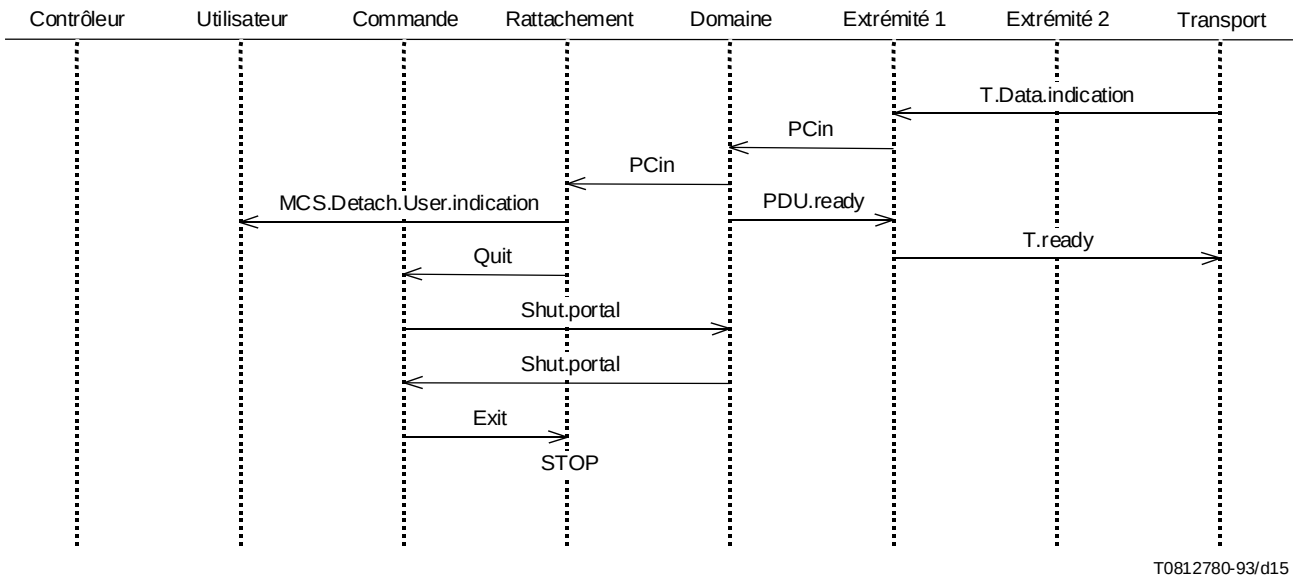



FIGURE VII.5/T.125

**Détachement déclenché par le fournisseur**

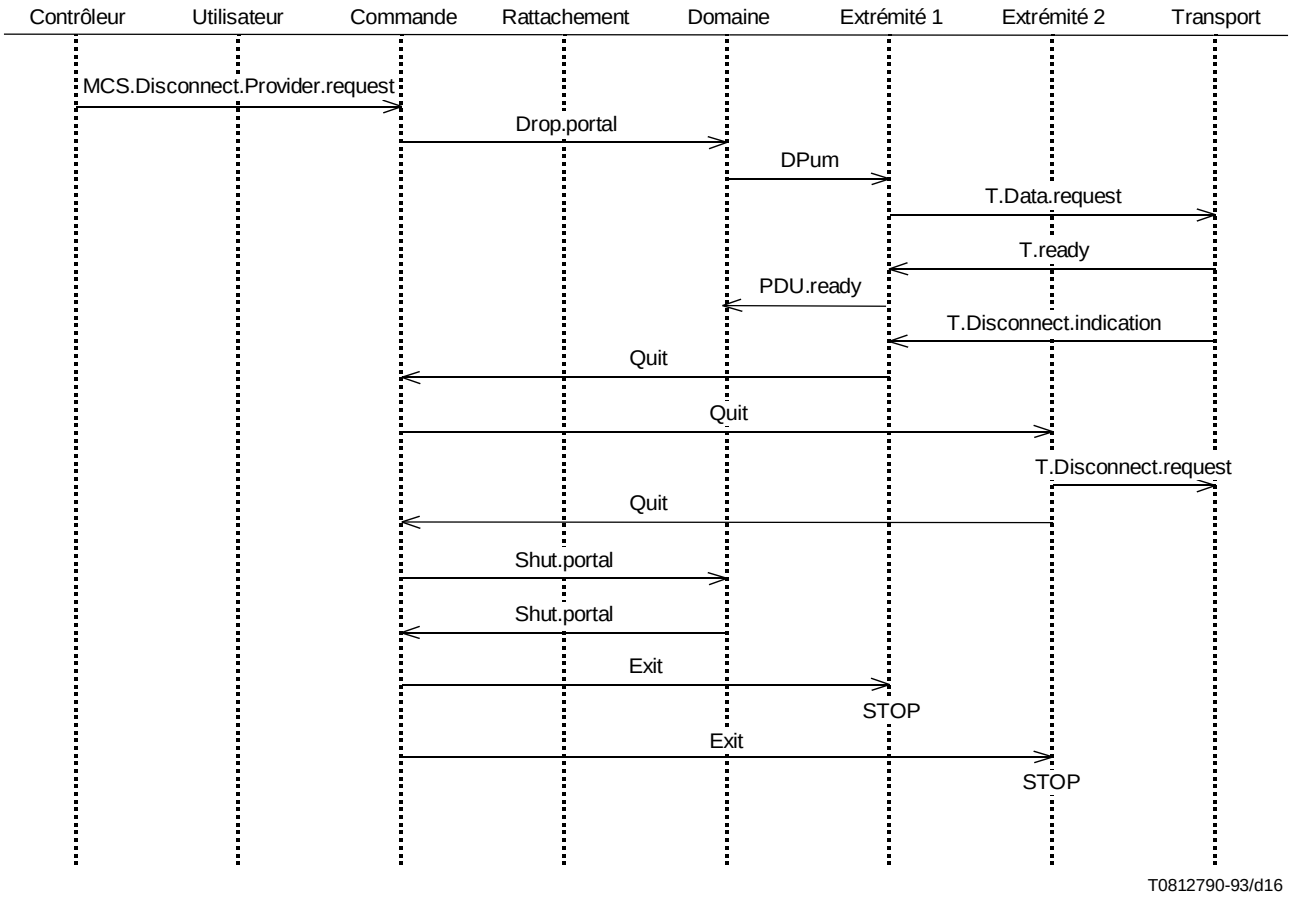



FIGURE VII.6/T.125

**Déconnexion demandée par le contrôleur local**

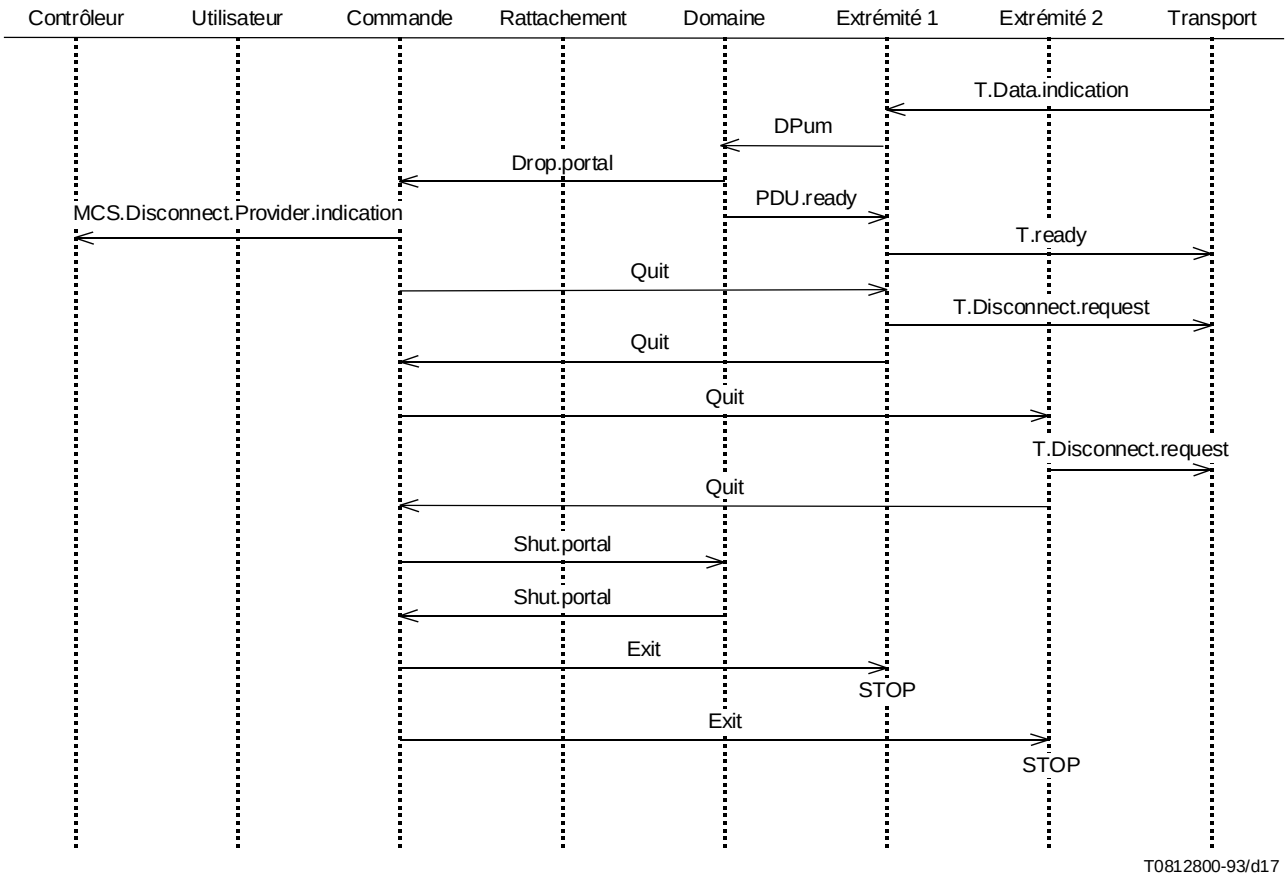



FIGURE VII.7/T.125

**Déconnexion demandée par le contrôleur distant**

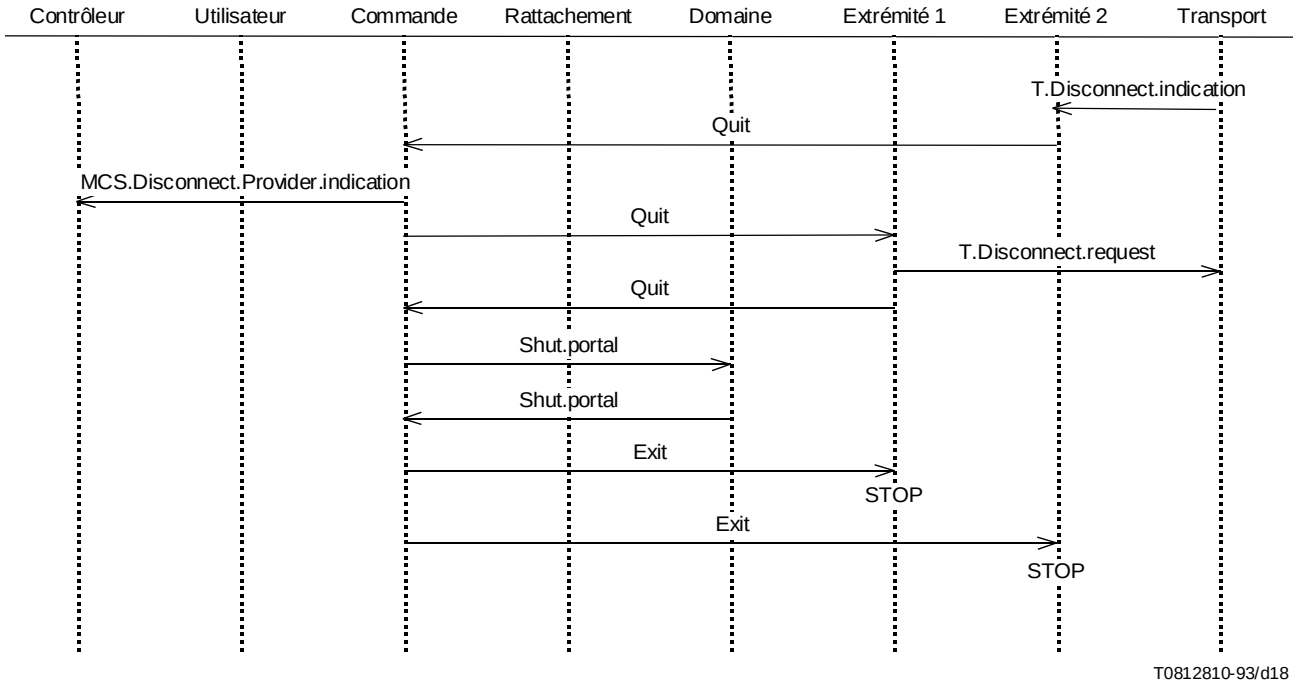



FIGURE VII.8/T.125

**Déconnexion déclenchée par le fournisseur**