UNION INTERNATIONALE DES TÉLÉCOMMUNICATIONS

UIT-T Z.130

(02/99)

SECTEUR DE LA NORMALISATION
DES TÉLÉCOMMUNICATIONS
DE L'UIT

SÉRIE Z: LANGAGES ET ASPECTS INFORMATIQUES GÉNÉRAUX DES SYSTÈMES DE TÉLÉCOMMUNICATION

Techniques de description formelle

Langage de définition d'objet de l'UIT

Recommandation UIT-T Z.130

(Antérieurement Recommandation du CCITT)

RECOMMANDATIONS UIT-T DE LA SÉRIE Z

LANGAGES ET ASPECTS INFORMATIQUES GÉNÉRAUX DES SYSTÈMES DE TÉLÉCOMMUNICATION

TECHNIQUES DE DESCRIPTION FORMELLE	
Langage de description et de spécification (SDL)	Z.100–Z.109
Application des techniques de description formelle	Z.110–Z.119
Diagrammes des séquences de messages	Z.120–Z.129
LANGAGES DE PROGRAMMATION	
CHILL: le langage de haut niveau de l'UIT-T	Z.200–Z.209
LANGAGE HOMME-MACHINE	
Principes généraux	Z.300–Z.309
Syntaxe de base et procédures de dialogue	Z.310–Z.319
LHM étendu pour terminaux à écrans de visualisation	Z.320–Z.329
Spécification de l'interface homme-machine	Z.330–Z.399
QUALITÉ DES LOGICIELS DE TÉLÉCOMMUNICATION	Z.400–Z.499
MÉTHODES DE VALIDATION ET D'ESSAI	Z.500–Z.599

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

RECOMMANDATION UIT-T Z.130

LANGAGE DE DEFINITION D'OBJET DE L'UIT

Résumé

La présente Recommandation spécifie le langage de définition d'objet de l'UIT (ITU-ODL, *ITU object definition language*). L'ODL est utilisé pour spécifier des systèmes du point de vue traitement réparti ouvert (ODP, *open distributed processing*) [3]. Elle contient les définitions des modèles d'interfaces opérationnelles, d'interfaces de flux, d'objets d'interface multiple et de groupes d'objets.

L'ITU-ODL est une extension du langage de définition d'interface ODP-IDL [8] avec des adjonctions permettant la prise en charge de la spécification des concepts de point de vue du traitement ODP à un niveau syntaxique. L'ITU-ODL est un surensemble de l'ODP-IDL. Cette relation entre l'ITU-ODL et l'ODP-IDL accepte la réalisation de systèmes via les réalisations de courtier de requêtes d'objets (ORB, *object request broker*) spécifiée par le groupe de gestion d'objets (OMG, *object management group*) [1]. On suppose que le lecteur connaît le langage ODP-IDL.

Source

La Recommandation UIT-T Z.130, élaborée par la Commission d'études 10 (1997-2000) de l'UIT-T, a été approuvée le 12 février 1999 selon la procédure définie dans la Résolution n° 1 de la CMNT.

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.

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

L'approbation des Recommandations par les Membres de l'UIT-T s'effectue selon la procédure définie dans la Résolution n° 1 de la CMNT.

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, le terme *exploitation reconnue (ER)* désigne tout particulier, toute entreprise, toute société ou tout organisme public qui exploite un service de correspondance publique. Les termes *Administration*, *ER* et *correspondance publique* sont définis dans la *Constitution de l'UIT (Genève, 1992)*.

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

Page

Recommandation Z.130

LANGAGE DE DEFINITION D'OBJET DE L'UIT

(Genève, 1999)

1 Domaine d'application

L'ITU-ODL a été élaboré pour:

- donner les spécifications de traitement. Ainsi, par exemple, une composante de service peut être décrite du point de vue traitement au moyen d'une spécification ITU-ODL;
- disposer d'une syntaxe pour le développement d'applications d'aides au génie logiciel tels les analyseurs syntaxiques ITU-ODL, les générateurs de codes, les éditeurs de spécifications de traitement et les outils CASE associés.

L'ITU-ODL est une extension du langage de définition d'interface du groupe de gestion d'objets (ODP-IDL) [8]. L'ITU-ODL prend en charge des caractéristiques qui ne sont pas (actuellement) couvertes par l'ODP-IDL. Ces caractéristiques trouvent leur origine dans le langage de traitement du modèle de référence ODP [3] et inclut plusieurs modèles d'objets d'interface, de modèles de groupes, de modèles d'interfaces de flux et descriptions de qualité de service (QS)[1]. On trouvera dans l'Appendice II une comparaison entre l'ITU-ODL et l'ODP-IDL.

Il convient de noter que l'ITU-ODL est fortement influencé par les travaux du consortium TINA et s'inspire du langage TINA-ODL [4]. Toutefois, certains concepts n'ont pas pu être adoptés et il est nécessaire de les modifier. En particulier, le concept de modèle de groupe a été doté d'une nouvelle sémantique.

L'ITU-ODL définit une syntaxe permettant de décrire les aspects statiques du point de vue du traitement des systèmes ODP. Cela signifie que le langage couvre les structures et les signatures de ces systèmes, mais pas le comportement (de manière formelle). Les aspects qui peuvent s'exprimer par la syntaxe ITU-ODL sont les suivants:

- la description des modèles d'objets de traitement qui prennent en charge les modèles d'interfaces multiples. Les modèles d'interface différents pris en charge représentent des services distincts assurés par le même objet;
- la capacité d'un objet à traiter des instances multiples d'un même modèle d'interface. Cela permet à un objet de conserver des contextes propres au client;
- la capacité à contrôler l'accès et la visibilité de parties d'une fonctionnalité d'objet;
- la possibilité de décrire les services ou la fonctionnalité dont un objet a besoin de la part de son environnement. Cette caractéristique permet de vérifier la compatibilité des modèles d'interface à un niveau de spécification statique;
- la capacité de structuration de la spécification par des modèles de groupes d'objets. Les modèles d'objet qui partagent une propriété commune peuvent être regroupés dans un modèle de groupe d'objets. On peut citer par exemple des aspects réalisation ou des problèmes de gestion;

[1] La notation de qualité de service est présentée à l'Appendice I.

- la possibilité de décrire des aspects statiques de modèles de d'interfaces flux;
- la capacité à associer des attributs de QS à des opérations et à des flux (ce concept est décrit dans l'Appendice I).

L'ITU-ODL est la base de la description de composantes logicielles réutilisables.

2 Références

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] Spécification OMG adoptée (1998), *The Common Object Request Broker: Architecture and Specification, Revision 2.2*.

[2] Recommandation UIT-T X.902 (1995) | ISO/CEI 10746-2:1995, *Technologies de l'information – Traitement réparti ouvert – Modèle de référence: Fondements.*

[3] Recommandation UIT-T X.903 (1995) | ISO/CEI 10746-3:1995, *Technologies de l'information – Traitement réparti ouvert – Modèle de référence de base: Architecture.*

[4] Spécification TINA-C (1996), *Object Definition Language Manual, Version 2.3*.

[5] Recommandation UIT-T Z.100 (1993), *Langage de description et de spécification du CCITT*.

[6] Recommandation UIT-T Z.100 (1993)/Add.1 (1996), *Langage de description et de spécification du CCITT – Addendum 1*.

[7] Recommandation UIT-T Z.105 (1995), *Langage de description et de spécification combiné avec la notation de syntaxe abstraite numéro un*.

[8] Recommandation UIT-T X.920 (1997) | ISO/CEI 14750:1999, *Technologies de l'information – Traitement ouvert réparti – Langage de définition d'interface*.

[9] http://www.fokus.gmd.de/research/cc/platin/products/y-sce/ – An acceptor for ITU-ODL.

[10] ISO/CEI 14882:1988, *Langages de programmation – C++*.

3 Abréviations

La présente Recommandation utilise les abréviations suivantes:

CASE génie logiciel assisté par ordinateur (*computer-aided software engineering*)

CORBA architecture commune de courtage d'objets (*common object request broker architecture*)

DPE environnement de traitement réparti (*distributed processing environment*)

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

ODP traitement réparti ouvert (*open distributed processing*)

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

4 Définitions

La présente Recommandation utilise les termes ci-dessous définis dans les Recommandations [2], [3] et [8].

X.902	X.903	X.920
compatibilité de comportement	annonce	exception
objet client	interface de traitement	fichier
environnement d'un objet	objet de traitement	module
héritage	flux	héritage multiple
instanciation d'un gabarit	groupe (d'interaction)	
objet	interrogation	
qualité de service	opération	
objet serveur	signature d'interface opération	
service	signature d'interface flux	
gabarit	terminaison	

NOTE – Il y a divergence entre les définitions du terme objet dans l'ODP et dans l'OMG. Comme le langage ITU-ODL inclut l'ODP-IDL comme un sous-ensemble, le terme objet peut être utilisé par référence à la définition OMG. Si tel est le cas, cela est indiqué.

5 Fondements et règles

Dans le présent paragraphe sont exposés les principes de l'ITU-ODL, sa métastructure et sa sémantique. La syntaxe ITU-ODL y est présentée pour illustrer des cas. La signification de cette syntaxe est expliquée dès qu'elle apparaît pour la première fois, mais la présentation détaillée de la syntaxe ITU-ODL fait l'objet du paragraphe suivant.

Dans un premier temps, les définitions et conventions de base sont présentées. Les règles de nomage de base et le domaine d'application suivent. D'autres règles de nomage de base et de visibilité sont ajoutées. L'une des caractéristiques de l'ITU-ODL est qu'il permet d'utiliser en commun ou de réutiliser des spécifications existantes. Sont ensuite exposés les principes sur la base desquels le comportement est spécifié. Comme indiqué plus haut, une forme de réutilisation des spécifications se fait par composition, tandis que la seconde forme se fait par spécialisation ou par héritage. L'architecture d'héritage spécifique adoptée pour l'ITU-ODL est présentée au 5.5.

Dans le présent paragraphe, les règles sont mises en évidence par des étiquettes de la forme Rn (ex.: R1, R2, etc.). Les définitions des termes sont indiquées par des étiquettes de la forme Dn (ex.: D1, D2, etc.).

5.1 Définitions et Conventions

5.1.1 Définitions

Les définitions suivantes sont utilisées dans la suite de la présente Recommandation:

5.1.1.1 (D1) création: une création ITU-ODL est un modèle de groupe d'objets, un modèle d'objet, un modèle d'interface, une opération ou un flux. La spécification de signature d'une création ITU-ODL est déclarée dans l'ITU-ODL.

5.1.1.2 (D2) définition corrélative: ce sont les définitions de types de données, de constantes et de déclarations d'exceptions.

Des définitions complémentaires sont introduites au besoin dans les sous-paragraphes suivants sous forme de concepts additionnels.

5.1.2 Conventions graphiques

On a utilisé les conventions suivantes pour la représentation graphique des exemples donnés dans la présente Recommandation.

- Les modèles d'objets sont représentés par des rectangles (voir la Figure 1).

Figure 1/Z.130 – Modèles d'objets

- Un modèle d'interface opérationnelle est représenté par un petit rectangle plein contigu au rectangle représentant le modèle d'objet auquel il appartient (voir la Figure 2). Certains modèles d'interface peuvent être mis en évidence par un motif spécial. Une opération est représentée par une flèche pointant sur le modèle d'interface auquel elle appartient. Les autres éléments des modèles d'interface opérationnelle ne sont pas représentés.

Figure 2/Z.130 – Modèles d'objets avec interfaces

- Un modèle d'interface flux associé est représenté par un rectangle contenant des cercles ouverts ou pleins, contigu au rectangle représentant le modèle d'objet auquel il appartient (voir la Figure 3). Certains modèles d'interface peuvent être mis en évidence par un motif spécial.
- Un puits de flux est représenté par une flèche pointant sur un cercle plein dans le modèle d'interface flux auquel il appartient. Une source de flux est représentée par une flèche sortant du cercle ouvert dans le modèle d'interface flux auquel il appartient. Les autres éléments des modèles d'interface opérationnelle ne sont pas représentés.

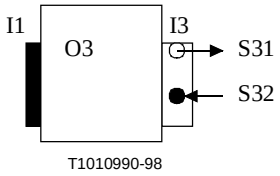

Figure 3/Z.130 – Modèle d'objet avec source et puits

- Les modèles de groupes d'objets sont représentés par des rectangles en pointillé. L'appartenance à un modèle de groupe d'objets est matérialisée par l'inclusion dans un rectangle en pointillé (voir la Figure 4).

Figure 4/Z.130 – Modèle de groupe

Nous avons présenté les termes de base et les conventions graphiques utilisés dans la présente Recommandation, nous allons maintenant étudier le cadre de nomage et domaine d'application de l'ITU-ODL.

5.2 Nomage et visibilité

Les règles de nomage et de visibilité sont définies pour permettre l'identification univoque des réalisations ITU-ODL. On pourra aux fins de comparaison se reporter aux règles équivalentes de l'ODP-IDL [8].

(R1) Un fichier ITU-ODL complet forme un domaine d'application de nomage.

(R2) Les types de définitions suivants forment des domaine d'application imbriqués:

- module;
- modèle de groupe d'objets;
- modèle d'objet;
- modèle d'interface;
- structure;
- union;
- opération;
- exception.

Ainsi, par exemple, les définitions ITU-ODL suivantes sont contenues dans un fichier. Il spécifie un module, M1, contenant un modèle de groupe d'objets, G1, contenant un modèle d'objet, O1, contenant un modèle d'interface, I1, contenant un type de données, DataType1, et une opération operation1. Le domaine de visibilité de M1 est global. Le domaine de visibilité de G1, se trouve inclus dans M1, celui de O1 dans G1, celui de I1 dans O1 et enfin ceux de DataType1 et operation1 dans I1.

```
module M1 {
        ...
        group G1 {
                ...
                CO O1 {
                        ...
                        interface I1 {
                                ...
                                typedef … DataType1;
```

```
                    ...
                    void operation1(in DataType1 variable11…);
                    ...
                }; // end of I1
            }; // end of O1
        }; // end of G1
    }; // end of M1
```

(R3) Les identificateurs des types de définition suivants sont associés à des champs de visibilité:

- modèle de groupe d'objets;
- modèle d'objet;
- modèle d'interface;
- opérations;
- types de données;
- constantes;
- valeurs énumératives;
- exceptions;
- attributs.

(R4) Un identificateur ne peut être défini qu'une seule fois dans un domaine de visibilité: les identificateurs peuvent être redéfinis dans des domaines de visibilité imbriqués.

(R5) Les identificateurs peuvent indifféremment être saisis en majuscules ou en minuscules.

(R6) Les identificateurs définis dans un domaine de visibilité peuvent être immédiatement utilisés dans ce domaine de visibilité.

(R7) Un nom qualifié (de la forme <scoped-name>::<identifier>) est résolu en repérant la définition de <identifier> dans le domaine de visibilité. L'identificateur doit être défini directement dans le domaine de visibilité. L'identificateur n'est pas recherché dans les domaines de visibilité englobants.

Par exemple, dans l'exemple d'ITU-ODL ci-dessus, le nom qualifié de G1 est M1::G1. De même, le nom qualifié de DataType1 est M1::G1::O1::I1::DataType1.

(R8) Un nom non qualifié (de la forme <identifier>) peut être utilisé dans un domaine de visibilité particulier. Il sera résolu en effectuant des recherches successives dans les domaines de visibilité englobants. Lorsqu'un nom non qualifié a été utilisé dans un domaine de visibilité, il ne peut plus être redéfini.

Par exemple, le texte ITU-ODL ci-dessous montre un DataType1 défini dans le domaine de visibilité du module M1. Il est ensuite utilisé (comme nom non qualifié) dans le domaine de visibilité du modèle d'interface I1. Puis dans le domaine de visibilité of I1, DataType1 est de nouveau redéfini. Cette redéfinition de DataType1 est "illégale". Si la deuxième définition a été insérée avant la première déclaration, la redéfinition sera "légale" et sera utilisée pour résoudre le nom non qualifié dans la déclaration operation1.

```
module M1 {
    ...
    typedef … DataType1;
    interface I1 {
        ...
        void operation1(in DataType1 variable11…);
        ...
        typedef … DataType1; // Illegal statement
```

```
            ...
        }; // end of I1
    }; // end of M1
```

(R9) Chaque définition ITU-ODL dans un fichier a un nom global dans ce fichier. La règle de création d'un nom global est la même que pour l'ODP-IDL [8].

"Avant de commencer l'exploration d'un fichier contenant une spécification ODP-IDL, le nom de la racine actuelle est initialement vide ("") et le nom du domaine de visibilité actuel est initialement vide (""). Chaque fois qu'un mot clé **`module`** est rencontré, on ajoute la chaîne "::" et l'identificateur associé au nom de la racine actuelle; dès détection de la fin du mot **`module`**, le postambule "::" et l'identificateur sont supprimés du nom de la racine actuelle. Chaque fois qu'un mot clé de type **`interface`**, **`struct`**, **`union`** ou **`exception`** est rencontré, la chaîne "::" et l'identificateur associé sont ajoutés au nom du domaine actuel; dès détection de la fin du mot clé **`interface`**, **`struct`**, **`union`** ou **`exception`**, le postambule "::" et l'identificateur sont supprimés du nom du domaine actuel. Par ailleurs, un nouveau domaine sans nom est ouvert lorsque les paramètres d'une déclaration d'opération sont traités; cela permet aux noms des paramètres de copier d'autres identificateurs; lorsque le traitement du paramètre est effectué, le domaine sans nom est fermé.

Le nom global d'une définition ODP-IDL est la concaténation de la racine actuelle, du domaine de visibilité actuel, d'une définition "::" et de <identifier>, qui est le nom local de cette définition."

En outre, à cette fin, un modèle de groupe d'objet et un modèle d'objet sont traités de manière analogue à une (un modèle d') interface, struct, union, et exception.

5.2.1 (D3) mot étiqueté: un mot étiqueté prend l'une des formes <scoped_name>"."<scoped_name>.

Cette création est utilisée pour avoir un accès qualifié aux modèles d'interface pris en charge d'un modèle d'objet ou d'un modèle de groupe d'objets dans une spécification d'interface requise.

Après avoir présenté certaines créations de base en langage ITU-ODL, nous allons maintenant examiner comment les spécifications ITU-ODL peuvent être présentées sous forme de Recommandations, en particulier afin de pouvoir réutiliser les spécifications ITU-ODL (ou ODP-IDL).

5.3 Modèle d'interface, modèle d'objet et modèle de groupe d'objets et utilisation commune de ces modèles

Une certaine liberté est offerte au développeur de spécifications informatiques en ce qui concerne les déclarations indépendantes de modèles d'interface, de modèles d'objet et de modèles de groupe d'objets. Chaque modèle d'interface en ITU-ODL peut être réutilisé dans un nombre quelconque de modèles d'objets. De même, les modèles d'objet peuvent être spécifiés comme des définitions individuelles et réutilisés dans un nombre quelconque de modèles de groupes d'objets.

Etant donné que l'ITU-ODL est un sous-ensemble de l'ODP-IDL, le mappage entre un modèle d'interface opérationnelle ITU-ODL et une spécification ODP-IDL équivalente est triviale. Un des avantages d'un mappage aussi direct est qu'il est possible d'utiliser des outils CORBA existants dans la chaîne de développement des logiciels. Un autre avantage est qu'il est possible de réutiliser des définitions d'interface ODP-IDL existantes.

De ce qui précède, on constate que la séparation des déclarations de créations présente l'avantage de pouvoir disposer d'un moyen direct d'utiliser des déclarations de création communes. Les principes d'utilisation commune sont présentés de façon plus détaillée ci-après.

5.3.1 Types de données

(R10) Les types de données peuvent être déclarés dans une domaine de visibilité ITU-ODL quelconque. L'utilisation de déclarations de types de données qui sont communes à plusieurs opérations ou flux de modèles différents est autorisée.

5.3.2 Opérations

(R11) Les signatures d'opérations sont déclarées dans les modèles d'interface. Comme les signatures d'opérations ne sont pas déclarées de manière distincte des modèles d'interface, et qu'il n'existe pas de mécanisme d'utilisation en commun approprié, l'utilisation de la même déclaration de signature d'opération par plusieurs modèles d'interface n'est pas possible.

NOTE – Cette règle est directement issue de l'ODP-IDL. Dans une nouvelle version de l'ODP-IDL cette règle sera peut-être assouplie. La question des conséquences d'une telle modification de l'ITU-ODL reste ouverte.

(R12) Deux opérations avec le même identificateur déclarées dans des modèles d'interface distincts sont considérées comme différentes.

5.3.3 Flux

(R13) Les signatures de flux sont déclarées dans les modèles d'interface. Comme les signatures de flux ne sont pas déclarées de manière distincte des modèles d'interface, et qu'il n'existe pas de mécanisme d'utilisation en commun approprié, l'utilisation de la même déclaration de signature de flux par plusieurs modèles d'interface n'est pas possible.

(R14) Deux opérations avec le même identificateur déclarées dans des modèles d'interface distincts sont considérées comme différentes.

5.3.4 Modèles d'interface

On suppose que l'objectif de l'utilisation d'une même déclaration de modèle d'interface est de permettre à plusieurs modèles d'objets d'utiliser la même spécification ITU-ODL d'un modèle d'interface. La syntaxe ITU-ODL est définie pour permettre la distinction entre déclarations de modèles d'interface et déclarations de modèles d'objet. Les spécifications de modèles d'interface peuvent être incluses dans une déclaration de modèle d'objet en indiquant qu'il s'agit d'une interface prise en charge ou d'une interface requise.

5.3.4.1 (D4) interfaces prises en charge déclarées/interfaces offertes: les modèles d'interface listés comme étant pris en charge sur un modèle d'objet sont les seuls modèles d'interface pour lesquels des instances peuvent exister sur l'instance d'objet[2]. Les interfaces offertes d'une instance d'objet sont les instances d'interfaces existant sur cet objet à un instant donné.

5.3.4.2 (D5) interfaces requises déclarées: les interfaces requises déclarées sur un modèle d'objet contient la liste des modèles d'interface dont une instance du modèle d'objet à besoin pour déclencher les opérations la concernant[3].

Les règles suivantes concernent la relation des déclarations d'opération, de flux, de modèle d'interface et de modèle d'objet:

---

[2] Les interfaces prises déclarées en charge d'une classe de base sont considérées comme des interfaces prises en charge de la sous-classe et peuvent être également instanciées dans le modèle de sous-classe.

[3] Les interfaces requises déclarées d'une classe de base sont considérées comme des interfaces requises de la sous-classe.

(R15) Les signatures d'opération et de flux sont uniquement déclarées dans les modèles d'interface. Les modèles d'interface peuvent être déclarés à l'intérieur et à l'extérieur des modèles d'objet ou des modèles de groupe d'objets.

5.3.5 Modèles d'objet

On suppose que l'objectif de l'utilisation d'une même déclaration de modèle d'objet est de permettre à plusieurs modèles de groupe d'utiliser la même spécification ITU-ODL d'un modèle d'objet. La syntaxe ITU-ODL est définie pour permettre de séparer les déclarations de modèles d'objet des déclarations de modèles de groupe. Les spécifications de modèle d'objet peuvent être considérés dans un modèle de groupe comme des membres. Les spécifications d'un modèle d'interface peuvent être considérées dans un modèle de groupe d'objets comme des contrats requis ou pris en charge, qui sont des modèles d'interface dont les instances peuvent être utilisées par des entités externes au groupe d'objets (pris en charge) ou qui sont nécessaires aux instances des membres du groupe depuis l'environnement (requis).

5.3.5.1 (D6) membres déclarés: les membres déclarés d'un modèle de groupe d'objets sont les modèles d'objet, ou modèles de groupe d'objets, appartenant au modèle de groupe. Les membres déclarés sont énumérés comme "membres" dans un modèle de groupe.

5.3.5.2 (D7) contrat déclaré requis/pris en charge: un contrat déclaré requis/pris en charge d'un modèle de groupe d'objets est l'une des interfaces requises/prises en charge d'un modèle d'objet ou modèle de groupe membre de ce modèle de groupe d'objets. Les contrats déclarés requis/pris en charge d'un modèle de groupe d'objets représente seulement les interfaces capables d'être utilisées (prises en charge) par des entités externes à ce groupe d'objets ou auxquelles les instances des membres peuvent accéder dans l'environnement considéré (requises). Les contrats déclarés sont énumérés comme étant requis ou pris en charge dans un modèle de groupe (analogue aux modèles d'objet). Si aucune interface requise ou prise en charge n'est spécifiée dans un modèle de groupe, les interfaces des membres du groupe ne sont pas soumises à des restrictions de visibilité[4].

La règle est la suivante:

(R16) Les modèles d'objet peuvent être déclarés en dehors des modèles de groupes d'objets.

5.3.6 Règles de visibilité

Les règles de visibilité suivantes s'applique aux spécifications communes.

(R17) S'agissant de la visibilité d'un modèle d'objet, un identificateur d'interface peut être défini en déclarant le modèle d'interface associé intégré, ou utilisé en le déclarant comme étant pris en charge ou requis. Dans chaque cas, le nom global du modèle d'interface sera différent, de la forme ...<object-identifier>::<interface-identifier> dans le premier cas ou ...<interface-identifier> dans le second cas.

(R18) S'agissant de la visibilité du modèle de groupe d'objets, un identificateur d'objet peut être défini en déclarant le modèle d'objet associé intégré, ou utilisé en le déclarant membre. Dans chaque cas, le nom global du modèle d'objet sera différent.

(R19) S'agissant de la visibilité d'un modèle d'objet, un identificateur d'interface peut être défini en déclarant le modèle d'interface associé intégré, ou utilisé en le déclarant comme étant un contrat pris en charge ou requis. Dans chaque cas, le nom global du modèle d'interface sera différent.

---

[4] Cela est dû au fait qu'il est possible de définir les groupes par plusieurs différents critères et que pour certains d'entre eux la spécification de contrats n'est pas utile.

5.4 Comportement

Le comportement d'une entité (interface, objet, modèle de groupe d'objets), au sens le plus général, se compose de toutes les interactions possibles qu'une entité peut entreprendre avec son environnement. L'ITU-ODL n'est pas à un stade suffisamment avancé pour permettre une spécification complète et détaillée. Au lieu de cela, une spécification de comportement informelle est décrite pour des entités particulières comme suit:

Modèles d'interface

La présente spécification décrit le service assuré par une instance du modèle en cours de définition. Elle décrit également l'utilisation envisagée de l'interface. La présente spécification documente les contraintes d'ordonnancement (ou de séquencement) sur les opérations définies dans le modèle d'interface. Les invocations d'opérations sur une instance d'un tel modèle doivent respecter ces contraintes. Dans la version actuelle de l'ITU-ODL, cette spécification est un littéral de chaîne.

Modèles d'objet

La présente spécification décrit les responsabilités d'un objet dans la fourniture de services via chacune de ses interfaces telles que prises en charge dans l'ITU-ODL.

Modèles de groupes d'objets

La présente spécification décrit le critère valable pour toutes les entités contenues dans le modèle de groupe. En fonction des critères, la spécification du prédicat du groupe peut contenir des informations supplémentaires. Par exemple, si le critère est que les membres du groupe exécutent ensemble un service particulier, la description du prédicat peut en plus définir une fonctionnalité assurée pour chacun des contrats pris en charge.

NOTE – Il convient de noter que la chaîne fournie comme étant une spécification de comportement peut être une référence à une spécification formelle de comportement faite dans un autre langage. Par exemple, c'est le cas d'une liaison depuis une définition de modèle d'interface opérationnelle vers un spécification de type processus SDL [7] qui contient une spécification formelle de comportement pour le modèle d'interface considéré. La spécification et l'interprétation de ses liaisons fait appel aux outils CASE en utilisant l'ITU-ODL comme technique de description pour les spécifications du point de vue traitement.

5.5 Héritage

5.5.1 Introduction et justification

Les modèles d'interface, les modèles d'objet et les modèles de groupes d'objets assurent la modularité des spécifications. Comme ces trois modèles représentent aussi des types, il est utile de définir un mécanisme de réutilisation dans lequel:

- une création peut directement reposer sur des entités définies dans une autre création (du même type de modèle). Par exemple, un type données défini dans un modèle d'interface est utilisé dans un autre modèle d'interface;
- une réalisation est dérivée d'une autre réalisation (du même type de modèle). Par exemple, une spécification de modèle d'objet est dérivée d'une autre spécification de modèle d'objet.

Dans les publications classiques sur les objets, ce mécanisme de réutilisation est connu sous le nom d'héritage. En ITU-ODL, les règles de définition pour l'héritage permet de déclarer les nouveaux modèles d'interface, modèles d'objet et modèles de groupes d'objets comme étant des extensions ou des restrictions de modèles précédemment définis.

Dans la suite du présent paragraphe, on décrit les principes sous-jacents à la réutilisation, en ITU-ODL, des spécification par héritage. On présente tout d'abord les éléments essentiels de

l'héritage de modèle d'interface, de modèle d'objet et de modèle de groupe d'objets. On explique ensuite les règles de nomage et de visibilité associées à l'héritage.

5.5.2 Définitions

Les définitions suivantes s'appliquent à l'héritage:

5.5.2.1 (D8) réalisation de base/dérivée/spécialisée: une réalisation (modèle de groupe, modèle d'objet ou modèle d'interface) est dite dérivée ou spécialisante d'une autre réalisation, appelée réalisation de base de la réalisation dérivée, si elle a hérité de la réalisation de base.

5.5.2.2 (D9) spécialisation ultime: les modèles d'objet spécialisés ultimes (modèle de groupe ou modèles d'interface) dans un ensemble de modèles d'objet (modèle de groupe ou modèles d'interface) sont les éléments de l'ensemble à partir desquels aucune autre réalisation dans cet ensemble n'est dérivée (c'est-à-dire qui ne constituent pas la base d'une autre réalisation de l'ensemble).

5.5.2.3 (D10) base directe/indirecte: une réalisation est dite base directe d'une réalisation si elle est mentionnée dans la spécification d'héritage de la déclaration de réalisation et base indirecte si elle n'est pas une base directe mais la base d'une base directe ou indirecte (sous-héritage).

5.5.2.4 (D11) graphe d'héritage/d'héritage partiel: le graphe d'héritage de modèles d'objet (modèle de groupe ou modèles d'interface) le graphe acyclique dirigé représentant les relations d'héritage entre modèles d'objet (modèle de groupe ou modèles d'interface). Un graphe d'héritage partiel d'un type donné est un graphe d'héritage limité à en ensemble de réalisations.

NOTE – Les feuilles du graphe d'héritage pour un type donné de réalisation (c'est-à-dire: modèle de groupes, modèles d'objet ou modèles d'interface) sont les réalisations les plus spécialisées de ce type considéré.

5.5.2.5 (D12) ensemble restriction/restreint: la restriction d'un ensemble de réalisations est opérée en supprimant de cet ensemble les réalisations qui sont la base de toute autre réalisation à partir de l'ensemble. Le résultat de la restriction d'un ensemble est appelé ensemble restreint.

NOTE 1 – Un ensemble restreint peut également être vu comme un ensemble de feuilles du graphe d'héritage partiel.

NOTE 2 – L'ensemble restreint de toutes les réalisations du type modèle d'objet (modèle de groupe ou modèle d'interface) est l'ensemble des modèles d'objet les plus spécialisés (modèle de groupes ou modèles d'interface).

Supposons que les modèles d'interface dotés des règles d'héritage suivantes sont définis:

- modèle d'interface I1;
- modèle d'interface I2, qui hérite du modèle d'interface I1;
- modèle d'interface I3, qui hérite du modèle d'interface I1;
- modèle d'interface I4.

La restriction de l'ensemble des modèles d'interface (I1, I2, I3, I4) est constituée par l'ensemble (I2, I3, I4).

5.5.3 Héritage de déclarations de réalisation

5.5.3.1 Héritage de modèle d'interface

On suppose que l'héritage du modèle d'interface indique "réutilisation par spécialisation" d'un (spécification ITU-ODL de) modèle d'interface. Le modèle d'interface d'où on a hérité est appelé modèle d'interface de base. Le modèle d'interface qui hérite est appelé modèle d'interface dérivé. Cette spécialisation peut prendre deux formes:

- addition de nouvelles opérations, ou flux, à la liste des modèles d'interface de base;
- redéfinition des signatures d'opérations ou flux dans les modèles d'interface de base.

NOTE 1 – La version actuelle de l'ODP-IDL n'autorise pas ce type d'héritage pour les modèles d'interface opérationnels (comme le montre la règle 24); il en est de même de l'ITU-ODL.

NOTE 2 – Tous les modèles d'interface dérivés d'un modèle d'interface de base sont considérés comme étant "compatible avec" le modèle d'interface de base.

La syntaxe définie pour l'ITU-ODL prend totalement en charge les règles d'héritage (de modèle) d'interface ODP-IDL, et adopte des règles homogènes pour les modèles d'interface opérationnelles ou flux. Les règles d'héritage ITU-ODL des modèles d'interface sont les suivantes:

Règles générales

(R20) Un modèle d'interface peut être dérivé d'un ou plusieurs autres modèles d'interface, dont chacun est appelé modèle d'interface de base du modèle d'interface dérivé. Dans le cas d'une dérivation depuis plusieurs interfaces de base (héritage multiple), l'ordre de dérivation n'a pas d'importance.

(R21) Un modèle d'interface peut ne pas être spécifié comme modèle d'interface de base direct d'un modèle d'interface dérivé plus d'une fois. Il peut être un modèle d'interface de base indirecte plusieurs fois (c'est-à-dire pouvant éventuellement être représenté par un graphe d'héritage "en diamant").

(R22) Un modèle d'interface dérivé peut déclarer des sous-réalisations nouvelles (types données, opérations ou flux). Sauf si elles sont redéfinies, les sous-réalisations du modèle d'interface de base peuvent être considérées comme des sous-réalisations du modèle d'interface dérivé. L'héritage du modèle d'interface déclenche l'importation dans la visibilité de nomage courante de tous les identificateurs dans l'enceinte du graphique d'héritage.

(R23) L'héritage depuis deux modèles d'interface ayant les même identificateurs d'opération ou les même identificateurs de flux n'est pas autorisé.

(R24) Il est interdit de redéfinir une opération dans le modèle d'interface dérivé.

(R25) Il est interdit de redéfinir un flux dans le modèle d'interface dérivé.

(R26) Un modèle d'interface dérivé peut redéfinir des identificateurs de type de données hérités. Un identificateur de type de données depuis un domaine de visibilité englobant peut être redéfini dans le domaine de visibilité courant.

Comportement

(R27) Le bloc behaviourText de modèles d'interface de base n'est pas disponible dans un modèle d'interface dérivé.

(R28) L'attribut d'usage des modèles d'interface de base n'est pas disponible dans un modèle d'interface dérivé.

A titre d'exemple, supposons que le modèle d'objet O4 déclare prendre en charge:

– le modèle d'interface I4, spécialisation du modèle d'interface I1 réalisé par adjonction de l'operation41;

– le modèle d'interface S2, spécialisation du modèle d'interface S1, avec adjonction du flux source videoFlow21.

Il est possible de déclarer le modèle d'interface I4 comme héritant de l'operation11 depuis I1, et d'ajouter l'opération operation41. De même, S2 peut hériter de S1 le flux source voiceDownStream et le flux puits voiceUpStream, et ajouter le flux source videoFlow21.

I1 et S1 sont définis comme suit:

```
interface I1{
        ...
        // data types
        typedef … DataType11;
        typedef … DataType12;

        void operation11 (in DataType11 …, out DataType12 …);
}; // end of I1

interface S1{
        ...
        // flow types
        typedef … VoiceFlowType;

        source VoiceFlowType voiceDownStream;
        sink VoiceFlowType voiceUpStream;
}; // end of S1
```

Les modèles d'interface héritiers peuvent alors être définis comme suit:

```
interface I4: I1{
        ...
        typedef … DataType41;
        void operation41 (in DataType41 …);
}; // end of I4

interface S2: S1{
        ...
        typedef … FlowTypeS21;
        source FlowType21 videoFlow21;
}; // end of S2
```

Le modèle d'objet O4, utilisant les modèles d'interface spécialisés hérités, peut être défini comme suit:

```
CO O4{
        behaviour
        ...
        supports
                I4, S2;
        ...
}; // end of O4
```

5.5.3.2 Héritage de modèles d'objet

On suppose que l'héritage de modèles d'objet est destiné à permettre la "réutilisation par spécialisation" d'un (d'une spécification ITU-ODL d'un) modèle d'objet. Le modèle d'objet qui est donné en héritage est appelé modèle d'objet de base. Le modèle d'objet qui reçoit l'héritage est appelé modèle d'objet dérivé. Cette spécialisation peut prendre deux formes de base:

- addition de modèles d'interface: de nouveaux modèles d'interface peuvent être ajoutés à la liste des interfaces prises en charge/requises de l'objet de base;
- affinement de modèles d'interface: interfaces prises en charge sur les objets de base peuvent être spécialisées dans l'objet dérivé.

Les règles d'héritage pour les modèles d'objet sont les suivantes:

Règles générales et interfaces prises en charge

(R29) Un modèle d'objet peut être dérivé d'un ou plusieurs modèles d'objet, appelé chacun modèle d'objet de base du modèle d'objet dérivé. Dans le cas de la dérivation à partir de plusieurs modèles d'objet de base (héritage multiple), l'ordre de dérivation n'est pas significatif.

NOTE 1 – Le graphique d'héritage pour les modèles d'objet est totalement distinct de celui des modèles d'interface.

(R30) Un modèle d'objet dérivé peut déclarer de nouvelles sous-réalisations (types de données, modèles d'interface). Sauf lorsqu'elles sont redéfinies, les sous-réalisations du modèle d'objet de base peuvent être considérées comme s'il s'agissait des sous-réalisations du modèle d'objet dérivé. L'héritage d'objet provoque l'importation vers le champ de visibilité de nomage courant de tous les identificateurs dans l'enceinte du graphique d'héritage.

(R31) Un modèle d'objet peut ne pas être spécifié comme étant un modèle d'objet de base direct d'un modèle d'objet dérivé plus d'une fois. Il peut être un modèle d'objet de base indirecte plusieurs fois (c'est-à-dire pouvant éventuellement être représenté par un graphe d'héritage "en diamant").

(R32) Les modèles d'interface qui peuvent être proposés sur un objet dérivé sont constitués par l'union des modèles d'interface pris en charge sur tous les objets de base, plus les modèles d'interface additionnels déclarés pris en charge sur le modèle d'objet dérivé.

NOTE 2 – Pour ajouter un nouveau modèle d'interface à la liste des modèles d'interface hérités depuis des modèles d'objet de base, il suffit de déclarer un modèle d'interface additionnel pris en charge dans le modèle d'objet (addition d'interface).

NOTE 3 – Pour affiner un modèle d'interface pris en charge par les modèles d'objet de base d'un objet, il suffit de déclarer un modèle d'interface pris en charge dans le modèle d'objet, dans lequel le modèle d'interface est dérivé depuis le premier (affinement de modèle d'interface). L'objet peut alors offrir des instances d'un de ces modèles d'interface.

(R33) Un modèle d'objet dérivé peut redéfinir des identificateurs de type de données reçus en héritage. Un identificateur de type de données depuis un champ de visibilité englobant peut être redéfini dans le champ de visibilité courant.

Comportement

(R34) Le bloc behaviourText des modèles d'objet de base n'est pas disponible dans un modèle d'objet dérivé.

(R35) Les interfaces requises du modèle d'objet de base est l'union des interfaces requises des modèles d'objet de base, plus les éventuelles interfaces additionnelles requises propres au modèle d'objet dérivé.

Initial

(R36) Les interfaces initiales des modèles d'objet de base ne sont pas disponibles dans un modèle d'objet dérivé; les interfaces initiales ne sont pas reçues en héritage.

Le modèle d'interface initiale d'un modèle d'objet dérivé doit être dérivé (ou peut être identique, dans le cas d'un héritage d'un seul modèle d'objet) de modèles d'interface initiale de modèles d'objet de base directs. Dans les autres cas, un système de gestion (pour l'instanciation) aura des difficultés à voir les modèles d'objet dérivés comme équivalents à ses modèles de base.

Par exemple, considérons un modèle d'objet O6 prenant en charge:

- un modèle d'interface I1 qui est pris en charge par les modèles d'objet O1 and O2;
- un modèle d'interface I5 prenant en charge l'opération Op21 qui est également définie sur le modèle d'interface I2 pris en charge par O1, et en plus par Op51;

- un modèle d'interface I3 pris en charge par O2;
- interface I6, prenant en charge l'opération Op61.

Il est possible d'établir les relations d'héritage suivantes entre les modèles d'objet O1, O2 et O6 (voir la Figure 5):

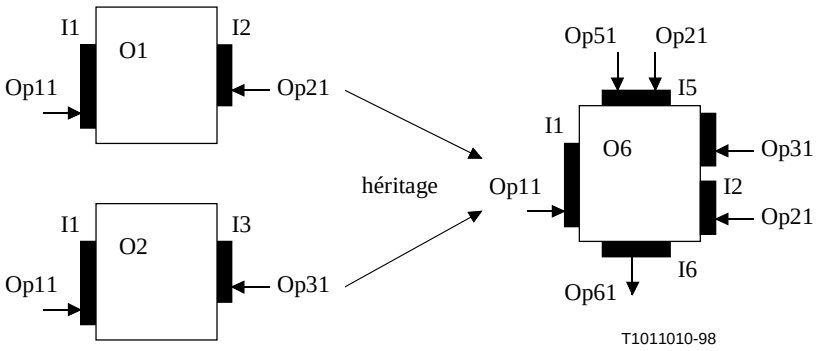

Figure 5/Z.130 – Héritage entre modèles d'objet

Dans ce cas, la définition de O6 peut être établie à partir de O1 et de O2 par l'utilisation des règles d'héritage et l'ajout de nouveaux membres.

Le modèle d'interface I5 est déclaré comme étant dérivé de l'interface I2, et le modèle d'interface I6 est déclaré isolément:

```
interface I5: I2{
        ...
        typedef … DataType51;
        void Op51(in DataType51 …);
}; // end of I5

interface I6{
        ...
        typedef … DataType61;
        void Op61(in DataType61 …);
}; // end of I6
```

Le modèle d'objet O6 est déclaré comme étant dérivé des modèles d'objet O1 et O2, et les modèles d'interface I5 et I6 sont déclarés dans la liste des modèles d'interface pris en charge de O6:

```
CO O6: O1, O2{
        ...
        supports
                I5, I6;
        ...
}; // end of O6
```

Puisqu'il n'existe pas de relation d'héritage depuis le modèle d'interface I6 avec un des modèles d'interface déclarés dans les modèles d'objet de base O1 ou O2, le modèle d'interface I6 est simplement considéré comme une interface prise en charge de O6.

Les modèles d'interface I3 et I2 qui apparaissent sur O2, et par l'intermédiaire des règles d'héritage, sont également considérés comme des interfaces prises en charge de O6.

Le modèle d'interface I1 qui apparaît sur O1 et sur O2, et par l'intermédiaire des règles d'héritage, est également considéré comme une interface prise en charge de O6.

Le modèle d'interface I5 hérite de I2 et offre les opérations Op21 et Op51. Il convient de noter que l'objet peut instancier le type de base (I2) ainsi que le sous-type (I5).

5.5.3.3 Héritage des modèles de groupe d'objets

On suppose que l'héritage de modèles de groupe d'objets est destiné à permettre la "réutilisation par spécialisation" d'un (d'une spécification ITU-ODL d'un) modèle de groupe. Le modèle de groupe qui est donné en héritage est appelé modèle de groupe de base. Le modèle de groupe qui reçoit l'héritage est appelé modèle de groupe dérivé. Cette spécialisation peut prendre deux formes de base:

- addition de modèles d'objet/de groupe: de nouveaux modèles d'objet peuvent être ajoutés à la liste des membres du modèle de groupe de base;
- affinement de modèles d'objet/de groupe: les membres sur les modèles de groupe de base peuvent être spécialisés dans les modèles de groupe dérivés.

Les règles d'héritage pour les modèles d'objet sont les suivantes:

Règles générales et membres

(R37) Un modèle de groupe peut être dérivé d'un ou de plusieurs autres modèles de groupes, appelés chacun modèle de base du modèle de groupe dérivé. Dans le cas de la dérivation depuis plusieurs modèles de groupes de base (héritage multiple), l'ordre de dérivation n'est pas significatif.

NOTE 1 – Le graphique d'héritage pour les modèles d'objet est totalement distinct de ceux correspondant aux objets et aux modèles d'interface.

(R38) Un modèle de groupe dérivé peut déclarer de nouveaux types de sous-réalisations (types de données, modèles d'interface, modèles d'objet, modèles de groupe). Sauf lorsqu'elles sont redéfinies, les sous-réalisations du modèle de groupe de base peuvent être considérées comme des sous-réalisations du modèle de groupe dérivé. L'héritage de modèle de groupe provoque l'importation de tous les identificateurs dans l'enceinte du graphique d'héritage vers le champ de visibilité de nomage courant.

(R39) Un modèle de groupe peut ne pas être spécifié comme modèle de groupe de base direct d'un modèle de groupe dérivé plus d'une fois. Il peut être un modèle de groupe de base indirecte plusieurs fois (c'est-à-dire pouvant éventuellement être représenté par un graphe d'héritage "en diamant").

(R40) Les modèles de groupe/d'objet qui peuvent comprendre un modèle de groupe dérivé sont constitués par l'union des modèles de groupe/d'objet déclarés sur tous les modèles de groupe de base, plus tout modèle de groupe/d'objet déclaré pris en charge sur le modèle de groupe dérivé.

NOTE 2 – Pour ajouter un nouvel objet/groupe à la liste des modèle membres hérités des modèles de groupe de base, il suffit de déclarer un modèle de groupe/d'objet additionnel membre dans le modèle de groupe (addition d'objet/groupe).

NOTE 3 – Pour affiner un modèle de groupe/d'objet pris en charge des modèles de groupe de base du groupe, il suffit de déclarer un modèle de groupe/d'objet membre dans le modèle de groupe, où ce modèle de groupe/d'objet membre est dérivé du premier (affinage de modèles de groupe/d'objet). Le groupe peut alors inclure des instances de ces modèles de groupe/d'objet.

(R41) Un modèle de groupe dérivé peut redéfinir des identificateurs de types de données reçus en héritage. Un identificateur provenant d'un champ de visibilité enfermant peut être redéfini dans le champ de visibilité courant.

Comportement

(R42) Les prédicats de groupes de base ne sont pas disponibles dans un groupe dérivé. Le prédicat de groupe dérivé définit les critères que tous les membres du groupe doivent respecter.

Contrats

(R43) Les contrats requis/pris en charge comprenant un modèle de groupe dérivé sont constitués par l'union des contrats requis/pris en charge comprenant les modèles de groupe, plus tout autre contrat requis/pris en charge déclarés sur le modèle de groupe dérivé.

Du moment que leurs définitions sont compatibles[5], les contrats peuvent être hérités même si les prédicats d'un sous-modèle diffèrent de ceux du modèle de base. L'utilisateur doit éviter d'utiliser l'héritage lorsque les prédicats de modèles de groupe de base et les sous-modèles de groupe ne sont pas compatibles.

5.5.3.4 Nomage et visibilité relativement à l'héritage

Les règles suivantes de visibilité sont ajoutées pour prendre en charge les capacités d'héritage.

(R44) L'héritage introduit des identificateurs dans le modèle d'interface, modèle d'objet ou modèle de groupe d'objets dérivé.

(R45) L'héritage de modèles d'interface, modèles d'objet ou modèles de groupes d'objets introduit plusieurs identificateurs ITU-ODL globaux pour les identificateurs reçus en héritage.

(R46) Un nom qualifié (de la forme <scoped-name>::<identifier>) est résolu par localisation de la définition de <identifier> dans le champ de visibilité. L'identificateur doit être directement défini dans le champ de visibilité ou (si le champ de visibilité est un modèle de groupe d'objets, modèle d'objet ou modèle d'interface) reçu en héritage dans le domaine de visibilité. On ne recherche pas l'identificateur dans les champs de visibilité englobants.

6 Spécification de l'ITU-ODL

Dans le présent paragraphe, la syntaxe ITU-ODL est définie. Ce paragraphe comporte quatre parties traitant de principales réalisations ITU-ODL:

- déclaration de types et constantes;
- modèles d'interface;
- modèles d'objet;
- modèles de groupes d'objets.

6.1 Déclaration de type et de constante

6.1.1 Structure

Les types de données et les constantes peuvent être déclarées dans pratiquement tout domaine de visibilité d'une spécification ITU-ODL. Ces types ou constantes peuvent être utilisés pour la déclaration d'opération, d'exceptions, de flux, et autres réalisations de modèles. Comme pour toute déclaration de modèle, il faut qu'un type ou une constante soit déclarée avant son utilisation (c'est-à-dire précédemment dans le fichier).

La syntaxe prise en charge par l'ITU-ODL pour la déclaration de type ou de constante est identique à celle de l'ODP-IDL. Le lecteur trouvera dans l'Appendice II une description de cette syntaxe.

---

[5] La présente Recommandation ne définit pas la compatibilité des prédicats.

6.1.2 Exemple de déclaration de type et de constante

Dans l'exemple ci-dessous on montre la déclaration de trois types de données : Bps, qui est un synonyme de flottante; Guarantee, qui est une énumération; et AudioQoS, qui est une structure.

```
typedef float Bps;

enum Guarantee {
        Deterministic,
        Statistical,
        BestEffort
};
struct AudioQoS {
        union Throughput switch (Guarantee){
                case Statistical:       Bps   mean;
                case Deterministic:     Bps   peak;
                case BestEffort:        range struct Interval {
                                                Bps min;
                                                Bps maxd;
                                        };
        };
        union Jitter switch (Guarantee) {
                case Statistical:       Bps   mean;
                case Deterministic:     Bps   peak;
        };
};
```

6.2 Modèle d'interface

6.2.1 Structure

Un modèle d'interface informatique se compose:

- d'une spécification (textuelle) de comportement;

  et si nécessaire:

- d'une signature d'interface opérationnelle; ou bien
- d'une signature d'interface flux.

Cette structure est reflétée dans la règle ITU-ODL pour <interface_body>. Dans les sous-paragraphes qui suivent on examine les éléments de cette structure de façon plus détaillée.

La syntaxe suivante est définie pour la déclaration de modèle d'interface:

```
<interface_template>        ::=  <interface_header> "{" <interface_body> "}"
<interface_header>          ::=  "interface" <identifier>
                                 [ <interf_inheritance_spec> ]
<interf_inheritance_spec>   ::=":" <scoped_name> { "," <scoped_name> }*
<interface_body>            ::=  [<interf_behaviour_spec>]
                                 { <op_sig_defns> | <stream_sig_defns> }
<interf_behaviour_spec>     ::=  "behaviour" {
                                 {<interf_behaviour_text> [<interf_usage_spec>]}
                                 | <interf_usage_spec> }
<interf_behaviour_text>     ::=  "behaviourText" <string_literal> ";"
<interf_usage_spec>         ::=  "usage" <string_literal> ";"
```

6.2.2 Héritage de modèle d'interface

Les règles applicables à l'héritage de modèle d'interface sont celles spécifiées pour l'ODP-IDL, avec des extensions permettant de traiter des flux.

La syntaxe suivante est définie pour l'héritage de modèle d'interface:

```
<interface_header>          ::=  "interface" <identifier> [ <interf_inheritance_spec> ]
<interf_inheritance_spec>   ::=  ":" <scoped_name> { "," <scoped_name> }*
```

Il convient de noter que la spécification ODP-IDL, sous sa forme actuelle, interdit la redéfinition d'un identificateur d'opération dans une spécification de modèle d'interface dérivé et interdit aussi l'héritage de deux opérations du même identificateur. Initialement, l'ITU-ODL sera restreint conformément à ces limitations dans ces définitions de modèles d'interface opérationnelle et flux.

Cohérent avec l'ODP-IDL, l'héritage de modèle d'interface est équivalent à la simple inclusion de tous les attributs (dont les attributs d'interface), opérations et flux depuis le modèle d'interface de base dans le modèle d'interface dérivé. Cette inclusion implique tous les attributs, opérations et flux du modèle d'interface de base, dont ceux obtenus par héritage d'autres spécifications de modèles d'interface. Ainsi, un modèle dérivé doit toujours pouvoir fournir les services du modèle d'interface de base.

Un modèle d'interface flux ne peut pas hériter d'un modèle d'interface opérationnel et réciproquement.

6.2.3 Spécification de comportement de modèle d'interface

La signature d'interface décrit seulement la structure syntaxique d'un modèle d'interface. La compatibilité de signature est moins précise que la compatibilité de comportement. Il est réellement possible que deux interfaces présentent deux signatures compatibles mais diffèrent totalement de comportement. Dans le présent paragraphe on décrit comment au moins un comportement textuel est spécifié dans l'ITU-ODL.

La syntaxe suivante est définie pour la spécification du comportement du modèle d'interface:

```
<interf_behaviour_spec> ::=  "behaviour" {
                                  {<interf_behaviour_text> [<interf_usage_spec>]}
                                  | <interf_usage_spec> }
<interf_behaviour_text>  ::=  "behaviourText" <string_literal> ";"
<interf_usage_spec>      ::=  "usage" <string_literal> ";"
```

6.2.4 Signature d'interface opérationnelle

Une signature d'interface opérationnelle se compose d'un ensemble de signatures d'interrogation et d'annonce, à raison d'une par type d'opération dans le modèle d'interface. Une signature d'interface opérationnelle spécifie les informations suivantes (analogue à l'ODP-IDL):

- un attribut d'opération facultatif qui spécifie la sémantique d'invocation que le système de communication doit offrir lorsque l'opération est invoquée (interrogation ou annonce);
- le type de résultat renvoyé par l'opération (vide dans les autres cas);
- l'identificateur d'opération;
- une liste d'opération (absence ou présence de paramètre à l'opération);
- une expression "fait état" facultative qui indique les exceptions dont on peut faire état suite à l'invocation de cette opération.

La syntaxe suivante est définie pour la signature de modèles d'interface opérationnels. Elle est analogue à la syntaxe ODP-IDL pour la déclaration d'interface (modèle):

```
<op_sig_defns>          ::=   { <op_sig_defn> ";" }*
<op_sig_defn>           ::=   { <announcement> | <interrogation>
<announcement>          ::=   "one-way" "void" <identifier> <parameter_dcls>
<interrogation>         ::=   <attr_dcl>
                        |     <oper_dcl>
<attr_dcl>              ::=   ["readonly"] "attribute" <param_type_spec>
                              <declarators>
<oper_dcl>              ::=   <op_type_spec> <identifier>
                              <parameter_dcls>
                              [<raises_expr>] [<context_expr>]
<op_type_spec>          ::=   <param_type_spec>
                        |     "void"
<parameter_dcls>        ::=   "(" <param_dcl> { "," <param_dcl> }* ")"
                        |     "(" ")"
<param_dcl>             ::=   <param_attribute> <param_type_spec>
                              <declarator>
<param_attribute>       ::=   "in" | "out" | "inout"
<raises_expr>           ::=   "raises"
                              "(" <scoped_name> { "," <scoped_name> }* ")"
<context_expr>          ::=   "context"
                              "(" <string_literal> { "," <string_literal> }* ")"
<param_type_spec>       ::=   <base_type_spec>
                        |     <string_type>
                        |     <scoped_name>
```

6.2.5 Attributs d'interface opérationnelle

Les attributs d'interface opérationnelle équivalent du point de vue logique à définir une paire de fonctions auxiliaires; une pour fixer la valeur de l'attribut et une autre pour obtenir cette valeur.

6.2.6 Signature de flux

Un modèle d'interface flux se compose d'un ensemble de types de flux. Chaque type de flux contient l'identificateur du flux, le type d'information du flux et une indication précisant s'il s'agit d'un producteur ou d'un consommateur (mais pas les deux à la fois) relativement à l'objet qui assure le service défini par le modèle.

La syntaxe définie ici présuppose une directivité relativement aux définitions du modèle d'interface flux. Si deux objets sont impliqués dans un lien de flux, l'un est désigné producteur et l'autre consommateur ou client. Le modèle d'interface décrivant les interactions entre eux est exprimé du point de vue du client (définissant le serveur). Dans de nombreux cas, en particulier lorsque les flux circulent dans les deux sens, le choix de client et de serveur peut apparaître assez arbitraire. Cependant ce modèle est cohérent avec beaucoup de modèles de service usuels. Il convient de noter que le modèle de serveur inclut une déclaration indiquant qu'il "prend en charge" le modèle d'interface flux, tandis que le modèle de client inclut une déclaration selon laquelle il "requiert" le modèle d'interface flux.

Par exemple, afin de jouer à un jeu vidéo, un client (le Joueur) localise une interface appropriée (Jeu vidéo) avec le serveur (le Jeu) (voir la Figure 6). Le service est naturellement défini en termes de sources d'information (la Vidéo et l'Audio) et des puits (les commandes étiquetées joystick1 et joystick2). Toutefois, toutes ces définitions présupposent une directivité, ou point de vue, plus précisément ceux du client. La vue du service tenue par le jeu lui-même, impliquant une source de fonctions de contrôle et un puits d'informations vidéo et audio, peut facilement être déduite des autres définitions par simple mappage. Par conséquent, une de ces définitions de service est

redondante. Des substituts pour l'objet joueur (en qualité de client) ou l'objet jeu (en qualité de serveur) peuvent être produits à partir d'une seule spécification de modèle d'interface.

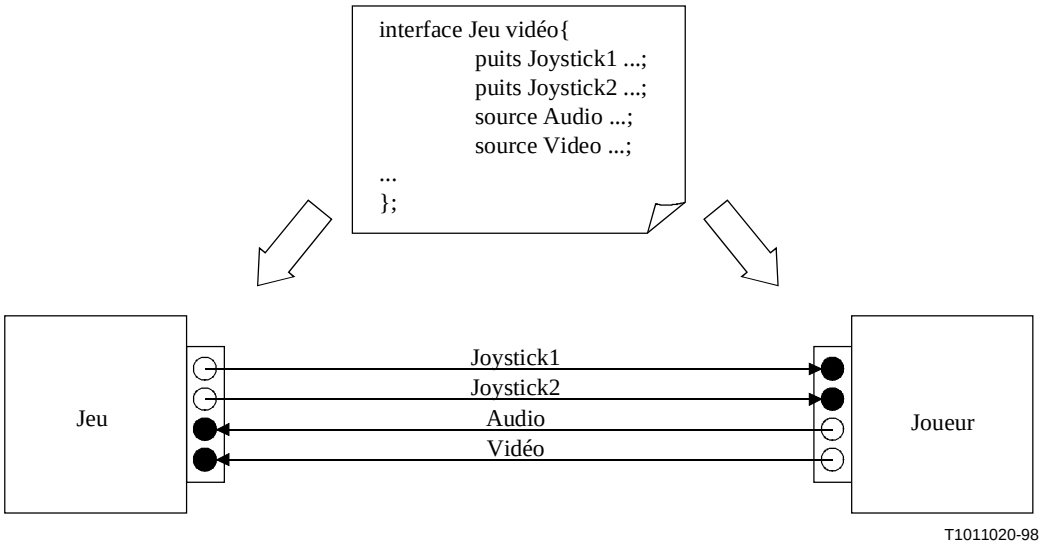

Figure 6/Z.130 – Exemple de modèle d'interface flux

En ITU-ODL, les modèles d'interface flux sont définis comme étant la vue du client sur le serveur. Chaque flux est spécifié comme source si l'information circule du serveur jusqu'au client, et comme puits s'il circule en sens inverse. Dans la définition du modèle d'objet du serveur, le modèle d'interface flux est listé comme étant une interface prise en charge, tandis que sur le client, le modèle d'interface est listé comme étant une interface requise.

```
<stream_sig_defns>          ::=   { <stream_flow_defn> ";" }*
<stream_flow_defn>          ::=   <flow_direction> <flow_type>
                                  <identifier>
<flow_direction>            ::=   "source" | "sink"
<flow_type>          ::=    <param_type_spec>
```

6.2.7 Exemple de déclaration de modèle d'interface

Nous donnons ci-dessous un exemple de modèle d'interface CSMConfiguration qui est dérivé par héritage d'un modèle d'interface ServiceManagement tel que défini dans le Management de module. Il contient les définitions des opérations, des attributs et une définition de comportement.

```
interface CSMConfiguration: Management:: ServiceManagement {
    behaviour
        behaviourText
        "This interface serves to configure the co CSM.

        The ReadState operation returns a complete
        representation of the CSM state. The WriteState operation
        allows the complete CSM state to be set.";

        usage
        "Operation init must be invoked prior to other
        operations defined on the service."
```

6.3 Modèle d'objet

6.3.1 Structure

Une spécification de modèle d'objet se compose de deux parties haut niveau. La première prend en charge l'héritage et est associée à la déclaration de l'identificateur du modèle d'objet. La deuxième est le corps du modèle d'objet, qui comporte les principales sous-parties du modèle suivant:

- une spécification de comportement;
- une spécification des interfaces requises;
- une spécification des interfaces prises en charge;
- une spécification d'initialisation.

Ci-dessous sont examinées ces spécifications de manière plus détaillée, ainsi que la syntaxe qui prend en charge l'héritage.

La syntaxe suivante est définie pour la déclaration du modèle d'objet:

```
<object_template>           ::=  <object_template_header>
                                 "{" <object_template_body> "}"
<object_template_header>    ::=  "CO" <identifier>
                                 [ <object_inheritance_spec> ]
<object_inheritance_spec>   ::=  ":" <scoped_name> { "," <scoped_name> }*
<object_template_body>      ::=  [<supporting_def_spec>]
                                 [<interface_def_spec>]
                                 [<object_behaviour_spec>]
                                 [<reqrd_interf_templates>]
                                 <suptd_interf_templates>
                                 [<object_init_spec>]
```

6.3.2 Héritage de modèles d'objet

L'héritage de modèles d'objet est destiné à permettre la prise en charge de la réutilisation de spécification et à disposer d'un mécanisme pour définir la compatibilité via des relations de sous-typage.

La syntaxe suivante est définie pour l'héritage de modèle d'objet:

```
<object_template_header>    ::=  "CO" <identifier>[ <object_inheritance_spec> ]
<object_inheritance_spec>   ::=  ":" <scoped_name> { "," <scoped_name> }*
```

L'héritage de modèles d'objet équivaut à la simple inclusion dans le modèle d'objet dérivé, de tous les attributs de contrainte, définitions de type, spécifications de modèles d'interface opérationnelle et flux requis et pris en charge à partir des modèles d'objet de base. Cette inclusion fait intervenir tous les attributs, types et modèles d'interface du modèle d'objet de base, y compris ceux obtenus par héritage depuis les autres spécifications de modèle d'objet.

Il n'y a pas restriction sur les noms de modèles ou de types d'interface hérités des modèles d'objet de base ITU-ODL.

L'interface initiale spécifiée dans un modèle d'objet dérivé doit être du même type que, ou dérivée de, tous les modèles d'interface initiaux des modèles d'objet de base correspondant.

6.3.3 Spécification du comportement de modèle d'objet

Le comportement de modèle d'objet est spécifié comme une chaîne dans le modèle d'objet, qui doit décrire le rôle d'un objet dans la fourniture de services via chacune de ses interfaces.

La syntaxe suivante est définie pour la spécification du comportement de modèle d'objet:

```
<object_behaviour_spec>        ::=   "behaviour"
                                     <string_literal> ";"
```

6.3.4 Modèles d'interface requise

Le deuxième comporte les interfaces requises (déclarées) qui spécifient les modèles d'interface utilisés par les instances du modèle d'objet pour exécuter leurs fonctions et fournir leurs services.

Il est possible de désigner une interface requise via un bloc <tagged_name> directement depuis un modèle d'objet particulier. Dans ce cas, le modèle d'objet spécifié doit avoir un modèle d'interface pris en charge, qui est compatible au modèle d'interface requis spécifié. Les règles de compatibilité sont ouvertes; on peut citer à titre d'exemple, les règles définies dans [2] et [3]. Cette notation garantit que la compatibilité entre les interfaces requises et prises en charge peut être vérifiée à ce niveau de spécification statique.

La syntaxe suivante est définie pour la définition du modèle d'interface requise:

```
<reqrd_interf_templates>   ::=   "requires" <req_interf_defn>
                                 {"," <req_interf_defn> }* ";"
<req_interf_defn>          ::=   <scoped_name> | <tagged_name>
```

6.3.5 Modèles d'interface prise en charge

Les interfaces prises en charge (déclarées) d'un modèle d'objet sont les interfaces listées comme étant prises en charge dans les spécifications du modèle. Les instances des modèles d'interface déclarées comme étant prises en charge peuvent être proposées par des instances des modèles en cours de définition.

La syntaxe suivante est définie pour les déclarations d'interfaces prises en charge:

```
<suptd_interf_templates>   ::= "supports" <suptd_interf> ";"
<suptd_interf>             ::= <suptd_interf_defn> {"," <suptd_interf_defn> }*
<suptd_interf_defn>        ::= <scoped_name> |      <interface_template>
```

6.3.6 Spécification d'initialisation des modèles d'objet

La spécification d'initialisation identifie un modèle d'interface, dont une référence sera retournée à l'instanciateur du modèle d'objet en cours de définition. Cette interface peut être utilisée pour initialiser l'objet nouvellement instancié. Il convient de noter que l'interface initiale est aussi l'une des interfaces opérationnelles prises en charge[6].

La syntaxe suivante est définie pour la spécification de l'initialisation:

```
<object_init_spec>         ::=   "initial"
                                 { <scoped_name> | <interface_template> } ";"
```

6.3.7 Exemple de déclaration de modèle d'objet

Dans ce qui suit, un exemple est présenté montrant comment un modèle d'objet est déclaré. Cela commence par le mot clé "CO", qui est suivi de l'identificateur du modèle d'objet, CSMfactory. Ce modèle n'hérite d'aucun autre, comme l'indique l'absence de spécification d'héritage. Le corps du modèle est ensuite déclaré entre deux accolades.

```
CO CSMfactory {
        requires
                QoSmanagerIF;
```

---

[6] L'interface initiale n'a pas besoin d'être incluse dans la clause de définition des interfaces prises en charge.

```
        supports
                Management,
                LcgFactory,
                CSMConfiguration;

        initial
                Management;

}; // end CSMfactory
```

Le modèle d'interface MyManagement hérite de l'interface initiale du modèle d'objet de base. Il convient de noter que le modèle d'objet dérivé prend en charge un modèle d'interface, MyCSMConfiguration, qui est dérivé d'un modèle d'interface pris en charge par le modèle de base, CSMConfiguration. L'instanciation de l'un ou des deux types à un instant donné est une décision qui dépend de l'implémentation. Une implémentation du modèle d'objet MyCSMfactory peut instancier aucune ou plusieurs instances de CSMConfiguration par exemple, et est encore conforme à cette spécification. Le nombre d'instances d'un modèle d'interface particulier peut être limité par la spécification de comportement du modèle d'objet.

```
interface MyManagement: Management{
        ...
}; // end MyManagement


interface MyCSMConfiguration: CSMConfiguration{
        ...
}; // end MyCSMConfiguration


CO MyCSMfactory: CSMfactory{

        requires
                AccountingEventIF;
        supports
                MyManagement,
                MyCSMConfiguration;

        initial
                MyManagement;

}; // end MyCSMfactory
```

Nous donnons ci-après un exemple de spécification de comportement:

```
CO Timer{

        behaviour
                "Instances of this co periodically call the
                tick function of a specified TimerInterrupt
                interface.";

        requires
                TimeInterrupt;
...
};
```

6.4 Modèle de groupe d'objets

6.4.1 Structure

Une spécification de modèle d'objet se compose de deux parties de haut niveau. La première prend en charge l'héritage et est associée à la déclaration de l'identificateur du modèle de groupe d'objets. La deuxième est le corps du modèle de groupe, qui comporte les principales sous-parties du modèle suivant:

- une spécification de comportement;
- une spécification des modèles d'objets et de modèles de groupe d'objets contenus;
- une spécification des modèles d'interface, dont les instances sont visibles en dehors du groupe.

Ci-dessous sont examinées ces spécifications de manière plus détaillée, ainsi que la syntaxe qui prend en charge l'héritage.

La syntaxe suivante est définie pour la déclaration de modèles de groupe d'objets:

```
<group_template>          ::=  <group_template_header>
                               "{" <group_template_body> "}"
<group_template_header>   ::=  "group" <identifier>
                               [ <group_inheritance_spec> ]
<group_template_body>     ::=  [<supporting_def_spec>]
                               [<interface_def_spec>]
                               [<object_def_spec>]
                               [<group_def_spec>]
                               [<group_predicate_spec>]
                               <supp_comp_templates>
                               [<supported_contract_interfaces>]
                               [<required_contract_interfaces>]
```

6.4.2 Héritage de modèle de groupe d'objets

L'héritage de modèle de groupe d'objets est destiné à permettre la prise en charge de la réutilisation de spécification et à se pourvoir d'un mécanisme pour définir la compatibilité via des relations de sous-typage. L'objectif de la compatibilité pour les spécifications de modèle d'objet est de permettre la prise en charge de l'utilisation de spécifications de cadre d'objet.

La syntaxe suivante est définie pour l'héritage de groupe:

```
<group_template_header>   ::=  "group" <identifier> [ <group_inheritance_spec> ]
<group_inheritance_spec>  ::=  ":" <scoped_name> { "," <scoped_name> }*
```

L'héritage de modèle de groupe équivaut à la simple inclusion dans le modèle d'objet dérivé, de toutes définitions de type, contrats et spécifications de membres depuis les modèles de groupe de base dans le modèle de groupe dérivé. Cette inclusion fait intervenir tous les attributs, types et modèles d'objet du modèle de groupe de base, y compris ceux obtenus par héritage depuis les autres spécifications de modèle de groupe d'objets.

Il n'y a pas restriction sur les identificateurs de modèles ou de types d'objet hérités des modèles d'objet de base ITU-ODL

6.4.3 Spécification des prédicats d'un modèle de groupe d'objets

La spécification des prédicats d'un modèle de groupe d'objets a pour but d'identifier les critères, que tous les membres du groupe doivent respecter. La gamme des critères possible est ouverte, on peut citer par exemple:

- objectif de structuration;
- questions de gestion (appartenance à un domaine, application des mêmes politiques);
- questions d'implémentation (les membres du groupe assurent un service particulier).

Selon les critères, la spécification du prédicat du modèle de groupe peut contenir toute information additionnelle qui est utile dans le contexte.

La syntaxe suivante est définie pour la spécification du prédicat de groupe:

```
<group_predicate_spec>      ::=   "predicate" <string_literal> ";"
```

6.4.4 Modèles d'objets membres et modèles d'objets

Les modèles d'objets membres et les modèles de groupes d'objets sont les modèles d'objet et les modèles de groupes d'objets qui appartiennent au modèle de groupe d'objets.

NOTE – Un objet ou un groupe peut être contenu dans plusieurs groupes.

La syntaxe ci-dessous prend en charge les modèles d'objet membres.

```
<supp_comp_templates>       ::=   "members" <suptd_comp> ";"
<suptd_comp>                ::=   <suptd_comp_defn> {"," <suptd_comp_defn> }*
<suptd_comp_defn>           ::=   <scoped_name>
                            |     <object_template>
                            |     <group_template>
```

6.4.5 Contrats

Les contrats sont les interfaces des membres du groupe d'objets qui sont visibles pour les entités extérieures au groupe d'objets. On ne peut avoir accès aux membres depuis l'environnement que par ces interfaces. Dans la spécification du modèle de groupe d'objets ils sont indiqués sous la forme d'une simple liste d'identificateurs (facultativement visibles).

La syntaxe ci-dessous prend en charge les contrats:

```
<supported_contract_interfaces>     ::= "supports" <scoped_name> { "," <scoped_name> }*  ";"
<required_contract_interfaces>      ::= "requires" {<scoped_name> | <tagged_name>}
                                            { "," {<scoped_name> | <tagged_name>} }*  ";"
```

6.4.6 Exemple de déclaration de modèle de groupe

Dans ce qui suit, un exemple est présenté montrant comment un modèle de groupe est déclaré. L'exemple présenté est un modèle de groupe d'objets subnetManager.

```
interface Configuration {...};
interface Configurator {...};
interface Trail {...};
interface TC {...};

CO CMC {

        requires
                Configuration;
        supports
                Configurator;
        ...
};
CO NetworkCoordinator {
        requires
                TC, SncService, SncServiceFactory;
        supports
                Trail, TC, Configuration;
```

```
        ...
};
CO NetworkCP {
        supports
                SncService, SncServiceFactory, Configuration;
        ...
};
CO ElementCP {...};


group SubnetManager {
        predicate
                "This group manages a subnetwork"

        members
                CMC, NetworkCoordinator, CMC, NetworkCP, ElementCP;

        supported
                Configurator, Trail, TC;
};
```

ANNEXE A

BNF

A.1 Conformité

Un accepteur de référence pour l'ITU-ODL est disponible dans [9]. En cas d'ambiguïté entre le texte de la présente Recommandation et l'accepteur de référence, c'est la présente Recommandation qui fait foi.

NOTE – Un accepteur est un outil qui détermine si une spécification ITU-ODL est conforme ou non.

A.2 Conventions lexicales

L'ITU-ODL utilise les conventions lexicales et les conventions de préprocesseur de l'ODP-IDL [8].

A.3 Mots clés

La plupart des mots clés sont importés de l'ODP-IDL mais plusieurs mots clés ont été ajoutés pour prendre en charge les extensions de l'ITU-ODL relativement à l'ODP-IDL. Ces mots clés sont soulignés.

any	attribute	behaviour	behaviourText	boolean	case
char	members	const	context	default	double
enum	exception	FALSE	fixed	float	group
in	initial	inout	interface	long	module
Object	CO	octet	one-way	out	predicate
raises	requires	readonly	sequence	short	sink
source	string	struct	supports	switch	TRUE
typedef	unsigned	union	usage	void	wchar
wstring					

A.4 Notation BNF étendue

Les méta-symboles suivants sont utilisés pour décrire la syntaxe de l'ITU-ODL. La description est en formalisme de Backus-Naur (BNF, *Backus-Naur form*) étendu[7].

Symbole	Signification
`::=`	Défini comme étant
`|`	Alternativement
`<text>`	non terminale
`"text"`	terminale (c'est-à-dire, littérale)
`*`	l'unité syntaxique précédente peut être répétée zéro fois ou plusieurs fois
`+`	l'unité syntaxique précédente peut être répétée une ou plusieurs fois
`{}`	les unités syntaxiques englobées sont groupées en une seule unité syntaxique
`[]`	l'unité syntaxique englobée est facultative et peut apparaître zéro fois ou une fois

A.5 Syntaxe

La syntaxe pour l'ITU-ODL est présentée ci-dessous. Les expressions provenant de l'ODP-IDL sont indiquées par une "*".

Syntaxe de haut niveau

```
<odl_spec>          ::=  <definition>*
<definition>        ::=  <module> ";"
                    |    <group_dcl> ";"
                    |    <object_dcl> ";"
                    |    <interface_dcl> ";"
                    |    <supporting_def> ";"
```

A.5.1 Syntaxe de module

```
<module>            ::=  "module" <identifier> "{" <definition>+ "}"
<scoped_name>       ::=  <identifier>
                    |    "::"<identifier>
                    |    <scoped_name> "::" <identifier>
<tagged_name>       ::=  <scoped_name> "." <scoped_name>
```

A.5.2 Syntaxe de groupe

```
<group_dcl>                 ::=  <group_forward_dcl>
                            |    <group_template>
<group_forward_dcl>         ::=  "group" <identifier>
<group_template>            ::=  <group_template_header>
                                 "{" <group_template_body> "}"
<group_template_header>     ::=  "group"<identifier>
                                 [ <group_inheritance_spec> ]
<group_inheritance_spec>    ::=  ":" <scoped_name> { "," <scoped_name> }*
```

---

[7] Noter que la concaténation de symboles a une priorité plus grande que |. Par exemple, X X X | Y Y Y est équivalent à {X X X} | {Y Y Y}.

```
<group_template_body>          ::=   [<supporting_def_spec>]
                                     [<interface_def_spec>]
                                     [<object_def_spec>]
                                     [<group_def_spec>]
                                     [<group_predicate_spec>]
                                     <supp_comp_templates>
                                     [<supported_contract_interfaces>]
                                     [<required_contract_interfaces>]
<supporting_def_spec>          ::=   {<supporting_def> ";"}*
<interface_def_spec>           ::=   {<interface_dcl> ";"}*
<object_def_spec>              ::=   {<object_dcl> ";" }
<group_def_spec>               ::=   {<group_dcl> ";" }
<group_predicate_spec>         ::=   "predicate" <string_literal> ";"
<supp_comp_templates>          ::=   "members" <supp_comp> ";"
<supp_comp>                    ::=   <supp_comp_defn> {"," <supp_comp_defn> }*
<supp_comp_defn>               ::=   <scoped_name>
<supported_contract_interfaces ::=   "supports" <scoped_name>
                                     { "," <scoped_name> }* ";"
<required_contract_interfaces> ::=   "requires" {<scoped_name> | <tagged_name>}
                                     { "," {<scoped_name> | <tagged_name>} }* ";"
```

A.5.3 Syntaxe d'objet

```
<object_dcl>                   ::=   <object_forward_dcl>
                               |     <object_template>
<object_forward_dcl>           ::=   "CO" <identifier>
<object_template>              ::=   <object_template_header>
                                     "{" <object_template_body> "}"
<object_template_header>       ::=   "CO" <identifier>
                                     [ <object_inheritance_spec> ]
<object_inheritance_spec>      ::=   ":" <scoped_name> { "," <scoped_name> }*
<object_template_body>         ::=   [<supporting_def_spec>]
                                     [<interface_def_spec>]
                                     [<object_behaviour_spec>]
                                     <suptd_interf_templates>
                                     [<reqrd_interf_templates>]
                                     [<object_init_spec>]
<object_behaviour_spec>        ::=   "behaviour" <string_literal> ";"
<reqrd_interf_templates>       ::=   "requires" <req_interf_defn>
                                     {"," <req_interf_defn> }* ";"
<req_interf_defn>              ::=   <scoped_name>
                               |     <tagged_name>
<suptd_interf_templates>       ::=   "supports" <suptd_interf> ";"
<suptd_interf>                 ::=   <suptd_interf_defn> {"," <suptd_interf_defn> }*
<suptd_interf_defn>            ::=   <scoped_name>
<object_init_spec>             ::=   "initial" <scoped_name> ";"
```

A.5.4 Syntaxe d'interface

```
<interface_dcl>                ::=   <interface_forward_dcl>
                               |     <interface_template>
<interface_forward_dcl>        ::=   "interface" <identifier>
<interface_template>           ::=   <interface_header> "{" <interface_body> "}"
<interface_header>             ::=   "interface" <identifier>
                                     [ <interf_inheritance_spec> ]
<interf_inheritance_spec>      ::=    ":" <scoped_name> { "," <scoped_name> }*
<interface_body>               ::=   [<supporting_def_spec>]
                                     [<interf_behaviour_spec>]
                                     { <op_sig_defns> | <stream_sig_defns> }
<interf_behaviour_spec>        ::=   "behaviour" {
```

```
                                    {<interf_behaviour_text> [<interf_usage_spec>]}
                        |           <interf_usage_spec> }
<interf_behaviour_text> ::=  "behaviourText" <string_literal> ";"
<interf_usage_spec>     ::=  "usage" <string_literal> ";"
```

A.5.5 Syntaxe d'interface (opérationnelle)

```
<op_sig_defns>          ::=  { <op_sig_defn> ";" }*
<op_sig_defn>           ::=  { <announcement> | <interrogation> }
 <announcement>         ::=  "one-way" "void" <identifier> <parameter_dcls>
<interrogation>         ::=  <attr_dcl>
                        |    <oper_dcl>
<attr_dcl>              ::=  ["readonly"] "attribute" <param_type_spec>
                             <declarators>
<oper_dcl>              ::=  <op_type_spec> <identifier>
                             <parameter_dcls>
                             [<raises_expr>] [<context_expr>]
<op_type_spec>          ::=  <param_type_spec>
                        |    "void"
<parameter_dcls>        ::=  "(" <param_dcl> { "," <param_dcl> }* ")"
                        |    "(" ")"
<param_dcl>             ::=  <param_attribute> <param_type_spec>
                             <declarator>
<param_attribute>       ::=  "in" | "out" | "inout"
<raises_expr>           ::=  "raises"
                             "(" <scoped_name> { "," <scoped_name> }* ")"
<context_expr>          ::=  "context"
                             "(" <string_literal> { "," <string_literal> }* ")"
<param_type_spec>       ::=  <base_type_spec>
                        |    <string_type>
                        |    <wide_string_type>
                        |    <fixed_pt_type>
                        |    <scoped_name>
```

A.5.6 Syntaxe d'interface (flux)

```
<stream_sig_defns>      ::=  { <stream_flow_defn> ";" }*
<stream_flow_defn>      ::=  <flow_direction> <flow_type>
                             <identifier>
<flow_direction>        ::=  "source" | "sink"
<flow_type>        ::=  <param_type_spec>
```

A.5.7 Syntaxe de définition de prise en charge

```
<supporting_def>        ::=  <const_dcl> ";"
                        |    <type_dcl> ";"
                        |    <except_dcl> ";"
<const_dcl>             ::=  "const" <const_type>
                             <identifier> "=" <const_exp>
<const_type>            ::=  <integer_type>
                        |    <char_type>
                        |    <wide_char_type>
                        |    <boolean_type>
                        |    <floating_pt_type>
                        |    <string_type>
                        |    <wide_string_type>
                        |    <fixed_pt_const_type>
                        |    <scoped_name>
<const_exp>             ::=  <or_expr>
<or_expr>               ::=  <xor_expr>
                        |    <or_expr> "|" <xor_expr>
```

```
<xor_expr>              ::=   <and_expr>
                        |     <xor_expr> "^" <and_expr>
<and_expr>              ::=   <shift_expr>
                        |     <and_expr> "&" <shift_expr>
<shift_expr>            ::=   <add_expr>
                        |     <shift_expr> ">>" <add_expr>
                        |     <shift_expr> "<<" <add_expr>
<add_expr>              ::=   <mult_expr>
                        |     <add_expr> "+" <mult_expr>
                        |     <add_expr> "-" <mult_expr>
<mult_expr>             ::=   <unary_expr>
                        |     <mult_expr> "*" <unary_expr>
                        |     <mult_expr> "/" <unary_expr>
                        |     <mult_expr> "%" <unary_expr>
<unary_expr>            ::=   <unary_operator> <primary_expr>
                        |     <primary_expr>
<unary_operator>        ::=   "-"
                        |     "+"
                        |     "~"
<primary_expr>          ::=   <scoped_name>
                        |     <literal>
                        |     "(" <const_exp> ")"
<literal>               ::=   <integer_literal>
                        |     <string_literal>
                        |     <wide_string_literal>
                        |     <character_literal>
                        |     <wide_character_literal>
                        |     <fixed_pt_literal>
                        |     <floating_pt_literal>
                        |     <boolean_literal>
<boolean_literal>       ::=   "TRUE"
                        |     "FALSE"
<typedcl>               ::=   "typedef" <type_declarator>
                        |     <struct_type>
                        |     <union_type>
                        |     <enum_type>
<type_declarator>       ::=   <type_spec> <declarators>
<type_spec>             ::=   <simple_type_spec>
                        |     <constr_type_spec>
<simple_type_spec>      ::=   <base_type_spec>
                        |     <template_type_spec>
                        |     <scoped_name>
<base_type_spec>                ::=     <floating_pt_type>
                        |     <integer_type>
                        |     <char_type>
                        |     <wide_char_type>
                        |     <boolean_type>
                        |     <octet_type>
                        |     <any_type>
                        |     <object type>
<object type>           ::=  "Object"
<floating_pt_type>      ::=   "float"
                        |     "double"
                        |     "long" "double"
<integer_type>          ::=   <signed_int>
                        |     <unsigned_int>
<signed_int>            ::=   <signed_long_int>
                        |     <signed_short_int>
                        |     <signed_longlong_int>
<signed_longlong_int>   ::=   "long" "long"
<signed_long_int>       ::=   "long"
```

```
<signed_short_int>          ::=  "short"
<unsigned_int>              ::=  <unsigned_long_int>
                            |    <unsigned_short_int>
                            |    <unsigned_longlong_int>
<unsigned_longlong_int>     ::=  "unsigned" "long" "long"<unsigned_long_int>
                            ::=  "unsigned" "long"
<unsigned_short_int>        ::=  "unsigned" "short"
<char_type>                 ::=  "char"
<wide_char_type>            ::=  "wchar"
<boolean_type>              ::=  "boolean"
<octet_type>                ::=  "octet"
<any_type>                  ::=  "any"
<template_type_spec>        ::=  <sequence_type>
                            |    <string_type>
                            |    <wide_string_type>
                            |    <fixed_pt_type>
<sequence_type>             ::=  "sequence" "<" <simple_type_spec> ","
                                 <positive_int_const> ">"
                            |     "sequence" "<" <simple_type_spec> ">"
<string_type>               ::=  "string" "<" <positive_int_const> ">"
                            |    "string"
<wide_string_type>          ::=  "wstring" "<" <positivi_int_const> ">"
                            |    "wstring"
<fixed_pt_type>             ::=  "fixed" "<" postive_int_const> "," <integer_literal> ">"
<fixed_pt_const_type>       ::=  "fixed"
<constr_type_spec>          ::=  <struct_type>
                            |    <union_type>
                            |    <enum_type>
<struct_type>               ::=  "struct" <identifier> "{" <member_list> "}"
<member_list>               ::=  <member>+
<member>                    ::=  <type_spec> <declarators> ";"
<union_type>                ::=  "union" <identifier>
                                 "switch" "(" <switch_type_spec> ")"
                                 "{" <switch_body> "}"
<switch_type_spec>          ::=  <integer_type>
                            |    <char_type>
                            |    <boolean_type>
                            |    <enum_type>
                            |    <scoped_name>
<switch_body>               ::=  <case>+
<case>                      ::=  <case_label>+ <element_spec> ";"
<case_label>                ::=  "case" <const_exp> ":"
                            |    "default" ":"
<element_spec>              ::=  <type_spec> <declarator>
<declarators>               ::=  <declarator> { "," <declarator> }*
<declarator>                ::=  <simple_declarator>
                            |    <complex_declarator>
<simple_declarator>         ::=  <identifier>
<complex_declarator>        ::=  <array_declarator>
<array_declarator>          ::=  <identifier> <fixed_array_size>+
<fixed_array_size>          ::=  "[" <positive_int_const> "]"
<positive_int_const>        ::=  <const_exp>
<enum_type>                 ::=  "enum" <identifier>
                                 "{" <enumerator> { "," <enumerator> }* "}"
<enumerator>                ::=  <identifier>
<except_dcl>                ::=  "exception" <identifier> "{" <member>* "}
```

ANNEXE B

Mappage avec le SDL et l'ASN.1

B.1 Justification

Comme indiqué plus haut, le langage ITU-ODL est destiné à être utilisé comme technique de description pour les aspects statiques d'une spécification du point de vue traitement. Toutefois, pour le développement de systèmes non triviaux, une spécification du comportement du système pourrait être aussi utile. Une telle spécification de comportement peut être faite au moyen d'autres langages normalisés par l'UIT. Un de ces langages est le SDL.

Pour combiner les deux langages (l'ITU-ODL et le SDL) un mappage de langage de l'ITU-ODL vers le SDL est proposé dans la présente annexe. Dans ce processus, tous les modèles d'objet, modèles d'interface, signatures d'opération et types de données sont spécifiés en ITU-ODL. Afin de spécifier le comportement de l'application, il est possible de dériver un talon SDL de la spécification ITU-ODL définissant les structures, signatures et types de données au moyen du mappage. Ce talon peut être enrichi par comportement en appliquant un mécanisme d'héritage. Cette spécification complète permet de tester ou de simuler l'application avant implémentation.

Ainsi, il est nécessaire d'avoir un mappage de langage bien défini de l'ITU-ODL vers le SDL combiné avec l'ASN.1. Ce mappage de langage offre la possibilité de dériver automatiquement un système SDL depuis une spécification ITU-ODL. Le comportement peut être ajouté de manière directe par surcharge des procédures virtuelles.

B.2 Conditions de base

Puisque l'ITU-ODL est un surensemble de l'ODP-IDL, les mappages de langage contenus en [1] pourraient aider à définir un mappage de langage pour l'ITU-ODL. Les mappages suivants doivent être effectués relativement au mappage de talon en langage C présenté en [1].

Réalisations ITU-ODL qui existent également en ODP-IDL:

- tous les types de données de base ITU-ODL;
- tous les types de données ITU-ODL construits;
- les constantes définies en ITU-ODL;
- références à des objets CORBA[8] définis en ITU-ODL;
- invocation d'opérations, dont les paramètres transférés et les résultats de réception;
- exceptions, y compris ce qui se passe lorsqu'une opération soulève une exception et la manière d'accéder aux paramètres d'exception;
- accès aux attributs.

Autres réalisations propres à l'ITU-ODL:

- modèles d'objet (CO);
- modèles de groupes d'objets;
- spécification de comportement;
- signatures de flux.

---

[8] Il y a divergence de définition du terme "objet" entre l'OMG et l'ODP. La présente Recommandation utilise la définition ODP. Si l'on renvoie à la définition OMG, celle-ci est notée comme objet CORBA.

B.3 Structure

Un fichier ITU-ODL est constitué de deux blocs SDL:

- <name>_interface;
- <name>_definition;

dans lesquels <name> est le nom de la spécification ITU-ODL (le nom de fichier par exemple).

Le bloc <name>_interface contient toutes les informations qui sont pertinentes à la fois pour le côté client et pour le côté serveur du système. Sont détaillées les définitions de type de données, les définitions de constantes, les définitions des signaux pour les opérations, flux, attributs et exceptions et les définitions de listes de signaux. En outre, il y a des définitions de procédure qui peuvent être utilisées par le client pour déclencher une opération sur un serveur et qui gèrent l'échange des signaux entre le client et le serveur (il faut noter qu'il s'agit seulement d'une notation abrégée).

Le bloc <name>_definition contient les squelettes pour le côté serveur du système. En fait, il y a des types de processus pour les modèles d'interface ITU-ODL et des types de bloc pour les modèles d'objet et modèles de groupes ITU-ODL.

B.4 Noms à visibilité définie

Tous les modèles, types, constants et exceptions doivent impérativement être définis en SDL en utilisant leur nom global. Cela est impératif car le SDL ne connaît pas de champ de visibilité.

Si des noms globaux sont utilisés dans une spécification ASN.1, tous les soulignements doivent être remplacés par des traits d'union et tous les identificateurs de type doivent commencer par une lettre majuscule. Tous les autres identificateurs doivent commencer par une lettre minuscule.

B.5 Mappage de module

Le module ITU-ODL ne peut être mappé avec le SDL car en SDL, il n'existe pas d'opérateur champ de visibilité. Toutes les réalisations ITU-ODL comme les modèles d'interface ou les types de données définis dans un module doivent être visibles pour tous les modèles d'objet définis en dehors de ce module. Il faut donc que toutes les définitions pour les modèles d'interface, types de données, modèles d'objet, etc. inclus dans un module soient faites dans les progiciels SDL en utilisant leurs noms globaux (voir B.4).

B.6 Mappage de modèles d'interface, d'opération, de flux et d'attribut

Les modèles d'interface ITU-ODL sont mappés en des types de processus contenus dans le bloc <name>_definition. Ces types de processus contiennent toute la spécification de comportement du modèle d'interface sous forme de commentaires (texte informel), les opérations assurées au niveau de cette interface sous forme de procédures virtuelles, les flux sous forme de déclarations de variables distantes et les attributs sous forme de déclarations de variables locales.

Opérations

Les opérations ne peuvent pas actuellement être mappées en procédures distantes. En effet, le SDL ne contient pas de mécanisme permettant de traiter les exceptions. Toutefois, le concept d'exception est une exigence fondamentale lorsqu'il s'agit de décrire des systèmes distribués. Par conséquent, les opérations sont mappées en un ensemble de signaux comme suit:

- pCALL_<global name of operation>;
- pREPLY_<global name of operation> (non valable pour les opérations unidirectives).

Le signal pCALL achemine tous les paramètres entrée et entrée-sortie de l'opération; le signal pREPLY chemine tous les paramètres entrants et entrants-sortants de l'opération et la valeur retour. Les signaux son définis dans le bloc <name>_interface. Le type de processus généré pour le modèle d'interface dans le bloc <name>_definition contient une procédure locale pour l'opération. Le nom de la procédure est le nom de l'opération. Les paramètres de l'opération sont déclarés comme paramètres entrée ou entrée/sortie dans la procédure SDL. Si un paramètre est spécifié comme étant un paramètre sortie en ITU-ODL, il est mappé en un paramètre entrée/sortie en SDL.

Un client peut invoquer une opération au niveau du serveur en envoyant un signal pCALL approprié. Le serveur reçoit le signal et (selon son état) appelle la procédure locale qui contient l'implémentation de l'opération. Le résultat est renvoyé au client en utilisant le signal pREPLY.

NOTE 1 – Pour simplifier le mécanisme d'appel du côté client, le bloc <name>_definition contient une procédure pour chaque opération qui envoie le signal pCALL, attend la réponse dans l'état attente et renvoie le résultats vers le client. Voir les exemples ci-dessous.

NOTE 2 – Les exceptions sont également mappées en signaux, et soulever une exception consiste seulement à renvoyer un signal d'exception approprié au client à la place du signal pREPLY.

NOTE 3 – Pour s'assurer que les signaux pCALL et pREPLY ne sont pas mélangés, chaque invocation doit contenir un identitificateur propre, qui est ensuite inclus dans la terminaison. Ce mappage n'impose pas une façon concrète de l'implémenter.

```
exception ex{
};

interface i{
      behaviour
            behaviourText
              "The interface serves for contacting type management";
            usage
              "Operation op must be invoked prior to other operations defined on the service ";

      string op (
            in boolean p1,
            inout char p2,
            out octet p3
      )
       raises ( ex);

      one-way void op1(
            in double p1
      );
} /*end opr interface */;

use idltypes ;
package name_interface ;
     /* name shall be replaced by the name of the ODL specification */
     signal pCALL_i_op(ODL_boolean,ODL_char);
     signal pREPLY_i_op(ODL_char,ODL_octet,ODL_string);
     signal pCALL_i_op1(ODL_double);
     signal pREPLY_i_op1;
     signal pRAISE_ex;
     signallist i_INVOCATIONS=pCALL_i_op,pCALL_i_op1 ;
     signallist i_TERMINATIONS=pREPLY_i_op,pRAISE_ex ;
     procedure i_op ;
          fpar in p1 ODL_boolean,
               in/out p2 ODL_char,
               in/out p3 ODL_octet,
               in server ODL_Object ;
```

```
          returns ODL_string ;
          dcl ex_var ex,return_var ODL_string ;
          start;
               decision (server);
                    (Null): output pCall_i_op(p1,p2);
                    else: output pCall_i_op(p1,p2) to server ;
               enddecision;
               nextstate Wait ;
          state Wait ;
               save * ;
               input pReply_i_op(p2,p3,return_var);
                    output pReply_i_op(p2,p3,return_var) to self ;
                    return return_var;
               input pRAISE_ex;
                    output pRAISE_ex to self;
               return return_var;
          endstate Wait ;
     endprocedure i_op ;
endpackage name_interface ;

use idltypes;
use name_interface ;
package name_definition ;
     /* name shall be replaced by the name of the ODL specification */
     process type <<package name_definition>> i /*interface definition i*/
          comment
               "The interface serves for contacting type management
               Operation op must be invoked prior to other operations defined on
               the service ";
          gate i_INVOCATIONS in with (i_INVOCATIONS) ;
          gate i_TERMINATIONS out with (i_TERMINATIONS) ;
          dcl  op_p1 ODL_boolean,op_p2 ODL_char,op_p3 ODL_octet,op_return
          ODL_string,op1_p1 ODL_double ;
          virtual procedure <<package name_definition/process type i>> op ;
               fpar in p1 ODL_boolean,
                    in/out p2 ODL_char,
                    in/out p3 ODL_octet ;
               returns ODL_string ;
          endprocedure <<package name_definition/process type i>> op ;
          virtual procedure <<package name_definition/process type i>> op1 ;
               fpar in p1 ODL_double ;
          endprocedure <<package name_definition/process type i>> op1 ;
     endprocess type <<package name_definition>> i ;
endpackage name_definition ;
```

Flux

Les signatures de flux sont représentées en SDL comme des déclarations de variables distantes. Le producteur exporte la variable de flux et le consommateur l'importe. Cela signifie que du côté serveur, un flux spécifié comme un puits est mappé en une spécification de variable importée et qu'un flux spécifié comme une source est mappé en une déclaration de variable importée. Ces déclarations de variables sont contenues dans un type de processus généré pour le modèle d'interface en utilisant leurs noms globaux. Du côté client il doit être traité réciproquement, mais cela n'est pas automatiquement généré.

NOTE – Il convient de noter que les variables importées et exportées doivent être déclarées comme étant distantes dans le bloc <name>_definition.

Références

Les références aux interfaces ITU-ODL seront indiquées sous la forme de PId SDL. Ces PId peuvent être des arguments, des résultats retournés ou des composantes de définitions de types.

Pour tous les modèles d'interface un nouveau type est introduit représentant la référence à cette interface.

Exemple:

```
interface i { /* ITU-ODL */
...
};
IRef ::= PId; /* SDL + ASN.1 */
```

Un appel d'un client vers un serveur désigné par une référence spéciale est exécuté par une procédure d'appel contenue dans le bloc <name>_interface en ajoutant le PId du serveur comme dernier paramètre.

Attributs

Les déclarations d'attributs ITU-ODL définis dans un modèle d'interface sont mappées en des définitions de variables locales dans la définition de type de processus représentant le modèle d'interface ITU-ODL. De plus, deux procédures distantes exportées par le type de processus sont définies permettant l'accès à distance (en lecture-écriture) à ces attributs par le client qui importe ces procédures distantes. Lorsqu'un attribut est déclaré lecture seulement (readonly) dans la spécification ITU-ODL, une seule procédure distante est fournie permettant de lire les valeurs de l'attribut correspondant. La déclaration distance est fait dans le bloc <name>_interface.

Exemple:

```
interface i3 {       /* ITU-ODL */
  attribute boolean attr1;
  readonly attribute short attr2;
};
use idltypes ;
package name_interface ;
     /* name shall be replaced by the name of the ODL specification */
     remote procedure i3__set_attr1;
          fpar in ODL_boolean;
     remote procedure i3__get_attr1;
          returns ODL_boolean;
     remote procedure i3__get_attr2;
          returns ODL_short;
endpackage;
use idltypes;
use name_interface;
package name_definition;
     /* name shall be replaced by the name of the ODL specification */
     process type i3;   /* SDL + ASN.1 */
          dcl
               attr1 ODL_boolean,
               attr2 ODL_short;
          exported procedure i3__set_attr1 referenced;
          exported procedure i3__get_attr1 referenced;
          exported procedure i3__get_attr2 referenced;
     endprocess type i3;
     exported procedure i3__set_attr1;
          fpar
               in value ODL_boolean;
```

```
          start;
               task
           attr1 := value;
          return;
     endprocedure i3__set_attr1;
     exported procedure i3__get_attr1;
          returns ODL_boolean;
          start;
               return attr1;
     endprocedure i3__get_attr1;
     exported procedure i3__get_attr2;
          returns ODL_short;
          start;
               return attr2;
     endprocedure i3__get_attr2;
endpackage;
```

B.7 Héritage de modèles d'interface

L'héritage de modèles d'interface définie en ITU-ODL doit être représenté sous forme d'une structure plate. Cela signifie que toutes les réalisations pour des opérations, flux et attributs définis dans le modèle d'interface de base doivent être définies de nouveau dans le modèle d'interface dérivé. Les spécifications de comportement ne sont pas reçues en héritage.

Le mécanisme d'héritage du SDL ne peut pas être utilisé car seul l'héritage simple est pris en charge et non pas l'héritage multiple comme en ITU-ODL.

Si la spécification ne contient qu'un héritage simple, la capacité héritage du SDL peut être utilisée[9].

B.8 Mappage des modèles d'objet

Les modèles d'objet sont représentés en SDL par des types de bloc définis dans le bloc <name>_definition en utilisant le nom global du modèle d'objet. De même que pour les modèles d'interface la spécification de comportement des modèles d'objet est mappée en un commentaire. Les autres parties du modèle d'objet sont mappées comme suit:

Interfaces requises

La liste des modèles d'interfaces requises est mappée en un commentaire.

Interfaces prises en charge

Les modèles d'interface annoncés dans la liste des interfaces prises en charge sont mappés comme suit:

pour chaque modèle d'interface un nouveau type de processus virtuel est introduit dans le type de bloc représentant le modèle d'objet. Ces types de processus héritent des types de processus correspondants définis pour le modèle d'interface (voir B.6). Il a le nom global du modèle d'interface.

Spécification d'initialisation

Pour le modèle d'interface annoncé dans la réalisation initiale, un nouveau type de processus virtuel est introduit dans la définition de type de bloc. Ces types de processus héritent des types de processus correspondants définis au niveau type de système (voir B.6). Il a le nom global du modèle d'interface.

---

9 Il appartient aux outils utilisés de décider d'utiliser ou non l'héritage SDL.

Héritage

L'héritage de modèles d'objet doit être mappé sous forme d'une structure plate. Cela est dû au fait que le SDL n'est pas prévu pour les héritages multiples. Si un héritage simple est contenu dans le fichier ITU-ODL, l'héritage SDL peut être utilisé.

Exemple:

```
interface i;          /*ITU-ODL*/
interface i1;
interface i2;
CO o {
      behaviour
            "use i to initialize object" ;
      requires
            i2;
      supports
            i1;
      initial
            i;
} /*end object */;

use idltypes ;
package name_interface ;
...
endpackage name_interface ;

use idltypes ;
use name_interface ;
package name_definition ;
     /* name shall be replaced by the name of the ODL specification */
     process type <<package name_definition>> i /*interface definition i*/;
          ...
     endprocess type <<package name_definition>> i ;
     process type <<package name_definition>> i1 /*interface definition i1*/;
          ...
     endprocess type <<package name_definition>> i1 ;
     process type <<package name_definition>> i2 /*interface definition i2*/;
          ...
     endprocess type <<package name_definition>> i2 ;
     block type <<package name_definition>> o /*object definition o*/
          comment "use i to initialize object"
          comment "requires i2";
          process type <<Block type o>> i1 /*supported interface i1*/
               inherits <<package name_definition>> i1;
          endprocess type <<Block type o>> i1;
          process type <<Block type o>> i /*initial interface i*/
               inherits <<package name_definition>> i;
          endprocess type <<Block type o>> i;
     endblock type <<package name_definition>> o ;
endpackage name_definition ;
```

B.9 Mappage des modèles de groupes d'objets

Les modèles de groupes d'objets sont mappés en types de bloc dans le bloc <name>_definition en utilisant le nom global du modèle de groupe. Ces types de bloc contiennent une sous-structure de bloc avec des définitions de type de bloc pour chaque membre du modèle de groupe qui héritent des types de blocs générés pour les membres eux-mêmes. Il a le nom global du modèle composé des membres.

Le prédicat du modèle de groupe est mappé en un commentaire.

Les contrats ne sont pas mappés.

L'héritage de modèles de groupe doit être mappé sous forme d'une structure plate. Cela est dû au fait que le SDL n'est pas prévu pour les héritages multiples. Si un héritage simple est contenu dans le fichier ITU-ODL, l'héritage SDL peut être utilisé.

Exemple:

```
interface i;          /*ITU-ODL*/
interface i1;
interface i2;

CO o;

group g{
      predicate "All group members have the following security policy:..." ;
      members ::o;
} /*end group */;

use idltypes ;
package name_interface ;
...
endpackage name_interface ;

use idltypes ;
use name_interface ;
package name_definition ;
     /* name shall be replaced by the name of the ODL specification */
     ...
     block type <<package name_definition>> o /*object definition o*/ ;
     ...
     endblock type <<package name_definition>> o ;

     block type <<package name_definition>> g /*group definition g*/;
          block type <<Block type g>> o /*group member definition o*/
               inherits <<package name_definition>> o;
          endblock type <<Block type g>> o;
     endblock type <<package name_definition>> g ;
endpackage name_definition ;
```

B.10 Mappage des constantes

Les constantes définies en ITU-ODL sont mappées en synonymes en SDL.

Exemple:

```
const float x = 7.5 + 3.4;        /* ITU-ODL */

synonym x ODL_REAL = 7.5 + 3.4;   /* SDL + ASN.1 */
```

B.11 Mappage des types de données de base

Dans le présent sous-paragraphe on montre comment les types de données de base en ITU-ODL sont exprimés en ASN.1. Tous les types de données de base sont définis sous forme de types ASN.1

comme le montre le tableau ci-dessous[10]. Chaque type reçoit un nom de la forme: ODL_<typename> dans lequel le nom de type est un des types de données de base ITU-ODL. Les variables de ces types peuvent être déclarées dans une déclaration dcl SDL et peuvent être utilisés de la même manière que les variables des types de données SDL normaux. La manière d'utiliser les types de données ASN.1 dans une spécification SDL est décrite en [7].

Type de données de base ITU-ODL	Types de données ASN.1 correspondants
Short	**INTEGER(-2**^{15}**..2**^{15}-1)
Unsigned short	**INTEGER(0..2**^{16}-1)
Long	**INTEGER(-2**^{31}**..2**^{31}-1)
Unsigned long	**INTEGER(0..2**^{32}-1)
Float	**REAL** (représente des nombres simple précision à virgule flottante IEEE[11])
Double	**REAL** (représente des nombres double précision à virgule flottante IEEE)
Char	**VisibleString SIZE(1)**
Boolean	**BOOLEAN**
Octet	**BIT STRING SIZE(8)**
Any	**ANY**

B.12 Mappage des types de données réalisés

B.12.1 Mappage des types de structure

Les types de structure définis dans l'ITU-ODL comme structure sont représentés en ASN.1 par une SEQUENCE. Les membres du même type dans la structure peuvent être notés comme une liste comme dans l'ITU-ODL. Cela n'est pas possible dans une SEQUENCE ASN.1. Par conséquent, chaque membre doit être spécifié séparément.

Exemple:

```
struct S {    /* ITU-ODL */
  long mem1;
  short mem2;
};
S ::= SEQUENCE {    /* SDL + ASN.1 */
    mem1 ODL-long,
    mem2 ODL-short
};
```

Il existe des opérateurs qui génèrent une instance du type défini ci-dessus et pour accéder à ses membres et modifier ces derniers. Ces opérateurs sont explicités en [7].

---

[10] On peut aussi mapper les types de données en types de données SDL. Ce mappage n'est pas décrit dans la présente annexe.

[11] Norme IEEE sur l'arithmétique binaire à virgule flottante (*binary floating point arithmetic*), ANSI/IEEE Std. 754-1985.

```
Dcl
     s S,
     i ODL_long,
     j ODL_short;
...
     task
          s := (. 2, 1 .),
          i := s!mem1,
          j := s!mem2,
          s!mem2 := 5;
```

B.12.2 Mappage des unions

Une union ITU-ODL est mappée en une SEQUENCE ASN.1 contenant un marqueur indiquant de quel type il s'agit et un CHOICE de tous les types définis dans l'union ITU-ODL.

Exemple:

```
union u switch (short) { /* ITU-ODL */
  case 1: long l;
  case 2: short s;
  default: float f;
};
```

```
U ::= SEQUENCE {   /* SDL + ASN.1 */
   tag ODL_short,
   union CHOICE {
      l ODL-long,
      s ODL-short,
      f ODL-float
   }
};
```

Le discriminateur est toujours appelé marqueur et CHOICE, union.

```
dcl
   u U,
   ll ODL_long,
   ss ODL_short,
   ff ODL_float;
...
/* making a call to get a value for u */
...
decision (u!tag);
   (1): task
             ll := u!union!l;
   (2): task
             ss := u!union!s;
   else: task
             ff := u!union!f;
enddecision;
```

B.12.3 Mappage des énumérations

Les énumérations (enum) ITU-ODL sont représentées en ASN.1 par ENUMERATED. L'ordre des éléments dans l'énumération ne doit pas être modifié. Un ordre explicite dans l'énumération ASN.1 doit être mis en évidence.

Pour les types énumération, il existe des opérateurs relationnels pour obtenir le premier et le dernier énumérateur et le prédécesseur et le successeur de chaque énumérateur. Ces opérateurs sont donnés en [7].

B.12.4 Mappage des types de séquence

Une SEQUENCE ITU-ODL est mappée en un type SEQUENCE OF ASN.1, avec ou sans la limite de taille selon la description de la SEQUENCE ITU-ODL.

Exemple:

```
typedef sequence <short, 30> seq;  /* ITU-ODL */
SEQ ::= SEQUENCE SIZE(30) OF ODL-short;     /*SDL + ASN.1 */
```

Les opérateurs pour accéder aux membres de SEQUENCE OF et modifier ces derniers sont définis en [7].

B.12.5 Mappage de chaîne

Une chaîne STRING est représentée en ASN.1 comme une VisibleString, avec ou sans la limite de taille selon la description de la chaîne STRING ITU-ODL.

Le type ASN.1 prend le nom ODL_string.

B.12.6 Mappage des matrices

Une matrice ITU-ODL est mappée sur un type ASN.1 séquence-de pour chaque dimension définie avec une limite de taille.

Exemple:

```
<type> a[8][7][67];        /* ITU-ODL */
```

Si elle fait partie d'une structure, la déclaration ASN.1 correspondante sera:

```
a SEQUENCE SIZE(8) OF SEQUENCE SIZE(7) OF SEQUENCE
                                SIZE(67) OF <type>;
```

S'il s'agit d'une spécification de type, la déclaration ASN.1 correspondante sera:

```
A ::= SEQUENCE SIZE(8) OF SEQUENCE SIZE(7) OF SEQUENCE
                                        SIZE(67) OF <type>;
```

B.13 Mappage des exceptions

Pour chaque EXCEPTION définie dans une spécification ITU-ODL, un type SEQUENCE ASN.1 est défini dans le bloc <name>_interface avec le nom global de l'exception. Cette SEQUENCE contient les paramètres de l'exception. En outre, il y a une définition de signal pRAISE_<name> dans laquelle le nom est nom global de l'exception. Ce signal achemine le type de données défini pour les paramètres de l'exception. Le serveur qui souhaite soulever une exception envoie simplement le signal pRAISE au client. Le client peut alors extraire du signal les valeurs des paramètres de l'exception.

Exemple:

```
exception ex{                    /*ITU-ODL*/
     long p1;
};

use idltypes ;                /*SDL,ASN.1*/
package name_interface ;
     /* name shall be replaced by the name of the ODL specification */
     Ex ::= SEQUENCE {
          ODL_long
     };
     signal pRAISE_ex( Ex );
endpackage;
```

B.14 Définitions additionnelles

Certaines définitions additionnelles sont ici proposées qui ne sont pas nécessaires mais qui permettent de faciliter le traitement de la spécification générée en SDL.

Liste de signaux (signallists)

Pour chaque modèle d'interface une liste de signaux est définie dans le bloc <name>_interface contenant les signaux (pCALL) pour les opérations qui peuvent être invoquées à cette interface. Une deuxième liste de signaux contient toutes les terminaisons possibles, à savoir tous les signaux pREPLY et pRAISE pour les terminaisons possibles des opérations. Ces deux listes de signaux sont appelées <name>_INVOCATIONS et <name>_TERMINATIONS, où <name> est le nom global du modèle d'interface.

Portes (Gates)

Les types de processus générés pour les modèles d'interface opérationnelle contiennent deux définitions de porte, une porte de sortie avec la liste de signaux TERMINATIONS et une porte d'entrée avec le liste des signaux INVOCATIONS. Les portes sont appelées <name>_INVOCATIONS et <name>_TERMINATIONS où<name> est le nom du modèle d'interface.

ANNEXE C

Mappage en C++

C.1 Justification

L'ITU-ODL constitue un surensemble strict de l'ODP-IDL [8]. Tous les aspects concernant la communication entre les composantes d'un système sont décrits au moyen de descriptions d'interfaces ODP-IDL. Afin de pouvoir se servir des produits CORBA existants comme plate-forme pour implémenter une spécification ITU-ODL, le mappage de la partie ODP-IDL est adaptée de l'OMG [1]. Il s'agit de faire en sorte que les compilateurs ODP-IDL propres à l'ORB existant puissent être utilisés pour mapper la partie ODP-IDL d'une spécification ITU-ODL en C++.

De plus, la composante ou l'information relative à la composante doit être reflétée dans le langage d'implémentation aussi. Cette information peut être considérée comme une information de conception. Il n'est pas exigé de la mapper dans le langage d'implémentation. L'application fonctionnera même si seule la partie ODP-IDL est mappée (seules les interfaces ont de l'importance pour la communication). Mais l'ITU-ODL doit être compris comme un langage informatique de conception et l'information qu'il contient doit faciliter la programmation des applications distribuées. Par conséquent, un mappage de langage de l'ITU-ODL en C++ [10] est ici proposé.

C.2 Conditions de base

Puisque l'ITU-ODL est un surensemble de l'ODP-IDL, les mappages de langage contenus en [1] pourraient aider à définir un mappage de langage pour l'ITU-ODL. Les mappages suivants doivent être effectués relativement au mappage de talon en langage C++ présenté en [1].

Réalisations ITU-ODL qui existent également en ODP-IDL:

- tous les types de données de base ITU-ODL;
- tous les types de données ITU-ODL construits;
- les constantes définies en ITU-ODL;
- références à des interfaces définies en ITU-ODL;

- invocation d'opérations, dont les paramètres transférés et les résultats de réception;
- exceptions, y compris ce qui se passe lorsqu'une opération soulève une exception et la manière d'accéder aux paramètres d'exception;
- accès aux attributs.

Autres réalisations propres à l'ITU-ODL:

- modèle d'objet (CO);
- modèles de groupes d'objets;
- spécification de comportement;
- signatures de flux (actuellement aucun mappage de flux n'est défini ici).

C.3 Structure

Un fichier ITU-ODL est mappé en en-tête et éventuellement des fichiers d'implémentation C++. Il n'existe pas de structure prescrite pour les fichiers générés.

C.4 Noms associés à des domaines de visibilité

Les règles relatives aux noms globaux en ITU-ODL sont les mêmes qu'en ODP-IDL. Il y a seulement trois entités de plus formant un domaine de visibilité de noms: modèles (namescope) d'interface flux, modèles CO et modèles de groupes. On peut accéder à ces entités en C++ au moyen de l'opérateur domaine de visibilité "::". Aucune autre règle n'est nécessaire.

C.5 Mappage de modules

Les modules sont supposés être mappés en espace nom (namespace) C++. Puisqu'un namespace peut être rouvert, aucun problème ne se pose concernant l'utilisation d'un compilateur ODP-IDL spécifique à l'ORB et la génération de fichiers additionnels pour les modèles de groupes et d'objet ITU-ODL.

Si pour des raisons tenant au compilateur ou à des restrictions ORB, un mappage sur les namespaces n'est pas possible, le problème de réouverture doit être résolu autrement. Par exemple en localisant les définitions des modèles de groupe et d'objet dans une classe additionnelle ::ODL. Dans ce cas, les noms globaux renvoyant aux CO ou groupes doivent être préfixés avec ce nom de classe.

C.6 Mappage de modèles d'interface, d'opération, de flux et d'attribut

Les modèles d'interfaces opérationnelles, les opérations et les attributs sont mappés en classes C++ de la même manière que celle décrite pour le mappage de l'ODP-IDL en C++ [1].

C.6.1 Clauses de comportement et d’utilisation

Les descriptions de comportement et les descriptions d'utilisation se font en texte clair seulement. Elles seront mappées en commentaires. Avec un compilateur ODP-IDL existant, elles ne seront pas mappées.

C.6.2 Flux

Non disponible actuellement.

C.6.3 Héritage de modèle d'interface

L'héritage de modèle d'interface est mappé de la même manière que pour le mappage de l'ODP-IDL en C++.

C.7 Mappage des modèles d'objet

Les modèles d'objet sont mappés en classes C++ (classes CO). La principale fonctionnalité qui est offerte par les classes générées est qu'elles contiennent des méthodes de création et de suppression des interfaces prises en charge de la définition des CO en ITU-ODL.

D'autres fonctionnalités tel le repérage ou la migration peuvent être offertes mais elles ne sont pas prescrites et dépendent de la plate-forme sur laquelle l'application doit tourner. C'est pourquoi il peut être utile d'avoir une classe de base commune analogue à ::CORBA::Object, et laisser toutes les classes générées en hériter.

C.7.1 Spécification d'interface requise

Les modèles d'interfaces requises ne sont pas elles-mêmes mappées. Il existe seulement le mappage client CORBA pour le modèle d'interface.

Toutefois, l'information d'interface requise est nécessaire si une implémentation de serveur minimale (par rapport au décompte de classes générées) doit être réalisée. Dans ce cas seulement, la partie client du mappage CORBA des interfaces requises et de la partie serveur des interfaces qui sont prises en charge doivent être générées.

C.7.2 Spécification d'interface prise en charge

Les modèles d'interfaces annoncées dans la liste des interfaces prises en charge sur un CO arbitraire sont mappés comme suit:

- il existe une classe C++ générée qui implémente la classe générée conformément au mappage ODP-IDL. Cette classe contient les implémentations pour toutes les opérations de l'interface. Le comportement consiste à déléguer un appel entrant à une classe d'implémentation. Le programmeur ne doit pas toucher à ces classes de délégation;
- la classe d'implémentation est générée pour toutes les interfaces listées quelque part comme étant prises en charge. La classe d'implémentation contient toutes les opérations comme étant des méthodes purement virtuelles. Le programmeur a la responsabilité de fournir des implémentations de ces classes d'implémentation;
- il existe une possibilité de faire une référence à une classe d'implémentation connue de l'instance de classe de délégation. Cela peut être implémenté dans le constructeur de la classe de délégation ou par un bloc method__set_implementation.

Le CO peut créer des instance de ses modèles d'interfaces prises en charge comme suit:

1) créer une instance de la classe de délégation;
2) obtenir une référence à une instance de la classe d'implémentation (peut exister ou être créée);
3) faire connaître à l'instance de classe de délégation, la référence à une classe d'implémentation.

C.7.3 Spécification d'initialisation

L'interface initiale est mappée dans le même sens que les interfaces prises en charge.

C.7.4 Héritage

L'héritage de modèle d'objet est réalisé comme un héritage C++ entre les classes C++ générées.

C.7.5 Exemple

Supposons que la définition ITU-ODL suivante doit être mappée en C++.

```
interface i3;        /*ITU-ODL*/
interface i1;
interface i2;
CO o {
        behaviour
                "use i to initialize object" ;
        requires
                i1;
        supports
                i2;
        initial
                i3;
} /*end object */;
```

La Figure 7 montre les classes générées et leurs relations. Les flèches représentent la relation d'héritage et les flèches en pointillés symbolisent la délégation. Les classes suffixées par _impl sont générées par la plupart de compilateurs ODP-IDL pour l'implémentation des interfaces. Les classes suffixées par _dpe sont les classes de délégation et les classes suffixées par _IMPL sont les classes d'implémentation présentées plus haut. La classe dpe est une classe de base pour toutes les classes de délégation et la classe dpe_o est une classe de base pour toutes les classes CO. Elles ne sont pas prescrites par ce mappage.

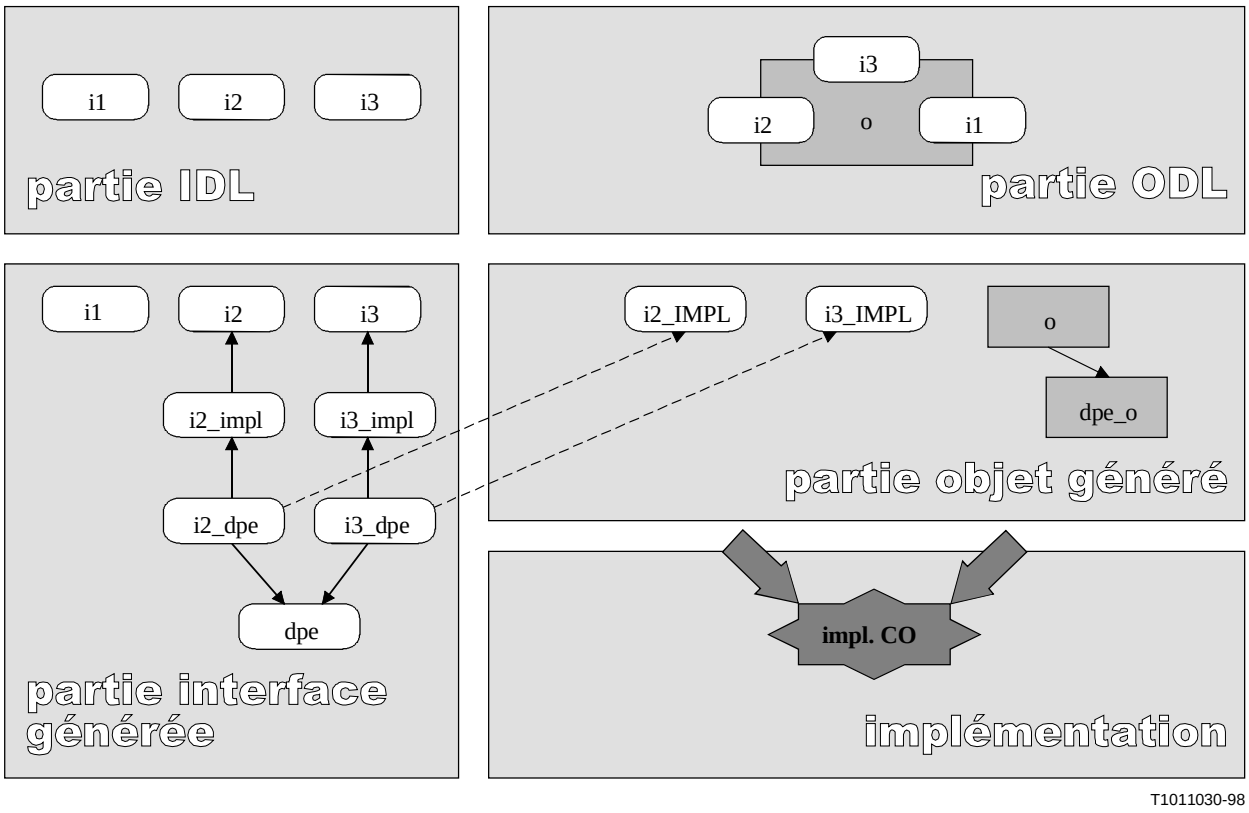

Figure 7/Z.130 – Mappage des modèles d'objets et des modèles d'interfaces prises en charge

Le programmeur est libre de choisir la manière de structurer l'application. Il doit seulement définir des implémentations pour les classes d'implémentation et éventuellement pour les classes CO.

La Figure 8 illustre certains exemples sur la manière de structurer l'implémentation:

- il est possible d'avoir une implémentation par classe d'implémentation d'interface. C'est le cas normalement lorsqu'il y a beaucoup d'informations d'état liées à l'interface. La synchronisation et l'échange d'informations est effectuée via l'implémentation de classe CO;
- il est possible d'avoir une implémentation pour un certain nombre de classes d'implémentation;
- s'il n'existe pas d'information d'état liée aux interfaces, l'implémentation peut être effectuée directement dans l'implémentation de classe CO.

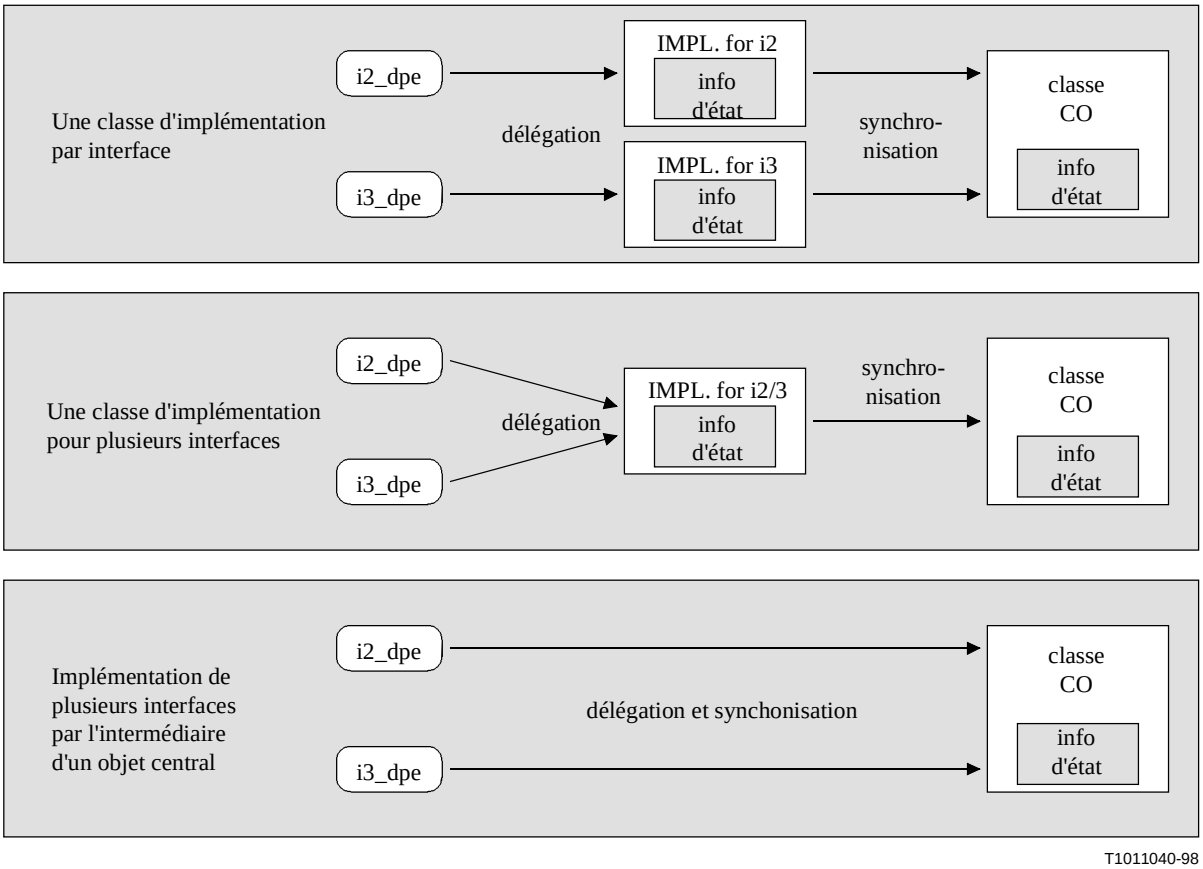

Figure 8/Z.130 – Différentes stratégies d'implémentation

C.8 Mappage des modèles de groupe

Les modèles de groupe sont des ensembles de CO et de groupes de calcul qui peuvent être regroupés ensemble pour un motif quelconque. Le motif est indiqué dans la spécification de prédicat du groupe. Comme cette spécification peut varier, il n'existe pas de mappage prescrit pour les groupes.

Toutefois, si l'objet du groupe est de regrouper ses membres pour l'implémentation, le groupe peut être mappé en une définition de classe de la même manière que les CO. Les contrats du groupe (interfaces prises en charge ou requises) sont alors mappés de la même manière que pour les objets.

C.9 Mappage de constantes

Les constantes sont mappées de la même façon que celle décrite pour le mappage de l'ODP-IDL en C++.

C.10 Mappage des types de données de base

Les types de données de base sont mappés de la même façon que celle décrite pour le mappage de l'ODP-IDL en C++.

C.11 Mappage des types de données réalisées

Les types de données réalisées sont mappés de la même façon que celle décrite pour le mappage de l'ODP-IDL en C++.

C.12 Mappage des exceptions

Les exceptions sont mappées de la même façon que celle décrite pour le mappage de l'ODP-IDL en C++.

APPENDICE I

Qualité de service

I.1 Justification

Il peut être nécessaire de compléter la spécification des aspects fonctionnels d'une opération ou flux par une spécification du "niveau de service" exigé. Il existe de nombreuses façons de spécifier la qualité de service, par exemple:

- déclaration de capacités obligatoires: par exemple, un flux doit prendre en charge des connexions présentant une certaine largeur de bande (non supérieure à ou non inférieure à); dans le cas contraire, elles ne sont pas offertes;
- déclaration des attentes: par exemple, une opération doit être déclenchée dans un certain délai dans la plupart des cas;
- déclaration de prise en charge: par exemple, lorsqu'on la lie à une instance d'interface flux, la qualité de service doit être négociée en termes, par exemple, de largeur de bande et gigue.

Pour ce qui est précisément de l'objet auquel l'information de qualité de service est associé, une définition vague est seulement donnée ici. Les qualités de service associées à une opération ou flux se composent des contraintes de synchronisation, du degré de parallélisme, des garanties de disponibilité, etc., à respecter par un objet. L'information de qualité de service intéresse la fourniture du service et moins le service lui-même. Les spécifications de qualité de service permettent de faire des déclarations sur "le niveau de service" offert par les réalisations. En général, cette information peut être donnée au moment de la spécification, ou dynamiquement par le système de gestion responsable du déclenchement de la création d'objets. Elle peut même être modifiée pendant la durée de vie de l'offre de service.

Le problème de l'ajout de spécifications de qualité de service à l'ITU-ODL peut être considéré comme étant plus un problème de sémantique que de syntaxe. Cela est évident lorsque pour une syntaxe donnée, les trois interprétations ci-dessus sont possibles puisque la signification d'une simple syntaxe doit être devinée. Par exemple, une déclaration de qualité de service associée à un flux avec une largeur de bande BandwidthType de 3 Mbit/s peut signifier que l'interface avec ce flux ne peut être offerte sauf si elle accepte exactement un flux source/puits de 3 Mbit/s exactement, ou si typiquement, des connexions au flux soient faites à 3 Mbit/s, mais d'autres valeurs peuvent être négociées, ou bien si la largeur de bande maximale d'une connexion est de 3 Mbit/s, et seules des valeurs inférieures peuvent être négociées, etc. La sémantique particulière de qualité de service à prendre en charge en ITU-ODL n'a pas été finalisée.

Dans la présente version de l'ITU-ODL, la sémantique est laissée au libre choix du programmeur.

NOTE – Une fois approuvé, le travail mené à l'UIT-T sur le "Traitement réparti ouvert – Modèle de référence – Qualité de service" devrait être intégré à la présente Recommandation.

I.2 Syntaxe

L'ITU-ODL permet d'associer un identificateur de type et variable de qualité de service type à toute définition:

- d'opération;
- de flux.

Les valeurs ne sont pas imputées aux variables de qualité de service pendant l'étape de spécification. Mais elles doivent être spécifiées au moment de l'instanciation. La sémantique dépend du service. On admet que cette capacité est fortement limitée. La sémantique d'une spécification de qualité de service peut être décrite dans la spécification de comportement d'interface.

Ci-après est donnée la syntaxe d'une spécification de qualité de service opérationnelle. Il convient de noter que les paramètres de qualité de service sont ajoutés après le mot clé "with" et que l'identificateur de la variable définie doit être unique à l'intérieur de l'interface:

```
<op_sig_defn>        ::=  { <announcement> | <interrogation> }
                          ["with" <QoS_attribute>]
<QoS_attribute>      ::=  <QoS_attr_type> <QoS_attr_name>
<QoS_attr_type>      ::=  <simple_type_spec>
<QoS_attr_name>      ::=  <simple_declarator>
```

La syntaxe pour l'adjonction d'une spécification de qualité de service spécification à un flux est la suivante:

```
<stream_flow_defn>   ::=  <flow_direction> <flow_type>
                          <identifier> ["with" <QoS_attribute>]
```

I.3 Exemple

Ci-après est donné un exemple de spécification de qualité de service de flux. Les paramètre de qualité de service apparaissent après le mot clé "with". La qualité de service est offerte en s'aidant d'une instance de VideoQoS. Une instance de VideoQoS est représentée sous forme de "flot", mais qui dépend de la valeur de Guarantee ("Statistical" ou "Deterministic"), le "float" doit être interprété comme un débit de trames "mean" ou "peak"(moyen ou crête).

```
struct VideoQoS {
        union Throughput switch (Guarantee) {          /* in frames/s */
                case Statistical:       float   mean;
                case Deterministic:     float   peak;
        };
}; // end of VideoQoS

interface I3 {
        ...
        sink VideoFlowType display with VideoQoS requiredQoS;
        ...
}; // end of I3
```

L'ITU-ODL ne spécifie pas de paramètres standards de qualité de service pour des opérations ou des flux.

I.4 Mappage avec le SDL

Le mappage d'une spécification de qualité de service en SDL est analogue au mappage des attributs d'interface à lecture seulement. Pour chaque paramètre de qualité de service, il existe une variable dans le type de processus généré pour le côté serveur ainsi qu'une procédure pour extraire cette valeur. Les conventions de nomage sont décrites au B.6.

APPENDICE II

Comparaison de l'ITU-ODL avec l'ODP-IDL et le TINA-ODL

II.1 Objectifs de l'ITU-ODL comparativement à l'ODP-IDL

Les objectifs de l'ITU-ODL sont les suivants:

- langage pour la spécification des applications (au stade du développement);
- langage pour la réutilisation des applications (au stade du développement);
- langage prenant en charge l'exécution et l'interaction des applications (au stade de l'exécution).

L'ODP-IDL partage la plupart de ces objectifs (prise en charge des spécifications des applications, de réutilisation des applications et interaction des applications).

II.2 Modèle d'objet

Le modèle d'objet qui est pris en charge par l'ITU-ODL étant le modèle OMG Objet Model [1] comme suit:

Au niveau objet, l'ITU-ODL permet la prise en charge des définitions:

- de comportement d'objet;
- des objets avec des interfaces multiples;
- des groupes d'objets.

Au niveau interface, l'ITU-ODL permet la prise en charge des définitions:

- des interfaces flux;
- de comportement d'interface.

Aux niveaux opération et flux, l'ITU-ODL permet la prise en charge des définitions:

- de signatures de flux.

II.3 Comparaison syntaxe ITU-ODL et syntaxe ODP-IDL

L'ITU-ODL dans sa version actuelle est un surensemble de l'ODP-IDL, ce qui veut dire qu'il est possible d'utiliser des spécifications ODP-IDL comme faisant partie des spécifications ITU-ODL (comme par exemple les déclarations d'interface opérationnelle). Par conséquent, la syntaxe définie en ITU-ODL pour les déclarations d'interface opérationnelle, englobe et prend en charge toutes les règles définies pour l'ODP-IDL [8].

II.3.1 Syntaxe générale

Pour une spécification ITU-ODL, en général:

- les structures ajoutées à l'ODP-IDL sont les déclarations d'objet (<object_dcl>), et les déclarations de groupe d'objets (<group_dcl>);

- les structures communes avec l'ODP-IDL sont les définitions de prise en charge (<supporting_def>) et les déclarations de module (<module>);
- la structure modifiée de l'ODP-IDL est la déclaration d'interface (<interface_dcl>).

II.3.2 Syntaxe d'interface

Dans la syntaxe de déclaration d'interface:

- les structures ajoutées à l'ODP-IDL sont la déclaration (facultative) de comportement d'interface (<interf_behaviour_spec>), spécification d'utilisation d'interface (<interf_usage_spec>) et la déclaration de signature de flux (<stream_sig_defns>);
- les structures communes avec l'ODP-IDL sont la déclaration anticipée d'interface (<interface_forward_dcl>), le mot clé et nom d'en-tête d'interface ("interface" and <identifier>) et la spécification d'héritage d'interface (<interf_inheritance_spec>);
- la structure modifiée de l'ODP-IDL est la déclaration de signature d'opération (<operation_sig_defns>).

II.3.3 Syntaxe d'opération

Dans la syntaxe de déclaration d'opération:

- les structures communes avec l'ODP-IDL sont la déclaration d'annonce (<announcement>) et la déclaration d'interrogation (<interrogation>).

Il convient de noter que toutes les adjonction à l'ODP-IDL ont été rendues facultatives.

SERIES 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 pour données et communication entre systèmes ouverts
Série Y	Infrastructure mondiale de l'information
Série Z	**Langages et aspects informatiques généraux des systèmes de télécommunication**