UNION INTERNATIONALE DES TÉLÉCOMMUNICATIONS

UIT-T X.780

(01/2001)

SECTEUR DE LA NORMALISATION
DES TÉLÉCOMMUNICATIONS
DE L'UIT

SÉRIE X: RÉSEAUX DE DONNÉES ET
COMMUNICATION ENTRE SYSTÈMES OUVERTS

Gestion OSI – Fonctions de gestion et fonctions ODMA

Directives concernant le RGT pour la définition d'objets gérés CORBA

Recommandation UIT-T X.780

(Antérieurement Recommandation du CCITT)

RECOMMANDATIONS UIT-T DE LA SÉRIE X

RÉSEAUX DE DONNÉES ET COMMUNICATION ENTRE SYSTÈMES OUVERTS

RÉSEAUX PUBLICS DE DONNÉES	
Services et fonctionnalités	X.1–X.19
Interfaces	X.20–X.49
Transmission, signalisation et commutation	X.50–X.89
Aspects réseau	X.90–X.149
Maintenance	X.150–X.179
Dispositions administratives	X.180–X.199
INTERCONNEXION DES SYSTÈMES OUVERTS	
Modèle et notation	X.200–X.209
Définitions des services	X.210–X.219
Spécifications des protocoles en mode connexion	X.220–X.229
Spécifications des protocoles en mode sans connexion	X.230–X.239
Formulaires PICS	X.240–X.259
Identification des protocoles	X.260–X.269
Protocoles de sécurité	X.270–X.279
Objets gérés des couches	X.280–X.289
Tests de conformité	X.290–X.299
INTERFONCTIONNEMENT DES RÉSEAUX	
Généralités	X.300–X.349
Systèmes de transmission de données par satellite	X.350–X.369
Réseaux à protocole Internet	X.370–X.399
SYSTÈMES DE MESSAGERIE	X.400–X.499
ANNUAIRE	X.500–X.599
RÉSEAUTAGE OSI ET ASPECTS SYSTÈMES	
Réseautage	X.600–X.629
Efficacité	X.630–X.639
Qualité de service	X.640–X.649
Dénomination, adressage et enregistrement	X.650–X.679
Notation de syntaxe abstraite numéro un (ASN.1)	X.680–X.699
GESTION OSI	
Cadre général et architecture de la gestion-systèmes	X.700–X.709
Service et protocole de communication de gestion	X.710–X.719
Structure de l'information de gestion	X.720–X.729
Fonctions de gestion et fonctions ODMA	**X.730–X.799**
SÉCURITÉ	X.800–X.849
APPLICATIONS OSI	
Engagement, concomitance et rétablissement	X.850–X.859
Traitement transactionnel	X.860–X.879
Opérations distantes	X.880–X.899
TRAITEMENT RÉPARTI OUVERT	X.900–X.999

Pour plus de détails, voir la Liste des Recommandations de l'UIT-T.

Recommandation UIT-T X.780

Directives concernant le RGT pour la définition d'objets gérés CORBA

Résumé

La présente Recommandation spécifie des directives pour la définition d'interfaces fondées sur CORBA avec des objets logiciels représentant des ressources gérables dans un RGT. Elle contient des directives pour la modélisation d'informations, des règles pour la traduction de modèles GDMO et des conventions sur le style IDL. Elle contient aussi un module IDL définissant des types de données, des superclasses et des notifications à utiliser dans les spécifications de modèle d'information fondé sur la technique CORBA.

Source

La Recommandation X.780 de l'UIT-T, élaborée par la Commission d'études 4 (2001-2004) de l'UIT-T, a été approuvée le 19 janvier 2001 selon la procédure définie dans la Résolution 1 de l'AMNT.

Mots clés

Architecture de courtier commun de requête d'objets (CORBA), langage de définition d'interface (IDL), directives pour la définition des objets gérés (GDMO), traitement réparti, interfaces du RGT, objets gérés, notation de syntaxe abstraite numéro un (ASN.1).

AVANT-PROPOS

L'UIT (Union internationale des télécommunications) est une institution spécialisée des Nations Unies dans le domaine des télécommunications. L'UIT-T (Secteur de la normalisation des télécommunications) est un organe permanent de l'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.

L'Assemblée mondiale de normalisation des télécommunications (AMNT), qui se réunit tous les quatre ans, détermine les thèmes d'étude 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 1 de l'AMNT.

Dans certains secteurs des technologies de l'information qui correspondent à la sphère de compétence de l'UIT-T, les normes nécessaires se préparent en collaboration avec l'ISO et la CEI.

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.

DROITS DE PROPRIÉTÉ INTELLECTUELLE

L'UIT attire l'attention sur la possibilité que l'application ou la mise en œuvre de la présente Recommandation puisse donner lieu à l'utilisation d'un droit de propriété intellectuelle. L'UIT ne prend pas position en ce qui concerne l'existence, la validité ou l'applicabilité des droits de propriété intellectuelle, qu'ils soient revendiqués par un Membre de l'UIT ou par une tierce partie étrangère à la procédure d'élaboration des Recommandations.

A la date d'approbation de la présente Recommandation, l'UIT n'avait pas été avisée de l'existence d'une propriété intellectuelle protégée par des brevets à acquérir pour mettre en œuvre la présente Recommandation. Toutefois, comme il ne s'agit peut-être pas de renseignements les plus récents, il est vivement recommandé aux responsables de la mise en œuvre de consulter la base de données des brevets du TSB.

TABLE DES MATIÈRES

Recommandation UIT-T X.780

Directives concernant le RGT pour la définition d'objets gérés CORBA

1 Domaine d'application

L'architecture du RGT définie dans l'UIT-T M.3010 utilise des concepts du traitement réparti ainsi que plusieurs protocoles de gestion. Les spécifications initiales des interfaces intra et interréseau RGT ont été élaborées sur la base de la notation GDMO (directives pour la définition des objets gérés) à partir de la gestion des systèmes OSI avec comme protocole le protocole commun d'informations de gestion (CMIP, *common management information protocol*). L'interface interRGT (X) comprend à la fois CMIP et CORBA GIOP/IIOP comme choix possibles au niveau de la couche Application.

On envisage d'utiliser CORBA, technique de traitement réparti, dans l'architecture de communication du RGT, essentiellement en raison de son acceptation par l'industrie des technologies de l'information. Cette acceptation devrait permettre de renforcer la disponibilité d'interfaces fondées sur CORBA grâce à de meilleurs outils de développement et à une compétence étendue dans le développement d'interfaces fondées sur CORBA. L'utilisation de cette technique, mise au point par le groupe de gestion d'objets (OMG, *object management group*), est également envisagée par plusieurs industries. Les spécifications fondées sur cette technique servent de base aux interfaces de programmation d'application (API) normalisées et aux liens avec les langages de programmation et elles facilitent aussi la portabilité de logiciel. Les solutions d'interopérabilité offertes par le courtier pour les requêtes sur des objets (ORB, *object request broker*) se combinent avec l'interopérabilité client-serveur en ce qui concerne les adresses de protocole inter-ORB. Tandis que le protocole CMIP et les modèles d'information fournissent des solutions pour l'interopérabilité entre les systèmes du gestionnaire et de l'agent, la technique CORBA définit des interactions interobjet pour lesquelles les objets peuvent être répartis.

1.1 Objectif

Plusieurs groupes élaborent des spécifications de gestion de réseau qui utilisent des techniques de modélisation CORBA avec le langage IDL comme notation conjointement avec des services CORBA. La présente norme vise à définir des directives à utiliser pour spécifier des interfaces de gestion de réseau fondées sur CORBA interopérables. Les exigences relatives aux interfaces X sont différentes de celles relatives aux interfaces utilisées "à l'intérieur" d'une administration, les interfaces "Q". La présente Recommandation porte sur toutes les interfaces du RGT où la technique CORBA peut être utilisée. Les capacités et modèles définis ici ne devraient pas tous être nécessaires dans toutes les interfaces du RGT. Cela signifie que le cadre général peut être utilisé pour les interfaces entre systèmes de gestion à tous les niveaux d'abstraction (inter et intra-administration) ainsi qu'entre systèmes de gestion et éléments de réseau.

L'UIT-T Q.816 [1] définit un ensemble de services qui sont nécessaires pour les interfaces du RGT fondées sur CORBA. La présente Recommandation définit des directives pour la spécification, en IDL CORBA, de modèles d'information auxquels les services s'appliquent. Elle contient aussi des règles permettant de traduire les modèles GDMO existants en IDL. Enfin, elle définit un certain code IDL de base à utiliser par tous les modèles d'information du RGT fondés sur CORBA. La combinaison de la présente Recommandation et de l'UIT-T Q.816 constitue un cadre général pour la définition et la mise en œuvre d'interfaces du RGT fondées sur CORBA.

L'utilisation d'un cadre général commun pour les interfaces de gestion des télécommunications présente plusieurs avantages, notamment: faciliter la réutilisation de modèles qui sont développés pour répondre aux caractéristiques génériques des télécommunications; définir un profil pour les

services CORBA à utiliser par l'industrie des télécommunications; faciliter la définition de nouveaux services pour le RGT; réutiliser la sémantique du riche ensemble existant de modèles et harmoniser l'approche de modélisation parmi les groupes utilisant une source unique analogue à l'UIT-T X.720, à l'UIT-T X.721 et à l'UIT-T X.722 pour le protocole CMIP. La réutilisation d'une approche commune concernant la modélisation des ressources et la réutilisation d'un modèle d'information générique pour diverses techniques de réseau et diverses applications de gestion de réseau accéléreront l'introduction de nouveaux services de réseau tout en maintenant à un bas niveau les coûts de développement des systèmes de gestion de réseau.

L'industrie des télécommunications a investi beaucoup de temps et d'énergie pour l'élaboration de modèles d'information pour le protocole de gestion de réseau CMIP. Un objectif essentiel du cadre général CORBA du RGT est de pouvoir réutiliser ces modèles d'information en les traduisant dans le langage de définition d'interface (IDL) CORBA sans apporter de modifications importantes à la sémantique. Les modèles d'information IDL initiaux devraient donc découler de modèles CMIP.

1.2 Application

L'UIT-T M.3020 définit trois phases dans l'élaboration d'une spécification sur le RGT, à savoir: caractéristiques, analyse et conception. La Figure 1 montre ce processus ainsi que le domaine d'application de la présente Recommandation en vue de l'élaboration de spécifications d'interface fondée sur CORBA en rapport avec ce processus.

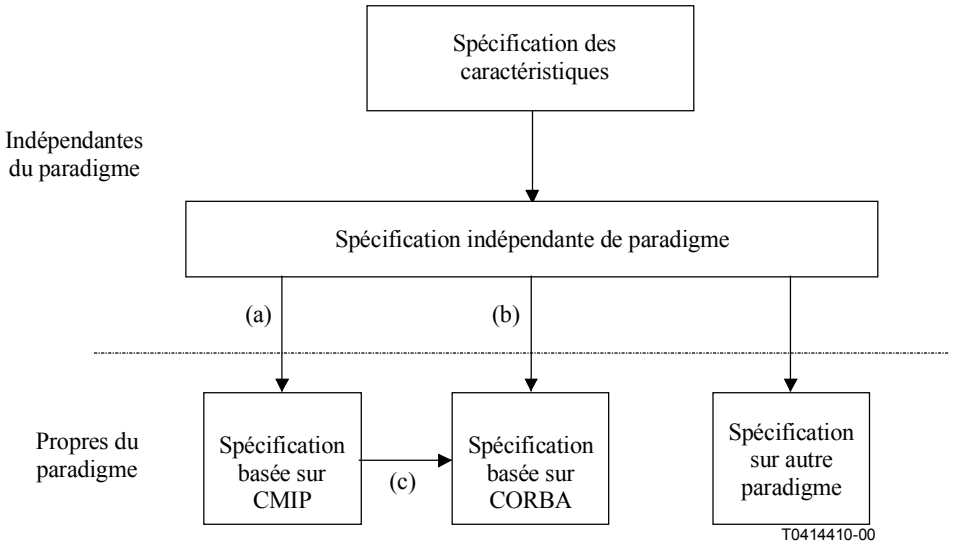

Figure 1/X.780 – Spécification fondée sur CORBA

Les caractéristiques et analyse sont spécifiées au moyen d'une approche indépendante du paradigme de technique de gestion de réseau. On utilise le résultat de la phase d'analyse, c'est-à-dire les caractéristiques indépendantes du paradigme, comme apport pour la phase de conception propre au paradigme.

Dans cette dernière, on utilise des particularités propres au paradigme de gestion de réseau pour définir des modèles d'information. Les caractéristiques propres au paradigme comportent tant le comportement (normalement en langage naturel) que les signatures d'interfaces formelles (telles que GDMO et IDL).

Les flèches désignées par (a) et (b) montrent que le résultat de l'analyse est projeté sur un modèle fondé sur GDMO/ASN.1 à utiliser respectivement avec des modèles CMIP ou IDL et avec

CORBA/IIOP. Il n'existe pas actuellement de règle établie permettant de générer ces modèles. Il sera peut-être possible d'élaborer de telles règles dans l'avenir dans l'UIT-T M.3020.

Dans la présente Recommandation, on aborde la question de la réutilisation de modèles existants élaborés dans le paradigme CMIP si CORBA/IIOP doit être utilisé à la place de CMIP [flèche désignée par (c)].

Pour définir la transformation des définitions GDMO/ASN.1 en IDL CORBA, deux méthodes sont possibles.

- Dans la première, chaque élément de la syntaxe est traduit en IDL CORBA au moyen d'un algorithme bien spécifié ou d'une définition établie. Cette méthode est celle qui est appliquée avec la gestion mixte interdomaine (JIDM) pour laquelle une passerelle peut être utilisée pour prendre en charge l'interopérabilité.
- La seconde (utilisée dans la présente Recommandation) ne traduit pas d'une façon formelle chaque élément de la syntaxe. Au lieu de cela, les éléments sont traduits à partir des définitions GDMO existantes d'une manière qui préserve la sémantique et utilise en outre les particularités de CORBA. Cette méthode n'est pas utilisée pour l'interfonctionnement en passant par des passerelles mais pour préserver les caractéristiques et la sémantique des modèles élaborés pour répondre au contexte des télécommunications. Elle est appliquée lorsque les systèmes gérant et géré sont conçus pour communiquer au moyen de CORBA/IIOP.

En plus des recommandations relatives à la traduction des modèles d'information GDMO définies ici, l'UIT-T Q.816 définit des recommandations relatives à l'utilisation de services CORBA pour la gestion des réseaux de télécommunication. Les aspects Q.816 du cadre général s'appliquent quelle que soit la manière dont les spécifications fondées sur CORBA sont élaborées [à savoir, en utilisant le chemin désigné par (b) ou celui désigné par (c) sur la Figure 1].

Outre la mise à profit des modèles d'information CMIP, un autre objectif des directives est de tirer parti de la technique CORBA. Le cadre général tire parti des fonctions définies dans les spécifications CORBA, y compris d'un ensemble de services d'objet communs. Par ailleurs, ces directives réutilisent les méthodes et modèles de conception CORBA chaque fois qu'il est judicieux de le faire. Enfin, il est important de réutiliser des modèles existants, mais il est tout aussi important que le cadre général permette d'élaborer de nouveaux modèles. Ces directives ne nécessitent pas d'élaborer un modèle GDMO avant d'élaborer un modèle IDL. En réalité, l'élaboration d'un nouveau modèle d'information IDL à utiliser dans ce cadre général est immédiate et des directives sont fournies en ce sens.

L'UIT-T M.3120 [11] fournit une version IDL CORBA du modèle d'information de réseau générique, dont la version originale est définie dans l'UIT-T M.3100. La version IDL est conforme aux directives pour la modélisation d'objet définies ici et elle est conçue pour utiliser les services du RGT fondés sur CORBA qui sont définis dans l'UIT-T Q.816.

1.3 Structure

La présente Recommandation a la structure suivante:

Paragraphe 1	Introduction, structure du document et mises à jour.
Paragraphe 2	Références.
Paragraphe 3	Signification des abréviations utilisées dans l'ensemble de la Recommandation.
Paragraphe 4	Caractéristiques applicables aux directives pour la modélisation d'objet. Il s'agit des objectifs de conception devant être atteints dans le cadre des directives.

Paragraphe 5 Description du module IDL CORBA définissant des interfaces à utiliser dans les spécifications d'interface de gestion de réseau, spécifications qui définiront des sous-classes de ces interfaces. Le véritable code IDL figure dans les Annexes A et B.

Paragraphe 6 Directives pour la définition de modèles d'information du RGT fondés sur CORBA. Ces directives sont expressément conçues pour les objets IDL utilisant les services fondés sur CORBA du RGT décrits dans l'UIT-T Q.816.

Paragraphe 7 Directives pour la traduction des modèles d'information GDMO en modèles IDL pouvant être utilisés avec les services fondés sur CORBA du RGT décrits dans l'UIT-T Q.816.

Paragraphe 8 Idiomes de style pour les spécifications d'interface de gestion de réseau en IDL CORBA.

Paragraphe 9 Directives relatives à la conformité.

Annexe A Module IDL concernant la spécification des directives pour la modélisation. Cette annexe est normative.

Annexe B Code IDL additionnel définissant les constantes utilisées par les directives pour la modélisation. Cette annexe est normative.

1.4 Conventions

Dans la présente Recommandation, on utilise quelques conventions pour faciliter la compréhension du texte par le lecteur. La plus grande partie de la Recommandation est normative, mais les paragraphes dans lesquels sont exposées succinctement des prescriptions obligatoires à respecter par un système de gestion (gérant et/ou géré) commencent par un "R" en gras et entre parenthèses, suivi d'un substantif bref indiquant l'objet de la prescription et d'un numéro.

Par exemple:

(R) EXEMPLE-1 Exemple de prescription obligatoire.

De manière analogue, les paragraphes présentant brièvement des prescriptions qui peuvent être facultativement implémentées par un système de gestion commencent par un "O" et non par un "R." Par exemple:

(O) OPTION-1 Exemple de prescription facultative.

On utilise ces prescriptions pour créer des profils de conformité.

De nombreux exemples de code IDL CORBA figurent dans la présente Recommandation et le code IDL spécifiant les types de données et les classes de base figure dans des annexes normatives. Le code IDL est présenté en caractères courrier à 9 points:

```
// Example IDL
interface foo {
        void operation1 ();
};
```

Le paragraphe 1.5 propose les instructions pour extraire le langage IDL d'une version électronique de la présente Recommandation et le compiler.

1.5 Compilation du langage IDL

Le langage IDL utilisé pour spécifier des interfaces de gestion du réseau présente l'avantage de pouvoir être "compilé" en code de programmation au moyen d'outils qui accompagnent un courtier ORB. Cela automatise en fait le développement d'une partie du code nécessaire pour permettre aux applications de gestion du réseau d'interfonctionner. La présente Recommandation a deux annexes

qui contiennent le code que les réalisateurs souhaiteront extraire et compiler. Les Annexes A et B sont toutes deux normatives; elles devraient être utilisées par les réalisateurs de logiciel qui implémentent des systèmes conformes à la présente Recommandation. Le langage IDL qui apparaît dans la présente Recommandation a été contrôlé au moyen de deux compilateurs afin de s'assurer de son exactitude. Il faut utiliser un compilateur prenant en charge la spécification CORBA 2.3.

Les annexes ont été formatées de telle manière qu'elles puissent être coupées et colées dans du texte en clair pouvant ensuite être compilé. Voici quelques conseils sur la manière de procéder.

1) Il est apparu que l'opération couper/coller fonctionne mieux à partir de la version Microsoft® Word® de la présente Recommandation. Dans le format Adobe® Acrobat®, l'opération englobe les en-têtes et les pieds de page, qui ne peuvent être compilés.

2) L'ensemble de l'Annexe A, depuis le début de la ligne"/* This IDL code ..." jusqu'à la fin, devrait être enregistré dans un fichier désigné "itu_x780.idl", dans un répertoire où il sera trouvé par le compilateur IDL.

3) L'ensemble de l'Annexe B, commençant par la ligne ""/* This IDL code ..." jusqu'à la fin devrait être enregistré dans un fichier nommé "itu_x780Const.idl" dans le même répertoire que le fichier contenant l'Annexe A.

4) Il n'y a pas lieu d'enlever les titres intégrés dans ces annexes. Ils ont été encapsulés dans des commentaires IDL mais seront ignorés par le compilateur.

5) Les commentaires qui commencent par la séquence spéciale "/**" sont reconnus par les compilateurs qui convertissent le langage IDL en HTML. Ces commentaires ont souvent des restrictions de formatage spéciales pour ces compilateurs. Ceux qui travailleront avec le langage IDL souhaiteront peut-être produire une version HTML étant donné que les fichiers HTML résultants ont des liens qui permettent de naviguer rapidement dans les fichiers.

Les annexes ont été formatées avec des tabulateurs à intervalles de 8 espaces et des retours à la ligne fixes qui devraient permettre à pratiquement tout éditeur de fonctionner avec ce texte.

2 Références

2.1 Références normatives

La présente Recommandation se réfère à certaines dispositions des Recommandations UIT-T et textes suivants qui, de ce fait, en sont partie intégrante. Les versions indiquées étaient en vigueur au moment de la publication de la présente Recommandation. Toute Recommandation ou tout texte étant sujet à révision, les utilisateurs de la présente Recommandation sont invités à se reporter, si possible, aux versions les plus récentes des références normatives suivantes. La liste des Recommandations de l'UIT-T en vigueur est régulièrement publiée.

[1] UIT-T Q.816 (2001), *Services du RGT fondés sur CORBA*.

[2] OMG Document formal/99-10-07, *The Common Object Request Broker: Architecture and Specification*, Révision 2.3.1.

[3] OMG Document formal/2000-08-01, *CORBA/TMN Interworking, Version 1*, Edition 4.31.

[4] UIT-T X.701 (1997) | ISO/CEI 10040:1998, *Technologies de l'information – Interconnexion des systèmes ouverts – Aperçu général de la gestion-systèmes*

[5] UIT-T X.703 (1997) | ISO/CEI 13244:1998, *Technologies de l'information – Architecture de gestion répartie ouverte*.

[6] ITU-T X.721 (1992) | ISO/CEI 10165-2:1992, *Technologies de l'information – Interconnexion des systèmes ouverts – Structure des informations de gestion: définition des informations de gestion*.

[7] ITU-T X.722 (1992) | ISO/CEI 10165-4:1992, *Technologies de l'information – Interconnexion des systèmes ouverts – Structure des informations de gestion: directives pour la définition des objets gérés*.

3 Définitions et abréviations

3.1 Termes définis dans l'UIT-T X.701

Les termes suivants utilisés dans la présente Recommandation sont définis dans l'aperçu général de la gestion-systèmes (UIT-T X.701):

– classe d'objets gérés

– gestionnaire

– agent

3.2 Termes définis dans l'UIT-T X.703

Le terme suivant utilisé dans la présente Recommandation est défini dans l'architecture de gestion répartie ouverte (UIT-T X.703):

– notification

3.3 Abréviations

La présente Recommandation utilise les abréviations suivantes:

ASN.1 notation de syntaxe abstraite numéro un (*abstract syntax notation No 1*)

ATM mode de transfert asynchrone (*asynchronous transfer mode*)

CMIP protocole commun d'informations de gestion (*common management information protocol*)

CORBA architecture de courtier commun de requête d'objets (*common object request broker architecture*)

COS services d'objet communs (*common object services*)

DN nom distinctif (*distinguished name*)

EMS système de gestion d'élément (*element management system*)

GDMO directives pour la définition des objets gérés (*guidelines for the definition of managed objects*)

GIOP protocole général d'interopérabilité (*general interoperability protocol*)

HTML langage de balisage hypertexte (*hypertext markup language*)

ID identificateur

IDL langage de définition d'interface (*interface definition language*)

IIOP protocole d'interopérabilité Internet (*Internet interoperability protocol*)

IOR référence d'objet interopérable (*interoperable object reference*)

JIDM gestion mixte interdomaine (*joint inter-domain management*)

MO objet géré (*managed object*)

NE élément de réseau (*network element*)

NMS système de gestion de réseau (*network management system*)

OAM&P exploitation, administration, maintenance et fourniture (*operations, administration, maintenance, and provisioning*)

OID identificateur d'objet (*object identifier*)

OMG groupe de gestion d'objets (*object management group*)

ORB courtier de requête d'objets (*object request broker*)

OSI interconnexion des systèmes ouverts (*open systems interconnection*)

PDU unité de données protocolaire (*protocol data unit*)

QS qualité de service

RDN nom distinctif relatif (*relative distinguished name*)

RGT réseau de gestion des télécommunications

TTP point de terminaison de chemin (*trail termination point*)

UID identificateur universel (*universal identifier*)

UIT-T Union internationale des télécommunications – Secteur de la normalisation des télécommunications

UML langage de modélisation unifié (*unified modelling language*)

UTC code temporel universel (*universal time code*)

4 Objectifs et caractéristiques applicables à la modélisation CORBA

Le présent paragraphe décrit les objectifs essentiels applicables à la modélisation de ressources du RGT au moyen de la technique CORBA ainsi que les caractéristiques que les directives pour la modélisation doivent respecter afin que ces objectifs soient atteints. Le paragraphe 4.1 présente les objectifs des directives pour la modélisation. Les sous-paragraphes qui suivent contiennent la terminologie et les caractéristiques. Les caractéristiques indiquées au paragraphe 4 sont celles que le cadre général doit respecter. Elles sont fondées sur les besoins en matière de gestion des télécommunications. Les paragraphes 5, 6, 7 et 8 décrivent ensuite les directives pour la modélisation qui permettent de répondre à ces besoins et définissent la manière de satisfaire aux caractéristiques du paragraphe 4 en utilisant CORBA d'une certaine façon. Les règles énoncées aux paragraphes 5, 6, 7 et 8 sur la manière d'utiliser CORBA constituent aussi des caractéristiques.

4.1 Objectifs

La présente Recommandation spécifie des directives pour la définition d'objets gérés CORBA à utiliser sur des interfaces prises en charge par des systèmes de gestion de réseau de télécommunication et par des éléments de réseau. Les objectifs essentiels des directives pour la modélisation comprennent notamment:

- l'interopérabilité des applications;
- l'utilisation commune de services d'objet communs CORBA;
- la transparence des modèles d'information.

Le présent paragraphe explicite ces trois objectifs.

4.1.1 Interopérabilité des applications

Un objectif essentiel de l'architecture du RGT, et en particulier de l'architecture d'information, est de promouvoir un cadre général normalisé pour assurer l'interopérabilité et l'échange d'informations entre systèmes de gestion de réseau provenant d'un ensemble de divers fournisseurs. L'interopérabilité entre systèmes fait intervenir de nombreux aspects de développement. Au niveau de la couche la plus basse, un mécanisme commun de communication doit être en place pour prendre en charge une syntaxe commune, l'établissement de la connectivité et l'échange de demandes/réponses concernant des opérations entre systèmes. Cet aspect de l'interopérabilité est pris en charge de façon intrinsèque par la spécification CORBA.

En ce qui concerne le RGT, il est nécessaire d'assurer l'interopérabilité des applications. Autrement dit, les systèmes de gestion provenant de divers fournisseurs seront utilisés dans un seul RGT d'une administration donnée pour remplir différentes fonctions nécessaires à la prise en charge de la gestion des réseaux de ladite administration. Pour simplifier l'intégration de leurs systèmes, ces divers fournisseurs doivent se mettre d'accord sur la sémantique des informations échangées. Pour cela, un modèle d'information est spécifié. Le présent document spécifie les règles permettant de définir ces modèles d'information.

4.1.2 Utilisation commune de services d'objet communs CORBA

Un second aspect de ces directives est qu'elles s'appuient sur une utilisation commune de l'environnement de traitement réparti choisi et sur la définition d'un profil pour cet environnement. Plutôt que de redéfinir, dans chaque modèle d'information, les capacités d'interface nécessaires pour assurer les fonctions communes de gestion de réseau telles que la dénomination d'objet et le filtrage des notifications, ces directives reposent sur un ensemble de services support. Ceux-ci permettent d'avoir des modèles d'information plus simples ainsi que de renforcer l'interopérabilité. Les services support nécessaires pour les interfaces fondées sur CORBA sont spécifiés dans l'UIT-T Q.816.

4.1.3 Transparence des modèles d'information

Si la technique CORBA est utilisée dans des endroits de l'architecture du RGT où des modèles d'information existants (par exemple GDMO) sont bien établis, le cadre général doit alors prendre en charge la réutilisation de ces modèles sans trop les modifier.

Un procédé normalisé unique permettant de projeter ces modèles d'information GDMO en IDL OMG est nécessaire de manière à ce que le protocole d'application présente toujours à l'application les mêmes modèles avec le même ensemble de services (capacités).

4.2 Entités

Un **type d'entité** décrit un genre de "chose" dans le monde réel. Chaque **type d'entité** a des propriétés particulières, appelées attributs.

Une **instance d'entité** (ou **entité**) (par exemple, bloc de circuits numéro 1) est un **type d'entité** (par exemple, bloc de circuits numéro 1). Les attributs de chaque **entité** ont des valeurs particulières qui représentent l'état de cette instance. En outre, chaque **entité** doit être identifiable de manière univoque.

Dans le cadre de CORBA, une **entité** est accessible au moyen de méthodes différentes. Elle est, par exemple, accessible par une structure de données IDL, un type de valeur ou un type d'interface. La présente Recommandation expose la manière dont la technique CORBA est utilisée pour définir des **types d'entité**.

4.2.1 Granularité à l'accès

Dans le contexte des opérations du RGT, la granularité définit le niveau d'abstraction qui est exposé entre systèmes. La granularité à l'accès identifie le niveau auquel on peut accéder aux **entités** (c'est-à-dire comment les informations sont exposées via une interface). Dans le cadre de CORBA, on attribue à chaque objet CORBA une adresse unique: la référence d'objet interopérable (IOR). Celle-ci fournit au système client une adresse identifiant le système serveur auquel il faut se raccorder pour pouvoir communiquer avec l'objet CORBA du côté serveur.

Dans le cadre de CORBA, il est possible de définir différentes abstractions concernant l'accès (c'est-à-dire différentes granularités à l'accès) pour les entités définies pour le RGT (par exemple l'UIT-T M.3100). Deux abstractions différentes concernant l'accès sont définies ici:

1) **granularité des instances**: chaque entité a sa propre référence IOR. Pour la création de nouvelles entités, cela implique l'instanciation d'un nouvel objet CORBA.

   -1 référence IOR/instance d'entité

   Par exemple, un **type d'entité** dans le domaine ATM est *atmLink*. Dans l'approche avec granularité des instances, on définit un objet CORBA qui prend en charge les mêmes attributs que le type d'entité qu'il représente. Pour chaque instance de *atmLink*, un objet CORBA indépendant est créé. Ainsi, chaque *atmLink* peut être adressé de manière univoque par sa référence IOR;

2) **granularité propre à l'application**: on accède aux instances d'un ensemble bien défini de types d'entité via une référence IOR unique (une interface unique).

   -1 référence IOR/famille (ensemble) de types d'entité

   Dans les interfaces IDL CORBA propres à l'application, on définit des opérations groupées, qui transmettent l'identité et l'état des entités gérées au moyen de paramètres d'opération employant des listes de types à structure IDL.

Les directives pour la modélisation d'objet CORBA définies dans la présente Recommandation s'appliquent à la spécification d'interfaces d'objet géré qui prennent en charge la granularité à l'accès avec granularité des instances. On peut aussi définir des normes sur le RGT utilisant la granularité à l'accès propre à l'application. Ces spécifications d'interface n'entrent toutefois pas dans le cadre de la présente Recommandation.

4.3 Principes de contenance et de dénomination

La contenance est une représentation logique de la manière dont les entités d'un certain type contiennent des entités d'un autre type. Un arbre de contenance définit la relation entre les instances d'entité. Une instance d'entité est contenue dans une instance d'entité contenante et une seule. Les instances d'entité contenantes peuvent elles-mêmes être contenues dans une autre instance d'entité, ce qui forme un graphe orienté. Le graphe orienté constitue ce qu'on appelle l'arbre de dénomination (ou de contenance).

La relation de contenance peut être utilisée pour modéliser de véritables hiérarchies de parties (par exemple assemblage, sous-assemblages et composantes) ou de véritables hiérarchies d'organisations (par exemple nom d'entreprise, nom d'organisation).

Un exemple d'arbre de contenance possible est montré sur la Figure 2 ci-dessous.

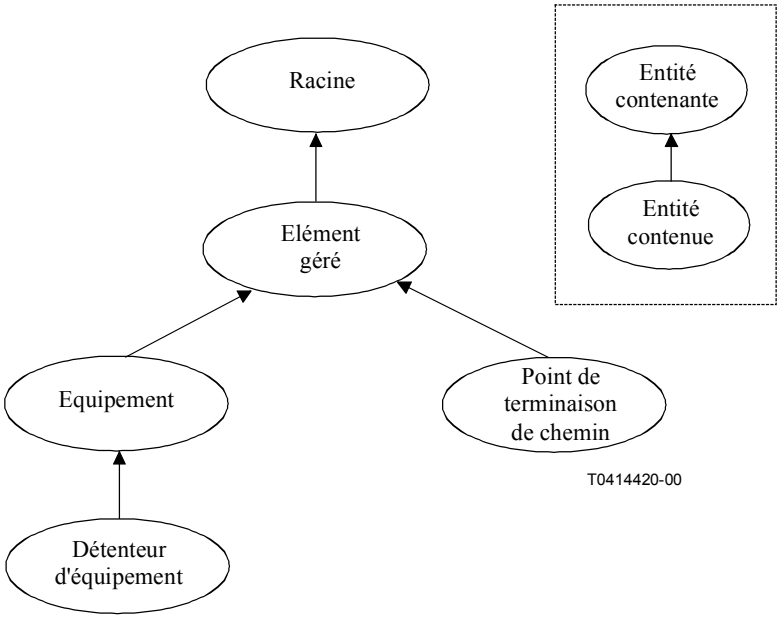

Figure 2/X.780 – Exemple de contenance

4.3.1 Dénomination

Les relations de contenance servent notamment pour les entités de dénomination. Les noms sont conçus pour être uniques dans un contexte spécifié; en ce qui concerne le RGT, ce contexte est déterminé par l'instance d'entité contenante.

Une entité qui est dénommée dans le contexte d'une autre entité est appelée "entité subordonnée". L'entité qui établit le contexte de dénomination (ce terme est employé dans un sens général et ne doit pas avoir la connotation directe d'un contexte de dénomination de service de dénomination COS) pour d'autres entités est appelée "entité supérieure".

Une "entité subordonnée" est dénommée par la combinaison:

– Du nom de son "entité supérieure".

– D'informations l'identifiant de manière univoque dans le cadre de son entité supérieure.

Le nom d'une entité, qui est unique dans un contexte de dénomination local, peut ne pas l'être dans un certain contexte de dénomination plus grand. Toutefois, si le contexte de dénomination local est unique dans le contexte plus grand, un nom local peut être rendu unique en lui ajoutant le nom de son contexte de dénomination, utilisé comme qualificatif. Cet arrangement peut être visualisé sous forme de graphe orienté dans lequel chaque arc (ou flèche) va d'un objet dénommé à un contexte de dénomination.

Concernant le nom du contexte de dénomination proprement dit, on peut lui ajouter comme qualificatif le nom d'un autre contexte de dénomination, et ce de façon récurrente, de sorte que la structure de dénomination complète peut être visualisée sous la forme d'une hiérarchie à racine unique. Cette hiérarchie est appelée arbre de dénomination. Ainsi, les "entités supérieures" deviennent les contextes de dénomination et leurs noms deviennent les noms des contextes. Un nom d'objet n'a besoin d'être unique que dans le contexte de ses entités supérieures; dans un contexte plus large, on lui ajoute toujours, comme qualificatifs, les noms de ses entités supérieures.

4.3.2 Identification d'entité

Comme une "entité supérieure" peut contenir plusieurs "entités subordonnées" du même type, chacune de ces entités contenues du même type doit pouvoir être distinguée par rapport à leur entité contenante. Le nom relatif d'une entité dans son entité contenante est appelé **nom distinctif relatif**

(RDN, *relative distinguished name*) de cette entité. Par exemple, il peut y avoir plusieurs détenteurs d'équipement dans un élément géré. Pour identifier de façon univoque chaque détenteur d'équipement dans l'élément géré, il faut fournir un nom RDN à chacun de ces détenteurs. Le nom RDN doit comprendre le nom du type d'entité (par exemple le détenteur d'équipement, qui est un type d'entité) et une valeur unique dans le cadre de l'entité contenante.

Un nom RDN est un élément de base d'un **nom distinctif** (DN, *distinguished name*), comme spécifié dans l'UIT-T X.720. Un nom DN est défini par une séquence de noms RDN à partir d'un contexte donné. Le nom DN permet d'obtenir un nom unique par rapport à ce contexte.

4.4 Classes d'objets gérés

Les directives pour la modélisation définies ici spécifient que chaque type d'entité est en mappage biunivoque avec une interface opérationnelle CORBA. Lorsqu'un type d'entité est mappé de cette manière, l'objet CORBA le représentant est appelé *classe d'objets gérés*. Une telle classe doit aussi être capable d'émettre des notifications (voir l'UIT-T X.703).

Le terme "classe d'objets gérés" est défini dans l'UIT-T X.720. Comme expliqué dans l'UIT-T X.703, les classes et sous-classes d'objets gérés sont mappées avec des interfaces et des interfaces dérivées.

4.5 Paquetages

Il est nécessaire de bien cerner la notion de paquetage en IDL CORBA. Les paquetages sont des groupes de capacités (attributs, actions ou notifications) qui peuvent être pris en charge sous certaines conditions par une instance d'objet géré. Un système gérant doit pouvoir déterminer quels paquetages sont pris en charge par une instance d'objet géré. Si des opérations sont effectuées sur un objet géré et qu'elles sont contenues dans un paquetage conditionnel qui n'est pas instancié pour cet objet géré, celui-ci doit indiquer une erreur.

4.6 Attributs

Les directives doivent prendre en charge la définition d'attributs (c'est-à-dire de propriétés visibles) pour les classes d'objets gérés.

4.6.1 GET et SET

La valeur d'un attribut peut être observable ou modifiable via une interface normalisée. Si elle est observable, le modélisateur d'information doit définir une méthode "get" (lecture) pour cet attribut. Si elle est modifiable, il doit définir une méthode "set" (écriture) pour cet attribut.

4.6.2 Lecture d'attribut générique

Les modèles d'information du RGT fondés sur CORBA doivent permettre à un système gérant de lire des groupes arbitraires d'attributs à partir d'un seul objet géré avec une opération unique. Ce service permet d'effectuer de nombreuses tâches de gestion avec une opération unique. La prise en charge de la lecture d'attribut générique est requise.

4.6.3 Attributs ayant un ensemble de valeurs

En ce qui concerne les attributs contenant des listes de valeurs, un modélisateur doit pouvoir faire en sorte que les systèmes gérants puissent ajouter ou supprimer des valeurs individuelles sur les listes sans avoir à renvoyer toutes les informations de la liste d'origine.

4.7 Création et suppression d'objets gérés

L'existence d'objets gérés (MO, *management object*) est étroitement liée à la relation de contenance entre les objets gérés. L'existence d'un objet géré est liée à l'existence de l'instance d'objet géré supérieur de cet objet géré. Si "l'objet géré supérieur" spécifié n'existe pas pour un "objet géré

subordonné", celui-ci ne peut pas être créé. De même, si "l'objet géré supérieur" d'un objet géré est supprimé, cet "objet géré subordonné" (et les subordonnés de cet "objet géré subordonné") ne peut (peuvent) plus exister. Cela étant, il faut que le cadre général CORBA du RGT applique une sémantique de création et de suppression.

Les sous-paragraphes qui suivent définissent les caractéristiques de haut niveau qui doivent être prises en charge pour la création et la suppression d'objet. L'UIT-T Q.816 décrit les services génériques utilisés pour mener à bien la création (c'est-à-dire la fabrique) et la suppression (c'est-à-dire la fabrique en coordination avec le service de terminaison). Le paragraphe 6 définit des directives pour la modélisation concernant la manière dont les caractéristiques définies dans le présent paragraphe sont prises en charge.

4.7.1 Création

Lors de la création d'un objet géré, trois aspects de l'existence de l'objet géré doivent être identifiés:

- le nom de l'objet géré;
- les valeurs des attributs de l'objet géré;
- les paquetages conditionnels de l'objet géré qui doivent être instanciés au moment de la création du nouvel objet géré.

Il est à noter que la définition de ces aspects dans la demande de création peut être explicite ou implicite. Les options relatives à l'identification de ces aspects de l'existence d'un objet géré sont définies dans les trois paragraphes suivants.

4.7.1.1 Identification du nom de l'objet géré

Le nom de l'objet géré à créer peut être déterminé de l'une des façons suivantes:

1) le gestionnaire peut spécifier, sous forme de paramètre de l'opération de création, une référence à un objet géré existant qui doit être le supérieur du nouvel objet géré et peut spécifier le nom RDN du nouvel objet géré dans la liste des attributs de l'opération de création. Ainsi, le gestionnaire fournit la spécification complète du nom de l'objet géré;
2) le gestionnaire peut spécifier, sous forme de paramètre de l'opération de création, une référence à un objet géré existant qui doit être le supérieur du nouvel objet géré et peut omettre de spécifier le nom RDN du nouvel objet géré. Dans ce cas, le nom RDN du nouvel objet géré est assigné par le système géré.

Si les informations associées ne sont pas correctes ou si, pour une autre raison, l'opération de création ne peut pas être effectuée, la fabrique essayant d'effectuer l'opération doit indiquer une erreur.

4.7.1.2 Identification des attributs de l'objet géré

Lorsqu'un objet géré est créé, on assigne à ses attributs des valeurs qui sont valides pour le type d'attribut. Ces valeurs sont déduites des informations associées à l'opération de création et à la définition de la classe d'objets gérés, de l'une des deux façons suivantes:

1) la demande de création peut spécifier une valeur explicite pour chacun des attributs. Lorsque l'objet géré est créé, les valeurs explicites sont assignées aux attributs en fonction des exigences de la définition de la classe d'objets gérés;
2) la définition de la classe d'objets gérés peut spécifier comment des valeurs par défaut sont assignées aux attributs pour lesquels l'opération de création ne spécifie pas de valeur.

Si aucune valeur par défaut n'est spécifiée pour un attribut, le système gérant doit fournir une valeur pour cet attribut dans la demande de création. Si aucune valeur n'est spécifiée pour cet attribut, une erreur doit se produire.

Si une valeur explicite est définie pour un attribut donné dans la demande de création, l'objet géré prendra cette valeur pour l'attribut spécifié de préférence à toute valeur par défaut potentielle pouvant être spécifiée pour cet attribut.

4.7.1.3 Identification des paquetages de l'objet géré à instancier

Pour faire en sorte que des ressources sous-jacentes puissent être instanciées avec les capacités requises, le gestionnaire doit être capable de spécifier les capacités (c'est-à-dire les paquetages conditionnels) que l'objet géré doit avoir instanciées.

Un paquetage conditionnel sera instancié si une condition associée est satisfaite pour l'objet géré instancié. Le gestionnaire peut aussi demander l'instanciation d'un paquetage conditionnel dans le cadre de la demande de création, en l'incluant dans l'attribut de paquetage de cette demande.

4.7.2 Suppression

En ce qui concerne la suppression, la sémantique de suppression peut prendre en charge la suppression de toutes les entités contenues tandis que dans d'autres cas, la méthode de suppression échoue immédiatement s'il existe des entités subordonnées contenues. Ces sémantiques doivent être conservées pour chaque type d'entité.

4.8 Héritage

Une "classe d'objets gérés" peut être définie comme une spécialisation d'une autre "classe d'objets gérés" en utilisant l'héritage. La spécialisation d'une "classe d'objets gérés" implique que toutes les méthodes et tous les attributs définis pour la superclasse seront aussi pris en charge par la sous-classe.

En IDL CORBA, un attribut ou une opération ne peut pas être hérité de plusieurs interfaces et une opération ou un attribut hérité ne peut pas être redéfini par une sous-classe. (Il est à noter qu'en général, il n'est pas prévu qu'un modèle d'information CORBA définisse une méthode ou un attribut dans une classe, lorsque cette méthode ou cet attribut peut aussi être défini dans la superclasse. Toutefois, dans certains cas de projection de GDMO sur IDL, cela peut se produire. Par exemple, comme les attributs GDMO spécifient des valeurs autorisées et des valeurs requises, une sous-classe en GDMO peut parfois redéfinir le même attribut. Il faut veiller, lors du mappage sur IDL, à ce que le même attribut ne soit pas redéfini.)

Une sous-classe en CORBA ne peut pas hériter le même attribut ou la même méthode (avec le même nom) de plusieurs superclasses (sauf si celles-ci l'héritent à leur tour de la même classe de base). En outre, une sous-classe ne peut pas redéfinir le même attribut ou la même méthode (avec le même nom) défini dans une de ses superclasses.

Les directives définies ici n'imposent aucune contrainte à l'héritage CORBA.

5 Module IDL du modèle d'objet

Avant de décrire les règles permettant de définir les objets gérés du RGT au moyen du langage de définition d'interface (IDL) [2] CORBA, le présent paragraphe expose un module de gestion de réseau comprenant un ensemble d'interfaces d'objet et prenant en charge des structures de données spécifiées en IDL CORBA. Ce module IDL est censé remplir un rôle dans la gestion de réseau fondé sur CORBA qui est analogue à celui rempli par les définitions GDMO et ASN.1 dans l'UIT-T X.721 [6] pour le protocole CMIP. Il fournit l'ensemble de base de définitions IDL à partir duquel les modèles d'information sont ensuite construits.

Le code IDL figure dans les Annexes A et B de la présente Recommandation. L'Annexe A contient les classes de base (interfaces), les structures de données et les notifications. L'Annexe B est un

fichier distinct contenant uniquement les définitions des constantes. Elles sont toutes deux fondées sur les définitions GDMO et ASN.1 figurant dans l'UIT-T X.721.

L'UIT-T X.721 est une bonne source concernant les capacités qui doivent être fournies dans les modèles d'information de gestion de réseau. Elle définit les classes d'objets gérés suivantes au moyen de GDMO:

- 9 types d'enregistrement (enregistrement de journalisation, enregistrement de journalisation d'événement, enregistrement d'alarme, enregistrement de modification de valeur d'attribut, enregistrement de création d'objet, enregistrement de suppression d'objet, enregistrement de relation, enregistrement de rapport d'alarme de sécurité, enregistrement de changement d'état);
- discriminateur et discriminateur de retransmission d'événement;
- journalisation;
- système;
- sommet.

Chacune d'elles a des attributs, des actions ainsi que des types de données et des paramètres support. En outre, l'UIT-T X.721 définit 15 notifications.

Si l'on regarde les classes d'objets gérés énumérées ci-dessus, il apparaît clairement que bon nombre d'entre elles sont couvertes par les services d'objet communs CORBA déjà inclus dans le cadre général (voir l'UIT-T Q.816 pour les détails sur les services du RGT fondés sur CORBA):

- le service de journalisation d'événement de télécommunication CORBA définit une structure pour la conservation des enregistrements de journalisation, de sorte qu'il n'est pas nécessaire de redéfinir les classes d'enregistrement. (Il est à noter qu'en spécifiant l'utilisation de ce service, le cadre général CORBA du RGT considère les enregistrements de journalisation comme des structures de données et non comme des objets.);
- le service de notification CORBA définit une capacité de filtrage, de sorte qu'il n'est pas nécessaire de redéfinir le discriminateur et le discriminateur de retransmission d'événement;
- le service de journalisation d'événement de télécommunication CORBA définit l'équivalent de la journalisation X.721.

Il reste donc uniquement les classes système et sommet, ainsi que les notifications. La classe système n'est pas vraiment une classe de cadre général et appartient plutôt à un modèle d'information générique (si elle est nécessaire). Le code IDL donné dans l'Annexe A définit donc une interface d'objet géré "du sommet", appelée "*ManagedObject*", qui est censée avoir pour sous-classes toutes les autres interfaces d'objet géré de la même manière que la classe d'objets gérés appelée "Top" (sommet) a pour sous-classes toutes les classes d'objets gérés CMIP. Est également inclus un objet "fabrique" générique. Les fabriques d'objet géré sont utilisées pour la création d'objet. (Les services du RGT fondés sur CORBA qui sont définis dans l'UIT-T Q.816 comprennent un service de *terminaison* qui traite les suppressions d'objet indépendamment du type d'objet, mais la création d'objet est traitée par des fabriques propres à la classe de sorte que les opérations de création d'objet peuvent être fortement typées.) Les notifications sont définies sur une troisième interface IDL. En outre, un certain nombre de types de données IDL sont définis. Enfin, certaines macros de précompilateur IDL sont définies afin de faciliter la spécification des interfaces d'objet géré. Chacune d'elles est examinée ci-dessous.

5.1 Interface d'objet géré (du sommet) de base

La première interface définie dans l'Annexe A est l'interface *ManagedObject*, située après toutes les définitions de type de données. Elle est censée être l'interface d'objet géré de base dont toutes les autres interfaces héritent. Elle définit un ensemble de capacités que toutes les instances d'objet géré doivent prendre en charge. Ces capacités sont les suivantes:

- une méthode qui renvoie le nom de l'objet;
- une méthode qui renvoie le nom de l'interface (de la véritable classe) de l'objet;
- une méthode qui renvoie les paquetages conditionnels pris en charge par l'instance d'objet;
- une méthode qui renvoie la source de création de l'objet (création autonome par la ressource gérée, création en réponse à une opération de gestion ou création inconnue);
- une méthode qui renvoie la politique de suppression applicable à l'instance. Il s'agit d'une valeur énumérée qui indique si l'objet ne peut pas être supprimé, s'il peut être supprimé uniquement s'il ne contient pas d'objet ou si tous les objets contenus seront supprimés lorsqu'il sera supprimé;
- une méthode qui renvoie un objet de type de valeur CORBA contenant tous les attributs lisibles de l'objet;
- une opération de destruction.

Le code IDL décrivant l'interface *ManagedObject* (sans les commentaires) est le suivant:

```
interface ManagedObject {
       NameType nameGet()
          raises (ApplicationError);
       ObjectClassType objectClassGet()
          raises (ApplicationError);
       StringSetType packagesGet()
          raises (ApplicationError);
       SourceIndicatorType creationSourceGet()
          raises (ApplicationError);
       DeletePolicyType deletePolicyGet()
          raises (ApplicationError);
       ManagedObjectValueType attributesGet (
          inout StringSetType attributeNames)
          raises (ApplicationError);
       void destroy()
          raises (ApplicationError, DeleteError);

}; // end of ManagedObject interface
```

5.1.1 Opération *nameGet*()

La première opération, *nameGet*(), renvoie le nom CORBA de l'objet. *NameType* est une définition de type pour le type *Name* du service de dénomination CORBA. *NameType* est utilisé en vue de la conformité aux conventions IDL définies plus loin dans la présente Recommandation. Cette méthode renvoie le nom composé de l'objet, commençant par le nom assigné au contexte de dénomination de la racine locale dans le cadre duquel l'objet est contenu. Autrement dit, la méthode renvoie le nom "unique à l'échelle globale" de l'objet. Voir l'UIT-T Q.816 pour plus de détails concernant l'assignation d'un nom unique au contexte de dénomination de la racine d'un système géré. Par définition, l'anomalie *ApplicationError* est signalée par n'importe quelle opération d'objet géré si l'opération ne peut pas être achevée en raison d'un problème de ressource. Voir le paragraphe 5.5 ci-dessous pour plus de détails à ce sujet ainsi que toutes les autres anomalies.

5.1.2 Opération *objectClassGet*()

L'opération *objectClassGet*() renvoie le nom limité en portée de l'interface (le nom de la véritable classe) de l'objet. Les noms d'interface limités en portée comprennent le ou les noms du ou des modules dans lesquels l'interface est définie. Le type de valeur renvoyée, *ObjectClassType*, est une définition de type pour une chaîne. Si la classe d'objets est une extension mineure d'une autre classe

(par exemple une classe "R1"), la chaîne renvoyée est le nom de la véritable classe (avec le "R1"). Par exemple, "EquipmentR1".

5.1.3 Opération *packagesGet*()

L'opération *packagesGet*() renvoie la liste des paquetages conditionnels pris en charge par une instance d'objet. La notion de paquetages conditionnels, chacun ayant un nom de chaîne, est prise en charge par les directives définies ici. Voir le paragraphe 6.6 pour plus de détails. *StringSetType* est une définition de type pour une liste de chaînes.

Il est à noter une légère différence par rapport à l'attribut *packages* des objets CMIP car ce cadre général ne prend pas en charge la définition des paquetages obligatoires mais uniquement de ceux qui sont conditionnels. Dans le protocole CMIP, il est possible que l'attribut *packages* contienne des paquetages obligatoires. Or, comme la définition des paquetages obligatoires n'est pas prise en charge par ce cadre général, de tels paquetages ne peuvent pas être énumérés dans l'attribut *packages* d'un objet géré.

5.1.4 Opération *creationSourceGet*()

L'opération *creationSourceGet*() renvoie une valeur indiquant le système qui a entraîné la création de l'objet. *SourceIndicatorType* est un type énuméré ayant trois valeurs: *resourceOperation*, *managementOperation* et *unknown*. Il indique si l'objet a été créé de façon autonome par la ressource, en réponse à une opération de gestion, ou si le motif de la création de l'objet est inconnu.

5.1.5 Opération *deletePolicyGet*()

L'opération *deletePolicyGet*() renvoie la politique de suppression applicable à cette instance d'objet. Il s'agit d'une valeur énumérée qui indique si l'objet ne peut pas être supprimé, s'il peut être supprimé uniquement s'il ne contient pas d'objet ou si tous les objets contenus seront supprimés lorsqu'il sera supprimé. (Il est interdit de supprimer un objet sans supprimer les objets qu'il contient.) Cette politique est fixée lorsque l'objet est créé par sa fabrique sur la base de l'information de rattachement de nom identifiée dans l'opération de création.

5.1.6 Opération *attributesGet*()

La méthode *attributesGet*() est utilisée pour renvoyer tout ou partie des valeurs d'attribut d'un objet dans une opération unique. Pour chaque interface d'objet géré dans un modèle d'information, on définira un type *valuetype* CORBA contenant les membres de données de chacun des attributs lisibles sur cette interface. (Les attributs lisibles sont ceux dotés d'une opération <attribute name>Get().) Cette méthode peut servir à récupérer ce type de valeur pour n'importe quel objet géré. Les types de valeur seront définis conformément à la hiérarchie d'héritage des interfaces d'objet géré (sauf qu'ils ne peuvent pas prendre en charge l'héritage multiple) et chacun sera finalement déduit du type *ManagedObjectValueType* défini pour l'interface *ManagedObject*. L'objet géré doit renvoyer un type de valeur défini pour son interface en réponse à cette méthode. Ainsi, lorsqu'un client invoque l'opération *attributesGet*() sur un objet géré, il recevra en retour une référence à un type *ManagedObjectValueType* qu'il peut alors réduire au type de valeur défini pour l'interface sur laquelle l'opération a été invoquée.

Quelques complications possibles: un client peut ne pas souhaiter récupérer toutes les valeurs d'attribut d'une instance et une instance peut ne pas prendre en charge tous les attributs qui sont dans des paquetages conditionnels. (Les types de valeur comprennent les attributs qui sont dans des paquetages conditionnels.) Pour cela, on utilise le paramètre in/out *attributeNames*. A l'invocation, le client peut soumettre la liste des noms des attributs par lesquels il est intéressé, une liste sans rien ayant la signification particulière que tous les attributs pris en charge doivent être renvoyés. Les noms de la liste qui sont des noms d'attribut non valides doivent être ignorés par l'objet géré. Dans sa réponse, l'objet renverra la véritable liste des attributs pour lesquels des valeurs sont fournies. Il est à noter que cette liste peut ne pas correspondre à la liste soumise. L'objet doit toujours renvoyer une

liste précise, même si la liste soumise était une liste sans rien ou comprenait des noms non valides. Si tous les noms de la liste soumise sont non valides, l'objet doit renvoyer une liste sans rien et un type de valeur vide.

Comme la structure du type de valeur est prédéfinie, l'objet doit inclure une certaine valeur pour les attributs non demandés ou non pris en charge. Fondamentalement, l'objet peut renvoyer des valeurs quelconques pour ces attributs, mais les valeurs doivent être aussi brèves que possible dans un souci d'efficacité. Ainsi, des valeurs néant doivent être renvoyées pour les chaînes, les références et les listes de toute sorte. On peut renvoyer une valeur quelconque pour les entiers et les types énumérés. Le client doit considérer que toute valeur associée à un attribut non dénommé sur la liste renvoyée par l'objet est non valide.

Actuellement, l'interface de base *ManagedObject* a uniquement une méthode qui renvoie un type de valeur CORBA contenant tous les attributs lisibles de l'objet. Elle ne contient pas de méthode analogue d'écriture pour les attributs car tous les attributs ne peuvent pas être fixés.

5.1.7 Opération *destroy*()

La dernière opération appliquée à l'objet, l'opération *destroy*(), sert à libérer les ressources associées à l'objet géré et à supprimer cet objet. L'objet signale l'anomalie *DeleteError* s'il a pour politique de suppression *NotDeletable*. L'anomalie *DeleteError* est également un moyen extensible de rendre compte de problèmes qui dépendent du modèle lors de la destruction d'un objet. Par exemple, essayer de supprimer un objet point de terminaison de chemin avant que le chemin ne soit supprimé pourrait se traduire par une anomalie *DeleteError*. L'UIT-T Q.816 définit toutefois un service appelé "service de terminaison" pour implémenter la logique nécessaire pour appliquer les politiques de suppression et pour conserver l'intégrité de l'arbre de dénomination. L'opération de destruction est en fait censée être utilisée par ce service et ne doit pas être invoquée directement par un système gérant. Voir l'UIT-T Q.816 pour plus de détails sur le service de terminaison.

(R) OBJET-1. Les interfaces utilisées pour modéliser les ressources d'un système géré héritent (directement ou indirectement) de l'interface *ManagedObject* décrite ci-dessus et définie dans le code IDL CORBA de l'Annexe A. Les capacités décrites ci-dessus sont prises en charge.

5.2 Fabrique d'objets gérés

Les objets gérés sont créés soit automatiquement par le système géré, soit à la suite d'une action sur un autre objet (par exemple un objet de connexion croisée qui est créé en réponse à une action de connexion sur une fabrique), soit en réponse à une demande émanant d'un gestionnaire de créer un objet. Dans ce dernier cas, dans les systèmes CMIP, l'opération de création est généralement prise en charge par le cadre général d'agent CMIP. Elle ne peut pas être prise en charge par l'objet proprement dit car celui-ci n'a pas encore été créé. Dans les implémentations CORBA, il n'existe pas de cadre général d'agent, de sorte qu'il est nécessaire d'avoir quelque chose dans le système géré qui permette au système gérant de créer des objets. Dans les systèmes CORBA, ce sont souvent des objets "fabrique" qui se chargent de cette création. L'interface *ManagedObjectFactory* (fabrique d'objets gérés) est censée être l'interface de base dont les autres interfaces de fabrique héritent. Elle définira les capacités que toutes les fabriques d'objets gérés sont censées prendre en charge. Actuellement, aucune capacité de ce genre n'a été identifiée, de sorte que l'interface est sans rien (elle n'hérite de rien et n'a ni attributs ni méthodes). Cette interface est un endroit où des capacités pourront être placées dans l'avenir si nécessaire. Elle sert aussi de superclasse commune à toutes les fabriques.

Les modèles d'information IDL CORBA sont censés inclure une interface de fabrique par interface d'objet géré (sauf si la classe d'objets gérés n'est pas instanciable). Les fabriques contiendront des opérations permettant de créer des objets gérés. Ces opérations auront un certain nombre de paramètres, tels que l'objet supérieur du nouvel objet, le nom du nouvel objet et les valeurs de

chacun des attributs fixés par création ou pouvant être modifiés, etc. Dès que la création du nouvel objet a abouti, la fabrique lui renverra une référence.

En plus de la création d'objets, les fabriques devraient aussi créer des rattachements de nom dans le service de dénomination CORBA pour les nouveaux objets. Cette fonctionnalité pourrait être implémentée ailleurs, mais on estime que le fait de l'implémenter dans les fabriques simplifiera les implémentations en évitant aux objets gérés d'implémenter cette tâche, ce qui leur permet de s'attacher à la représentation des ressources. Voir l'UIT-T Q.816 pour plus de détails sur la manière dont le cadre général CORBA du RGT utilise le service de dénomination CORBA.

Pour aider les clients à trouver des fabriques, l'UIT-T Q.816 définit un service de recherche de fabrique. Ce service joue le rôle de courtier entre les clients et les fabriques. Le principe est le suivant: les fabriques s'enregistrent auprès du service, puis les clients interrogent ce service pour trouver une fabrique d'un type particulier. Voir l'UIT-T Q.816 pour plus de détails sur le service de recherche de fabrique.

(R) FABRIQUE-1. Les objets de fabrique utilisés pour créer des objets gérés dans un système géré héritent (directement ou indirectement) de l'interface *ManagedObjectFactory* décrite ci-dessus et définie dans le code IDL CORBA de l'Annexe A.

(R) FABRIQUE-2. Toutes les fabriques sont enregistrées dans le ou les objets de recherche de fabrique instanciés dans ce système.

5.3 Interface des notifications

La troisième interface définie dans l'Annexe A est l'interface des notifications. Chacune des notifications définies dans l'UIT-T X.721 a une opération correspondante sur cette interface. Les notifications sont définies comme étant des appels de méthode typés, comme requis par l'UIT-T Q.816. Le service de notification OMG sert à filtrer et à diffuser les notifications. Les méthodes de notification typée peuvent être utilisées directement avec un service de notification qui prend en charge les notifications typées. Les mappages entre ces méthodes d'événement typé et les événements structurés figurent dans l'UIT-T Q.816.

Toutes les opérations de notification définies sur cette interface transmettent un certain nombre de paramètres, certains étant communs à toutes les notifications. Plusieurs notifications ont des paramètres identiques, mais sont utilisées pour des motifs légèrement différents. Le code IDL de l'interface des notifications ressemble à ce qui suit:

```
interface Notifications {
 void equipmentAlarm (
         in ExternalTimeType                 eventTime,
         in NameType                         source,
         in ObjectClassType                  sourceClass,
         in NotifIDType                      notificationIdentifier,
         in CorrelatedNotificationSetType        correlatedNotifications,
         in AdditionalTextType               additionalText,
         in AdditionalInformationSetType     additionalInfo,
         in ProbableCauseType                probableCause,
         in SpecificProblemSetType           specificProblems,
         in PerceivedSeverityType            perceivedSeverity,
         in BooleanTypeOpt                   backedUpStatus,
         in NameType                         backUpObject,
         in TrendIndicationTypeOpt           trendIndication,
         in ThresholdInfoType                thresholdInfo,
         in AttributeChangeSetType           stateChangeDefinition,
         in AttributeSetType                 monitoredAttributes,
         in ProposedRepairActionSetType      proposedRepairActions,
         in BooleanTypeOpt                   alarmEffectOnService,
         in BooleanTypeOpt                   alarmingResumed,
         in SuspectObjectSetType             suspectObjectList
         );

 ...
}; // end of Notifications interface
```

Les quatorze autres opérations de notification sont analogues à celle donnée ci-dessus. Les noms des 15 notifications définies sont les suivants:

- AttributeValueChange
  (modification de valeur d'attribut)
- CommunicationsAlarm
  (alarme de communication)
- EnvironmentalAlarm
  (alarme environnementale)
- EquipmentAlarm
  (alarme d'équipement)
- IntegrityViolation
  (violation de l'intégrité)
- ObjectCreation
  (création d'objet)
- Object Deletion
  (suppression d'objet)
- OperationalViolation
  (violation opérationnelle)
- PhysicalViolation
  (violation physique)
- ProcessingErrorAlarm
  (alarme pour erreur de traitement)
- QualityofServiceAlarm
  (alarme de qualité de service)
- RelationshipChange
  (modification de relation)
- SecurityViolation
  (violation de sécurité)
- StateChange
  (changement d'état)
- TimeDomainViolation
  (violation du domaine temporel)

Le cadre général CORBA requiert l'emploi d'identificateurs de notification là où ils ne sont pas nécessairement exigés dans d'autres interfaces (c'est le cas dans l'UIT-T X.733). Voici, à titre indicatif, quatre cas possibles dans lesquels le mappage des identificateurs de notification d'alarme entre l'interface élément de réseau/EMS et l'interface EMS/NMS doit avoir lieu:

1) l'élément de réseau utilise toujours des identificateurs de notification et l'objet géré est représenté dans les deux interfaces. Dans ce cas, le système EMS transmet l'alarme (avec son identificateur de notification) au système NMS;

2) l'élément de réseau n'utilise jamais d'identificateur de notification et l'objet géré est représenté dans les deux interfaces. Dans ce cas, le système EMS utilise un compteur interne, inclut cette valeur comme identificateur de notification et transmet l'alarme au système NMS;

3) l'élément de réseau utilise parfois des identificateurs de notification et l'objet géré est représenté dans les deux interfaces. Comme l'identificateur de notification est requis, le système EMS doit définir une valeur si aucune n'est proposée. Il peut être difficile de définir une valeur au niveau du système EMS étant donné que les valeurs d'identificateur de notification doivent être uniques parmi toutes les notifications d'une instance d'objet géré donnée aussi longtemps que la corrélation est significative [1]. Le système EMS doit donc choisir une valeur qui n'est pas en cours d'utilisation dans les alarmes en vigueur et qui ne sera pas utilisée dans des alarmes subséquentes. L'opération est délicate car l'algorithme utilisé pour choisir les valeurs d'identificateur de notification est défini par le système producteur (dans le cas présent, l'élément de réseau).

   Une solution serait que le système EMS fournisse sa propre valeur de l'identificateur de notification pour toutes les alarmes. Pour cela, il faudrait aussi mettre à jour toutes les listes de notification correspondantes, et le système EMS maintiendrait un mappage complet des valeurs d'identificateur de notification d'élément de réseau avec les valeurs d'identificateur de notification de système EMS.

   Une autre solution serait que le système EMS et l'élément de réseau conviennent de prendre en charge des sous-ensembles différents de numéros d'identificateur de notification.

Enfin, le système EMS pourrait fournir son propre numéro sans tenir compte des risques de collision, permettant à ceux-ci de se produire en de rares occasions.

4) une alarme est mappée entre un objet interface élément de réseau/EMS et un objet interface EMS/NMS différent. D'une manière analogue à celle qui précède, le système EMS doit fournir une valeur d'identificateur de notification qui est unique pour l'objet géré EMS/NMS. Les listes de notification correspondantes doivent également être mises à jour.

5.4 Définitions de types de données

Avant les définitions d'interface données dans l'Annexe A, on trouve un certain nombre de définitions de structures et de types de données. La plupart sont utilisées dans les notifications. Elles ont été déduites du module ASN.1 figurant dans l'UIT-T X.721 avec quelques modifications légères en vue d'en simplifier la syntaxe. Lorsque c'était possible, on a employé des concepts modernes orientés objet (par exemple paramètres in/out et anomalies), qui sont reflétés dans les types en question.

L'un des types de données à relever est le type temps. Les directives définies ici adoptent le code temporel universel défini pour le service temporel CORBA. Ce type de données est constitué d'un entier long servant à compter les centaines de nanosecondes qui se sont écoulées depuis le 15 octobre 1582 à minuit. Afin de tenir compte du temps dans le monde, la date est exprimée par rapport à la date dans le fuseau horaire de Greenwich en utilisant un entier court signé pour la différence. Cela signifie que les systèmes fondés sur ces directives doivent connaître leur fuseau horaire local. Cette approche facilite néanmoins la comparaison des dates, étant donné que celles-ci sont représentées sous la forme d'un entier. Des bibliothèques normalisées servant à la conversion entre la représentation sous forme d'entier et des formats plus familiers seront probablement largement disponibles.

5.5 Anomalies

Le module IDL donné dans l'Annexe A définit quelques anomalies à utiliser par les opérations d'objet géré. Ces anomalies peuvent être signalées dans le cadre de quelques opérations, comme défini ci-dessous. En outre, n'importe laquelle des anomalies CORBA normalisées peut être signalée dans le cadre de n'importe quelle opération. Par exemple, l'anomalie "CORBA:NO_PERMISSION" pourrait être signalée en cas de violation de sécurité. Les anomalies définies sont les suivantes:

```
valuetype ApplicationErrorInfoType {
         public UIDType          error;
         public Istring          details;
};
valuetype CreateErrorInfoType : ApplicationErrorInfoType {
         public MOSetType        relatedObjects;
         public AttributeSetType attributeList;
};
valuetype DeleteErrorInfoType : ApplicationErrorInfoType {
         public MOSetType        relatedObjects;
         public AttributeSetType attributeList;
};
valuetype PackageErrorInfoType : CreateErrorInfoType {
         public StringSetType    packages;
};
exception ApplicationError { ApplicationErrorInfoType info; };
exception CreateError { CreateErrorInfoType info; };
exception DeleteError { DeleteErrorInfoType info; };
```

5.5.1 Anomalie ApplicationError

Une anomalie *ApplicationError* (erreur d'application) est signalée lorsqu'une opération ne peut pas être achevée en raison d'une certaine condition au niveau application dans le système géré. Les informations renvoyées avec l'anomalie comprennent un identificateur correspondant à une condition particulière et une chaîne donnant des détails additionnels ou une explication.

Quelques identificateurs correspondant à des conditions d'erreur particulières sont définis par le cadre général. Ils doivent être utilisés chaque fois que c'est possible. Les modèles d'information peuvent toutefois définir des codes de condition d'erreur additionnels ou créer leurs propres anomalies.

Les données renvoyées avec l'anomalie d'erreur d'application constituent un type de valeur, ce qui signifie qu'elles peuvent être étendues. Autrement dit, pour certains codes de condition d'erreur, le véritable type de données renvoyé pourrait être une extension du type d'information d'erreur d'application de base. Comme le code d'erreur est dans le type de base, le code de client peut l'examiner et, si sa valeur est l'une de celles qui sont transmises en retour dans une sous-classe, le client peut réduire le type de valeur et accéder aux informations additionnelles.

L'anomalie *ApplicationError* doit être incluse dans la partie *raises* de chaque objet géré et de chaque opération de fabrique d'objet géré. Quelques valeurs de code d'erreur pour l'anomalie d'erreur d'application ont été définies pour le cadre général. Chacune d'elles est abordée dans les sous-paragraphes qui suivent.

5.5.1.1 invalidParameter

Une anomalie d'erreur d'application avec un code d'erreur *invalidParameter* (paramètre non valide) est signalée lorsque la valeur d'un certain paramètre d'opération n'est pas valide pour l'opération demandée. Le nom du paramètre non valide est renvoyé dans le champ details (détails).

5.5.1.2 resourceLimit

Une anomalie d'erreur d'application avec un code d'erreur *resourceLimit* (limite de ressource) est signalée lorsqu'une opération ne peut pas être achevée en raison d'une certaine erreur transitoire dans le système géré (manque de mémoire par exemple). Une chaîne contenant une explication est renvoyée dans le champ details.

5.5.1.3 downstreamError

Une anomalie d'erreur d'application avec un code d'erreur *downstreamError* (erreur en aval) est signalée lorsqu'une opération ne peut pas être achevée en raison d'une erreur en aval du système géré. C'est le cas par exemple lorsqu'une opération ne peut pas être achevée car un système EMS ne peut pas communiquer avec un élément de réseau.

5.5.2 Anomalie *CreateError*

L'anomalie *CreateError* (erreur de création) est signalée lorsqu'une erreur se produit dans le cadre d'une opération de création de fabrique. Elle doit être incluse dans la partie *raises* de chaque opération de création de fabrique d'objets gérés.

Les données renvoyées avec cette anomalie étendent celles d'une anomalie *ApplicationError* générale et ajoutent une liste associée à l'objet connexe ainsi que les valeurs d'attribut que l'objet aurait eues s'il avait été créé. Les codes d'erreur particuliers pour cette anomalie définis par ce cadre général sont présentés ci-dessous. Les implémentations doivent les utiliser chaque fois que c'est possible. Les modèles d'information peuvent ajouter de nouvelles valeurs ou définir de nouvelles anomalies pour des cas particuliers.

5.5.2.1 invalidNameBinding

Une anomalie d'erreur de création avec un code d'erreur *invalidNameBinding* (rattachement de nom non valide) est signalée lorsque le rattachement de nom inclus dans l'opération de création ne prend pas en charge la création de l'objet dans cette situation.

5.5.2.2 duplicateName

Une anomalie d'erreur de création avec un code d'erreur *duplicateName* (nom en double) est signalée lorsque le nom inclus dans l'opération de création est déjà utilisé.

5.5.2.3 unsupportedPackages

Une anomalie d'erreur de création avec un code d'erreur *unsupportedPackages* (paquetages non pris en charge) est signalée lorsqu'un ou plusieurs des paquetages demandés n'est pas pris en charge par l'implémentation. Il est à noter que lorsque ce code d'erreur est utilisé, la structure de données renvoyée est en fait une structure *PackagesErrorInfoType*, qui étend la structure *CreateErrorInfoType*. La structure *PackagesErrorInfoType* comprend une liste de paquetages, qui dans ce cas sera constituée des paquetages non pris en charge.

5.5.2.4 incompatiblePackages

Une anomalie d'erreur de création avec un code d'erreur *incompatiblePackages* (paquetages incompatibles) est signalée lorsque certains paquetages demandés ne sont pas compatibles entre eux ou avec la ressource pour laquelle l'objet est créé. Il est à noter que lorsque ce code d'erreur est utilisé, la structure de données renvoyée est en fait une structure *PackagesErrorInfoType*, qui étend la structure *CreateErrorInfoType*. La structure *PackagesErrorInfoType* comprend une liste de paquetages, qui dans ce cas sera constituée des paquetages incompatibles.

5.5.3 Anomalie *DeleteError*

L'anomalie *DeleteError* (erreur de suppression) est signalée lorsqu'une erreur se produit dans le cadre d'une opération de suppression. Elle est incluse dans la partie *raises* de l'opération de destruction sur l'interface *ManagedObject* de base, qui est ensuite héritée par chaque objet géré.

Les données renvoyées avec cette anomalie étendent celles d'une anomalie *ApplicationError* générale et ajoutent une liste associée à l'objet connexe ainsi que les valeurs d'attribut que l'objet avait lorsque la tentative de suppression a été faite. Les codes d'erreur particuliers pour cette anomalie définis par ce cadre général sont présentés ci-dessous. Les implémentations doivent les utiliser chaque fois que c'est possible. Les modèles d'information peuvent ajouter de nouvelles valeurs ou définir de nouvelles anomalies pour des cas particuliers.

5.5.3.1 notDeletable

Une anomalie d'erreur de suppression avec la valeur de constante *notDeletable* (ne pouvant pas être supprimé) est signalée en cas de tentative d'invocation de l'opération destroy() sur un objet géré qui ne doit pas être détruit conformément à sa politique de suppression. (Il est à noter que l'opération d'objet géré destroy() est définie pour être utilisée par d'autres parties du cadre général. Les systèmes gérants qui l'invoquent directement courent le risque de corrompre les données dans le système géré.)

En outre, le service de terminaison signalera cette anomalie lorsqu'un client essaiera de supprimer un objet ayant la politique de suppression *notDeletable*.

5.5.3.2 containsObjects

Une anomalie d'erreur de suppression avec la valeur de constante *containsObjects* (contient des objets) est signalée en cas de tentative de suppression d'un objet géré qui a des subordonnés et la politique de suppression *deleteOnlyIfNoContainedObjects* (suppression uniquement en cas d'absence d'objet contenu).

Ce ne sont pas les objets gérés qui sont responsables de la détection de cette condition, mais le service de terminaison.

5.6 Définitions de macros

Dans l'Annexe A, après les définitions d'interface, on trouve les définitions de certaines macros. Ces macros fournissent simplement des notations abrégées pour indiquer quelles notifications sont prises en charge par quels objets. En raison de la capacité limitée du langage IDL CORBA d'accepter ce genre d'indications, on a estimé que ces macros seraient utiles.

La macro *MandatoryNotification* (notification obligatoire) indique les notifications qui doivent obligatoirement être prises en charge par un objet et la macro *ConditionalNotification* (notification conditionnelle) indique les notifications qui doivent absolument être émises par un objet géré si celui-ci prend en charge un paquetage particulier. Les deux macros ont des arguments identifiant le nom d'une opération (on rappelle que les opérations sont utilisées pour acheminer des notifications) et le nom limité en portée de l'interface sur laquelle l'opération est définie. La macro *ConditionalNotification* accepte aussi un troisième paramètre, le nom du paquetage auquel la notification appartient.

Les macros de notification ne sont pas extensibles. Malheureusement, le langage IDL est simplement trop limité pour fournir un moyen de bien tenir compte de ces informations. Des commentaires peuvent être générés, mais ils sont immédiatement éliminés par le compilateur. Les commentaires formatés, comme ceux utilisés pour générer le HTML, ne peuvent malheureusement pas être utilisés car ils nécessitent une certaine construction IDL associée. On espérait que le modèle en composantes CORBA à venir fournirait une solution, mais les implémentations ne seront pas disponibles à temps pour les directives définies ici. Dans l'avenir, il sera peut-être possible de modifier les macros pour générer de l'IDL compatible avec le modèle en composantes CORBA. Pour le moment, les informations indiquant quelles notifications sont émises par quelles classes d'objets sont toutefois prises en compte par ces macros.

5.7 Définitions de constantes

Les spécifications d'interface contiennent toujours un certain nombre de constantes dont les valeurs signifient la même chose pour tout le monde. Par exemple, pour tout le monde, un "1" dans un certain champ signifie une perte de signal, un "2" signifie une perte de trame, etc. L'UIT-T X.721 ne fait pas exception et définit un certain nombre de constantes. Celles-ci sont reproduites en format IDL dans l'Annexe B. Pour plus de détails sur le mécanisme utilisé pour acheminer des constantes prédéfinies, voir le paragraphe 6.11.

6 Directives pour la modélisation des informations

Le présent paragraphe contient des directives pour l'élaboration de modèles d'information du RGT fondés sur CORBA. Le paragraphe qui suit contient des directives pour la traduction des modèles existants spécifiés en GDMO.

6.1 Modules

Les modules IDL sont utilisés pour regrouper des interfaces, des définitions de type, des anomalies et d'autres constructions IDL. Ils fournissent aussi une délimitation de l'espace de noms; les identificateurs contenus dans un module donné doivent être uniques mais ils peuvent être réutilisés dans d'autres modules. Dans presque tous les cas, il faut utiliser un module pour regrouper les constructions utilisées pour spécifier un modèle d'information. Des modules peuvent être imbriqués dans d'autres modules et des modules peuvent s'étendre sur plusieurs fichiers. Le code IDL spécifié dans ces directives est contenu dans un seul module, appelé "itut_x780":

```
module itut_x780 {
…
}; // end of module itut_x780
```

Ce module comporte des sous-modules pour les définitions de constantes.

6.2 Interfaces

Il faut qu'une interface IDL soit définie pour chaque *entité* accessible via l'interface de gestion de réseau CORBA. Les interfaces regroupent un ensemble d'attributs et de méthodes qui peuvent être considérés comme étant fournis par un seul objet logiciel. Elles peuvent hériter des capacités d'autres interfaces et les interfaces définies pour modéliser une *entité* doivent hériter (directement ou indirectement) de l'interface appelée *ManagedObject* définie dans la présente Recommandation. Par exemple:

```
interface Equipment : ManagedObject {
…
}; // end of interface Equipment
```

Ces interfaces sont appelées "interfaces d'objets gérés". Les objets qui prennent en charge ces interfaces sont des "objets gérés". Comme l'interface *ManagedObject* définie dans la présente Recommandation a un ensemble de capacités qui sont héritées par toutes les interfaces d'objets gérés, chaque objet géré doit implémenter un ensemble de base de fonctions devant exister dans le cadre général CORBA du RGT.

L'un des problèmes auxquels les modélisateurs peuvent être confrontés est la prise en charge limitée de l'héritage multiple par CORBA. Une interface peut hériter une opération ou un attribut de plusieurs superclasses uniquement si celles-ci l'héritent à leur tour de la même superclasse. C'est ce qu'on appelle l'héritage "en losange", représenté sur la Figure 3 ci-dessous.

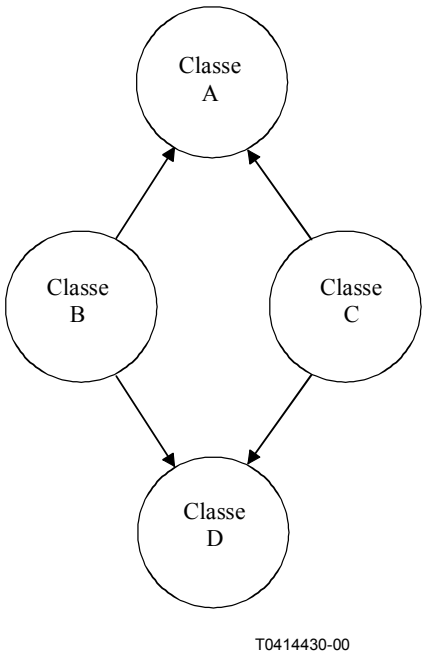

Figure 3/X.780 – Héritage en losange

Si un modélisateur d'information se trouve dans la situation où il doit hériter la même capacité de deux classes différentes qui ne partagent pas de superclasse commune, il sera peut-être amené à modifier les classes et à créer une superclasse virtuelle de laquelle la capacité peut être héritée. Par exemple, pour créer "D" à partir de "B" et de "C" ci-dessus lorsque "A" n'existe pas, le modélisateur devra peut-être modifier les superclasses en créant une nouvelle classe virtuelle ("A") avec la capacité commune, qui est alors héritée par "B" et par "C."

6.3 Attributs

Les attributs sont modélisés dans les interfaces sous forme d'opérations utilisées pour accéder à leur valeur. Le nom de l'opération et les types d'entrée et de résultat indiquent le nom de l'attribut et le type d'opération. (Le langage IDL CORBA prend en charge les attributs en plus des opérations, mais actuellement seules les opérations sont autorisées à signaler des anomalies définies par l'utilisateur. Comme on le verra, des anomalies définies par l'utilisateur sont nécessaires pour l'accès aux attributs. C'est la raison pour laquelle on définit aussi des opérations pour accéder aux attributs et on ne définit pas seulement des attributs. Dans des versions futures de CORBA, on prévoit d'autoriser les anomalies définies par l'utilisateur pour l'accès aux attributs et les directives définies ici pourront être modifiées afin d'en tirer parti.)

6.3.1 Attributs pouvant être lus

Les objets gérés doivent avoir une opération appelée "<attribute name>Get" sur leur interface pour chaque attribut pouvant être lu. Le type de résultat de cette opération reflète le type de l'attribut. Par exemple:

```
AdministrativeStateType administrativeStateGet()
        raises (ApplicationError);
```

Les attributs pouvant être fixés mais ne pouvant pas être lus, ce qui est rare, ne doivent pas avoir d'opération de lecture définie sur l'interface.

Les opérations de lecture d'attribut susceptibles de renvoyer de grandes quantités de données doivent définir un itérateur pour permettre au système client de contrôler le flux d'informations renvoyé. On trouvera un exemple d'utilisation d'itérateurs dans l'UIT-T Q.816.

6.3.2 Attributs pouvant être fixés

Les interfaces d'objets gérés doivent avoir une opération appelée "<attribute name>Set" pour chaque attribut pouvant être fixé. Le type de résultat de l'opération doit être *void* et le paramètre d'entrée doit refléter le type de l'attribut. Par exemple:

```
void administrativeStateSet (in AdministrativeStateType adminState)
        raises (ApplicationError);
```

Les attributs ne pouvant pas être fixés ne doivent pas avoir cette opération sur l'interface.

6.3.3 Attributs ayant un ensemble de valeurs

De nombreux attributs d'objets gérés peuvent contenir des ensembles de valeurs. Dans ces cas aussi, les opérations définies ci-dessus doivent être prises en charge (si l'attribut peut être lu et/ou fixé). Comme CORBA ne définit pas explicitement de type complexe pour les ensembles, les types d'entrée ou de résultat de ces opérations seront des séquences CORBA. Les valeurs renvoyées pour ces attributs ne doivent pas contenir de valeurs en double et l'ordre des valeurs n'est pas important. En outre, il peut être nécessaire de prendre en charge l'ajout ou la suppression de valeurs pour ces attributs. Les opérations correspondantes doivent être appelées "<attribute name>Add" et "<attribute name>Remove". Les types de résultat de ces opérations doivent être *void* et le paramètre d'entrée de chacune doit être une séquence reflétant le type de l'attribut. Par exemple:

```
void supportedByObjectsAdd (in ManagedObjectSetType objects)
        raises (ApplicationError);
void supportedByObjectsRemove (in ManagedObjectSetType objects)
        raises (ApplicationError);
```

6.3.4 Anomalies

Les opérations d'accès aux attributs peuvent en outre signaler les anomalies suivantes:

1) *ApplicationError*. Cette anomalie doit être incluse dans la partie *raises* de chaque opération d'objet géré, y compris les opérations d'accès aux attributs. Elle peut être utilisée pour signaler un certain nombre de situations, par exemple une valeur hors intervalle, une limitation de ressources dans le système géré, etc.

2) Anomalies relatives aux paquetages conditionnels. Si l'attribut fait partie d'un paquetage conditionnel, l'anomalie définie pour ce paquetage doit être incluse dans la partie *raises* des opérations d'accès à l'attribut. Cette anomalie est signalée lors d'une tentative d'accès à l'attribut alors que le paquetage auquel l'attribut appartient n'est pas pris en charge par l'instance. On trouvera plus de détails sur les paquetages conditionnels au paragraphe 6.6 ci-dessous.

Outre ces anomalies, une implémentation peut aussi signaler l'une quelconque des anomalies CORBA normalisées. Les opérations qui signalent des anomalies ne doivent pas modifier la valeur de l'attribut. Donnons un exemple d'opération d'accès à un attribut qui signale une anomalie:

```
void supportedByObjectsRemove (in ManagedObjectSetType objects)
        raises (ApplicationError);
```

6.3.5 Attributs normalisés

Les objets gérés modélisent des ressources et présentent souvent des caractéristiques communes. Cela est parfois représenté au moyen d'une relation d'héritage entre les classes d'objets, mais les objets peuvent aussi présenter des caractéristiques communes sans qu'il n'existe de relation d'héritage. Un bon exemple est celui des attributs analogues. De nombreux objets gérés ont des attributs analogues. Pour faciliter l'implémentation d'interfaces de gestion, les directives données ici définissent certains types de données normalisés qui doivent être utilisés pour les attributs chaque fois que c'est possible. Autrement dit, les modélisateurs doivent tenter d'utiliser ces définitions de type au lieu de définir de nouveaux types. En outre, le nom d'attribut et les noms des opérations

permettant d'accéder à l'attribut doivent être utilisés. En fait, lors de la définition d'un nouveau modèle, il convient de réutiliser les types et noms d'attribut provenant des modèles existants chaque fois que c'est possible. Les attributs normalisés sont définis dans le Tableau 1:

Tableau 1/X.780 – Attributs normalisés

Type de données	Nom de l'attribut	Méthode d'accès
AdministrativeStateType	administrativeState	administrativeStateGet()
AvailabilityStatusSetType	availabilityStatus	availabilityStatusGet()
BackedUpStatusType	backedUpStatus	backedUpStatusGet()
ControlStatusSetType	controlStatus	controlStatusGet()
SourceIndicatorType	creationSource (voir Note)	creationSourceGet()
DeletePolicyType	deletePolicy (voir Note)	deletePolicyGet()
ExternalTimeType	externalTime	externalTimeGet()
NameType	name (voir Note)	nameGet()
ObjectClassType	objectClass (voir Note)	objectClassGet()
OperationalStateType	operationalState	operationalStateGet()
StringSetType	packages (voir Note)	packagesGet()
ProceduralStatusSetType	proceduralStatus	proceduralStatusGet()
StandbyStatusType	standbyStatus	standbyStatusGet()
SystemLabelType	systemLabel	systemLabelGet()
UnknownStatusType	unknownStatus	unknownStatusGet()
UsageStateType	usageState	usageStateGet()
NOTE – Ces attributs sont hérités par tous les objets gérés.		

6.4 Actions

En plus des attributs, de nombreux objets gérés auront des *actions* – méthodes servant à d'autres fins que celle d'accéder à un attribut. Les paramètres et les types de résultat de ces opérations sont simplement définis pour répondre aux besoins de l'action. Le nom de l'opération doit refléter l'objectif de l'opération. Par définition, les anomalies suivantes peuvent être signalées sur des opérations corrrespondant à des actions:

1) *ApplicationError*. Cette anomalie doit être incluse dans la partie *raises* de chaque opération d'objet géré, y compris les opérations correspondant à des actions. Elle peut être utilisée pour signaler un certain nombre de situations, par exemple une valeur de paramètre hors intervalle, une limitation de ressources dans le système géré, etc.

2) Anomalies relatives aux paquetages conditionnels. Si l'action fait partie d'un paquetage conditionnel, l'anomalie définie pour ce paquetage doit être incluse dans la partie *raises* des opérations corrrespondant à l'action. Cette anomalie est signalée lors d'une tentative d'invocation de l'action alors que le paquetage auquel l'action appartient n'est pas pris en charge par l'instance. On trouvera plus de détails sur les paquetages conditionnels au paragraphe 6.6 ci-dessous.

Outre ces anomalies, une implémentation peut aussi signaler l'une quelconque des anomalies CORBA normalisées. D'autres anomalies propres à l'action peuvent et doivent être définies pour d'autres situations d'erreur. Autre solution: un modèle d'information peut étendre les points de code d'erreur définis pour l'anomalie *ApplicationError*.

Les actions susceptibles de renvoyer de grandes quantités de données doivent définir un itérateur pour permettre au système client de contrôler le flux d'informations renvoyé. On trouvera un exemple d'utilisation d'itérateurs dans l'UIT-T Q.816.

6.5 Notifications

Il est prévu que la plupart des objets gérés émettent des notifications dans certaines conditions. Dans le cadre général CORBA du RGT, les notifications sont acheminées par des invocations de méthode depuis un objet géré vers un système gérant en retour, avec l'aide du service de notification. Ainsi, l'opération de notification est véritablement définie pour l'interface CORBA du système gérant et non pour l'interface de l'objet géré. Les directives données ici définissent un certain nombre de notifications normalisées, mais si une nouvelle notification doit être définie, elle doit l'être sous forme d'opération sur une interface appelée "Notifications" dans le module du modèle d'information. Le nom de l'opération doit être le nom de la notification. Les paramètres de l'opération doivent refléter les données qui doivent être rapportées dans la notification. Le type de résultat de l'opération de notification doit être void et il ne doit avoir que des paramètres "in". Il est à noter que le mot clé "oneway" précédant la définition de l'opération de notification ne doit pas être utilisé. Les notifications conformes à ces directives sont confirmées. Autrement dit, lorsqu'un objet géré envoie une notification sur un canal, la réception de cette notification sera confirmée en retour à l'objet géré par le canal. De même, dès que le canal envoie la notification à chaque destinataire, une confirmation est reçue par le canal. Les garanties de qualité de service, spécifiées dans l'UIT-T Q.816, définissent la fiabilité du canal proprement dit. Ainsi, la remise des notifications aux destinataires est garantie.

Par ailleurs, il est nécessaire d'avoir un moyen permettant d'indiquer quels objets gérés émettent quelles notifications. Plutôt que de simplement noter cela par le biais de commentaires dans un fichier IDL, on utilise une déclaration de macro. En réalité, ces directives définissent deux macros, l'une à utiliser lorsque la notification est obligatoire et l'autre lorsque la notification fait partie d'un paquetage conditionnel. Les macros sont censées être utilisées sur une interface d'objet géré et sont définies comme suit:

```
MANDATORY_NOTIFICATION(<interface name>,
        <notification operation name>);
CONDITIONAL_NOTIFICATION(<interface name>,
        <notification operation name>, <package name>);
```

Par exemple:

```
interface Equipment : ManagedObject {
…
MANDATORY_NOTIFICATION(itut_x780::Notifications, objectCreation);
CONDITIONAL_NOTIFICATION(itut_x780::Notifications,
        equipmentAlarm, equipmentAlarmPackage);
…
}; // end of Equipment interface
```

Le nom de paquetage utilisé dans la macro de notification conditionnelle est le même que celui qui est utilisé ailleurs. Voir le paragraphe 6.6 pour plus de détails sur les paquetages. En réalité, les macros ne sont pas extensibles car il n'y a pas vraiment de bonne solution en IDL CORBA. Ainsi, les macros servent à des fins d'indication et ne se traduisent pas par une génération de code. Un point appelle un complément d'étude: la modification des macros pour générer du code IDL qui identifierait les notifications prises en charge par un objet. La publication de la spécification du modèle en composantes CORBA sera l'occasion de procéder à cette modification d'une manière qui

soit compatible avec ce modèle. Une seule notification peut être énumérée dans chaque macro et ce, afin de simplifier la future modification éventuelle des macros.

6.6 Paquetages conditionnels

Les directives définies ici pour la modélisation d'informations prennent en charge la notion selon laquelle les capacités définies pour une classe d'objets gérés n'ont pas besoin d'être toutes prises en charge par toutes les instances. En fait, on peut définir des groupes de capacités tels que, soit toutes les capacités sont prises en charge, soit aucune ne l'est. Ces groupes de capacités sont appelés *paquetages*. En ce qui concerne la représentation des paquetages en IDL, les choix sont limités. Définir une interface distincte pour chaque paquetage conduirait à un trop grand nombre de paquetages: on utilise donc l'approche décrite ici.

Chaque opération qui fait partie d'un paquetage conditionnel peut signaler une anomalie définie pour le paquetage. Le nom de l'anomalie doit être NO<package_name>. Par exemple:

```
exception NOadministrativeStatePackage {};
…
AdmininstrativeStateType administrativeStateGet()
        raises (NOadministrativeStatePackage);
```

Les notifications qui sont émises dans le cadre d'un paquetage conditionnel sont repérées par la déclaration *CONDITIONAL_NOTIFICATION* comme décrit ci-dessus.

Les règles précisant dans quel cas les capacités incluses dans un paquetage doivent être prises en charge et dans quel cas elles ne doivent pas l'être sont énoncées dans des commentaires associés à l'interface d'objet géré. On peut inclure une opération dans plusieurs paquetages conditionnels en énumérant plusieurs anomalies NO<package name> dans la partie *raises* de ladite opération. Une anomalie sera signalée uniquement si aucun des paquetages n'est présent puis l'une quelconque des anomalies relatives aux paquetages peut être signalée. Si une opération est obligatoire, elle ne doit contenir aucune anomalie relative aux paquetages dans sa partie *raises*. Une notification peut énumérer plusieurs paquetages dans la macro *CONDITIONAL_NOTIFICATION*.

6.7 Comportement

Il manque, en IDL CORBA, un moyen formel permettant de tenir compte du comportement des objets. Dans l'avenir, il est possible que les modèles d'information utiliseront le langage UML et incluront des cas d'utilisation et des diagrammes d'interaction entre les objets. Le langage IDL est, quant à lui, limité aux commentaires. Par conséquent, lorsque c'est nécessaire ou utile, il faut utiliser des commentaires pour décrire le comportement des objets.

Le code IDL figurant dans la présente Recommandation contient un certain nombre de commentaires. Ceux-ci sont formatés de manière à pouvoir être analysés par les compilateurs utilisés pour la conversion IDL vers HTML afin d'en faciliter la lecture. Un commentaire formaté commence par /** et se termine par */ et il est associé à la construction IDL suivante. Les étiquettes de formatage HTML sont autorisées avec ces commentaires, tout comme certains mots clés (précédés par le symbole '@') qui sont convertis par les compilateurs IDL vers HTML dans un formatage additionnel. L'affichage du code IDL avec un programme de lecture HTML est pratique, mais il est à noter que l'utilisation des macros décrites ci-dessus en est affectée. Comme l'extension de macro est réalisée dans le cadre de la conversion vers HTML, les informations de macro qui existaient avant l'extension seront perdues. Ainsi, les macros utilisées pour identifier les notifications prises en charge par chaque objet géré auront été étendues.

6.8 Informations de rattachement de nom

La contenance est une relation très importante dans la gestion de réseau. Dans le cadre général fondé sur CORBA du RGT, elle est représentée par l'intermédiaire de noms. Malheureusement, cela n'impose aucune restriction aux relations de contenance qui peuvent éventuellement exister. Rien n'empêche, par exemple, un objet de réseau d'être contenu dans un objet de connexion. Il est clairement souhaitable de disposer d'un moyen permettant de restreindre les relations de contenance possibles à celles qui sont sensibles. Ces restrictions doivent toutefois être extensibles sous le contrôle du modélisateur d'information.

Pour répondre à ces besoins, les directives définies ici nécessitent que les modules IDL spécifiant des modèles d'information du RGT fondés sur CORBA contiennent aussi des informations définissant les relations de contenance possibles parmi les classes d'objets gérés. Ces informations relatives aux relations de contenance sont appelées *informations de rattachement de nom d'objet géré*. (Malheureusement, cette appellation peut facilement être confondue avec les informations de rattachement de nom enregistrées dans le service de dénomination CORBA. Il y a une différence entre les deux.)

Les informations de rattachement de nom d'objet géré sont représentées en IDL CORBA au moyen des conventions suivantes:

1) chaque module IDL de modèle d'information doit contenir un sous-module appelé "NameBindings" pour les informations de rattachement de nom d'objet géré;

2) dans ce module de rattachement de nom, des sous-modules doivent être définis pour chaque relation de contenance autorisée;

3) chaque sous-module de rattachement de nom doit assigner des valeurs aux 7 constantes suivantes:

```
const string            superiorClass
const boolean           superiorSubclassesAllowed
const string            subordinateClass
const boolean           subordinateSubclassesAllowed
const boolean           managersMayCreate
const DeletePolicyType  deletePolicy
const string            kind
```

La constante *superiorClass* contient le nom de classe limité en portée de l'objet (contenant) supérieur. Si un objet peut être l'objet "sommital" dans un système géré, autrement dit, s'il peut être contenu directement sous un contexte de dénomination de racine local, la valeur de rattachement de nom *superiorClass* doit être une chaîne vide. La constante *superiorSubclassesAllowed* est un champ booléen qui aura la valeur *true* si des sous-classes du type de classe supérieure sont acceptables en utilisant ce rattachement de nom. La constante *subordinateClass* contient le nom de classe limité en portée de l'objet subordonné (l'objet devant être créé). La constante *subordinateSubclassesAllowed* indique si des sous-classes de l'objet subordonné peuvent être créées au moyen de ce rattachement de nom. Le fanion *managersMayCreate* indique si la création d'objet est assurée au niveau de l'interface de gestion au moyen de ce rattachement de nom. L'intérêt de mettre ce fanion à *false* est que cela permet d'indiquer en IDL toutes les informations relatives aux relations de contenance, même si l'objet subordonné n'est créé que par le système géré. La constante *deletePolicy* contient la valeur qui sera assignée à l'attribut *deletePolicy* de l'objet géré au moment de sa création. La constante *kind* contient la valeur qui sera assignée au champ *kind* dans le rattachement de nom CORBA pour l'objet au moment de sa création.

La valeur choisie pour le champ *kind* dans un rattachement de nom sera généralement le nom de classe de l'objet subordonné non limité en portée (on utilisera généralement des noms de classe non limités en portée pour réduire la longueur des noms). L'objet principal du champ *kind* est de segmenter l'espace de dénomination pour éviter des collisions de dénomination. Des modules de corrélation de noms pour les nouvelles versions des

interfaces existantes pourraient réutiliser les valeurs de *kind* utilisées pour les anciennes interfaces. A titre d'exemple, les modules de corrélation de noms pour les interfaces Equipment et Equipment R1 peuvent toutes deux utiliser la valeur "Equipment". Mais il sera toutefois plus sûr d'utiliser une valeur unique pour chaque classe d'interface.

4) Le nom d'un sous-module de rattachement de nom sera <subordinateClass>_<superiorClass>, où <subordinateClass> est la valeur assignée à la constante *subordinateClass* et <superiorClass> est la valeur assignée à la constante *superiorClass* dans le module. Si deux modules de rattachement de nom dans le même module parent partagent les mêmes valeurs *superiorClass* et *subordinateClass* mais diffèrent pour d'autres valeurs, on ajoutera au nom de l'un des modules un mot indiquant une différence entre les deux. Par exemple: "Equipment_Equipment" et "Equipment_Equipment_NotDeleteabe".

Donnons quelques exemples de rattachements de nom d'objet géré:

```
module itut_m3120 {
…
  /** The following module contains name binding information */
  module NameBindings {
        /** This name binding module allows Equipment objects to be
        created under Managed Element objects.
        */
        module Equipment_ManagedElement {
                const string    superiorClass = "itut_m3120::ManagedElement";
                const boolean   superiorSubclassesAllowed = TRUE;
                const string    subordinateClass = "itut_m3120::Equipment";
                const boolean   subordinateSubclassesAllowed = TRUE;
                const boolean   managersMayCreate = TRUE;
                const DeletePolicyType deletePolicy =
                                itut_x780::DeleteOnlyIfNoContainedObjects;
                const string    kind = "Equipment";
        }; // end of Equipment_ManagedElement name binding module
        /** This name binding module allows Equipment objects to be
        created under other Equipment objects.
        */
        module Equipment_Equipment {
                const string    superiorClass = "itut_m3120::Equipment";
                const boolean   superiorSubclassesAllowed = TRUE;
                const string    subordinateClass = "itut_m3120::Equipment";
                const boolean   subordinateSubclassesAllowed = TRUE;
                const boolean   managersMayCreate = TRUE;
                const DeletePolicyType deletePolicy =
                                itut_x780::DeleteOnlyIfNoContainedObjects;
                const string    kind = "Equipment";
        }; // end of Equipment_Equipment name binding module
  }; end of name binding module
}; end of itut_m3120 module
```

Il est à noter que la constante *deletePolicy* est de type énuméré et, conformément aux règles de définition des constantes en IDL CORBA, si ce type est défini dans un autre module, la portée de la valeur assignée à la constante est obligatoirement limitée à ce module. Le type *DeletePolicyType* est défini dans le module itut_x780 et le module IDL donné en exemple est itut_m3120. Par conséquent, il faut limiter la portée de la valeur *DeleteOnlyIfNoContained* en la faisant précéder de la chaîne "itut_x780::". La portée du type proprement dit, *DeletePolicyType*, doit aussi être limitée. Pour cela, on peut insérer une déclaration *typedef* au début du module.

6.9 Fabriques

Le cadre général fondé sur CORBA du RGT définit un service pour la suppression d'objets, mais les objets sont créés au moyen de *fabriques* propres à la classe. Les fabriques sont des objets avec des interfaces qui sont distincts des objets qu'elles ont l'habitude de créer, mais généralement connexes. Chaque classe d'objets gérés aura une classe de fabrique. On procède de manière à ce que les

opérations de création de fabrique puissent être fortement typées et propres à la classe associée aux objets qu'elles créent. Le résultat est que les modules IDL définissant des interfaces d'objets gérés contiendront aussi des interfaces pour les fabriques utilisées pour créer les objets. Le nom de l'interface IDL de la fabrique sera "<Managed Object Class Name>Factory".

La présente Recommandation définit une interface de fabrique d'objet géré de base de laquelle chaque interface de fabrique doit hériter. Les fabriques ne suivent pas la même hiérarchie d'héritage que les objets qu'elles créent. Elles héritent simplement de l'interface *ManagedObjectFactory*. Donnons un exemple de définition d'interface de fabrique:

```
interface EquipmentFactory : ManagedObjectFactory {
…
}; // end of EquipmentFactory interface
```

Comme les fabriques ne peuvent pas créer de sous-classes d'objets, il faut définir de nouvelles fabriques pour chaque sous-classe.

Une fabrique doit être définie pour chaque classe instanciable, même si, pour le moment, il n'est pas défini de module de rattachement de nom permettant aux gestionnaires de créer des instances. Le but est de pouvoir définir dans l'avenir des modules de rattachement de nom qui autorisent les gestionnaires à créer des instances.

6.9.1 Opérations de création

Chaque interface de fabrique doit définir une opération unique à utiliser par les clients pour créer des objets. Cette opération doit avoir pour nom "create" (création) et doit renvoyer une référence au type d'objet créé par la fabrique. Les quatre premiers paramètres de chaque opération de création sont toujours les mêmes. Viennent ensuite des paramètres pour chaque attribut fixé par création ou pouvant être modifié qui est défini pour l'objet géré. (Un attribut fixé par création est un attribut pour lequel l'objet n'a pas d'opération "set" (écriture), mais pour lequel une valeur est spécifiée dans l'opération de création.) Les noms de ces paramètres sont identiques à celui de l'attribut. (Il s'agit du nom d'une opération d'accès à l'attribut moins le suffixe "Get" ou "Set".) Chaque opération de création doit aussi accepter des paramètres permettant de fixer les valeurs de n'importe quel attribut fixé par création ou pouvant être modifié de toutes les superclasses de l'objet créé par la fabrique. Donnons un exemple d'opération de création pour une fabrique d'équipement:

```
Equipment create(
        in NameBindingType nameBinding, // module name containing NB info.
        in ManagedObject superiorObject,// Reference to containing object.
        inout string name,              // In/out, may be null if auto-create.
        in StringSetType packages,      // List of packages requested.
        …                               // Writeable and set-by-create values
                                        // for Equipment superclass attributes.
        …                               // Writeable and set-by-create values
                                        // for Equipment attributes.
);
```

6.9.1.1 Rattachement de nom

Le paramètre de rattachement de nom achemine le nom d'un module contenant les informations de rattachement de nom d'objet géré, comme décrit au paragraphe 6.8. La valeur pourrait par exemple être "itut_m3120::NameBindings::Equipment_Equipment". Cela étant, la fabrique peut vérifier si la valeur est un identificateur de rattachement de nom valide. (Une fabrique pourrait être "codée matériellement" avec les informations de rattachement de nom disponibles lorsque le système est compilé ou elle pourrait accéder aux informations situées dans le dépôt d'interface CORBA au moment de l'exécution.) Si les informations de rattachement de nom ne sont pas trouvées, la fabrique doit signaler une anomalie *invalidParameter ApplicationError*, renvoyant "nameBinding" (rattachement de nom) comme argument. (Il s'agit d'une anomalie *ApplicationError* avec le code *error* mis à *invalidParameter* et la chaîne *details* mise à "nameBinding".) Si les informations de

rattachement de nom sont trouvées, mais sont incomplètes, la fabrique doit signaler une anomalie *invalidNameBinding CreateError*.

La fabrique doit aussi vérifier si le type de classe subordonnée spécifié dans le module de rattachement de nom correspond au type d'objets qu'elle crée. Si ce n'est pas le cas, la fabrique peut vérifier si le type d'objets qu'elle crée est une sous-classe de la valeur de la constante de classe subordonnée. Si c'est le cas et si la constante *subordinateSubclassesAllowed* a la valeur *true*, la fabrique peut procéder à la création de l'objet. Sinon, elle rejette la demande en signalant une anomalie *invalidNameBinding CreateError*.

Enfin, si la constante *managersMayCreate* dans le module de rattachement de nom vaut *false*, la fabrique rejette aussi la demande en signalant une anomalie *invalidNameBinding CreateError*. (Les fabriques peuvent avoir une deuxième opération de création destinée à une utilisation interne par le système géré qui ne vérifie pas cette valeur et qui n'est pas exposé via l'interface de gestion.) L'inclusion de modules de rattachement de nom avec des valeurs *managersMayCreate* mises à *false* permet d'indiquer en IDL toutes les informations de contenance, comme cela est possible avec GDMO, même si les objets sont créés uniquement par le système géré proprement dit.

Les autres informations figurant dans le module de rattachement de nom seront utilisées par la fabrique au moment où cette dernière crée l'objet et le rattachement de nom du service de dénomination CORBA associé. La constante *deletePolicy* sera assignée à l'attribut du nouvel objet géré du même nom. La valeur de la constante *kind* sera utilisée lorsque la fabrique créera le rattachement de nom de l'objet géré dans le service de dénomination CORBA.

6.9.1.2 Objet supérieur

Le deuxième paramètre de l'opération de création est une référence à l'objet supérieur, dans le cadre duquel le nouvel objet doit être créé. En utilisant les capacités CORBA normalisées, la fabrique doit examiner la classe de l'objet supérieur pour déterminer si elle correspond au type spécifié dans la constante *superiorClass* définie dans le module de rattachement de nom. Si elle ne correspond pas, la fabrique doit ensuite vérifier si la référence fournie est une référence à une sous-classe du type spécifié dans la constante *superiorClass*. Si c'est le cas et si la constante *superiorSubclassesAllowed* figurant dans le rattachement de nom vaut *true*, la fabrique peut procéder à la création de l'objet. Dans le cas contraire, la fabrique doit rejeter la demande en signalant une anomalie *invalidNameBinding CreateError*, renvoyant "superiorObject"(objet supérieur) dans le champ *details*.

Si la constante *superiorClass* figurant dans le module de rattachement de nom est une chaîne vide, les objets de la classe subordonnée peuvent être créés sans objet supérieur (parent) et leur nom est lié directement à un contexte de dénomination de racine local. Généralement, ces objets seront créés par le système géré, mais dans ces cas, la référence à l'objet supérieur vaudrait néant.

6.9.1.3 Nom

Le troisième paramètre est le nom à assigner au nouvel objet. Cette chaîne deviendra le champ *ID* du rattachement de nom CORBA créé dans le service de dénomination CORBA pour le nouvel objet. Elle se rapportera au nom de l'objet supérieur. Si le paramètre est *inout*, il indique que la fabrique doit assurer l'autodénomination. Dans ce cas, un client peut soumettre une chaîne valant néant pour le nom et la fabrique choisira une chaîne adaptée et renverra la valeur choisie. Si, au contraire, le client soumet une chaîne, la fabrique utilisera alors cette valeur (et la renverra comme valeur *out*). Si le paramètre est uniquement *in*, l'autodénomination n'est pas assurée et le client doit fournir un nom. S'il ne le fait pas, la fabrique doit signaler une anomalie *badName CreateError*. La fabrique signale une anomalie *duplicateName CreateError* si le nom fourni est un nom en double. (Cela signifie que les champs *ID* et *kind* correspondent tous deux à un objet existant contenu dans l'objet supérieur.)

6.9.1.4 Paquetages

L'attribut de paquetages est important. Il indique à la fabrique non seulement quels paquetages une instance doit prendre en charge, mais aussi quelles valeurs de paramètre relatives à l'opération de création elle doit ignorer. Comme elles sont fortement typées, les méthodes de création comprennent un paramètre pour chaque attribut d'un objet fixé par création ou pouvant être modifié, même si un attribut fait partie d'un paquetage conditionnel. La fabrique doit ignorer les valeurs d'un attribut figurant dans un paquetage qui n'est pas demandé par le client, même si elle instancie tout de même l'objet avec le paquetage. (Si la fabrique instancie un objet avec un paquetage non demandé par le client, elle doit choisir les valeurs initiales.) Cela évite au client de devoir fournir des valeurs pour des attributs figurant dans des paquetages qu'il ne souhaite pas. En revanche, le client peut soumettre n'importe quelle valeur. Dans un souci d'efficacité, les valeurs soumises pour les attributs figurant dans des paquetages non demandés par le client doivent être brèves.

Si le client fournit un nom de paquetage non valide dans le paramètre de paquetages, la fabrique doit signaler une anomalie *unsupportedPackage CreateError* et renvoyer le nom du paquetage comme argument. Une anomalie *incompatiblePackages CreateError* peut par ailleurs être signalée si le client demande la création d'une instance mais spécifie des paquetages qui ne peuvent pas coexister dans la même instance.

6.9.1.5 Paramètres de superclasse

Après ces quatre premiers paramètres viennent des paramètres pour chacun des attributs fixés par création ou pouvant être modifiés pour toutes les superclasses du type des objets créés par la fabrique.

6.9.1.6 Paramètres de classe d'objets

Enfin, viennent des paramètres pour chacun des attributs fixés par création ou pouvant être modifiés pour la classe d'objets gérés créée par la fabrique.

6.9.2 Recherche de fabrique

Afin de faciliter la tâche de recherche d'une fabrique, l'UIT-T Q.816 définit une interface de recherche de fabrique. (La recherche de fabrique est un modèle de conception commun dans les applications CORBA.) Elle permet à un client de trouver facilement une fabrique en interagissant avec un courtier bien connu ayant connaissance de toutes les fabriques présentes dans un système géré.

6.10 Types de valeur de classe d'objets gérés

Chaque classe d'objets gérés conforme aux directives définies ici hérite une opération de la classe d'objets gérés de base qui renvoie tout ou partie des attributs d'objet dans un seul type de valeur *valuetype*. (CORBA 2.3 introduit le concept de types de valeur, correspondant à des objets qui sont transmis par valeur et non par référence.) Non seulement l'implémentation d'un objet géré doit prendre en charge cette caractéristique, mais le code IDL décrivant l'objet géré doit aussi inclure un type de valeur avec des attributs publics pour chacun des attributs pris en charge par l'objet géré. Ces directives définissent un type *ManagedObjectValueType* de base et les types de valeur définis pour les objets gérés doivent, au bout du compte, dériver de ce type de valeur de base. Les types de valeur définis pour les objets gérés doivent normalement suivre le modèle d'héritage de l'interface d'objet géré, mais comme les types de valeur CORBA ne prennent en charge que l'héritage simple, cela ne sera pas toujours possible. Mais il ne s'agit pas d'une limitation très grave. Cela signifie simplement que les types de valeur définis pour les interfaces utilisant l'héritage multiple devront hériter uniquement de l'un des types de valeur supérieurs et les autres attributs devront être ajoutés et mis à jour à la main.

A titre d'exemple, supposons que l'interface d'objet géré Equipment hérite directement de la classe *ManagedObject* de base et possède, entre autres, une fonction d'accès à un attribut appelée *userLabelGet* qui renvoie un type *UserLabelType*. Le code IDL décrivant le type de valeur pour l'objet géré Equipment serait par exemple:

```
valuetype EquipmentValueType : ManagedObjectValueType {
   public UserLabelType    userLabel;
   …                              // other attributes
};
```

Le nom du type de valeur est le nom de l'interface auquel on ajoute "ValueType". Il est par ailleurs à noter que le nom de l'attribut public dans le type de valeur est le nom de la méthode sur l'interface d'objet géré utilisée pour accéder à l'attribut sans le suffixe "Get". Cette convention doit être respectée pour tous les attributs dans des types de valeur. Le type de l'attribut est le même que le type renvoyé par la fonction d'accès à l'attribut.

Le code pour le côté client souhaitant récupérer les valeurs d'attribut pour un objet Equipment pourrait par exemple être le suivant:

```
ManagedObjectValueType   moValue;
EquipmentValueType       eqValue;
Equipment                eq;
eq = …   // code that sets eq to a CORBA proxy representing an
         // equipment object.
moValue = eq.getAttributes();
eqValue = (EquipmentValueType) moValue;  // cast return to proper type
System.out.println("User Label = " + eqValue.userLabel); // print label
```

Lorsque le code IDL est compilé en langage de programmation orienté objet, les interfaces (dans ce cas, Equipment) ainsi que les types de valeur (*ManagedObjectValueType* et *EquipmentValueType*) seront traduits en classes. Pour les interfaces, les classes sont en fait des proxys. Lorsque des méthodes sont invoquées pour ces classes, elles peuvent utiliser le courtier ORB pour renvoyer la demande au serveur. Les classes traduites à partir de types de valeur ne sont toutefois pas des proxys. Ce sont simplement des objets locaux.

Lorsque le client invoque l'appel au niveau du proxy de l'équipement pour lire des attributs, la réponse du serveur sera un type *EquipmentValueType*. Lorsque le courtier ORB reçoit cette réponse, il crée une instance locale d'un objet *EquipmentValueType* avec les valeurs d'attribut reçues du serveur. Comme le type de résultat de la méthode *attributesGet*(), définie sur l'interface d'objet géré de base, est *ManagedObjectValueType*, la référence à l'instance *EquipmentValueType* est renvoyée sous forme de référence de type *ManagedObjectValueType*. Cela fonctionne car *EquipmentValueType* est déduit de *ManagedObjectValueType*. Afin d'accéder aux attributs qui sont propres au type *EquipmentValueType*, le client doit toutefois réduire la référence au type *EquipmentValueType*.

Le traitement effectué en interne par le courtier ORB est légèrement compliqué, mais l'autre solution consisterait à utiliser des listes de types *any* CORBA pour conserver les valeurs d'attribut. Cette approche nécessiterait toutefois un traitement encore plus élaboré. Les types *any* seraient également beaucoup plus compliqués pour le programmeur. Comme indiqué dans l'exemple ci-dessus, l'utilisation des types de valeur est en réalité relativement simple.

6.11 Constantes

Les systèmes de gestion de réseau ont besoin de pouvoir échanger des informations concernant les significations adoptées précédemment. Par exemple, une notification de changement d'état avec la cause probable "1" pourrait signifier qu'elle a probablement été engendrée par une perte de signal, tandis qu'un "2" pourrait signifier une perte de trame, etc. Il est assez simple de définir une énumération ou un ensemble de valeurs entières à transmettre via une interface dans un certain champ, mais il est un peu plus compliqué de rendre ce mécanisme extensible par plusieurs groupes,

susceptibles d'agir en parallèle. Le mécanisme utilisé par ces directives, pour cela, est appelé "Identificateur universel (UID, *universal identifier*)".

Un identificateur UID est une structure de données comprenant deux champs. Le premier est une chaîne correspondant au nom limité en portée d'un module IDL contenant les constantes définies pour un certain champ. Le second est un entier signé de type *short* (16 bits) contenant la valeur. Par exemple, pour envoyer une valeur de "perte de signal" dans un champ de cause probable, un système construirait une structure UID avec une chaîne *moduleName* valant "itut_x780::ProbableCauseConst" et un entier valant 29. (L'Annexe B contient les constantes définies pour ces directives. Elle comporte un module appelé "ProbableCauseConst" (constante de cause probable) qui contient une constante appelée *lossOfSignal* (perte de signal) ayant pour valeur 29.)

Il est à noter qu'il s'agit de l'unique format pour les valeurs de constante utilisé dans ce cadre général. Aucune valeur "locale" n'est utilisée.

Les conventions suivantes doivent être respectées lors de la définition de constantes pour un modèle d'information:

1) les valeurs de constante doivent être définies dans des modules distincts, un pour chaque ensemble de constantes défini pour un champ donné. Ces sous-modules doivent être inclus dans le module de niveau sommital contenant les autres constructions définies pour le modèle d'information;

2) le nom du module doit être le nom du champ auquel on ajoute "Const". Par exemple, les valeurs du champ *probableCause* (défini en tant que type UIDType) sont incluses dans un module appelé "ProbableCauseConst";

3) les constantes définies dans le sous-module doivent être de type *const short*. Par exemple:

```
const short lossOfSignal = 29;
```

4) les constantes peuvent être conservées dans un fichier distinct, pour réduire la longueur et la complexité du fichier IDL principal. Même si les constantes sont dans un fichier distinct, les sous-modules doivent être dans une déclaration de module IDL portant le même nom que le module du fichier principal. En haut du fichier principal, doit se trouver une déclaration *include* de précompilateur et ce, afin que les constantes soient incluses dans toute compilation;

5) le sous-module doit aussi inclure une constante correspondant à une chaîne appelée "moduleName" qui contient le nom limité en portée de ce module. Par exemple:

```
module itut_x780 {
        …
        module ProbableCauseConst {
                const string moduleName = "itut_x780::ProbableCauseConst";
                …
        }; // end of module ProbableCauseConst
        …
}; // end of module itut_x780
```

En réalité, il s'agit simplement de permettre aux programmeurs de faire référence au nom du module par une constante plutôt que de coder matériellement des noms de chaîne de module.

Il est à noter que d'autres modèles d'information peuvent étendre les valeurs de la cause probable. Il pourrait par exemple y avoir un module "itut_m3120::ProbableCauseConst" avec des valeurs additionnelles pour le champ de cause probable. Ces modules peuvent même réutiliser la valeur 29. L'identificateur UID sera toujours unique car les noms de module seront différents.

6.12 Enregistrement

Le langage IDL CORBA nécessite que tous les identificateurs contenus dans un module donné soient uniques. Cela signifie que, tant qu'un nom de module est unique, l'ensemble de son contenu sera dénommé de façon univoque. Le langage IDL CORBA définit aussi une déclaration *pragma* de compilateur IDL pouvant être utilisée pour définir un préfixe unique pour les identificateurs de module lorsqu'ils sont enregistrés dans le dépôt d'interface CORBA, un dépôt central d'informations relatives aux interfaces et utilisées par les courtiers ORB CORBA. Ce cadre général nécessite que les documents IDL contiennent une déclaration de préfixe de pragma utilisant le nom de domaine Internet de l'organisation comme préfixe pour les modules contenus.

Cela évite de devoir enregistrer chaque construction individuellement.

6.13 Version de spécifications en IDL CORBA

Lorsqu'on utilise CORBA, une interface de gestion est spécifiée comme étant une ou plusieurs interfaces d'objet définies au moyen du langage IDL. Inévitablement, les interfaces de gestion changent. L'ajout d'une nouvelle interface d'objets CORBA à une interface de gestion est immédiat. Il faut simplement définir la nouvelle interface CORBA en IDL et l'ajouter à la spécification identifiant les interfaces d'objet à prendre en charge sur l'interface de gestion considérée.

La mise à jour d'une interface d'objet CORBA existante est toutefois un peu plus difficile. Les directives définies ici imposent une priorité à la compatibilité amont. Les règles données ci-après s'appliquent donc à l'extension d'une interface d'objet géré existante. Il est à noter que ces règles ne s'appliquent qu'aux extensions faites à une classe de base qui ne provoquent pas de modification de l'objectif de l'objet. Autrement dit, la nouvelle classe modélise la même ressource que l'ancienne classe, elle a simplement quelques capacités additionnelles.

1) Le nom de la nouvelle interface d'objet est le même que celui de l'interface existante auquel on ajoute la lettre "R" et un numéro, commençant à "1". Pour les extensions ultérieures, on incrémentera le numéro. Ainsi, l'extension d'une interface pour les objets gérés "Equipment" conduirait à une interface appelée "EquipmentR1."

2) La nouvelle interface est définie au sein du même nom de module que l'interface existante. (Les modules CORBA ne sont en fait que des espaces de noms et peuvent s'étaler sur plusieurs fichiers.)

3) La nouvelle interface hérite de l'interface existante.

4) Les capacités héritées de l'interface existante ne peuvent pas être supprimées ou modifiées dans la nouvelle interface. Si une définition correspondant à une opération doit être modifiée, il faut définir une nouvelle opération. Le nom de la nouvelle opération doit être le même que celui de l'opération existante auquel on ajoute la lettre "R" et un numéro, commençant à "1". Pour les extensions ultérieures, on incrémentera le numéro.

5) La valeur du champ *kind* utilisé dans les rattachements de nom continuera à être déterminée par une constante dans les modules de rattachement de nom donnés en référence lorsque l'objet est créé. Tout rattachement de nom valide pour l'interface existante doit être valide pour la nouvelle interface. Autrement dit, un module de rattachement de nom pour un objet Equipment doit aussi être valide pour un objet EquipmentR1, même si la constante *subordinateSubclassesAllowed* du module a pour valeur *false*.

6) Les références aux nouvelles interfaces doivent être du type le plus spécifique. (Dans le cas contraire, on ne peut accéder aux nouvelles capacités.) De même, la valeur de l'attribut *objectClass* signalée par un objet de la nouvelle classe doit être du type le plus spécifique. CORBA permet de déterminer la classe réelle d'une référence sur la base d'informations contenues dans la référence IOR.

Par exemple, considérons l'interface d'objet suivante:

```
interface Foo {
        void action(in int A, in int B);
}
```

On pourrait étendre l'action de la façon suivante:

```
interface FooR1: Foo {
        void actionR1(in int A, in int B, in int C);
}
```

L'ancienne action serait toujours une opération valide.

Il faut utiliser une méthode analogue – ajouter la lettre "R" et un numéro incrémenté – lorsque d'autres définitions en IDL sont remaniées, notamment les définitions de constante, de type et de valuetype.

7 Traduction de GDMO

Le présent paragraphe contient des directives pour la création de modèles d'information IDL à partir de modèles d'information existants décrits au moyen de GDMO. Les sous-paragraphes qui suivent décrivent la manière dont chacun des gabarits GDMO doit être traduit en IDL CORBA.

7.1 Classes d'objets gérés

Chaque classe d'objets gérés d'une spécification GDMO est traduite en interface d'objet géré. Les traductions de classes d'objets gérés déduites de la classe sommitale GDMO héritent de l'interface IDL CORBA *ManagedObject*. Les traductions de classes non déduites directement de la classe sommitale héritent de la traduction de la classe dont elles sont déduites. Toutes les interfaces d'objets gérés doivent hériter directement ou indirectement de l'interface *ManagedObject*. L'héritage multiple est autorisé sous réserve des règles associées au langage IDL CORBA. Il est à noter toutefois que ces règles diffèrent de celles du protocole CMIP. En particulier, CORBA ne permet pas à un attribut ou à une opération d'être hérité de plusieurs sources sauf si celles-ci l'héritent à leur tour de la même source commune. Si une traduction à partir du protocole CMIP correspondant à un héritage multiple ne respecte pas les règles CORBA, le traducteur devra choisir d'hériter d'une seule superclasse et d'ajouter manuellement les autres capacités provenant de l'autre classe. Une autre solution consiste à modifier les superclasses posant problème de sorte qu'elles héritent la capacité posant problème d'une source commune. Pour cela, il faudrait bien entendu redéfinir ces superclasses.

L'incapacité d'hériter d'une superclasse potentielle signifie aussi qu'un travail manuel peut être nécessaire si la superclasse potentielle ou l'une quelconque de ses superclasses est modifiée. Un problème plus grave est que le polymorphisme CORBA est fondé sur l'héritage. Si la sous-classe n'hérite pas d'une classe, elle ne peut pas lui être polymorphique. Malheureusement, il s'agit d'une limitation de CORBA et non des directives définies ici.

Les attributs, actions et notifications contenus dans des paquetages obligatoires ou conditionnels sont traduits en opérations sur l'interface conformément aux directives données plus loin. Un commentaire précédant l'interface doit décrire les conditions dans lesquelles les capacités d'un paquetage conditionnel doivent être prises en charge par une instance, sur la base de la partie PRESENT IF de ce paquetage. Il est à noter que CORBA ne permet pas de redéfinir une capacité présente dans une superclasse. Par conséquent, si une capacité est définie comme étant conditionnelle dans une superclasse, elle ne peut pas être redéfinie comme étant obligatoire dans une sous-classe. (Comme décrit ci-dessus, les capacités sont indiquées comme étant conditionnelles lorsqu'elles signalent une anomalie *NO<package_name>*. Cette anomalie ne peut pas être supprimée dans une sous-classe. La meilleure autre solution consistera à placer un commentaire indiquant que la sous-classe ne doit pas signaler l'anomalie. Une autre solution consisterait à abandonner l'héritage

et à ajouter manuellement la capacité, celle-ci étant alors rendue obligatoire. Cela peut toutefois conduire à des problèmes de polymorphisme et à une mise à jour manuelle.)

Il n'est pas nécessaire d'enregistrer les interfaces individuellement.

7.2 Paquetages

Malheureusement, le langage IDL ne permet pas de définir des paquetages à un endroit donné d'une façon autre que la traduction d'un paquetage en interface. Toutefois, cela conduirait à un grand nombre d'interfaces supplémentaires et augmenterait la complexité de l'interface CORBA. Les directives définies ici incluent donc le concept de prise en charge conditionnelle de groupes de capacités.

Comme décrit plus haut, chaque fois qu'un paquetage GDMO est inclus dans une classe d'objets gérés, la traduction de cette classe en interface IDL comprend une traduction de chacun des gabarits du paquetage.

Les attributs GDMO qui font partie d'un paquetage conditionnel sont traduits en opérations d'accès, chacune ayant une partie *raises* qui comprend l'anomalie définie pour ce paquetage. Les actions GDMO qui font partie d'un paquetage conditionnel sont traduites en opérations ayant également une partie *raises* qui comprend l'anomalie définie pour ce paquetage. Les notifications GDMO qui font partie d'un paquetage conditionnel sont traduites en déclarations de macro *CONDITIONAL_NOTIFICATION*.

La partie *present if* de la déclaration de paquetage conditionnel d'objet GDMO doit être traduite en commentaire précédant la traduction en IDL de l'objet.

On peut par ailleurs rencontrer des problèmes avec les traductions à partir du protocole CMIP lorsque la même capacité est incluse dans des paquetages conditionnels différents. Il faut alors suivre les règles suivantes:

1) si la capacité est obligatoire dans une source et conditionnelle dans une autre, elle doit être obligatoire dans la classe traduite;
2) si la capacité fait partie de plusieurs paquetages conditionnels, l'opération traduite inclura une anomalie pour chaque paquetage. Une anomalie ne sera signalée que si aucun des paquetages n'est présent puis n'importe laquelle des anomalies peut être signalée;
3) si le même paquetage conditionnel est inclus à partir de plusieurs superclasses, la condition dans laquelle le paquetage est inclus dans la nouvelle classe est un "OU" logique entre les conditions des superclasses;
4) les notifications qui font partie de plusieurs paquetages sont traduites en une seule déclaration de macro. Si l'un des paquetages est obligatoire, on utilise la déclaration de macro *MANDATORY_NOTIFICATION*. Sinon, on utilise la déclaration de macro *CONDITIONAL_NOTIFICATION* et on énumère toutes les anomalies de paquetage.

Si un gabarit GDMO est présent dans plusieurs paquetages conditionnels contenus dans un même objet, le modélisateur souhaitera peut-être envisager de rendre la capacité obligatoire ou de définir un nouveau paquetage conditionnel pour la capacité.

Il est à noter que l'utilisation d'anomalies pour représenter des paquetages n'est possible que pour les paquetages conditionnels. Si plusieurs paquetages obligatoires sont présents dans une classe GDMO, on ne pourra les distinguer sur l'interface traduite.

Les déclarations de comportement accompagnant une définition de paquetage sont traduites en commentaires dans les définitions des objets IDL traduits à partir des objets GDMO qui comprennent le paquetage.

Il n'est pas nécessaire d'enregistrer les paquetages.

7.3 Attributs

Comme décrit ci-dessus, les classes d'objets gérés GDMO contiennent la liste des paquetages à inclure dans la définition de classe. Le paquetage contient quant à lui la liste des attributs, actions et notifications qui le constituent. Lors de la traduction d'une classe d'objets gérés, chaque gabarit présent dans les paquetages inclus sera traduit en opération sur l'interface d'objet géré et la plupart comprendront des définitions d'attribut.

Une opération <Attribute Name>Get doit être définie pour les attributs qui prennent en charge les capacités *GET*. Le type de résultat de l'opération sera une traduction de la syntaxe ASN.1 de l'attribut.

Une opération <Attribute Name>Set doit être définie pour les attributs qui prennent en charge les capacités *REPLACE*. Le type de paramètre d'entrée de l'opération sera une traduction de la syntaxe ASN.1 de l'attribut.

Une opération <Attribute Name>Add doit être définie pour les attributs qui prennent en charge les capacités *ADD*. Une opération <Attribute Name>Remove doit être définie pour les attributs qui prennent en charge les capacités *REMOVE*. Le type de paramètre d'entrée de ces opérations correspondra aux séquences IDL traduites à partir de la syntaxe ASN.1 de l'attribut.

Les méthodes de création de fabrique accepteront une valeur initiale pour les attributs qui prennent en charge la capacité *set-by-create* mais ces attributs ne doivent pas avoir d'opération *SET*. (La méthode de création de fabrique acceptera aussi des valeurs pour les attributs pouvant être modifiés, mais pas pour les attributs pouvant simplement être lus.)

Les valeurs par défaut sont définies sous forme de constantes au sein d'une interface. L'identificateur de la constante est <AttributeName>Default. L'interface peut aussi avoir une opération permettant de mettre l'attribut à sa valeur par défaut ou le client peut simplement utiliser l'opération *SET* avec la constante par défaut. L'opération de mise à la valeur par défaut est appelée <AttributeName>SetDefault, elle n'accepte aucun paramètre et renvoie *void*. L'IDL CORBA permet de définir des constantes pour des types de base et des types énumérés seulement; donc, si le type d'attribut est complexe, on ne peut lui définir aucune valeur par défaut. Dans ce cas, il faut définir une opération de mise à la valeur par défaut, et un commentaire associé à cette opération doit décrire cette valeur par défaut.

Quelques autres capacités GDMO liées aux attributs ne peuvent pas être recréées en IDL. En ce qui concerne les attributs GDMO avec une partie *DERIVED-FROM*, il faudra ajouter manuellement les capacités de l'autre attribut à la spécification d'interface. (On utilisera la syntaxe de l'attribut derived-from.) Des règles de mise en correspondance sont définies par le langage de contrainte du service d'exploitation d'objets multiples, qui fait partie des services CORBA du RGT définis dans l'UIT-T Q.816. Ces règles de mise en correspondance dépendent simplement du type de base de l'attribut. Il n'y a pas de règle de mise en correspondance pour chaque attribut. Les valeurs initiales, les valeurs permises et les valeurs requises ne sont pas prises en charge.

Il sera souvent judicieux de définir un type IDL pour chaque attribut. Même si l'attribut est d'un type simple, on peut utiliser une déclaration *typedef* IDL pour définir un type pour lui. Un commentaire précédant la définition de type d'un attribut est le meilleur endroit pour mettre la traduction de la déclaration de comportement de l'attribut. Sinon, on peut traduire la déclaration de comportement en commentaire précédant l'opération d'accès à l'attribut sur l'interface d'objet.

Les attributs normalisés définis par les directives données ici devront être utilisés chaque fois que cela sera possible. Voir le paragraphe 6.3.5.

Il n'est pas nécessaire d'enregistrer les attributs.

7.4 Groupes d'attributs

Les directives définies ici ne prennent pas en charge le concept de groupes d'attributs. Les groupes d'attributs GDMO n'ont pas de traduction équivalente.

7.5 Actions

Les actions sont traduites en opérations IDL. Les paramètres d'entrée, les paramètres de sortie et le type de résultat de l'opération sont traduits à partir de la syntaxe ASN.1 de l'entrée et de la sortie de l'action. Autrement dit, la syntaxe de l'entrée doit être traduite en paramètres *in* IDL, tandis que la syntaxe de la sortie est traduite en un mélange comprenant des paramètres *out* et la valeur de résultat. On peut utiliser des paramètres *inout* (d'entrée/sortie) IDL lorsque c'est utile. En outre, il convient de définir des anomalies afin de renvoyer des valeurs pour les situations d'erreur plutôt que de renvoyer des unions de valeurs normales et de valeurs erronées.

Les actions GDMO ayant le mode *unconfirmed* (celles pour lesquelles il manque la partie *MODE CONFIRMED*) peuvent être traduites en méthodes ayant le mot clé IDL *oneway* juste avant le type de résultat. Toutefois, ces opérations doivent avoir le type de résultat *void* et aucun paramètre *out* ou *inout*. Les opérations IDL n'ayant pas le mot clé *oneway* sont confirmées.

7.6 Notifications

Les directives données ici définissent l'équivalent IDL des 15 notifications définies dans l'UIT-T X.721, qui sont les notifications utilisées dans la plupart des modèles d'information GDMO. Généralement, une notification présente dans un paquetage GDMO sera simplement traduite en une déclaration de macro de notification sur chaque interface qui inclut le paquetage. On utilise une déclaration *MANDATORY_NOTIFICATION* si la notification fait partie d'un paquetage obligatoire et une déclaration *CONDITIONAL_NOTIFICATION* si elle fait partie d'un paquetage conditionnel.

Le mappage d'attributs d'objet avec des champs de notification dans une déclaration de notification n'est pas pris en charge. Si un certain mappage spécial est requis, il doit faire l'objet d'un commentaire. Les réponses aux notifications ne sont pas prises en charge.

Si une nouvelle notification doit être définie, elle doit l'être sous forme d'opération sur une interface appelée "Notifications" dans le module du modèle d'information (cela ne sous-entend pas que cette interface doit hériter de l'interface des notifications de itut x780::Notifications interface). Le nom de l'opération doit être le nom de la notification. Les paramètres de l'opération doivent être traduits à partir de la syntaxe d'information de la notification. L'opération de notification doit avoir le type de résultat *void* et elle ne doit avoir que des paramètres *in*. L'UIT-T Q.816 contient des informations sur la manière dont les données sont placées dans une notification structurée. Il est à noter que les identificateurs d'attribut ne sont pas nécessaires. En revanche, les paramètres sont identifiés avec un nom et un type de données. On peut alors utiliser le nom d'interface limité en portée et l'opération de notification dans les déclarations de macro de notification. Un nouveau nom est donné à cette notification (par l'ajout de "R1", etc.) pour permettre au système de gestion de créer une interface par héritage multiple pour recevoir les alarmes qui comprennent les versions ancienne et étendue.

S'il est nécessaire d'étendre une notification, il faut définir une nouvelle opération. Celle-ci doit contenir les mêmes paramètres que l'ancienne. Par exemple, le code IDL ci-dessous étend l'alarme d'équipement en ajoutant un paramètre appelé "newData" de type "newType."

```
module newModule {
…
        interface Notifications {
                void equipmentAlarmR1 (
                        in ExternalTimeType             eventTime,
                        … (other equipmentAlarm parameters)
```

```
                        in SuspectObjectSetType         suspectObjectList,
                        in newType                      newData);
        }
}
```

7.7 Comportements

Les gabarits de comportement GDMO sont traduits en commentaires IDL formatés précédant immédiatement la construction IDL à laquelle chaque comportement est associé. Les comportements d'attribut sont traduits en commentaires IDL précédant la définition du type d'attribut. Les comportements de paquetage sont traduits en commentaires IDL précédant l'anomalie définie pour le commentaire.

7.8 Rattachements de nom

Chaque rattachement de nom GDMO est traduit en module de rattachement de nom IDL comme défini au paragraphe 6.8. Les diverses constructions présentes dans le rattachement de nom GDMO sont traduites comme suit.

Le nom de la classe supérieure figurant dans le rattachement de nom devient la valeur de la constante *superiorClass* figurant dans le module de rattachement de nom. Si la partie de classe supérieure GDMO a un modificateur *AND SUBCLASSES*, la constante de rattachement de nom IDL *superiorSubclassesAllowed* vaudra *true*. Sinon, elle vaudra *false*.

Le nom de la classe subordonnée figurant dans le rattachement de nom devient la valeur de la constante *subordinateClass* figurant dans le module de rattachement de nom. Si la partie de classe subordonnée GDMO a un modificateur *AND SUBCLASSES*, la constante de rattachement de nom IDL *subordinateSubclassesAllowed* vaudra *true*. Sinon, elle vaudra *false*.

Si le rattachement de nom GDMO a une partie *CREATE*, la constante de rattachement de nom IDL *managersMayCreate* vaudra *true*. Sinon, elle vaudra *false*.

Si le rattachement de nom GDMO n'a pas de partie *DELETE*, la constante de rattachement de nom IDL *deletePolicy* vaudra *notDeletable*. S'il a une partie *DELETE* sans modificateur ou avec un modificateur *ONLY-IF-NO-CONTAINED-OBJECTS*, la constante *deletePolicy* vaudra *deleteOnlyIfNoContainedObjects*. S'il a une partie *DELETE* avec un modificateur *CONTAINED-OBJECTS*, la constante *deletePolicy* vaudra *deleteContainedObjects*.

Si la partie CREATE du rattachement de nom a un modificateur *WITH-AUTOMATIC-INSTANCE-NAMING*, l'opération de création de la fabrique d'objet géré doit définir le paramètre de nom comme étant *inout* et doit inclure un commentaire indiquant que le client peut soumettre un nom valant néant et, si c'est le cas, la fabrique choisira un nom et le renverra.

Créer un objet en copiant un ensemble partiel de valeurs d'attribut à partir d'un objet de référence est impossible avec une méthode de fabrique fortement typée car rien ne permet à la fabrique d'indiquer les valeurs qu'elle doit copier et qu'elle doit utiliser à partir des paramètres de l'opération. On peut définir une opération fortement typée qui copie toutes les valeurs à partir d'une référence, mais l'utilité en est limitée. Une opération faiblement typée acceptant un objet de référence ainsi qu'une liste partielle d'attributs peut aussi être définie pour une fabrique, mais, étant donné la difficulté d'implémentation, une telle opération ne mérite pas d'être définie. Par conséquent, la traduction du modificateur *WITH-REFERENCE-OBJECT* figurant dans une partie CREATE de rattachement de nom n'est pas prise en charge.

Les paramètres associés aux parties CREATE sont traduits en anomalies *CreateError*. Pour cela, il peut être nécessaire de définir une nouvelle valeur pour l'identificateur d'erreur. Il convient d'insérer, dans le module IDL de rattachement de nom, un commentaire indiquant quels identificateurs d'erreur pour les anomalies *CreateError* s'appliquent aux objets créés avec ce rattachement de nom. S'il est impossible de traduire un paramètre de partie CREATE en anomalie *CreateError*, une autre solution,

moins souhaitable, consiste à définir une nouvelle fabrique et à traduire le paramètre en anomalie pour une opération de création pour cette fabrique. Compte tenu toutefois du caractère général de l'anomalie *CreateError*, la nécessité de recourir à cette solution devrait être rare. (On trouvera plus de détails sur les paramètres dans le paragraphe ci-dessous.)

7.9 Paramètres

Grâce aux paramètres GDMO, il est possible d'étendre les modèles d'information GDMO. On utilise des gabarits de paramètre pour étoffer une spécification existante dans le domaine des notifications, des actions (demandes, réponses et échecs) et des erreurs spécifiques lors de la définition de sous-classes. Les définitions GDMO de toutes les notifications et de nombreuses actions contiennent un champ d'extensibilité qui est défini plus en détail par des sous-classes (si nécessaire). Dans le cas d'erreurs spécifiques, on utilise des erreurs propres à la classe pour étoffer l'erreur générale "d'échec de traitement" dans le protocole CMIP. Le format de ces informations est souvent une liste de couples nom-valeur, où le nom définit le type de données de la valeur.

Traduire des paramètres GDMO en IDL est une bonne occasion pour faire en sorte que les extensions alors définies qui ont été jugées utiles avec de nombreuses classes d'objets constituent une partie "normale", fortement typée du modèle. Par exemple, trois paramètres GDMO définis pour les alarmes ont été inclus dans les notifications définies en IDL. (Les trois paramètres sont "Alarm Effect On Service", "Suspect Object List" et "Alarming Resumed.")

Il existe plusieurs mots clés utilisés dans les gabarits de paramètre GDMO pour spécifier la sémantique des extensions. La traduction des diverses capacités d'extension disponibles avec des gabarits de paramètre fondés sur ces mots clés est examinée ci-dessous.

7.9.1 ACTION-INFO et ACTION-REPLY

Conformément au fort typage recommandé dans le cadre général, les paramètres GDMO ayant le mot clé "ACTION-INFO" dans le gabarit ne sont pas traduits sous forme de champ d'extension. En revanche, on définit une nouvelle interface comme sous-classe d'une interface existante, qui spécifie l'action mais ajoute les extensions sous forme de paramètres "in" normaux de cette méthode. Le nom du paramètre IDL doit provenir du nom du paramètre et le type de données du paramètre doit être traduit à partir de la syntaxe du paramètre GDMO. De même, les paramètres ayant le mot clé "ACTION-REPLY" seraient traduits en paramètres "out" pour l'opération.

La méthode ci-dessus implique que l'ajout ultérieur d'un paramètre à une opération IDL existante n'est pas pris en charge. En revanche, le modélisateur d'informations peut utiliser les approches plus classiques fournies par CORBA pour l'extension d'une interface, par exemple la définition d'une sous-classe d'une interface d'objet et la définition d'une nouvelle méthode pour cette sous-classe, avec des paramètres *in* et/ou *out* additionnels ou des anomalies additionnelles. On trouvera les directives à ce sujet au paragraphe 6.13.

7.9.2 EVENT-INFO et EVENT-REPLY

Dans les cas où des paramètres ayant le mot clé "EVENT-INFO" ont déjà été définis, ils sont traduits en paramètres "in" normaux pour les opérations IDL utilisées pour acheminer une notification. Ces directives ne prennent pas en charge les réponses aux notifications, il n'y a donc pas de traduction des paramètres ayant le mot clé "EVENT-REPLY".

Comme ce cadre général définit déjà un ensemble de notifications, la traduction de paramètres ayant le mot clé EVENT-INFO pourrait signifier la redéfinition d'une des opérations de notification. Voir le paragraphe 7.6.

Dans la plupart des cas toutefois, la réutilisation d'une définition de notification existante sera préférée. Dans les cas où des extensions GDMO sont prédéfinies (pour l'information d'alarme par exemple), elles doivent être incluses dans les spécifications IDL de notification traduites. Toutefois, le code IDL de notification du cadre général prend aussi en charge un champ

"additionalInformation" (informations additionnelles), qui correspond à une liste de couples nom-valeur faiblement typés. On peut l'utiliser pour ajouter des informations à ces notifications précédemment définies. Le type d'événement de notification ne changera pas. La nouvelle interface d'objet géré qui doit utiliser l'extension pour un paramètre donné doit toutefois indiquer l'utilisation de ce paramètre en commentaires. Malheureusement, il n'existe pas de mécanisme autre que celui consistant à utiliser les macros indiquées ci-dessus pour spécifier quelles notifications sont prises en charge par quels objets et ce mécanisme n'assure pas non plus la spécification de paramètres. L'avantage lié à l'utilisation du même type de notification est de permettre aux gestionnaires de recevoir les notifications sans avoir à se soucier de l'enregistrement d'un nouveau type de notification. Si les extensions ne sont pas comprises en raison de versions différentes du gestionnaire et de l'agent, les informations additionnelles sont éliminées.

La spécification des extensions correspondant aux informations additionnelles est décrite ci-dessous.

Les notifications définies par ce cadre général comprennent un champ appelé "additionalInformation" qui ressemble étroitement au champ "additionalInformation" présent dans les notifications CMIP. La syntaxe IDL du champ "additionalInformation" dans les notifications est de type "AdditionalInformationSetType":

```
struct ManagementExtensionType {
        UIDType id;             // identifies the type of info
        any     info;           // type will depend on id
};
typedef sequence <ManagementExtensionType> AdditionalInformationSetType;
```

On traduit les paramètres ayant le mot clé EVENT-INFO en définissant un identificateur unique (UID, *unique indentifier*) pour chaque paramètre. Voir le paragraphe 6.11 pour plus de détails. Brièvement toutefois, le modélisateur définit un sous-module appelé "AdditionalInformationConst" dans lequel on définit des constantes de type de valeur "short". Les noms de ces constantes sont les noms des paramètres GDMO. La valeur de chaque constante peut aussi être déduite du code GDMO, sur la base, éventuellement, du dernier numéro d'enregistrement du paramètre. Sinon, il convient de choisir un entier unique pour les constantes de ce module. Cette définition doit aussi comprendre un commentaire indiquant le type de données de la valeur qui accompagne l'identificateur UID dans le champ "additionalInformation". A titre d'exemple, si le paramètre Alarm Effect On Service n'a pas été transformé en membre normal de la structure de données Alarm Info utilisée par les alarmes de ce cadre général, il pourrait avoir été traduit comme suit:

```
module itut_m3100 {
…
  module AdditionalInformationConst {
  /** Alarm effect on service parameters are accompanied by a boolean
  value in the "any" field indicating if service has been affected. */
  const short alarmEffectOnService = 1;
  …
  }; // end of module AdditionalInformationConst
}; // end of module itut_m3100
```

Une interface IDL d'objet géré peut ensuite identifier les notifications qu'elle prend en charge comme d'habitude, mais un commentaire doit indiquer les paramètres qui seront inclus dans les notifications.

7.9.3 Mot clé de contexte

Les paramètres ayant un mot clé de contexte identifient des informations devant être transmises dans un champ nommé d'une unité PDU CMIP. Ce champ nommé est généralement une séquence de structures de données composées d'un identificateur et d'un type de données "any" qui contient une valeur dont le type dépend de l'identificateur. Dans le protocole CMIP, ces paramètres ayant un mot clé de contexte peuvent être transmis dans des paramètres d'action ou dans des notifications. La traduction de paramètres ayant un mot clé de contexte relatifs à des actions n'est pas prise en charge par ce cadre général en raison de la préférence pour le fort typage. En revanche, les informations

additionnelles relatives aux actions doivent être traduites en paramètres d'opération normaux. (Voir les paramètres ayant le mot clé ACTION-INFO ci-dessus.)

En ce qui concerne les notifications, sauf pour les extensions (voir l'explication ci-dessus), si des champs sont définis comme étant de type faible, on peut utiliser la même approche que pour le champ d'extension. Toutefois, cette approche n'a pas été utilisée dans la plupart des normes GDMO. La distinction entre le cas avec mot clé EVENT-INFO et celui avec mot clé de contexte est que le premier est conçu en vue d'une extensibilité dans laquelle un ou plusieurs paramètres peuvent être ajoutés. L'approche recommandée dans le cas d'une extension multiple est l'utilisation de EVENT-INFO et toutes les normes ont donc défini des paramètres utilisant ce mot clé.

7.9.4 SPECIFIC-ERROR

Les paramètres ayant le mot clé SPECIFIC-ERROR sont renvoyés dans des messages d'échec de traitement CMIP. Ils indiquent une issue anormale d'opération. Il existe deux options pour la traduction de ces paramètres. D'abord, ils peuvent être traduits en anomalies IDL signalées par l'opération pour laquelle ils sont définis. Le nom de l'anomalie doit provenir du nom du paramètre GDMO et le type de données renvoyé avec l'anomalie doit provenir de la syntaxe du paramètre GDMO. Comme des paramètres ayant le mot clé SPECIFIC-ERROR peuvent être définis pour différentes sortes de gabarits GDMO, ceux qui sont associés à des actions doivent être traduits en anomalies signalées par l'action et ceux qui sont associés à des attributs doivent être traduits en anomalies signalées par l'opération d'accès à l'attribut. En outre, ceux qui sont associés à la partie "Create" d'un rattachement de nom doivent être traduits en anomalies pour l'opération de création sur l'interface de fabrique. En ce qui concerne ceux qui sont associés aux notifications, ce cadre général ne prend en charge aucune traduction étant donné que les réponses aux notifications ne sont pas autorisées.

La seconde option concernant la traduction d'un paramètre ayant le mot clé SPECIFIC-ERROR consiste à traduire le paramètre en un nouveau point de code pour l'une des trois anomalies normalisées définies par ce cadre général: l'anomalie *CreateError*, signalée pour les opérations de création de fabrique, l'anomalie *DeleteError*, signalée pour les opérations de suppression d'objet géré et l'anomalie *ApplicationError*, signalée pour toutes les autres opérations d'objet géré. L'anomalie *ApplicationError* renvoie un identificateur unique de l'erreur d'application spécifique et une explication de texte. Les anomalies correspondant aux erreurs de création et de suppression étendent ces informations en ajoutant une liste d'objets connexes qui peuvent être invoqués ainsi que les attributs de l'objet sur lequel l'opération a été tentée. La liste des objets connexes pourrait par exemple indiquer certains objets qui doivent être supprimés avant que l'objet cible ne puisse être supprimé. Les attributs pourraient contenir des informations sur l'état de l'objet qui sont appropriées pour l'erreur.

La traduction d'un paramètre ayant le mot clé SPECIFIC-ERROR en point de code utilisé par l'une de ces anomalies normalisées devrait être utilisée chaque fois que c'est possible. Comme les types de données renvoyés dans les anomalies sont des types de valeur, ils peuvent être étendus pour des points de code spécifiques. Comme l'opération de suppression est héritée de l'interface d'objet géré de base, les paramètres ayant le mot clé SPECIFIC-ERROR qui figure dans les parties "Delete" de rattachement de nom GDMO doivent être traduits en points de code pour l'anomalie *DeleteError*. Cette traduction est semblable à celle des paramètres ayant le mot clé EVENT-INFO décrite ci-dessus. Fondamentalement, le modélisateur définit un sous-module d'erreur de suppression pour des constantes d'identificateur UID. Les définitions de constante doivent inclure un commentaire indiquant quelles données seront placées dans les champs "relatedObjects" et "attributeList" accompagnant une erreur avec cet identificateur. En outre, si le modélisateur a étendu le type de valeur normalisé renvoyé pour le point de code, un commentaire doit indiquer le véritable type de données renvoyé de sorte que le système gérant puisse réduire le type et accéder aux informations additionnelles. En réalité, le cadre général comprend certains points de code pour l'erreur de suppression qui étendent le type de valeur d'erreur de suppression normalisé.

Enfin, un commentaire sur l'interface IDL d'objet géré indique les valeurs d'erreur de suppression qui pourraient être signalées dans une anomalie lors d'une tentative incorrecte de suppression de l'objet. Donnons un exemple de traduction:

```
module itut_m3100 {
…
 module DeleteErrorConst {
 /** Network TTP Terminates Trail delete errors are raised when an
 attempt is made to delete a TTP before the trail has been deleted.
 It includes a reference to the Trail in the "relatedObjects" field. */
 const short networkTTPTerminatesTrail = 54;
 …
 }; // end of module DeleteErrorConst
}; // end of module itut_m3100
```

7.10 Types de données ASN.1

Les directives GDMO utilisent le langage ASN.1 pour définir la syntaxe des attributs ainsi que des paramètres d'opération et de notification; ainsi, pour convertir des gabarits GDMO en IDL, ces définitions de syntaxe devront aussi être traduites. Le présent paragraphe contient des directives pour la traduction de la syntaxe ASN.1 en IDL CORBA.

7.10.1 Types de base

Le langage IDL CORBA définit les types de base suivants vers lesquels les types de base ASN.1 peuvent être traduits: any, boolean, char, double (pour les nombres à virgule flottante en double précision), enum (pour les types énumérés), fixed, float (pour les nombres à virgule flottante en simple précision), long (pour les grands entiers), object (pour les références d'objet), octet, short (pour les petits entiers), string, wchar (pour les caractères larges) et wstring (pour les chaînes de caractères larges).

Ce cadre général utilise[1] le type *string* pour toutes les chaînes et définit "Istring", de type wstring (chaînes larges), pour les cas où la chaîne est susceptible de contenir des caractères internationaux d'échappement. Les chaînes Istring sont des chaînes composées de caractères larges (16 bits).

En outre, le service temporel CORBA définit un type temporel appelé "UtcT", utilisé par ce cadre général.

7.10.2 Séquence

Le langage IDL CORBA prend en charge la définition de structures de données utilisant le mot clé *struct*, de manière analogue à la prise en charge des types *séquence* en ASN.1.

7.10.3 Séquence de

Le langage IDL CORBA prend en charge la définition de séquences de types – de base ou complexes – essentiellement de manière analogue à la prise en charge du type *séquence de* en ASN.1.

---

[1] NOTE – Des contributions sont sollicitées sur l'utilisation, pour les chaînes, de Istring plutôt que de wstring car les chaînes peuvent comporter des ensembles de caractères internationaux lorsqu'on utilise la négociation de l'ensemble de codes, prise en charge par le protocole GIOP version 1.1 ou ultérieure. Le type wstring est mappé par les liens de langage CORBA avec le type wstring du langage de programmation, qui est souvent associé uniquement à Unicode.

7.10.4 Ensemble de

Le langage IDL CORBA ne prend pas en charge la définition de types d'ensemble complexes, contrairement à l'ASN.1. En revanche, les ensembles sont traduits en séquences IDL. La convention selon laquelle le nom de type se termine par "SetType" doit être respectée. Lors de la manipulation de valeurs d'ensemble, il convient d'éliminer les doubles et d'ignorer l'ordre.

7.10.5 Choix

Le langage IDL CORBA prend en charge la définition d'unions discriminées, remplissant les mêmes objectifs que les types choix en ASN.1.

Dans un souci de simplification de la mise en œuvre des normes du RGT fondées sur CORBA, ce cadre général recommande d'utiliser de façon prudente les unions discriminées. Lors de la traduction de code ASN.1 en code IDL CORBA, le type traduit peut souvent être simplifié sans perte de sémantique. Par exemple, en général, un choix entre une chaîne et néant peut simplement être traduit par une chaîne. Un commentaire précisant que la chaîne peut éventuellement valoir néant peut être ajouté pour indiquer cette possibilité. De même, un choix entre une séquence de (ou un ensemble de) et néant peut être traduit simplement par la séquence.

7.10.6 Identificateur d'objet (OID)

Ce cadre général définit un type appelé "identificateur universel" (UID), conçu pour remplacer les identificateurs d'objet ASN.1.

7.10.7 Instance d'objet

Le cadre général prend en charge deux traductions possibles des types d'instance d'objet ASN.1. Comme chaque objet géré a un nom, on peut utiliser le type de nom défini par le service de dénomination CORBA. (Ce cadre général contient une définition *typedef* pour les noms du service de dénomination CORBA, appelé *NameType*.) On peut aussi utiliser les références d'objet CORBA. Comme toutes les interfaces d'objets gérés doivent hériter de l'interface *ManagedObject*, le type *ManagedObject* doit être utilisé chaque fois qu'une référence générale à un objet est nécessaire. Le modélisateur peut aussi utiliser un type propre à la classe d'objets gérés (par exemple *Equipment*), ce qui présente l'avantage de rendre un modèle plus fortement typé.

7.10.8 Chaîne binaire

Le présent paragraphe définit deux mappages de ASN.1 BIT STRING à utiliser dans la traduction de spécifications d'objets GDMO en objets gérés CORBA et dans la spécification de syntaxes à utiliser dans la spécification de nouveaux objets gérés CORBA.

Etant donné que les chaînes binaires ASN.1 sont utilisées de différentes manières, deux mappages se justifient.

```
BitStringType ::=
          BIT STRING |
          BIT STRING{NamedBitList}
```

Ces deux mappages sont:

- ASN.1 BIT STRING établit un mappage avec une séquence d'octets IDL. Celle-ci n'achemine aucun drapeau sémantique.
- ASN.1 BIT STRING établit un mappage avec le type de valeur IDL, avec une représentation d'état, des fonctions d'aide locale pour manipuler des valeurs et des constantes drapeau sémantique associées.

7.10.8.1 Représentation simple de ASN.1 BIT STRING

La représentation de l'état ASN.1 BIT STRING est mappée avec une définition typedef en IDL de BitString, en tant que *sequence<octet>*. La déclaration suivante est incluse dans "itut_x780.idl" pour utilisation avec des représentations par chaînes binaires simples:

```
typedef sequence<octet> BitString;
```

L'interprétation d'une chaîne BitString est définie de la manière suivante par souci de correspondance avec le codage BER de BIT STRING. La séquence d'octets doit avoir un octet initial suivi de zéro, de un ou de plusieurs octets. Les bits de la chaîne binaire doivent être placés, en commençant par le premier et en terminant par le dernier, dans les bits 8 à 1 du deuxième octet, suivi des bits 8 à 1 de chaque octet, tour à tour, suivi du nombre de bits nécessaires dans l'octet final, en commençant par le bit 8 (les termes " bit zéro" (premier bit) et "bit de fin" (dernier bit) sont définis dans l'UIT-T X.208 | ISO/CEI 8824). L'octet initial doit coder, sous la forme d'un entier binaire non signé dont le bit 1 est le bit de plus faible poids, le nombre de bits inutilisés dans l'octet final. Ce nombre doit être compris entre zéro et sept. Si la chaîne binaire est vide, il ne doit pas y avoir d'octet subséquent et l'octet initial doit être zéro.

Lorsque, dans cette représentation simple, on utilise le protocole GIOP pour acheminer une valeur de chaîne binaire dans un "*any*" IDL, un identificateur de dépôt informant le récepteur (c'est-à-dire l'utilisateur de courtier orb) qu'il faut interpréter la séquence d'octets comme une chaîne d'octets codée, sera envoyé.

Les constantes de chaîne binaire ASN.1 sont représentées par une séquence d'octets utilisant une variante de la forme "bstring" X.208, sans les guillemets. Les constantes ASN.1 définies utilisant la forme "hstring" devront être traduites en "bstring"pour la constante en IDL associée.

Tableau 2/X.780 – Exemple de mappage de chaîne ASN.1 BIT STRING simple

ASN.1	IDL
CCDScan ::= BIT STRING scan CCDScan ::= '1001101001000011101 10'B	typedef ITUT_X780::BitString CCDScanType; const string scan = "1001101001000011101 10B" ;
G3FacsimilePage ::= BIT STRING -- a seq of bits conforming to Rec. T.4 image G3FacsimilePage ::= '1001101001000011101 10'B trailer BIT STRING ::= '01'H	typedef ITUT_X780::BitString G3FacsimilePageType; //string constants generated for BIT STRING constants const string image = "1001101001000011101 10B"; const string trailer = "00000001B";

7.10.8.2 Représentation valuetype de chaîne ASN.1 BIT STRING

La représentation d'état pour le type de valeur est ITUT_X780::BitString. La représentation de constantes pour le type BitStringValType est la même que pour le type BitString simple. Des méthodes d'aide (qui établissent une correspondance avec les appels de fonction d'objet de langage de programmation) sont comprises dans le type de valeur défini.

```
exception InvalidLength { long length } ;
valuetype BitStringValue {
  public BitString bitStringVal;
  factory initValue (in unsigned long number_of_bits);
  factory InitFromBitString (in BitString desiredValue);
  // local operations
  short getBit (in unsigned long position)
        raises (InvalidLength);
  void setBit (in unsigned long position, in short new_bit_value)
        raises (InvalidLength);
```

```
  unsigned long length ();
  string asString ();   // produces a string with binary values ("1001011B")
  // input a string with binary values ("1001011B")
  void setFromString (in string string_value)
        raises (InvalidString);
};
```

Les constantes IDL sont générées dans des types valuetype hérités de BitStringValue pour les drapeaux sémantiques associés aux positions des bits nommés. Chaque bit nommé est mappé en tant que constante IDL de type long non signé de valeur égale au décalage dans la chaîne binaire. Le nom de la constante est le nom attribué, rendu non ambigu par les règles de gestion JIDM (c'est-à-dire "a_n" pour la $n^{ième}$ utilisation de l'identificateur "a" dans le même contexte) s'il y a des collisions d'identificateurs dans le domaine de la définition du type de valeur dérivé.

```
const unsigned long <bitname> = <offset>;
```

Tableau 3/X.780 – Exemples de mappage avec représentation ValueType de chaîne Bit String

ASN.1	IDL
T0 ::= BIT STRING	valuetype T0Type : ITUT_X780::BitStringValue {} //could have used typedef to BitString for simpler mapping

MessageFlag ::= BIT STRING {
posResp (0), negResp (1),
doNotForward (2) }
	valuetype MessageFlagType : ITUT_X780::BitStringValue {
	const unsigned long posResp = 0;
	const unsigned long negResp = 1;
	const unsigned long doNotForward = 2;
	}

a INTEGER ::= 1
T1::= INTEGER { a(2) }
T2 ::= BIT STRING { a(3), b(a) }
	const long a = 1;
	typedef long T1Type;
	const T1Type a_1 = 2;
	valuetype T2Type : ITUT_X780::BitStringValue {
	const unsigned long a = 3;
	const unsigned long b = a;
	}

NOTE – Dans certains cas de traduction, il est peut être nécessaire de rendre les noms de constante non ambigus, même dans le contexte d'une déclaration valuetype. Pour éviter les confusions de noms, on pourrait utiliser dans le contexte de valuetype les règles relatives aux collisions de la gestion JIDM.

8 Idiomes de style pour les spécifications en IDL CORBA

Le présent paragraphe définit un ensemble d'idiomes de style pour le langage de définition d'interface (IDL, *interface definition language*) de l'architecture commune de courtage pour les requêtes sur des objets (CORBA) à utiliser dans les spécifications d'interface. Le fait d'avoir un ensemble d'idiomes de style permettra d'avoir des spécifications en IDL CORBA avec un style cohérent. Pour cela, les éditeurs devront peut-être fournir un effort supplémentaire afin d'augmenter la lisibilité des spécifications en IDL CORBA. Il est important d'avoir à l'esprit que les conventions de style sont destinées à aider le lecteur, mais pas nécessairement l'auteur.

8.1 Utiliser une indentation cohérente

Le présent paragraphe montre le style d'indentation pouvant être utilisé dans les modules IDL. A titre d'exemple, on donne ci-dessous un extrait du module de non-répudiation du service de sécurité CORBA:

```
enum EvidenceType {
  SecProofofCreation,
  SecProofofReceipt,
  SecProofofApproval,
  SecProofofRetrieval,
  SecProofofOrigin,
  SecProofofDelivery,
  SecNoEvidence // used when request-only token desired
};
interface NRPolicy {
  void get_NR_policy_info (
    out Security::ExtensibleFamily NR_policy_id,
    out unsigned long        policy_version,
    out Security::TimeT       policy_effective_time,
    out Security::TimeT       policy_expiry_time,
    out EvidenceDescriptorListType supported_evidence_types,
    out MechanismDescriptorListType supported_mechanisms
  );
};
```

8.2 Utiliser une casse cohérente pour les identificateurs

Plusieurs langages appliquent des règles relatives à la casse (par exemple l'ASN.1) tandis que d'autres ont des règles de fait. Ces règles permettent aux lecteurs de distinguer facilement les identificateurs ayant un type différent, ce qui accroît la lisibilité. Le langage IDL n'applique pas de règle relative à la casse, on propose donc les règles suivantes:

- pour les opérations, paramètres, attributs, membres et constantes, tout mot incorporé doit être en majuscules, à l'exception du premier;
- pour tous les autres identificateurs, la première lettre de chaque mot incorporé doit être en majuscules.

```
module CarModule {
  struct EngineType {
    PistonType piston;
    RodType  pistonRod;
  };
  typedef string KeyType;
  enum WontStartReasonType {
    BatteryIsDead,
    NoGas
  };
  exception WontStart {
    WontStartReasonType reasonEngineWontStart;
  };
  interface FordRanger {
    void startEngine(
      in KeyType key
    )
    raises (
      WontStart;
    );
    attribute EngineType engine;
  };
};
```

8.3 Respecter l'approche JIDM pour IMPORT

Au début d'un module qui importe un type d'un autre module, il convient de créer une définition *typedef* locale. Celle-ci contient explicitement le type importé du module exportant. (NOTE – Il n'est pas nécessaire que le nom de l'identificateur local soit le même que celui de l'identificateur dans le module exportant.) Cette utilisation de typedef ne convient pas pour les définitions d'interface importée ou de valuetype, pour lesquelles il convient d'utiliser des noms entièrement limités en portée.

```
module ImportingModule {
  // Imports
  typedef ExportingModule::SomeType             SomeType;
  typedef ExportingModule::SomeOtherType        SomeOtherType;
  typedef ExportingModule::SomethingElse        SomethingElseType;
  ...
};
```

8.4 Utiliser l'approche JIDM pour OPTIONAL et CHOICE

Pour le type énuméré et les types numériques (entier et à virgule flottante), il convient d'utiliser les mappages de OPTIONAL et CHOICE ASN avec le langage IDL comme prescrit dans la spécification Inter-Domain Management: Specification Translation [3] du groupe Open Group and Open-Network Management Forum Joint Inter-domain Management (JIDM).

Un exemple est donné ci-dessous:

```
// Choice
enum CarChoiceType {
  Ford,
  Cheverolet,
  Chrysler
};
union CarType switch (CarChoiceType) {
  case Ford:            FordType                fordValue;
  case Cheverolet:      ChevroletType           chevroletValue;
  case Chrysler:        ChryslertType           chryslerValue;
}
// Optional
union SunRoofTypeOpt switch(boolean) {case TRUE: SunRoofType the_value};
```

Pour les chaînes, séquences et références d'objet, une valeur néant peut généralement être utilisée pour représenter des cas facultatifs où aucune valeur n'est présente. Dans les cas où il y a une différence sémantique entre un 'néant' et un 'non présent', on peut utiliser la méthode ci-dessus.

Pour les structures et unions, on peut utiliser la méthode ci-dessus ou on peut prendre la décision d'utiliser des valeurs néant dans la structure pour représenter des valeurs facultatives qui ne sont pas présentes. Par exemple, pour une structure composée de deux chaînes, deux néants peuvent représenter une valeur facultative qui n'est pas présente. Si une valeur est facultative, elle doit être indiquée comme étant facultative au moyen d'un commentaire.

Comme toujours, les directives doivent être utilisées avec intelligence. La traduction résultante doit être évaluée du point de vue de la clarté et de la facilité d'utilisation. Si la traduction est trop complexe, le modélisateur souhaitera peut-être essayer de la simplifier.

8.5 Utiliser un suffixe cohérent pour les types

Il convient d'ajouter le suffixe "Type" à tous les types IDL. Cela permet aux identificateurs de type et aux membres d'utiliser le même nom sans collision car le langage IDL est sensible à la casse. En outre, cet idiome accroît la lisibilité en séparant clairement les identificateurs de type des autres identificateurs.

8.6 Utiliser un suffixe cohérent pour les types de séquence

En ce qui concerne les séquences (ordonnées, doubles autorisés), il convient d'utiliser le suffixe "SeqType" pour les distinguer des singletons.

8.7 Utiliser un suffixe cohérent pour les types d'ensemble

En ce qui concerne les ensembles (non ordonnés, doubles non autorisés), il convient d'utiliser le suffixe "SetType" pour les distinguer des singletons.

8.8 Utiliser un suffixe cohérent pour les types facultatifs

En ce qui concerne les types facultatifs, il convient d'utiliser le suffixe "TypeOpt" pour les distinguer des types non facultatifs.

8.9 Placer les paramètres d'opération de manière cohérente

Un ordonnancement cohérent des paramètres accroît la lisibilité. Il convient de placer les paramètres des opérations dans l'ordre suivant: *in*, *inout*, puis *out*.

8.10 Supposer l'absence d'espace d'identificateurs global

Pour réduire les collisions de nom et promouvoir la réutilisation, la portée de tous les identificateurs doit être limitée à un contexte particulier (par exemple, module et interface).

8.11 Définitions au niveau module

Toutes les définitions de type doivent être au niveau module. L'imbrication de définitions de type dans un contexte inférieur conduit à des difficultés de réutilisation et de duplication.

8.12 Utiliser des anomalies et des codes de résultat

Il convient d'utiliser des anomalies pour les situations exceptionnelles telles que les situations d'erreur. Pour les résultats normaux, on utilisera des codes de résultat et des paramètres de sortie.

8.13 Opérations explicites et opérations implicites

Une opération doit réaliser une fonction explicite. L'utilisation d'un paramètre comme fanion pour modifier implicitement le comportement de l'opération peut prêter à confusion. Il convient de prendre en compte chaque comportement dans une opération explicite distincte.

8.14 Ne pas créer un grand nombre d'anomalies

Le fait d'avoir un grand nombre d'anomalies augmente les difficultés de compréhension d'une définition d'interface. Il convient de regrouper les anomalies par catégorie ou d'utiliser les anomalies normalisées (*ApplicationError*, *CreateError* et *DeleteError*) en définissant de nouveaux points de code d'erreur pour elles, si nécessaire.

9 Conformité

Le présent paragraphe définit les critères que doivent respecter les autres documents normatifs déclarés conformes aux directives définies ici ainsi que les fonctions qui doivent être mises en œuvre par les systèmes déclarés conformes à la présente Recommandation.

9.1 Conformité des documents normatifs

Toute spécification déclarée conforme aux directives définies ici doit:

1) être telle que toutes les interfaces qui modélisent des ressources soient déduites (directement ou indirectement) de l'interface *ManagedObject* décrite au paragraphe 5.1 et définie en IDL CORBA dans l'Annexe A;
2) définir, pour chaque classe d'objet géré pouvant être instanciée, une interface de fabrique déduite (directement ou indirectement) de l'interface *ManagedObjectFactory* décrite au paragraphe 5.2 et définie en IDL CORBA dans l'Annexe A;
3) utiliser les constantes définies en IDL CORBA dans l'Annexe B chaque fois qu'il est judicieux de le faire;
4) utiliser les notifications décrites au paragraphe 5.3 et définies en IDL CORBA dans l'Annexe A chaque fois qu'il est judicieux de le faire;
5) respecter les conventions pour la définition des objets gérés du RGT CORBA spécifiées au paragraphe 6;
6) respecter les conventions relatives au langage IDL spécifiées au paragraphe 8;
7) spécifier des notifications sous forme de méthodes sur une interface "Notifications" si aucune des notifications définies dans la présente Recommandation ne s'applique;
8) définir et utiliser une anomalie NO<package name> pour identifier les attributs et les actions qui font partie d'un paquetage conditionnel;
9) utiliser les macros définies dans la présente Recommandation pour identifier les notifications qui doivent être prises en charge par un objet géré;
10) utiliser les définitions de types d'attribut générique figurant au paragraphe 6.3.5 chaque fois que cela s'applique;
11) définir des modules de rattachement de nom IDL pour identifier les relations de contenance admissibles;
12) indiquer, dans sa partie de conformité, une référence au ou aux modules à partir desquels d'autres attributs génériques sont utilisés;
13) suivre les règles de mappage de GDMO avec IDL définies au paragraphe 7 si le modèle IDL est une traduction à partir de GDMO.

9.2 Conformité des systèmes

Une implémentation déclarée conforme à la présente Recommandation doit:

1) prendre en charge toutes les capacités de l'interface ManagedObject décrites au paragraphe 5.1;
2) prendre en charge le comportement de l'opération de création décrit au paragraphe 6.9.

9.3 Directives pour les déclarations de conformité

Les utilisateurs des directives définies ici doivent soigner leurs déclarations de conformité. Comme les modules IDL sont utilisés sous forme d'espaces de noms, ils peuvent, comme cela est autorisé par les règles IDL OMG, être subdivisés en plusieurs fichiers. Par conséquent, lorsqu'un module est étendu, son nom ne changera pas mais un nouveau fichier IDL sera simplement ajouté. L'indication pure et simple du nom d'un module dans une déclaration de conformité ne suffira donc pas à identifier un ensemble d'interfaces IDL. La déclaration de conformité doit mentionner un document et une année de publication afin de garantir que c'est la bonne version de IDL qui est identifiée.

ANNEXE A

Module CORBA IDL du modèle d'objet

```
/* This IDL code is meant to be stored in a file named "itut_x780.idl"
located in the search path used by IDL compilers on your system. */
#ifndef ITUT_X780_IDL
#define ITUT_X780_IDL
#include <CosNaming.idl>
#include <CosTime.idl>
#include <itut_x780Const.idl>
#pragma prefix "itu.int"
/* Most comments in this file are formatted to be parsed by an IDL-to-HTML
converter such as idldoc or orbacus hidl. */
```

// MODULE itut_x780

```
/** This module provides the fundamental capabilities for implementing network
management interfaces and defines the "managed object" interface. The
interfaces below are modeled after the managed object specifications
found in the ITU-T CMIP specification document X.721. */
module itut_x780 {
```

// IMPORTED TYPES

```
        // Types imported from CosNaming
        typedef CosNaming::Name NameType;
        // Types imported from CosTime
        typedef TimeBase::UtcT UtcT;
```

// FORWARD DECLARATIONS AND TYPEDEFS

```
        /** International strings are strings of wide (16 bit unicode)
        characters. */
        typedef wstring Istring;
        /** Istring Sets are just sets of Istrings */
        typedef sequence <Istring> IstringSetType;
        /** Additional Text Type is often used in notifications to convey a
        text explanation for the notification.
        */
        typedef Istring AdditionalTextType;
        /** Avalibility Type is used in a sequence to indicate the
        availability of a resource. Zero or more of these conditions may be
        indicated.
        */
        typedef short AvailabilityStatusType;
        const AvailabilityStatusType inTest = 0;
        const AvailabilityStatusType failed = 1;
        const AvailabilityStatusType powerOff = 2;
        const AvailabilityStatusType offLine = 3;
        const AvailabilityStatusType offDuty = 4;
        const AvailabilityStatusType dependency = 5;
        const AvailabilityStatusType degraded = 6;
        const AvailabilityStatusType notInstalled = 7;
        const AvailabilityStatusType logFull = 8;
        /** Availability status is used to indicate the availability of a
        resource. It is represented as a sequence of integers because several
        of the conditions may exist at once.
        */
        typedef sequence<AvailabilityStatusType> AvailabilityStatusSetType;
        /** Backed Up Status Type is used to indicate if an object has a back
        up. */
        typedef boolean BackedUpStatusType;
        /** BitStrings are used to hold strings of bits. They may be of any
        length. */
        typedef sequence<octet> BitString;
        /** Control Status Type is used in a sequence to indicate the
        control status of a resource. Zero or more of these may be indicated.
        */
        typedef short ControlStatusType;
```

```
const ControlStatusType subjectToTest = 0;
const ControlStatusType partOfServicesLocked = 1;
const ControlStatusType reservedForTest = 2;
const ControlStatusType suspended = 3;
/** Control status set is used to indicate the control status of a
resource. It is represented as a sequence of integers because several
of the conditions may exist at once.
*/
typedef sequence<ControlStatusType> ControlStatusSetType;
/** Generalized time is a basic ASN.1 type. It is usually represented
as a string in computing languages but it has certain, parseable
formats. The 3 possible forms are: <ol><li>
Local time only. "YYYYMMDDHHMMSS.fff", where the optional fff is
accurate to three decimal places, <li>
Universal time (UTC time) only. "YYYYMMDDHHMMSS.fffZ", and <li>
Difference between local and UTC times. "YYYYMMDDHHMMSS.fff+-HHMM".
</ol>
The options for representing this in IDL seem to be either a string or
the UtcT structure from the CORBA Time Service. UtcT makes it a little
easier to compare times from different zones, but requires managed
systems to know their time zones. UtcT was picked.
*/
typedef UtcT GeneralizedTimeType;
/** External Time is generalized time. */
typedef GeneralizedTimeType ExternalTimeType;
/** Forward declaration. CORBA uses object references
of type "object" to identify objects. These are used instead of ASN.1
object instances. For network management interfaces, all objects will
inherit from the "ManagedObject" interface. */
interface ManagedObject;
/** MO Set is a set of ManagedObject references. */

typedef sequence <ManagedObject> MOSetType;
/** MO Seq is a sequence of ManagedObject references. */
typedef sequence <ManagedObject> MOSeqType;
/** A set of names is definded as a sequence of names. */
typedef sequence <NameType> NameSetType;
/** Notification IDs are long integers. */
typedef long NotifIDType;
/** This defines a set of notification IDs. */
typedef sequence <long> NotifIDSetType;
/** Procedural Status Type is used in a sequence to indicate the
procedural status of a resource. Zero or more of these may be
indicated.
*/
typedef short ProceduralStatusType;
const ProceduralStatusType initializationRequired = 0;
const ProceduralStatusType notInitialized = 1;
const ProceduralStatusType initializing = 2;
const ProceduralStatusType reporting = 3;
const ProceduralStatusType terminating = 4;
/** Procedural Status Set is used to indicate the procedural status of
a resource. It is represented as a sequence of integers because
several of the conditions may exist at once.
*/
typedef sequence<ProceduralStatusType> ProceduralStatusSetType;
/** ScopedName is just a string. */
typedef string ScopedNameType;
/** Scoped Name Sets are simply sets of Scoped Names. */
typedef sequence <ScopedNameType> ScopedNameSetType;
/** In CORBA, strings containing scoped names are used to identify
object classes (actually, "interfaces"). */
typedef ScopedNameType ObjectClassType;
/** Object Class Set is a set of object classes */
typedef sequence <ObjectClassType> ObjectClassSetType;
/** Name Binding Modules are identified with scoped names. */
typedef ScopedNameType NameBindingType;
/** StartTimeType is used to specify a time when something starts.
It is often paired with a StopTimeType to control the activation of
some function.
*/
```

```
typedef GeneralizedTimeType StartTimeType;
/** String sets are sets of strings. */
typedef sequence <string> StringSetType;
/** System Labels are strings used to identify systems. */
typedef string SystemLabelType;
/** Unknown status is used to indicate if the status of a resource is
not known. A value of true indicates the status is unknown. */
typedef boolean UnknownStatusType;
```

// ENUMERATED TYPES

```
/* The following state objects are used in many interfaces and parallel
the state objects in CMIP standards. */
/** Administrative State is read/write. A "locked" object is usually
one that may not be changed or one which is not providing service.
Setting the Admininstrative State of an object to "shuttingDown" begins
the shutdown process for that object. */
enum AdministrativeStateType {locked, unlocked, shuttingDown};
/** Operational State is read only. It simply reports the current
capability of the object to provide service. */
enum OperationalStateType {disabled, enabled};
/** Usage state is read only. If "idle," the resource is completely
unused. If "busy," the total capacity of the resource is in use.
"Active" is in between. */
enum UsageStateType {idle, active, busy};
/** Delete Policy indicates if an object can be deleted and if so if
any contained objects should automatically be deleted. Since objects
must not be orphaned, if an object has a delete policy of
"deleteOnlyIfNoContainedObjects" the object must not be deleted if it
has contained objects. A value of "deleteContainedObjects" means if
the object is deleted its contained objects should also be deleted. */
enum DeletePolicyType {notDeletable, deleteOnlyIfNoContainedObjects,
                deleteContainedObjects};
/** PerceivedSeverity reports the severity of an alarm. "Indeterminate"
is used when it is not possible to assign one of the other values */
enum PerceivedSeverityType {indeterminate, critical, major, minor,
                                warning, cleared};
/** Source Indicator is used in many notifications. It identifies
whether the notification is a result of a management operation or
something that occurred on the managed system. */
enum SourceIndicatorType {resourceOperation, managementOperation,
                                unknown};
/** The standby status attribute is single-valued and read-only.
The value is only meaningful when the back-up relationship role exists.
If "hot standby" the resource is not providing service, but is
operating in synchronism with another resource that is to be backed-up.
If "cold standby" the resource is to back-up another resource, but is
not synchronized with that resource. If "providing service" the back-up
resource is providing service and is backing up another resource.
*/
enum StandbyStatusType {hotStandby, coldStandby, providingService};
/** Stop times are used to specify when some function should cease.
There are normally two choices, the function runs continually (in
which case no actual time is specified) or the function ends at
a specified time.
*/
enum StopTimeChoice {specific, continual};
/** Threshold indication describes if the threshold crossed was in the
up or down direction. */
enum ThresholdIndicationType {up, down};
/** TrendIndication values indicate if some observed condition is
getting better, worse, or not changing. */
enum TrendIndicationType {lessSevere, noChange, moreSevere};
```

// STRUCTURES AND UNIONS

```
/* The structures defined below are used to pass values that may be
optionally included. For some types of values, like strings, lists,
and pointers, it is easy to tell if the value is included. For others,
like enumerations, numbers, and structures, it is not. */
```

```
/** AdministrativeStateTypeOpt is an optional type. If the
discriminator is true the value is present, otherwise the value is
null. */
union AdministrativeStateTypeOpt switch (boolean) {
        case TRUE:      AdministrativeStateType value;
};
/** BooleanTypeOpt is an optional type. If the discriminator is
true the value is present, otherwise the value is null. */
union BooleanTypeOpt switch (boolean) {
        case TRUE:      boolean value;
};
/** FloatTypeOpt is an optional type. If the discriminator is
true the value is present, otherwise the value is null. */
union FloatTypeOpt switch (boolean) {
        case TRUE:      float   value;
};
/** LongTypeOpt is an optional type. If the discriminator is
true the value is present, otherwise the value is null. */
union LongTypeOpt switch (boolean) {
        case TRUE:      long    value;
};
/** OperationalStateTypeOpt is an optional type. If the discriminator
is true the value is present, otherwise the value is null. */
union OperationalStateTypeOpt switch (boolean) {
        case TRUE:      OperationalStateType    value;
};
/** ShortTypeOpt is an optional type. If the discriminator is
true the value is present, otherwise the value is null. */
union ShortTypeOpt switch (boolean) {
        case TRUE:      short   value;
};
/** TrendIndicationTypeOpt is an optional type. If the discriminator
is true the value is present, otherwise the value is null. */
union TrendIndicationTypeOpt switch (boolean) {
        case TRUE:      TrendIndicationType     value;
};
/** UnsignedShortTypeOpt is an optional type. If the discriminator is
the value is present, otherwise the value is null. */
union UnsignedShortTypeOpt switch (boolean) {
        case TRUE:      unsigned short value;
};
/** UsageStateTypeOpt is an optional type. If the discriminator is
true the value is present, otherwise the value is null. */
union UsageStateTypeOpt switch (boolean) {
        case TRUE:      UsageStateType  value;
};
/** Many times interface specifications need to define standard values
to be passed across the interface. Also, often the scheme used to
define these values needs to be extensible as new interfaces are
subclassed, so enumerations don't work well. CMIP uses OIDs, strings
of numbers that are often appended, in standards. To serve this
purpose, the Unique ID is used. It consists of two parts, a string
containing a scoped module name, and an integer value defined as a
constant within that module. These UIDs, and the ObjectClass type
defined above, replace ASN.1 OIDs. It is expected that each module
will contain a constant string named "moduleName" that contains the
name of the module for error-free use by the programmer. A null module
name will indicate a null value for the UID. <p>
Code to interpret a UID might look like the following code snippet:<br>
<code><pre>
UIDType pc;     // probable cause
...
if (pc.moduleName ==
        itut_x780::ProbableCauseConst::moduleName) //string compare
        switch (pc.value) {
        case itut_x780::ProbableCauseConst::adapterError:
        ...
        case
        itut_x780::ProbableCauseConst::applicationSubsystemFailure:
        ...
```

```
        case itut_x780::ProbableCauseConst::bandwidthReduced:
        ...
        }
else if (pc.moduleName == MyLocal::ProbableCauseConst::moduleName)
        switch (pc.value) {
        ...
        }
</pre></code>
@member moduleName      The scoped module name where values are
                        defined.
@member value           The value defined as a constant within the
                        module.
*/
struct UIDType {
        string moduleName;      // module where value is defined
        short value;            // constant within the module
};
typedef sequence <UIDType> UIDSetType;
/** Management Extension is a structure for flexibly reporting
information. It is typically used in the Additional Information field
of notifications.
@see <a href="#AdditionalInformationSetType">
AdditionalInformationSetType </a>
@member id      identifies the type of information
@member any     contains the actual information, type will depend on
                the value of the id member.
*/
struct ManagementExtensionType {
        UIDType id;             // identifies the type of info
        any     info;           // type will depend on id
};
/** Additional Information is a flexible way to report information that
does not fit into the structure of a notification. It contains a
sequence of a structure called "Management Extension". */
typedef sequence <ManagementExtensionType>
                        AdditionalInformationSetType;
/** An Attribute Value structure is used in a notification to report
the value of any attribute. The string used for the attribute's name
is the same as the name of the data member in the value object defined
for the object. In other words, it is the name of an attribute accessor
method minus the "get" or "set".
@member attributeName   the name of the attribute
@member value           contains the value of the attribute, type will
                        depend on the attributeName.
*/
struct AttributeValueType {
        string  attributeName;
        any     value;          // type will depend on the attribute
};
/** Attribute Value Sets are used to report attributes generically,
in a batch mode. */
typedef sequence <AttributeValueType> AttributeSetType;
/** An Attribute Value Change structure is used in a notification to
report an attribute that has been changed.
@see <a href="#AttributeValueType">AttributeValueType</a>
@member attributeName   the name of the attribute
@member oldValue        the old value, type will depend on the
                        attributeName
@member newValue        the new value, type will depend on the
                        attributeName.
*/
struct AttributeValueChangeType {
        string          attributeName;
        any             oldValue;       // type depends on attribute
        any             newValue;       // type depends on attribute
};
/** An Attribute Change Set is used to report the attributes that have
been changed in an attribute value change notification. */
typedef sequence <AttributeValueChangeType> AttributeChangeSetType;
```

```
/** A Correlated Notification is identified by the object that emitted
the notification and the notification ID. Both are included in case
the Notification IDs are not unique across objects.
@member source  Reference to object that emitted the correlated
                notification. If null, the correlated notifications
                are from the same source as the notification containing
                this data structure.
@member notifIDs IDs of the correlated notifications. Notification
                identifiers must be chosen to be unique across all
                notifications from a particular managed object
                throughout the time that correlation is significant.
*/
struct CorrelatedNotificationType {
        NameType        source;
        NotifIDSetType  notifIDs;
};
/** Correlated Notification sets are sets of Correlated Notification
structures. */
typedef sequence <CorrelatedNotificationType>
                CorrelatedNotificationSetType;
/** ProbableCause, in CMIP standards, may be either an integer or GDMO
OID, a dot-notation string. The UID type is used instead. */
typedef UIDType ProbableCauseType;
/** Proposed Repair Actions are sets of unique identifiers. */
typedef UIDSetType ProposedRepairActionSetType;
/** Security Alarm Causes are unique identifiers. */
typedef UIDType SecurityAlarmCauseType;
/** Security Alarm Detector can indicate either a mechanism or a
specific object. According to X.721 a choice is made between one or
the other, though it is not clear why. (Actually, X.721 adds a third
choice for an AE-title which has no equivalent here.) Unless otherwise
indicated, then, at most one of the members will be non-null. Two
nulls may be sent if the managed system does not support this property.
@member mechanism       the scheme or function detecting the alarm, may
                        be null
@member obj             the object detecting the alarm, may be null
*/
struct SecurityAlarmDetectorType {
        UIDType         mechanism;      // may be null
        NameType        obj;            // may be null
};
/** Service User
@member id      the id of the service user
@member details details about the service user, type will depend on id
*/
struct ServiceUserType {
        UIDType id;
        any     details;        // value will depend on id
};
/** Service Providers share the same representation as Service Users.
*/
typedef ServiceUserType ServiceProviderType;
/** Specific Problems are sets of unique identifiers. */
typedef UIDSetType SpecificProblemSetType;
/** A Stop Time Type is used to indicate when some function should
cease. In the specific case, an actual time is given. In the
continual case, the function runs continually and no value is
carried in this union.
*/
union StopTimeType switch (StopTimeChoice) {
        case specific:  GeneralizedTimeType time;
        /* case continual carries NULL value */
};
/** A SuspectObject identifies an object that may be the cause of a
failure. It is usually a component of a SuspectObjectList.
@member objectClass             Object class of the suspect object
@member suspectObjectInstance   Object instance of the suspect object
@member failureProbability      Optional failure responsibility
                                probability from 1 to 100
*/
```

```
struct SuspectObjectType {
        ObjectClassType         objectClass;
        ManagedObject           suspectObjectInstance;
        UnsignedShortTypeOpt    failureProbability;
};
/** Suspect Object Lists are used to identify objects that may be the
cause of a failure.
*/
typedef sequence<SuspectObjectType> SuspectObjectSetType;
/** Threshold Level Indication describes multi-level threshold
crossings. Up is the only permitted choice for a counter. In ASN.1,
if indication is "up", low value is optional.
@member indication      indicates up or down direction of crossing.
@member low             the low observed value.
@member high            the high observed value.
*/
struct ThresholdLevelIndType {
        ThresholdIndicationType         indication;
        FloatTypeOpt                    low;    // observed value
        float                           high;   // observed value
};
/** Threshold Level Ind Type Opt is an optional type. If the
discriminator is true the value is present, otherwise the value is
null. */
union ThresholdLevelIndTypeOpt switch (boolean) {
        case TRUE:      ThresholdLevelIndType   value;
};
/** Threshold Information indicates some guage or counter attribute
passed a set threshold. The structure differs from X.721 some to
simplify the syntax.
@member attributeID     Identifies the attribute that crossed the
                        threshold. Actually, it is an operation name
                        on an interface minus the "get" or "set". The
                        interface on which the operation is defined is
                        included elsewhere in the notification as
                        ObjectClass. A Null value indicates the entire
                        structure is null.
@member observedValue   Attributes that are of type integer will be
                        converted to floats.
@member thresholdlevel  This parameter is for multi-level threhsolds.
                        Optional.
@member armTime         May be null(0). */
struct ThresholdInfoType {
        string                          attributeID;
        float                           observedValue;
        ThresholdLevelIndTypeOpt        thresholdLevel;
        ExternalTimeType                armTime;
};
```

// EXCEPTIONS

```
/** Application error info types are passed back in managed object
exceptions.
@member error   A unique identifier identifying the problem.
@member details A text message with additional information about the
problem.
*/
valuetype ApplicationErrorInfoType {
        public UIDType          error;
        public Istring          details;
};
/** Create error info types are passed back in managed object create
exceptions. They extend application error info types.
@member relatedObjects  objects that have some relationship to the
                        object to be created that somehow prevented the
                        creation.
@member attributeList   the values that would have been assigned to the
                        created object. These may hold some key to why
                        the object could not be created.
*/
```

```
valuetype CreateErrorInfoType : ApplicationErrorInfoType {
        public MOSetType        relatedObjects;
        public AttributeSetType attributeList;
};
/** Delete error info types are passed back in managed object delete
exceptions. They extend application error info types.
@member relatedObjects  objects that have some relationship to the
                        object to be deleted that somehow prevented the
                        deletion.
@member attributeList   the attribute values assigned to the object to
                        be deleted. These may hold some key to why the
                        object could not be deleted.
*/
valuetype DeleteErrorInfoType : ApplicationErrorInfoType {
        public MOSetType        relatedObjects;
        public AttributeSetType attributeList;
};
/** A package error info type is a special create error. It will be
passed back in a managed object create exception as a create error. If
the UID error code matches the package error info type, the client
application may narrow the value type from create error info type to
package error info type to access the additional information.
@member packages        the list of requested packages that conflicted
                        or could not be supported.
*/
valuetype PackageErrorInfoType : CreateErrorInfoType {
        public StringSetType    packages;
};
/** Application error exceptions may be raised on any managed object
operation to identify a problem preventing the operation from being
completed. */
exception ApplicationError { ApplicationErrorInfoType info; };
/** Create error exceptions may be raised on any managed object create
operation to identify a problem preventing the object from being
completed. */
exception CreateError { CreateErrorInfoType info; };
/** Delete error exceptions may be raised by a managed object in
response to an attempt to delete the object. They may also be raised
by the terminator service. */
exception DeleteError { DeleteErrorInfoType info; };
/** Invalid length exceptions are raised when an invalid length is
supplied on an operation invocation. */
exception InvalidLength { long length; };
/** Invalid string exceptions are raised when an invalid string is
supplied on an operation invocation. */
exception InvalidString {};
```

// VALUE TYPES

```
/** Bit string value types are used to represent bit strings with
associated semantic tags representing the bit string positions. */
valuetype BitStringValue {
        /** The state of a bit string is kept in a data member
        of type BitString (sequence of octets). The first octed
        is the count of unused bits in the last octed. */
        public BitString bitStringVal;
        /** This initializer shall set all bit positions to '0' */
        factory initValue (in unsigned long number_of_bits);
        /** This initializer shall create a new BitStringValue with
        the same length and value as the supplied BitString. */
        factory InitFromBitString (in BitString desiredValue);
        // local operations
        /** This local operation returns 0 or 1 for the bit value
        at the specified position. If the position requested is
        beyond the length of the bit string an invalid length
        exception shall be raised. */
        short getBit (in unsigned long position)
                raises (InvalidLength);
```

```
        /** This local operation is used to set the value of the
        bit at the requested position. If the position requested is
        beyond the length of the bit string an invalid length
        exception shall be raised. If the new_bit_value is 0,
        the bit shall be set to 0, otherwise it shall be set to 1. */

        void setBit (in unsigned long position, in short new_bit_value)
                raises (InvalidLength);
        /** This local operation returns the number of bits in the bit
        string. Since the first octet contains the number of unused
        bits in the last octet, and the last octet may contain unused
        bits, the value returned is
        (NumberOfOctets – 2)*8 + (8 – firstOctetVal) */
        unsigned long length ();
        /** This local operation returns the value of a bit string as a
        character string. Each bit of value 0 shall be represented as
        a '0' character, and each bit of value 1 shall be represented
        as a '1' character. A 'B' character shall be appended to the
        end of the string ("1001011B").
        string asString ();
        /** This local operation sets the value of the bit string given
        a character string. Each '0' character in the string is
        converted to a bit of value 0, and each '1' character is
        converted to a 1. If the string has any characters other than
        '0', '1', or a terminating 'B', the InvalidString exception
        shall be raised. */
        void setFromString (in string string_value)
                raises (InvalidString);
};
```

// **MANAGED OBJECT INTERFACE**

```
/** This valuetype object contains members for each of the attributes
accessible on this interface. */
valuetype ManagedObjectValueType {
        public NameType                 name;
        public ObjectClassType          objectClass;
        public StringSetType            packages;
        public SourceIndicatorType      creationSource;
        public DeletePolicyType         deletePolicy;
};
/** The Managed Object interface is intended to be the base interface
from which all other managed object interfaces inherit. It is a
central place to specify basic functions which all managed objects are
expected to support. */
interface ManagedObject {
        /** This method returns the fully-qualified name for the
        object. This method is used rather than having a "get*ID"
        method defined for each interface, as is done in CMIP
        specifications. This will ensure that objects have only a
        single operation to retrieve names when they are sub-classed.
        <p>
        The response is a sequence of name component structures,
        starting with the name assigned to the "local root" naming
        context under which this object is contained. The client may
        find the superiors of this object by removing components from
        the tail end of this sequence and performing a resolve
        operation on the first part of the name. */
        NameType nameGet()
                raises (ApplicationError);
        /** This method returns the scoped name of the most-specific
        class of the interface (e.g. "EquipmentR1"). */
        ObjectClassType objectClassGet()
                raises (ApplicationError);
        /** This method returns a list of all the conditional packages
        supported by this instance. */
        StringSetType packagesGet ()
                raises (ApplicationError);
        /** This method returns an indication of how the object was
        created. */
```

```
        SourceIndicatorType creationSourceGet()
                raises (ApplicationError);
        /** This method returns a value indicating if the object may be
        deleted and if it may, if all contained objects are
        automatically deleted. */
        DeletePolicyType deletePolicyGet ()
                raises (ApplicationError);
        /** This method may be used to generically get all of the
        attributes supported by an instance. Each interface is
        expected to sub-class the Managed Object value type and add the
        other attributes supported by that interface. The managed
        object must return a value object of that type. The client
        must then narrow the reference to access all the attributes.
        <p>
        The client may also submit a list of names indicating the
        attributes it wishes to receive. These names must match the
        member names in the value object. For members not on the list,
        and for members that are part of packages that are not
        supported, the server may return any value but it should be as
        short as possible. The server also returns the list of
        attributes, which may be shorter due to exclusion of attributes
        in unsupported packages. The client must regard the value of
        any member not in the list as garbage. <p>
        A null attribute names list indicates that all supported
        attributes are to be returned. The server must return the
        actual list. */
        ManagedObjectValueType attributesGet (
                inout StringSetType attributeNames)
                raises (ApplicationError);
        /** This method destroys the object. It is used to simply
        release any resources associated with the managed object. It
        does not check for contained objects or remove name bindings
        from the naming tree. <p>
        The intent of this operation is to allow support services to
        destroy the managed object. <p><b>
        NOTE: Direct invocation of this operation from a managing
        system could corrupt the naming tree and is recommended only
        under extraordinary circumstances. Clients wishing to delete
        an object should instead use the terminator service. </b> */
        void destroy()
                raises (ApplicationError, DeleteError);
};  // end of ManagedObject interface
```

// MANAGED OBJECT FACTORY INTERFACE

```
/** This interface defines the generic managed object factory
interface. All Managed Object factories should inherit from this
interface. <p>
In addition to providing the means for creating objects by management
operation, the factories are assumed to take responsibility for
maintaining the integrity of the naming tree by creating name bindings
for the objects they create. <p>
Currently, this interface is null. It is included, however, as a
placeholder for capabilities that must be supported by all managed
object factories.
*/
interface ManagedObjectFactory {

};  // end of ManagedObjectFactory interface
```

// NOTIFICATIONS INTERFACE

```
/** This interface contains the definitions of notifications emitted by
many managed objects. <p>
The use of "typed" notifications is done here so that the notifications
can be documented in IDL and to support typed notifications for those
manager and managing systems that wish to use them. Note that the
OMG's Notification Service supports both structured and typed
notifications. It is not clear if implementations of the Notification
Service will support translation between them. It is expected that the
implementation agreement between the managing and managed system will
```

```
        specify the use of structured or typed notifications. <p>
        Notification users wishing to use typed notifications need only support
        the interfaces below. Notification publishers and subscribers wishing
        to use structured notifications based on the operations defined below
        should follow these rules for constructing and reading the notification
        structure: <ul><li>
        The domain_type string in the fixed header of the structure should be
        set to "telecommunications". <li>
        The event_type string in the fixed header of the structure should be
        set to the scoped name of the operation. For example, for the
        Attribute Value Change notification defined below this field would be
        "itut_x780::Notifications::attributeValueChange". <li>
        The event_name string in the fixed header of the structure is not used
        by this framework. It can be set to null or used for other purposes.
        <li>
        Optional header fields may be included to support features like Quality
        of Service as appropriate. <li>
        Each parameter in the operation should be placed in a name-value pair
        in the filterable body portion of the notification. The fd_name string
        of this pair shall be set to the name of the parameter and the type
        placed in the associated fd_value will be the type specified for the
        parameter. For example, each of the notifications defined below has a
        parameter named "eventTime" that is an "ExternalTimeType." This
        parameter would be placed in the filterable data portion of the event.
        The fd_name string of this pair would be set to "eventTime" and
        fd_value would contain an ExternalTimeType value. <li>
        The remainder of the body of the notification (the unfilterable part)
        should be null. </ul>
        Unfortunately, typed notifications are mapped to notification
        structures differently, so if one system wants to use typed
        notifications and the other structured, the structured notification
        user must be aware of how the CORBA Notification Service translates
        typed notifications to structured notifications. See the specification
        for details. In short, however, each of the parameters in the
        operations below will be converted into a name-value pair in the
        filterable data protion of the structured notification. Also, the
        event_type field in the fixed header of the structured notification
        will be set to the special value "%TYPED" and the domain_type field
        will be an empty string. Finally, a name-value pair will be added as
        the first element in the filterable data portion of the notification
        with the name "operation". The value associated with this name will be
        a string with the value set to the scoped name of the operation used to
        emit the notification
        (e.g. itut_x780::Notifications::attributeValueChange). <p>
        Also, structured notification publishers may exclude notification
        parameters that are marked "optional" or are of an optional type (a
        type name ending in "TypeOpt." This should be done for efficiency.
        This will, however, preclude the automatic conversion of structured
        notifications to typed, so managers must be capable of accepting
        structured notifications. (They do not strictly have to support typed
        notifications, but if managed systems emit typed notifications managers
        should accept them rather than translations because it will be more
        efficient.) If an "optional" parameter is included in a notification,
        the "optional" type (discriminated union) must be used. <p>
        Parameters named "operation" should be avoided in notification
        operations to support the use of typed notifications. While the
        notification channel should be able to differentiate the real parameter
        from the one added based on their positions in the filterable data
        list, it could have an impact on filtering as the default filtering
        language does not have a way to differentiate parameters based on
        position. <p>
        Because the scoped operation name is placed in either the type_name
        string (when structured notifications are used) or a filterable body
        name-value pair with the name "operation" (when typed notifications are
        used), there is no "event type" parameter explicitly included in any of
        the notification data structures. */
    interface Notifications {
        /** An Attribute Value Change notification is used to report changes to
        the attributes of an object such as addition or deletion of members to
        one or more set-valued attributes and replacement of the value of one
        or more attributes.
```

```
@param eventTime                Managed system's current time.
@param source                   Object emitting notification.
@param sourceClass              Actual class of source object.
@param notificationIdentifier   A unique identifier for this
                                notification. Must be unique for
                                an object instance. (Optional in X.721
                                but not here. See text for
                                discussion of possible implications)
@param correlatedNotifications  List of correlated notifications.
                                Optional. Zero length sequence
                                indicates absence of this parameter.
@param additionalText           Text message. Optional. Zero length
                                string indicates absence of this
                                parameter.
@param additionalInfo           Optional. Zero length sequence
                                indicates absence of this parameter.
param sourceIndicator           Cause of event. Optional. Use
                                "unknown" if not supported.
@param attributeChanges         Changed attributes
*/
void attributeValueChange (
        in ExternalTimeType                     eventTime,
        in NameType                             source,
        in ObjectClassType                      sourceClass,
        in NotifIDType                          notificationIdentifier,
        in CorrelatedNotificationSetType        correlatedNotifications,
        in AdditionalTextType                   additionalText,
        in AdditionalInformationSetType         additionalInfo,
        in SourceIndicatorType                  sourceIndicator,
        in AttributeChangeSetType               attributeChanges
);
/** A Communications Alarm notification is used to report when an
object detects a communications error.
@param eventTime                Managed system's current time.
@param source                   Object emitting notification.
@param sourceClass              Actual class of source object.
@param notificationIdentifier   A unique identifier for this
                                notification. Must be unique for
                                an object instance. (Optional in X.721
                                but not here. See text for
                                discussion of possible implications)
@param correlatedNotifications  List of correlated notifications.
                                Optional. Zero length sequence
                                indicates absence of this parameter.
@param additionalText           Text message. Optional. Zero length
                                string indicates absence of this
                                parameter.
@param additionalInfo           Optional. Zero length sequence
                                indicates absence of this parameter.
@param probableCause
@param specificProblems         Optional. Zero length sequence
                                indicates absence of this parameter.
@param perceivedSeverity
@param backedUpStatus           "True" if backed up
@param backUpObject             Will be null if backedUpStatus is
                                "false"
@param trendIndication          Optional. See type for details.
@param thresholdInfo            Optional. See type for details.
@param stateChangeDefinition    Optional. Zero length sequence
                                indicates absence of this parameter.
@param monitoredAttributes      Optional. Zero length sequence
                                indicates absence of this parameter.
@param proposedRepairActions    Optional. Zero length sequence
                                indicates absence of this parameter.
@param alarmEffectOnService     True if alarm is service effecting.
@param alarmingResumed          True if alarming was just resumed,
                                possibly resulting in delayed reporting
                                of an alarm
@param suspectObjectList        Objects possibly involved in failure.
*/
```

```
void communicationsAlarm (
        in ExternalTimeType                     eventTime,
        in NameType                             source,
        in ObjectClassType                      sourceClass,
        in NotifIDType                          notificationIdentifier,
        in CorrelatedNotificationSetType        correlatedNotifications,
        in AdditionalTextType                   additionalText,
        in AdditionalInformationSetType         additionalInfo,
        in ProbableCauseType                    probableCause,
        in SpecificProblemSetType               specificProblems,
        in PerceivedSeverityType                perceivedSeverity,
        in BooleanTypeOpt                       backedUpStatus,
        in NameType                             backUpObject,
        in TrendIndicationTypeOpt               trendIndication,
        in ThresholdInfoType                    thresholdInfo,
        in AttributeChangeSetType               stateChangeDefinition,
        in AttributeSetType                     monitoredAttributes,
        in ProposedRepairActionSetType          proposedRepairActions,
        in BooleanTypeOpt                       alarmEffectOnService,
        in BooleanTypeOpt                       alarmingResumed,
        in SuspectObjectSetType                 suspectObjectList
);
/** An Environmental Alarm notification is used to report a problem in
the environment.
@param eventTime                Managed system's current time.
@param source                   Object emitting notification.
@param sourceClass              Actual class of source object.
@param notificationIdentifier   A unique identifier for this
                                notification. Must be unique for
                                an object instance. (Optional in X.721
                                but not here. See text for
                                discussion of possible implications)
@param correlatedNotifications  List of correlated notifications.
                                Optional. Zero length sequence
                                indicates absence of this parameter.
@param additionalText           Text message. Optional. Zero length
                                string indicates absence of this
                                parameter.
@param additionalInfo           Optional. Zero length sequence
                                indicates absence of this parameter.
@param probableCause
@param specificProblems         Optional. Zero length sequence
                                indicates absence of this parameter.
@param perceivedSeverity
@param backedUpStatus           "True" if backed up
@param backUpObject             Will be null if backedUpStatus is
                                "false"
@param trendIndication          Optional. See type for details.
@param thresholdInfo            Optional. See type for details.
@param stateChangeDefinition    Optional. Zero length sequence
                                indicates absence of this parameter.
@param monitoredAttributes      Optional. Zero length sequence
                                indicates absence of this parameter.
@param proposedRepairActions    Optional. Zero length sequence
                                indicates absence of this parameter.
@param alarmEffectOnService     True if alarm is service effecting.
@param alarmingResumed          True if alarming was just resumed,
                                possibly resulting in delayed reporting
                                of an alarm
@param suspectObjectList        Objects possibly involved in failure.
*/
void environmentalAlarm (
        in ExternalTimeType                     eventTime,
        in NameType                             source,
        in ObjectClassType                      sourceClass,
        in NotifIDType                          notificationIdentifier,
        in CorrelatedNotificationSetType        correlatedNotifications,
        in AdditionalTextType                   additionalText,
        in AdditionalInformationSetType         additionalInfo,
        in ProbableCauseType                    probableCause,
        in SpecificProblemSetType               specificProblems,
```

```
        in PerceivedSeverityType                perceivedSeverity,
        in BooleanTypeOpt                       backedUpStatus,
        in NameType                             backUpObject,
        in TrendIndicationTypeOpt               trendIndication,
        in ThresholdInfoType                    thresholdInfo,
        in AttributeChangeSetType               stateChangeDefinition,
        in AttributeSetType                     monitoredAttributes,
        in ProposedRepairActionSetType          proposedRepairActions,
        in BooleanTypeOpt                       alarmEffectOnService,
        in BooleanTypeOpt                       alarmingResumed,
        in SuspectObjectSetType                 suspectObjectList
);
/** An Equipment Alarm notification is used to report a failure in the
equipment.
@param eventTime                Managed system's current time.
@param source                   Object emitting notification.
@param sourceClass              Actual class of source object.
@param notificationIdentifier   A unique identifier for this
                                notification. Must be unique for
                                an object instance. (Optional in X.721
                                but not here. See text for
                                discussion of possible implications)
@param correlatedNotifications  List of correlated notifications.
                                Optional. Zero length sequence
                                indicates absence of this parameter.
@param additionalText           Text message. Optional. Zero length
                                string indicates absence of this
                                parameter.
@param additionalInfo           Optional. Zero length sequence
                                indicates absence of this parameter.
@param probableCause
@param specificProblems         Optional. Zero length sequence
                                indicates absence of this parameter.
@param perceivedSeverity
@param backedUpStatus           "True" if backed up
@param backUpObject             Will be null if backedUpStatus is
                                "false"
@param trendIndication          Optional. See type for details.
@param thresholdInfo            Optional. See type for details.
@param stateChangeDefinition    Optional. Zero length sequence
                                indicates absence of this parameter.
@param monitoredAttributes      Optional. Zero length sequence
                                indicates absence of this parameter.
@param proposedRepairActions    Optional. Zero length sequence
                                indicates absence of this parameter.
@param alarmEffectOnService     True if alarm is service effecting.
@param alarmingResumed          True if alarming was just resumed,
                                possibly resulting in delayed reporting
                                of an alarm
@param suspectObjectList        Objects possibly involved in failure.
*/
void equipmentAlarm (
        in ExternalTimeType                     eventTime,
        in NameType                             source,
        in ObjectClassType                      sourceClass,
        in NotifIDType                          notificationIdentifier,
        in CorrelatedNotificationSetType        correlatedNotifications,
        in AdditionalTextType                   additionalText,
        in AdditionalInformationSetType         additionalInfo,
        in ProbableCauseType                    probableCause,
        in SpecificProblemSetType               specificProblems,
        in PerceivedSeverityType                perceivedSeverity,
        in BooleanTypeOpt                       backedUpStatus,
        in NameType                             backUpObject,
        in TrendIndicationTypeOpt               trendIndication,
        in ThresholdInfoType                    thresholdInfo,
        in AttributeChangeSetType               stateChangeDefinition,
        in AttributeSetType                     monitoredAttributes,
        in ProposedRepairActionSetType          proposedRepairActions,
        in BooleanTypeOpt                       alarmEffectOnService,
```

```
        in BooleanTypeOpt                       alarmingResumed,
        in SuspectObjectSetType                 suspectObjectList
);
/** An Integrity Violation notification is used to report that a
potential interruption in information flow has occurred such that
information may have been illegally modified, inserted or deleted.
@param eventTime                Managed system's current time.
@param source                   Object emitting notification.
@param sourceClass              Actual class of source object.
@param notificationIdentifier   A unique identifier for this
                                notification. Must be unique for
                                an object instance. (Optional in X.721
                                but not here. See text for
                                discussion of possible implications)
@param correlatedNotifications  List of correlated notifications.
                                Optional. Zero length sequence
                                indicates absence of this parameter.
@param additionalText           Text message. Optional. Zero length
                                string indicates absence of this
                                parameter.
@param additionalInfo           Optional. Zero length sequence
                                indicates absence of this parameter.
@param securityAlarmCause
@param securityAlarmSeverity    Clears allowed? X.721 appears to
                                restrict the "cleared" value on this
                                alarm but clears should be allowed.
@param securityAlarmDetector
@param serviceUser
@param serviceProvider
*/
void integrityViolation (
        in ExternalTimeType                     eventTime,
        in NameType                             source,
        in ObjectClassType                      sourceClass,
        in NotifIDType                          notificationIdentifier,
        in CorrelatedNotificationSetType        correlatedNotifications,
        in AdditionalTextType                   additionalText,
        in AdditionalInformationSetType         additionalInfo,
        in SecurityAlarmCauseType               securityAlarmCause,
        in PerceivedSeverityType                securityAlarmSeverity,
        in SecurityAlarmDetectorType            securityAlarmDetector,
        in ServiceUserType                      serviceUser,
        in ServiceProviderType                  serviceProvider
);
/** An Object Creation notification is used to report the creation of a
managed object to another open system. Note that the source field
should be set to the created object, not the factory.
@param eventTime                Managed system's current time.
@param source                   Object emitting notification.
@param sourceClass              Actual class of source object.
@param notificationIdentifier   A unique identifier for this
                                notification. Must be unique for
                                an object instance. (Optional in X.721
                                but not here. See text for
                                discussion of possible implications)
@param correlatedNotifications  List of correlated notifications.
                                Optional. Zero length sequence
                                indicates absence of this parameter.
@param additionalText           Text message. Optional. Zero length
                                string indicates absence of this
                                parameter.
@param additionalInfo           Optional. Zero length sequence
                                indicates absence of this parameter.
@param sourceIndicator          Cause of event. Optional. Use
                                "unknown" if not supported.
@param attributeSet             Attribute values. Optional. Zero length
                                sequence indicates absence of this
                                parameter.
*/
void objectCreation (
        in ExternalTimeType                     eventTime,
```

```
        in NameType                             source,
        in ObjectClassType                      sourceClass,
        in NotifIDType                          notificationIdentifier,
        in CorrelatedNotificationSetType        correlatedNotifications,
        in AdditionalTextType                   additionalText,
        in AdditionalInformationSetType         additionalInfo,
        in SourceIndicatorType                  sourceIndicator,
        in AttributeSetType                     attributeList
);
/** An Object Deletion notification is used to report the deletion of a
managed object. Note that the source field should be set to
the object being deleted.
@param eventTime                Managed system's current time.
@param source                   Object emitting notification.
@param sourceClass              Actual class of source object.
@param notificationIdentifier   A unique identifier for this
                                notification. Must be unique for
                                an object instance. (Optional in X.721
                                but not here. See text for
                                discussion of possible implications)
@param correlatedNotifications  List of correlated notifications.
                                Optional. Zero length sequence
                                indicates absence of this parameter.
@param additionalText           Text message. Optional. Zero length
                                string indicates absence of this
                                parameter.
@param additionalInfo           Optional. Zero length sequence
                                indicates absence of this parameter.
@param sourceIndicator          Cause of event. Optional. Use
                                "unknown" if not supported.
@param attributeSet             Attribute values. Optional. Zero length
                                sequence indicates absence of this
                                parameter.
*/
void objectDeletion (
        in ExternalTimeType                     eventTime,
        in NameType                             source,
        in ObjectClassType                      sourceClass,
        in NotifIDType                          notificationIdentifier,
        in CorrelatedNotificationSetType        correlatedNotifications,
        in AdditionalTextType                   additionalText,
        in AdditionalInformationSetType         additionalInfo,
        in SourceIndicatorType                  sourceIndicator,
        in AttributeSetType                     attributeList
);
/** An Operational Violation notification is used to report that the
provision of the requested service was not possible due to the
unavailability, malfunction or incorrect invocation of the service.
@param eventTime                Managed system's current time.
@param source                   Object emitting notification.
@param sourceClass              Actual class of source object.
@param notificationIdentifier   A unique identifier for this
                                notification. Must be unique for
                                an object instance. (Optional in X.721
                                but not here. See text for
                                discussion of possible implications)
@param correlatedNotifications  List of correlated notifications.
                                Optional. Zero length sequence
                                indicates absence of this parameter.
@param additionalText           Text message. Optional. Zero length
                                string indicates absence of this
                                parameter.
@param additionalInfo           Optional. Zero length sequence
                                indicates absence of this parameter.
@param securityAlarmCause
@param securityAlarmSeverity    Clears allowed? X.721 appears to
                                restrict the "cleared" value on this
                                alarm but clears should be allowed.
@param securityAlarmDetector
@param serviceUser
```

```
@param serviceProvider
*/
void operationalViolation (
        in ExternalTimeType                     eventTime,
        in NameType                             source,
        in ObjectClassType                      sourceClass,
        in NotifIDType                          notificationIdentifier,
        in CorrelatedNotificationSetType        correlatedNotifications,
        in AdditionalTextType                   additionalText,
        in AdditionalInformationSetType         additionalInfo,
        in SecurityAlarmCauseType               securityAlarmCause,
        in PerceivedSeverityType                securityAlarmSeverity,
        in SecurityAlarmDetectorType            securityAlarmDetector,
        in ServiceUserType                      serviceUser,
        in ServiceProviderType                  serviceProvider
);
/** A Physical Violation notification is used to report that a physical
resource has been violated in a way that indicates a potential security
attack.
@param eventTime                Managed system's current time.
@param source                   Object emitting notification.
@param sourceClass              Actual class of source object.
@param notificationIdentifier   A unique identifier for this
                                notification. Must be unique for
                                an object instance. (Optional in X.721
                                but not here. See text for
                                discussion of possible implications)
@param correlatedNotifications  List of correlated notifications.
                                Optional. Zero length sequence
                                indicates absence of this parameter.
@param additionalText           Text message. Optional. Zero length
                                string indicates absence of this
                                parameter.
@param additionalInfo           Optional. Zero length sequence
                                indicates absence of this parameter.
@param securityAlarmCause
@param securityAlarmSeverity    Clears allowed? X.721 appears to
                                restrict the "cleared" value on this
                                alarm but clears should be allowed.
@param securityAlarmDetector
@param serviceUser
@param serviceProvider
*/
void physicalViolation (
        in ExternalTimeType                     eventTime,
        in NameType                             source,
        in ObjectClassType                      sourceClass,
        in NotifIDType                          notificationIdentifier,
        in CorrelatedNotificationSetType        correlatedNotifications,
        in AdditionalTextType                   additionalText,
        in AdditionalInformationSetType         additionalInfo,
        in SecurityAlarmCauseType               securityAlarmCause,
        in PerceivedSeverityType                securityAlarmSeverity,
        in SecurityAlarmDetectorType            securityAlarmDetector,
        in ServiceUserType                      serviceUser,
        in ServiceProviderType                  serviceProvider
);
/** A Processing Error Alarm notification is used to report a
processing failure in a managed object.
@param eventTime                Managed system's current time.
@param source                   Object emitting notification.
@param sourceClass              Actual class of source object.
@param notificationIdentifier   A unique identifier for this
                                notification. Must be unique for
                                an object instance. (Optional in X.721
                                but not here. See text for
                                discussion of possible implications)
@param correlatedNotifications  List of correlated notifications.
                                Optional. Zero length sequence
                                indicates absence of this parameter.
```

```
@param additionalText           Text message. Optional. Zero length
                                string indicates absence of this
                                parameter.
@param additionalInfo           Optional. Zero length sequence
                                indicates absence of this parameter.
@param probableCause
@param specificProblems         Optional. Zero length sequence
                                indicates absence of this parameter.
@param perceivedSeverity
@param backedUpStatus           "True" if backed up
@param backUpObject             Will be null if backedUpStatus is
                                "false"
@param trendIndication          Optional. See type for details.
@param thresholdInfo            Optional. See type for details.
@param stateChangeDefinition    Optional. Zero length sequence
                                indicates absence of this parameter.
@param monitoredAttributes      Optional. Zero length sequence
                                indicates absence of this parameter.
@param proposedRepairActions    Optional. Zero length sequence
                                indicates absence of this parameter.
@param alarmEffectOnService     True if alarm is service effecting.
@param alarmingResumed          True if alarming was just resumed,
                                possibly resulting in delayed reporting
                                of an alarm
@param suspectObjectList        Objects possibly involved in failure.
*/
void processingErrorAlarm (
        in ExternalTimeType                     eventTime,
        in NameType                             source,
        in ObjectClassType                      sourceClass,
        in NotifIDType                          notificationIdentifier,
        in CorrelatedNotificationSetType        correlatedNotifications,
        in AdditionalTextType                   additionalText,
        in AdditionalInformationSetType         additionalInfo,
        in ProbableCauseType                    probableCause,
        in SpecificProblemSetType               specificProblems,
        in PerceivedSeverityType                perceivedSeverity,
        in BooleanTypeOpt                       backedUpStatus,
        in NameType                             backUpObject,
        in TrendIndicationTypeOpt               trendIndication,
        in ThresholdInfoType                    thresholdInfo,
        in AttributeChangeSetType               stateChangeDefinition,
        in AttributeSetType                     monitoredAttributes,
        in ProposedRepairActionSetType          proposedRepairActions,
        in BooleanTypeOpt                       alarmEffectOnService,
        in BooleanTypeOpt                       alarmingResumed,
        in SuspectObjectSetType                 suspectObjectList
);
/** A Quality of Service Alarm notification is used to report a failure
in the quality of service of the managed object.
@param eventTime                Managed system's current time.
@param source                   Object emitting notification.
@param sourceClass              Actual class of source object.
@param notificationIdentifier   A unique identifier for this
                                notification. Must be unique for
                                an object instance. (Optional in X.721
                                but not here. See text for
                                discussion of possible implications)
@param correlatedNotifications  List of correlated notifications.
                                Optional. Zero length sequence
                                indicates absence of this parameter.
@param additionalText           Text message. Optional. Zero length
                                string indicates absence of this
                                parameter.
@param additionalInfo           Optional. Zero length sequence
                                indicates absence of this parameter.
@param probableCause
@param specificProblems         Optional. Zero length sequence
                                indicates absence of this parameter.
@param perceivedSeverity
@param backedUpStatus           "True" if backed up
```

```
@param backUpObject             Will be null if backedUpStatus is
                                "false"
@param trendIndication          Optional. See type for details.
@param thresholdInfo            Optional. See type for details.
@param stateChangeDefinition    Optional. Zero length sequence
                                indicates absence of this parameter.
@param monitoredAttributes      Optional. Zero length sequence
                                indicates absence of this parameter.
@param proposedRepairActions    Optional. Zero length sequence
                                indicates absence of this parameter.
@param alarmEffectOnService     True if alarm is service effecting.
@param alarmingResumed          True if alarming was just resumed,
                                possibly resulting in delayed reporting
                                of an alarm
@param suspectObjectList        Objects possibly involved in failure.
*/
void qualityOfServiceAlarm (
        in ExternalTimeType                     eventTime,
        in NameType                             source,
        in ObjectClassType                      sourceClass,
        in NotifIDType                          notificationIdentifier,
        in CorrelatedNotificationSetType        correlatedNotifications,
        in AdditionalTextType                   additionalText,
        in AdditionalInformationSetType         additionalInfo,
        in ProbableCauseType                    probableCause,
        in SpecificProblemSetType               specificProblems,
        in PerceivedSeverityType                perceivedSeverity,
        in BooleanTypeOpt                       backedUpStatus,
        in NameType                             backUpObject,
        in TrendIndicationTypeOpt               trendIndication,
        in ThresholdInfoType                    thresholdInfo,
        in AttributeChangeSetType               stateChangeDefinition,
        in AttributeSetType                     monitoredAttributes,
        in ProposedRepairActionSetType          proposedRepairActions,
        in BooleanTypeOpt                       alarmEffectOnService,
        in BooleanTypeOpt                       alarmingResumed,
        in SuspectObjectSetType                 suspectObjectList
);
/** A Relationship Change notification is used to report the change in
the value of one or more relationship attributes of a managed object,
that result through either internal operation of the managed object or
via management operation.
@param eventTime                Managed system's current time.
@param source                   Object emitting notification.
@param sourceClass              Actual class of source object.
@param notificationIdentifier   A unique identifier for this
                                notification. Must be unique for
                                an object instance. (Optional in X.721
                                but not here. See text for
                                discussion of possible implications)
@param correlatedNotifications  List of correlated notifications.
                                Optional. Zero length sequence
                                indicates absence of this parameter.
@param additionalText           Text message. Optional. Zero length
                                string indicates absence of this
                                parameter.
@param additionalInfo           Optional. Zero length sequence
                                indicates absence of this parameter.
@param sourceIndicator          Cause of event. Optional. Use
                                "unknown" if not supported.
@param relationshipChanges      Changed relationship attributes
*/
void relationshipChange (
        in ExternalTimeType                     eventTime,
        in NameType                             source,
        in ObjectClassType                      sourceClass,
        in NotifIDType                          notificationIdentifier,
        in CorrelatedNotificationSetType        correlatedNotifications,
        in AdditionalTextType                   additionalText,
        in AdditionalInformationSetType         additionalInfo,
```

```
        in SourceIndicatorType                  sourceIndicator,
        in AttributeChangeSetType               relationshipChanges
);
/** A Security Violation notification is used to report that a security
attack has been detected by a security service or mechanism.
@param eventTime                Managed system's current time.
@param source                   Object emitting notification.
@param sourceClass              Actual class of source object.
@param notificationIdentifier   A unique identifier for this
                                notification. Must be unique for
                                an object instance. (Optional in X.721
                                but not here. See text for
                                discussion of possible implications)
@param correlatedNotifications  List of correlated notifications.
                                Optional. Zero length sequence
                                indicates absence of this parameter.
@param additionalText           Text message. Optional. Zero length
                                string indicates absence of this
                                parameter.
@param additionalInfo           Optional. Zero length sequence
                                indicates absence of this parameter.
@param securityAlarmCause
@param securityAlarmSeverity    Clears allowed? X.721 appears to
                                restrict the "cleared" value on this
                                alarm but clears should be allowed.
@param securityAlarmDetector
@param serviceUser
@param serviceProvider
*/
void securityViolation (
        in ExternalTimeType                     eventTime,
        in NameType                             source,
        in ObjectClassType                      sourceClass,
        in NotifIDType                          notificationIdentifier,
        in CorrelatedNotificationSetType        correlatedNotifications,
        in AdditionalTextType                   additionalText,
        in AdditionalInformationSetType         additionalInfo,
        in SecurityAlarmCauseType               securityAlarmCause,
        in PerceivedSeverityType                securityAlarmSeverity,
        in SecurityAlarmDetectorType            securityAlarmDetector,
        in ServiceUserType                      serviceUser,
        in ServiceProviderType                  serviceProvider
);
/** A State Change notification is used to report the change in the the
value of one or more state attributes of a managed object, that result
through either internal operation of the managed object or via
management operation.
@param eventTime                Managed system's current time.
@param source                   Object emitting notification.
@param sourceClass              Actual class of source object.
@param notificationIdentifier   A unique identifier for this
                                notification. Must be unique for
                                an object instance. (Optional in X.721
                                but not here. See text for
                                discussion of possible implications)
@param correlatedNotifications  List of correlated notifications.
                                Optional. Zero length sequence
                                indicates absence of this parameter.
@param additionalText           Text message. Optional. Zero length
                                string indicates absence of this
                                parameter.
@param additionalInfo           Optional. Zero length sequence
                                indicates absence of this parameter.
@param sourceIndicator          Cause of event. Optional. Use
                                "unknown" if not supported.
@param stateChanges             Changed state attributes.
*/
void stateChange (
        in ExternalTimeType                     eventTime,
        in NameType                             source,
        in ObjectClassType                      sourceClass,
```

```
        in NotifIDType                          notificationIdentifier,
        in CorrelatedNotificationSetType        correlatedNotifications,
        in AdditionalTextType                   additionalText,
        in AdditionalInformationSetType         additionalInfo,
        in SourceIndicatorType                  sourceIndicator,
        in AttributeChangeSetType               stateChanges
);
/** A Time Domain Violation notification is used to report that an
event has occurred at an unexpected or prohibited time.
@param eventTime                Managed system's current time.
@param source                   Object emitting notification.
@param sourceClass              Actual class of source object.
@param notificationIdentifier   A unique identifier for this
                                notification. Must be unique for
                                an object instance. (Optional in X.721
                                but not here. See text for
                                discussion of possible implications)
@param correlatedNotifications  List of correlated notifications.
                                Optional. Zero length sequence
                                indicates absence of this parameter.
@param additionalText           Text message. Optional. Zero length
                                string indicates absence of this
                                parameter.
@param additionalInfo           Optional. Zero length sequence
                                indicates absence of this parameter.
@param securityAlarmCause
@param securityAlarmSeverity    Clears allowed? X.721 appears to
                                restrict the "cleared" value on this
                                alarm but clears should be allowed.
@param securityAlarmDetector
@param serviceUser
@param serviceProvider
*/
void timeDomainViolation (
        in ExternalTimeType                     eventTime,
        in NameType                             source,
        in ObjectClassType                      sourceClass,
        in NotifIDType                          notificationIdentifier,
        in CorrelatedNotificationSetType        correlatedNotifications,
        in AdditionalTextType                   additionalText,
        in AdditionalInformationSetType         additionalInfo,
        in SecurityAlarmCauseType               securityAlarmCause,
        in PerceivedSeverityType                securityAlarmSeverity,
        in SecurityAlarmDetectorType            securityAlarmDetector,
        in ServiceUserType                      serviceUser,
        in ServiceProviderType                  serviceProvider
);
/** These constants define the names of the notifications declared
above and are provided to help reduce errors. */
const string attributeValueChangeTypeName =
        "itut_x780::Notifications::attributeValueChange";
const string communicationsAlarmTypeName =
        "itut_x780::Notifications::communicationsAlarm";
const string environmentalAlarmTypeName =
        "itut_x780::Notifications::environmentalAlarm";
const string equipmentAlarmTypeName =
        "itut_x780::Notifications::equipmentAlarm";
const string integrityViolationTypeName =
        "itut_x780::Notifications::integrityViolation";
const string objectCreationTypeName =
        "itut_x780::Notifications::objectCreation";
const string objectDeletionTypeName =
        "itut_x780::Notifications::objectDeletion";
const string operationalViolationTypeName =
        "itut_x780::Notifications::operationalViolation";
const string physicalViolationTypeName =
        "itut_x780::Notifications::physicalViolation";
const string processingErrorAlarmTypeName =
        "itut_x780::Notifications::processingErrorAlarm";
const string qualityOfServiceAlarmTypeName =
        "itut_x780::Notifications::qualityOfServiceAlarm";
```

```
        const string relationshipChangeTypeName =
                "itut_x780::Notifications::relationshipChange";
        const string securityViolationTypeName =
                "itut_x780::Notifications::securityViolation";
        const string stateChangeTypeName =
                "itut_x780::Notifications::stateChange";
        const string timeDomainViolationTypeName =
                "itut_x780::Notifications::timeDomainViolation";
        /** These constants define the names of the parameters used in the
        notifications declared above and are provided to help reduce errors.
        */
        const string additionalInfoName = "additionalInfo";
        const string additionalTextName = "additionalText";
        const string alarmEffectOnServiceName = "alarmEffectOnService";
        const string alarmingResumedName = "alarmingResumed";
        const string attributeChangesName = "attributeChanges";
        const string attributeListName = "attributeList";
        const string backedUpStatusName = "backedUpStatus";
        const string backUpObjectName = "backUpObject";
        const string correlatedNotificationsName = "correlatedNotifications";
        const string eventTimeName = "eventTime";
        const string monitoredAttributesName = "monitoredAttributes";
        const string notificationIdentifierName = "notificationIdentifier";
        const string perceivedSeverityName = "perceivedSeverity";
        const string probableCauseName = "probableCause";
        const string proposedRepairActionsName = "proposedRepairActions";
        const string relationshipChangesName = "relationshipChanges";
        const string securityAlarmCauseName = "securityAlarmCause";
        const string securityAlarmDetectorName = "securityAlarmDetector";
        const string securityAlarmSeverityName = "securityAlarmSeverity";
        const string serviceProviderName = "serviceProvider";
        const string serviceUserName = "serviceUser";
        const string sourceName = "source";
        const string sourceClassName = "sourceClass";
        const string sourceIndicatorName = "sourceIndicator";
        const string specificProblemsName = "specificProblems";
        const string stateChangeDefinitionName = "stateChangeDefinition";
        const string stateChangesName = "stateChanges";
        const string suspectObjectListName = "suspectObjectList";
        const string thresholdInfoName = "thresholdInfo";
        const string trendIndicationName = "trendIndication";
  }; // end of Notifications interface

}; // end of itut_x780 module
```

// MACROS

```
/* The following macros are provided for quickly and concisely defining
the notifications to be supported by an object. Example usage (within an
interface):
MANDATORY_NOTIFICATION(itut_x780::Notifications, objectCreation);
CONDITIONAL_NOTIFICATION(itut_x780::Notifications, stateChange, statePackage);
The macros simply expand into nothing, as CORBA IDL doesn't really have
anything for them to expand into that makes sense. Eventually, these
may be changed to expand into IDL supporting the CORBA Component Model.
*/
#undef MANDATORY_NOTIFICATION
#define MANDATORY_NOTIFICATION(InterfaceName, NotificationName)
#undef CONDITIONAL_NOTIFICATION
#define CONDITIONAL_NOTIFICATION(InterfaceName, NotificationName, PackageName)
#endif // end of ifndef itut_x780_IDL
```

ANNEXE B

Définitions des constantes de gestion de réseau

```
/* This IDL code is intended to be stored in a file named "itut_x780Const.idl"
and located in the same directory as the file containing Annex A */
#ifndef ITUT_X780Const_IDL
#define ITUT_X780Const_IDL
#pragma prefix "itu.int"
module itut_x780 {
```

// ApplicationErrorConst Module

```
/** This module contains the constants defined for the error code contained in
Application Error Info structures returned with Application Error exceptions.
*/
module ApplicationErrorConst {
        const string moduleName = "itut_x780::ApplicationErrorConst";
        /** This application error exception code indicates the operation
          failed due to a problem downstream from the managed system,
          possibly a communication problem between the managed system
          and the resource */
        const short downstreamError = 1;
        /** An application error exception returining this code will return
          the name of the offending paramter in the details field. */
        const short invalidParameter = 2;
        /** This application error exception code indicates the operation
          failed due to a transient problem on the managed system. */
        const short resourceLimit = 3;
}; // end of module ApplicationErrorConst
```

// CreateErrorConst Module

```
/** This module contains the constants defined for the error code contained in
Create Error Info structures returned with Create Error exceptions.
*/
module CreateErrorConst {
        const string moduleName = "itut_x780::CreateErrorConst";

        /** This create error exception code indicates that the name included
          in the create operation is not valid. */
        const short badName = 1;
        /** This create error exception code indicates that the name included
          in the create operation is a duplicate. */
        const short duplicateName = 2;
        /** This create error exception code indicates some packages requested
          in the create operation are incompatible with each other. It must
          be included in a PackageErrorInfoType structure (subclass of
          CreateErrorInfoType). The packages list contains the names of the
          unsupported packages. */
        const short incompatiblePackages = 3;
        /** This create error exception code indicates that the name binding
          referenced in the create operation is not valid. */
        const short invalidNameBinding = 4;
        /** This create error exception code indicates a package requested in
          the create operation is not supported. It must be included in a
          PackageErrorInfoType structure (subclass of CreateErrorInfoType).
          The packages list contains the names of the unsupported packages.
          */
        const short unsupportedPackages = 5;
}; // end of module CreateErrorConst
```

// DeleteErrorConst Module

```
/** This module contains the constants defined for the error code contained in
Delete Error Info structures returned with Delete Error exceptions.
*/
```

```
module DeleteErrorConst {
        const string moduleName = "itut_x780::DeleteErrorConst";
        /** This delete error exceptin code indicates the object has both
          subordinates and a delete policy of deleteOnlyIfNoContained. */
        const short containsObjects = 1;
        /** This delete error exception code indicates the object has a delete
          policy of notDeletable, and cannot be deleted. */
        const short notDeletable = 2;
        /** This delete error exception code indicates the object had a
          subordinate object that could not be deleted, so the superior
          object(s) could not be deleted. */
        const short undeletableContainedObject = 3;
        /** This delete error exception code indicates the object is in
          a state in which it cannot be deleted. */
        const short invalidStateForDestroy = 4;
}; // end of module DeleteErrorConst
```

// ProbableCauseConst Module

```
/** This module contains the constant values defined for the
ProbableCause UID. These values were borrowed from X.721. */
module ProbableCauseConst {
const string moduleName = "itut_x780::ProbableCauseConst";
        const short indeterminate = 0;
        const short adapterError = 1;
        const short applicationSubsystemFailure = 2;
        const short bandwidthReduced = 3;
        const short callEstablishmentError = 4;
        const short communicationsProtocolError = 5;
        const short communicationsSubsystemFailure = 6;
        const short configurationOrCustomizationError = 7;
        const short congestion = 8;
        const short corruptData = 9;
        const short cpuCyclesLimitExceeded = 10;
        const short dataSetOrModemError = 11;
        const short degradedSignal = 12;
        const short dTE_DCEInterfaceError = 13;
        const short enclosureDoorOpen = 14;
        const short equipmentMalfunction = 15;
        const short excessiveVibration = 16;
        const short fileError = 17;
        const short fireDetected = 18;
        const short floodDetected = 19;
        const short framingError = 20;
        const short heatingOrVentilationOrCoolingSystemProblem = 21;
        const short humidityUnacceptable = 22;
        const short inputOutputDeviceError = 23;
        const short inputDeviceError = 24;
        const short lANError = 25;
        const short leakDetected = 26;
        const short localNodeTransmissionError = 27;
        const short lossOfFrame = 28;
        const short lossOfSignal = 29;
        const short materialSupplyExhausted = 30;
        const short multiplexerProblem = 31;
        const short outOfMemory = 32;
        const short ouputDeviceError = 33;
        const short performanceDegraded = 34;
        const short powerProblem = 35;
        const short pressureUnacceptable = 36;
        const short processorProblem = 37;
        const short pumpFailure = 38;
        const short queueSizeExceeded = 39;
        const short receiveFailure = 40;
        const short receiverFailure = 41;
        const short remoteNodeTransmissionError = 42;
        const short resourceAtOrNearingCapacity = 43;
        const short responseTimeExcessive = 44;
        const short retransmissionRateExcessive = 45;
        const short softwareError = 46;
        const short softwareProgramAbnormallyTerminated = 47;
```

```
        const short softwareProgramError = 48;
        const short storageCapacityProblem = 49;
        const short temperatureUnacceptable = 50;
        const short thresholdCrossed = 51;
        const short timingProblem = 52;
        const short toxicLeakDetected = 53;
        const short transmitFailure = 54;
        const short transmitterFailure = 55;
        const short underlyingResourceUnavailable = 56;
        const short versionMismatch = 57;
}; // end of ProbableCauseConst module

// SecurityAlarmCauseConst Module
/** This module contains the constant values defined for the
SecurityAlarmCause UID. These values were borrowed from
X.721. */
module SecurityAlarmCauseConst {
const string moduleName = "itut_x780::SecurityAlarmCauseConst";
        const short authenticationFailure = 1;
        const short breachOfConfidentiality = 2;
        const short cableTamper = 3;
        const short delayedInformation = 4;
        const short denialOfService = 5;
        const short duplicateInformation = 6;
        const short informationMissing = 7;
        const short informationModificationDetected = 8;
        const short informationOutOfSequence = 9;
        const short intrusionDetection = 10;
        const short keyExpired = 11;
        const short nonRepudiationFailure = 12;
        const short outOfHoursActivity = 13;
        const short outOfService = 14;
        const short proceduralError = 15;
        const short unauthorizedAccessAttempt = 16;
        const short unexpectedInformation = 17;
        const short unspecifiedReason = 18;
}; // end of SecurityAlarmCauseConst module

}; // end of itut_x780 module
#endif // end of ifndef ITUT_X780Const_IDL
```

APPENDICE I

Bibliographie

Les Recommandations et autres références ci-après contiennent des informations qui ont été utilisées dans l'élaboration des présentes lignes directrices, dont un des principaux objectifs, comme indiqué dans l'introduction, est de permettre la réutilisation des modèles d'information de gestion du réseau existants, du moins sans modifications sémantiques d'importance. Les documents cités contiennent nombre de précisions relatives au cadre du protocole CMIP de l'UIT-T, et pour cette raison définissent quelques-unes des fonctionnalités que ces lignes directrices pour la modélisation des objets CORBA doivent prendre en charge.

[8] UIT-T X.720 (1992) | ISO/IEC 10165-1:1993, *Technologies de l'information – Interconnexion des systèmes ouverts – Structure des informations de gestion: modèle d'information de gestion.*

[9] UIT-T X.733 (1992) | ISO/IEC 10164-4:1992, *Technologies de l'information – Interconnexion des systèmes ouverts – Gestion-systèmes: fonction de signalisation des alarmes.*

[10] UIT-T M.3010 (2000), *Principes des réseaux de gestion des télécommunications.*

[11] UIT-T M.3120 (2001), *Modèle informationnel de réseau générique d'architecture CORBA et de niveau élément de réseau*.

[12] UIT-T Q.821 (2000), *Description d'étape 2 et d'étape 3 de l'interface Q3 – Supervision des alarmes*.

SÉRIES DES RECOMMANDATIONS UIT-T

Série A	Organisation du travail de l'UIT-T
Série B	Moyens d'expression: définitions, symboles, classification
Série C	Statistiques générales des télécommunications
Série D	Principes généraux de tarification
Série E	Exploitation générale du réseau, service téléphonique, exploitation des services et facteurs humains
Série F	Services de télécommunication non téléphoniques
Série G	Systèmes et supports de transmission, systèmes et réseaux numériques
Série H	Systèmes audiovisuels et multimédias
Série I	Réseau numérique à intégration de services
Série J	Transmission des signaux radiophoniques, télévisuels et autres signaux multimédias
Série K	Protection contre les perturbations
Série L	Construction, installation et protection des câbles et autres éléments des installations extérieures
Série M	RGT et maintenance des réseaux: systèmes de transmission, de télégraphie, de télécopie, circuits téléphoniques et circuits loués internationaux
Série N	Maintenance: circuits internationaux de transmission radiophonique et télévisuelle
Série O	Spécifications des appareils de mesure
Série P	Qualité de transmission téléphonique, installations téléphoniques et réseaux locaux
Série Q	Commutation et signalisation
Série R	Transmission télégraphique
Série S	Equipements terminaux de télégraphie
Série T	Terminaux des services télématiques
Série U	Commutation télégraphique
Série V	Communications de données sur le réseau téléphonique
Série X	**Réseaux de données et communication entre systèmes ouverts**
Série Y	Infrastructure mondiale de l'information et protocole Internet
Série Z	Langages et aspects informatiques généraux des systèmes de télécommunication

Genève, 2001