UNION INTERNATIONALE DES TÉLÉCOMMUNICATIONS

UIT-T Z.130

SECTEUR DE LA NORMALISATION
DES TÉLÉCOMMUNICATIONS
DE L'UIT

(07/2003)

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

Techniques de description formelle – Langage étendu de définition d'objets

Langage étendu de définition d'objets (eODL): techniques de développement de composants logiciels répartis – Bases conceptuelles, notations et correspondances technologiques

Recommandation UIT-T Z.130

RECOMMANDATIONS UIT-T DE LA SÉRIE Z

LANGAGES ET ASPECTS GÉNÉRAUX LOGICIELS 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
Langage étendu de définition d'objets	**Z.130–Z.139**
Notation de test et de commande de test	Z.140–Z.149
Notation de prescriptions d'utilisateur	Z.150–Z.159
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.349
Interfaces homme-machine orientées données	Z.350–Z.359
Interfaces homme-machine pour la gestion des réseaux de télécommunication	Z.360–Z.369
QUALITÉ	
Qualité des logiciels de télécommunication	Z.400–Z.409
Aspects qualité des Recommandations relatives aux protocoles	Z.450–Z.459
MÉTHODES	
Méthodes de validation et d'essai	Z.500–Z.519
INTERGICIELS	
Environnement de traitement réparti	Z.600–Z.609

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

Recommandation UIT-T Z.130

Langage étendu de définition d'objets (eODL): techniques de développement de composants logiciels répartis – Bases conceptuelles, notations et correspondances technologiques

Résumé

La présente Recommandation est destinée aux concepteurs, implémenteurs et gestionnaires de systèmes répartis ainsi qu'à ceux qui développent des outils pour la prise en charge des systèmes répartis.

La présente Recommandation spécifie le langage étendu de définition d'objets de l'UIT (ITU-eODL). Ce langage sert à mettre au point des systèmes répartis sur la base de composants selon quatre points de vue différents mais liés entre eux: traitement, implémentation, déploiement et environnement cible. Chaque point de vue est associé à un objectif de modélisation précis, exprimé par des concepts d'abstraction particuliers. Les types d'objet de traitement avec les ports et interfaces (opération, flux, signal) constituent les principaux concepts du point de vue de traitement permettant de décrire de manière abstraite les composants logiciels répartis en fonction de leurs interfaces potentielles. Les artefacts – c'est-à-dire les abstractions de contextes concrets de langage de programmation – et leurs relations avec les interfaces constituent le point de vue implémentation. Le point de vue déploiement décrit les entités logicielles (composants logiciels) en représentation binaire ainsi que les entités de traitement qu'elles réalisent. Le point de vue environnement cible définit les concepts de modélisation d'un réseau physique dans lequel les composants logiciels seront déployés. Les concepts des différents points de vue sont tous liés entre eux. Ces relations constituent une base essentielle pour les techniques et les outils utilisés dans le processus de développement de logiciel, depuis la conception jusqu'au déploiement, en passant par l'implémentation et l'intégration. La présente Recommandation ne traite pas encore de la phase d'essai.

Le langage ITU-eODL est une extension du langage de définition d'objets de l'UIT (ITU-ODL) [1], qu'il annule et remplace. Au départ, le langage ITU-ODL était conçu comme une extension du langage ODP-IDL [9] et il définissait des concepts de traitement fondés sur la terminologie de l'ODP [2], [3]. Le langage eODL suit ce principe. Toutefois, les définitions sont fondées sur un métamodèle et non sur l'approche classique de la syntaxe abstraite, ce qui présente plusieurs avantages. En effet, les outils liés à l'architecture MOF [4] peuvent être utilisés pour l'automatisation des transitions de modèle entre les différentes phases du développement de logiciel. De plus, pour représenter les modèles concrets instanciés à partir du métamodèle, on peut utiliser les langages existants, permettant ainsi d'intégrer différentes approches en matière de conception.

Les lecteurs de la présente Recommandation sont supposés bien connaître les sujets suivants: IDL [5], UML [11] et MOF.

La définition de l'eODL est étayée par les annexes et appendices qui suivent:

- l'Annexe A présente une syntaxe textuelle du langage eODL, destinée à être utilisée pour la représentation de spécifications en eODL. On utilise le formalisme EBNF pour définir cette syntaxe;
- l'Annexe B définit le mappage entre le métamodèle eODL et la syntaxe textuelle définie dans l'Annexe A;
- l'Annexe C spécifie un mappage de l'eODL avec le langage SDL-2000 de l'UIT, permettant de transformer automatiquement un modèle eODL en modèle SDL-2000;

- l'Annexe D donne une référence à un logiciel contenant la représentation en XML [12] du métamodèle eODL conformément au format d'échange de métadonnées XML (XMI) [6]. Cette représentation est contenue dans un fichier distinct afin que les outils UML puissent importer et traiter le métamodèle eODL;
- le paragraphe 1 présente un aperçu de la manière dont l'eODL est utilisé par les concepteurs, implémenteurs et gestionnaires d'un système réparti. L'Appendice I donne un exemple concret d'utilisation;
- l'Appendice II décrit l'ensemble du processus de développement en cas d'utilisation de l'eODL ainsi que des outils possibles.

Source

La Recommandation Z.130 de l'UIT-T a été approuvée le 22 juillet 2003 par la Commission d'études 17 (2001-2004) de l'UIT-T selon la procédure définie dans la Recommandation UIT-T A.8.

AVANT-PROPOS

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

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

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

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

NOTE

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

Le respect de cette Recommandation se fait à titre volontaire. Cependant, il se peut que la Recommandation contienne certaines dispositions obligatoires (pour assurer, par exemple, l'interopérabilité et l'applicabilité) et considère que la Recommandation est respectée lorsque toutes ces dispositions sont observées. Le futur d'obligation et les autres moyens d'expression de l'obligation comme le verbe "devoir" ainsi que leurs formes négatives servent à énoncer des prescriptions. L'utilisation de ces formes ne signifie pas qu'il est obligatoire de respecter la Recommandation.

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
1	Domaine d'application	1
2	Références	3
3	Abréviations	3
4	Définitions	4
5	Métamodèle	7
5.1	Définitions et conventions	8
5.2	Dénomination et visibilité	9
5.3	Concepts de traitement	10
5.4	Concepts d'implémentation	20
5.5	Concepts de déploiement	22
5.6	Concepts de l'environnement cible	24
6	Bibliographie	28
Annexe A – Syntaxe de l'eODL		29
A.1	Introduction	29
A.2	Conventions lexicales et base grammaticale	29
A.3	Point de vue de traitement	29
A.4	Point de vue configuration	31
A.5	Point de vue implémentation	31
A.6	Point de vue déploiement	33
A.7	Environnement cible	35
A.8	Syntaxe de l'eODL	35
Annexe B – Mappage entre métamodèle et syntaxe		43
B.1	Introduction	43
B.2	Signal et paramètre de signal	44
B.3	Type de média, média, ensemble de médias	45
B.4	Consommer et produire	46
B.5	Puits et source	47
B.6	Type d'interface	47
B.7	Types de CO, prendre en charge et requérir	48
B.8	Port provisionné et port utilisé	49
B.9	Artefact et schéma d'instanciation	50
B.10	Relation implémenter	50
B.11	Elément d'implémentation	51
B.12	Composant logiciel	52
B.13	Assemblage et configuration initiale	53
B.14	Contraintes et propriétés	54
B.15	Environnement cible, nœud et liaison entre nœuds	55
B.16	Carte d'installation	56

Page

B.17 Carte d'instanciation 57
B.18 Plan de déploiement 58
B.19 Type externe 58
Annexe C – Mappage avec le SDL-2000 59
C.1 Introduction 59
C.2 Paquetage eodl 59
C.3 Structure 60
C.4 Noms visibles 60
C.5 Mappage des concepts de traitement 60
C.6 Mappage des concepts du point de vue configuration 67
C.7 Mappage des concepts d'implémentation 68
C.8 Omission du comportement généré automatiquement 73
C.9 Concepts de l'eODL non mappés 74
C.10 Paquetage eodl prédéfini 74
Annexe D – Représentation en XML du **métamodèle** eODL 77
Appendice I – Exemple: dîner des philosophes 77
I.1 Introduction 77
I.2 Description 78
I.3 Exemple en eODL 79
I.4 Exemple en SDL-2000 81
Appendice II – Traitement de l'information et outils 102
II.1 Introduction 102
II.2 Outils de modélisation 103
II.3 Outils de génération 103
II.4 Outils de déploiement 104

Recommandation UIT-T Z.130

Langage étendu de définition d'objets (eODL): techniques de développement de composants logiciels répartis – Bases conceptuelles, notations et correspondances technologiques

1 Domaine d'application

La fourniture de techniques efficaces et d'outils pour la mise au point et l'ingénierie des systèmes répartis joue un rôle essentiel pour l'évolution future des technologies de l'information. Les systèmes de télécommunication sont des systèmes répartis particuliers, constitués de composants qui sont répartis dans les réseaux, et dans lesquels les aspects de concurrence, d'autonomie, de synchronisation et de communication doivent être pris en considération. La mise au point de systèmes très efficaces et évolutifs est une tâche complexe et ardue et des outils sont nécessaires pour toutes les phases du processus de mise au point – depuis la définition du cahier des charges jusqu'à l'intégration, l'essai et le déploiement, en passant par la conception et l'implémentation.

La génération de code en dehors des modèles de conception orientés objet permet d'avoir des composants exécutables réutilisables. Des aspects dépendant de l'environnement d'exécution et de la technologie des plates-formes de type logiciel médiateur sont intégrées par ces composants, au modèle de conception orienté objet propre à une entreprise. Chaque composant logiciel a une représentation physique (par exemple fichier binaire), qui doit être disponible pour exécution au niveau d'un nœud particulier du système réparti considéré. La présente Recommandation porte essentiellement sur la conception de ces composants.

Les techniques utilisées pour la mise au point de systèmes répartis contribuent, dans une large mesure, à réduire le temps nécessaire à la mise sur le marché d'applications et de services de télécommunication répartis. Pour traiter comme il se doit tous les aspects liés aux communications, on s'appuie sur la nature même du domaine de l'application visée. Ces aspects vont des spécifications transactionnelles relatives aux interactions entre objets aux politiques de sécurité, en passant par les questions de qualité de service. La technologie de logiciel médiateur fondé sur des objets étant largement acceptée, les plates-formes de type logiciel médiateur constituent un environnement d'implémentation idéal pour les composants considérés. Les technologies sont notamment les suivantes: CORBA ordinaire [5], composants CORBA [7] et d'autres plates-formes de traitement réparti.

La présente Recommandation est censée s'intégrer dans tous les processus de développement de logiciel se rapportant aux phases suivantes du cycle de vie des logiciels:

- phase de conception;
- phase d'implémentation;
- phase d'intégration;
- phase d'exploitation.

La présente Recommandation ne porte pas sur la phase de définition du cahier des charges.

Dans la présente Recommandation, l'accent est tout particulièrement mis sur la prise en charge au niveau technologique des transitions entre phases pour en permettre l'automatisation. La technologie fondamentale repose sur la définition de modèles à partir d'un métamodèle bien défini (voir [11]). Ce métamodèle permet d'intégrer plusieurs langages de conception existants, comme SDL [8], UML et CORBA-IDL. Il est constitué par la définition des concepts relatifs aux phases considérées du cycle de vie des logiciels. Pour représenter les modèles instanciés à partir du métamodèle, on peut utiliser les langages existants. Comme ceux-ci ne possèdent pas certains concepts, la présente Recommandation définit par ailleurs une syntaxe concrète: eODL (ODL

étendu). L'approche de type métamodèle remplace l'approche de type syntaxe abstraite couramment utilisée pour la définition des langages. L'UIT-eODL est une révision de l'UIT-ODL. La syntaxe est définie dans l'Annexe A.

Le métamodèle est donc indépendant de toute notation utilisée à la conception. Pour développer des modèles de conception, on peut appliquer des notations différentes mais ces modèles sont fondés sur les mêmes concepts. Le métamodèle peut servir à échanger des informations de conception. La notation et le métamodèle sont tous deux indépendants de tout environnement d'exécution. Le même modèle peut être projeté dans différents environnements cibles, ce qui procure une grande souplesse et est également important pour l'aspect de réutilisation des modèles de conception de composants.

La Figure 1 illustre l'intégration de la présente Recommandation dans les processus de développement de logiciel. Un modèle de conception est spécifié à partir de la définition précise du cahier des charges. La présente Recommandation définit des concepts pour différents points de vue du modèle de conception. Chaque point de vue intègre différents aspects du système à mettre au point. Les concepts du métamodèle permettent de mettre au point des modèles suffisamment puissants pour déterminer automatiquement les squelettes des composants logiciels. Ces squelettes constituent le point de départ de la phase d'implémentation, au cours de laquelle ils sont complétés par la logique commerciale. Dans la phase d'intégration, les composants logiciels complétés doivent être intégrés dans l'environnement cible.

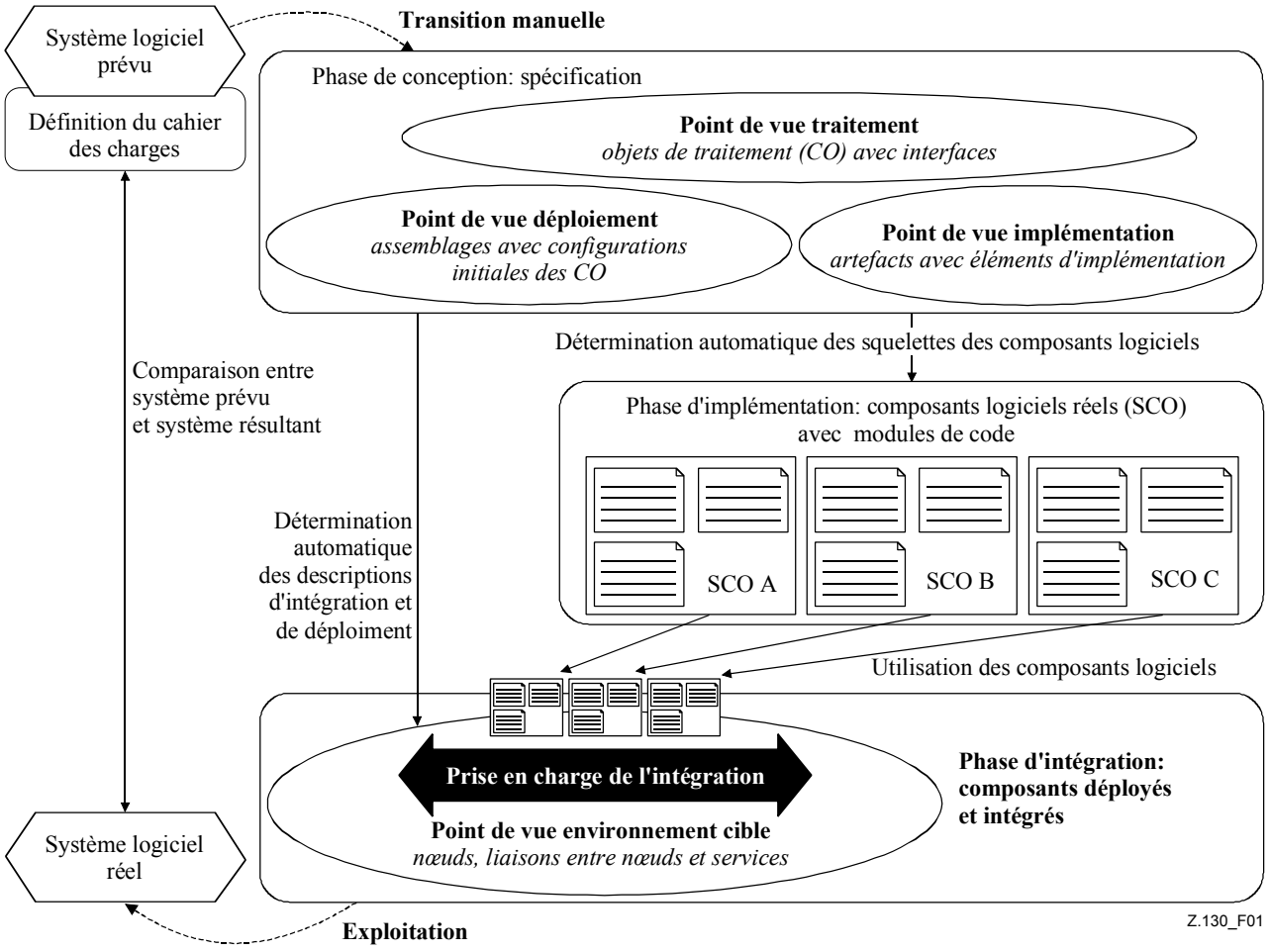


Figure 1/Z.130 – Développement de logiciel

La présente Recommandation contient également des concepts permettant de décrire la topologie et les propriétés de l'environnement cible. Conjointement avec les descriptions de déploiement et d'intégration générées automatiquement à partir de la phase de conception, la description de l'environnement cible permet d'automatiser le déploiement. Après la phase d'intégration, le système logiciel développé peut être mis en service.

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. La référence à un document figurant dans la présente Recommandation ne donne pas à ce document, en tant que tel, le statut d'une Recommandation.

[1] Recommandation UIT-T Z.130 (1999), *Langage de définition d'objet de l'UIT*.

[2] Recommandation UIT-T X.902 (1995) | ISO/CEI 10746-2:1996, *Technologies de l'information – Traitement réparti ouvert – Modèle de référence: fondements*.

[3] Recommandation UIT-T X.903 (1995) | ISO/CEI 10746-3:1996, *Technologies de l'information – Traitement réparti ouvert – Modèle de référence: architecture*.

[4] Groupe de gestion d'objets, *Meta Object Facility (MOF) Specification*, Version 1.3. Document formal/00-04-03 de l'OMG.

[5] Groupe de gestion d'objets, *The Common Object Request Broker Architecture and Specification*, Revision 2.4.2. Document formal/01-02-33 de l'OMG.

[6] Groupe de gestion d'objets, *XML Metadata Interchange (XMI) Specification*, Version 1.1. Document formal/00-11-02 de l'OMG.

[7] Groupe de gestion d'objets, *CORBA Component*, Version 3.0. Document formal/2002-06-65 de l'OMG.

[8] Recommandation UIT-T Z.100 (2002), *SDL: langage de description et de spécification*.

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

[10] Recommandation UIT-T Z.600 (2000), *Architecture de l'environnement de traitement réparti*.

3 Abréviations

La présente Recommandation utilise les abréviations suivantes:

API interface de programmation d'application (*application programming interface*)

ASN.1 notation de syntaxe abstraite numéro un (*abstract syntax notation one*)

CCM modèle de composants CORBA (*CORBA component model*)

CO objet de traitement (*computational object*)

COM modèle d'objets de composants (*component object model*)

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

CWM modèle de stockage commun (*common warehouse metamodel*)

DTD définition de type de document pour le langage XML (*document type definition for XML*)

EBNF formalisme Backus-Naur étendu (*extended Backus-Naur form*)

EJB Enterprise JavaBeans™

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

MDA architecture modélisée (*model driven architecture*)

MOF architecture métaobjet (*meta-object facility*)

OCL langage de contrainte d'objet (*object constraint language*)

ODL langage de définition d'objet (*object definition language*)

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

OSD description de logiciel ouvert (*open software description*)

RM-ODP modèle de référence du traitement réparti ouvert (*reference model for open distributed processing*)

SDL langage de description et de spécification (*specification and description language*)

TINA architecture d'intégration en réseau des informations de télécommunication (*telecommunication information networking architecture*)

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

XMI échange de métadonnées XML (*XML metadata interchange*)

XML langage balisé extensible (*extensible markup language*)

4 Définitions

La présente Recommandation définit les termes suivants:

4.1 **artefact**: abstraction de constructions concrètes de langage de programmation, par exemple une classe dans le cas de langages orientés objet (réalisée sous forme de modules de code), contenue dans un **composant logiciel**. Une instance d'**artefact réalise** (au sens d'un modèle) l'état, le comportement et l'identité d'un **CO**.

4.2 **assemblage**: description d'un système logiciel réparti par l'ensemble de tous les **types de CO** participants et la **configuration initiale**. (Utilisé par CCM.)

4.3 **anomalie**: type particulier de **terminaison** d'opération en cas d'erreur. (Défini par RM-ODP.)

4.4 **classe**: concept utilisé pour la classification des objets. Des objets peuvent être instanciés à partir de la description d'une classe. (Défini par MOF.)

4.5 **connexion**: concept utilisé pour l'échange de **références d'interface**, appartenant à des définitions de port conformément à leur type particulier de définition (relation d'*utilisation* ou de *fourniture*). Défini en tant que lien de traitement par RM-ODP utilisé par CCM.)

4.6 **architecture de composants**: **environnement de traitement réparti** prenant en charge l'interaction de **composants logiciels** répartis.

4.7 **plate-forme cible**: **architecture de composants** assurant le **déploiement** et l'exécution répartie, dans laquelle les composants sont censés être déployés.

4.8 objet de traitement (CO, *computational object*): unité fonctionnelle, qui résulte d'une décomposition fonctionnelle du système logiciel modélisé. (Défini par RM-ODP.)

4.9 type d'objet de traitement (type de CO): gabarit utilisé pour l'instanciation d'**objets de traitement**. (Défini par RM-ODP; identifié par CCM comme étant un composant CORBA.)

4.10 consommer: concept utilisé pour la modélisation de la réception d'un signal potentiel. (Utilisé par CCM.)

4.11 interaction de type média continu, signal et opération: interaction entre différents **CO** au moyen d'**éléments d'interaction** de type opération, signal ou *média continu*: **opération, attribut, consommer, produire, puits, source**. (Défini par RM-ODP et TINA (interaction de type opération et *média continu*).)

4.12 type de données: spécification d'une structure, d'un contenu et d'un comportement admissibles pour des données appartenant à un modèle de types de données (à savoir CORBA-IDL), lequel sert de base à des modèles d'information . (Défini par CORBA.)

4.13 environnement de traitement réparti: base technologique, prenant en charge les interactions entre les objets d'un système réparti. (Défini par TINA.)

4.14 déploiement: processus au cours duquel les représentations physiques des **composants logiciels** sont installées au niveau des **nœuds**, et ce de manière telle qu'elles soient prêtes à être exécutées et au cours duquel la **configuration initiale** est établie.

4.15 élément d'implémentation: relation entre un **élément d'interaction** et un **artefact**, dans laquelle une instance d'**artefact** est responsable du comportement de l'**élément d'interaction**.

4.16 implémenter: relation entre **artefacts** et **types de CO**, dans laquelle les instances d'**artefact réalisent** le comportement d'un **type de CO**.

4.17 CO initial: **CO** créé au début de l'**exécution** d'un système logiciel réparti.

4.18 configuration initiale: ensemble constitué par les **CO initiaux** et les **connexions initiales**. (Utilisé par CCM.)

4.19 connexion initiale: lien initialement établi au début de l'**exécution** d'un système logiciel réparti. (Utilisé par CCM.)

4.20 politique d'instanciation: description, sous forme de politique, de l'instanciation de divers concepts d'implémentation modélisés par des **artefacts**.

4.21 interaction: action impliquant l'environnement d'un objet. (Défini par RM-ODP.)

4.22 élément d'interaction: généralisation des concepts **opération, attribut, puits, source, consommer, produire**.

4.23 interface: ensemble des interactions possibles d'un **CO** auxquels on peut faire référence. (Défini par RM-ODP.)

4.24 attribut d'interface: type particulier d'opérations sous forme de notation abrégée pour les opérations *get* (obtenir) et *set* (fixer) pour un **type de données** particulier. (Utilisé par CORBA.)

4.25 référence d'interface: référence à une **interface**. (Correspond à la référence d'objet CORBA.)

4.26 type d'interface: description d'un ensemble d'**éléments d'interaction** sous forme de points d'extrémité nommés et identifiables d'une communication possible. (Défini par RM-ODP; correspond à l'interface IDL OMG.)

4.27 ensemble de médias: ensemble constitué de médias.

4.28 type de média: déclaration à utiliser pour le codage, la transmission et le décodage des données d'un **média**.

4.29 média: flux de données continu unidirectionnel atomique. (Remplace la notation de flux de la Rec. UIT-T Z.600 [10].)

4.30 métamodèle: définition de concepts de modélisation pour la construction de modèles dans un domaine particulier. (Défini par MOF.)

4.31 méta-métamodèle: définition de concepts de modélisation pour la construction de **métamodèles**. (Défini par MOF.)

4.32 port multiple: port assurant dynamiquement l'enregistrement et l'extraction de multiples **références d'interface**. (Défini par CCM.)

4.33 espace de noms: concept utilisé pour structurer les identificateurs des éléments d'un modèle. (Défini par RM-ODP.)

4.34 nœud: dispositif servant à interpréter les modules de code d'un **composant logiciel**. (Défini par RM-ODP.)

4.35 objet: modèle d'une entité, une entité pouvant être n'importe quel phénomène intéressant dans le domaine considéré; un objet a une identité, un état et un comportement. (Défini par RM-ODP.)

4.36 opération: élément d'une interaction de type opération, décrit par un ensemble de paramètres et de terminaisons possibles. (Défini par RM-ODP, CORBA.)

4.37 paramètre: élément de l'invocation d'une opération, décrit par un sens pour l'échange des informations et un **type de données**. (Défini par CORBA.)

4.38 port: entité utilisée pour l'enregistrement et l'extraction des **références d'interface** d'un **CO**. (Défini par CCM.)

4.39 produire: **élément d'interaction** utilisé pour envoyer des **signaux**. (Défini par CCM.)

4.40 port provisionné: **port** au niveau duquel les **références d'interface** du **CO** correspondant peuvent être extraites. (Défini par CCM.)

4.41 réaliser: correspondance de **composants logiciels** avec des **types de CO**. (Défini par UML.)

4.42 requérir: relation entre un **type d'interface** et un **type de CO**, dans laquelle les **CO** de ce **type de CO** requièrent les références d'interface du **type d'interface** provenant de l'environnement du **CO**. (Défini par TINA.)

4.43 exécution: phase pendant laquelle un **composant logiciel** est exécuté.

4.44 signal: **élément d'interaction** utilisé pour l'échange asynchrone de messages atomiques. (Défini par RM-ODP.)

4.45 port simple: **port** au niveau duquel une seule **référence d'interface** peut être enregistrée et extraite. (Défini par CCM.)

4.46 puits: **élément d'interaction** utilisé pour recevoir un **ensemble de médias**. (Défini par TINA.)

4.47 composant logiciel: entité qui est constituée de séquences d'instructions (modules de code), qui possède une représentation physique (sous la forme d'un format de données particulier) et qui peut être assemblée dans une structure de **composants logiciels** ou dans un système logiciel; pour pouvoir composer des **composants logiciels**, il faut préciser leur fonctionnalité via des **types d'interface** bien définis. Pendant l'exécution d'un **composant logiciel**, des objets sont instanciés à partir de **classes** (réalisées sous forme de modules de code).

4.48 paquetage logiciel: paquetage de **composants logiciels**. (Défini par CCM.)

4.49 source: **élément d'interaction** utilisé pour envoyer un **ensemble de médias**. (Défini par TINA.)

4.50 prendre en charge: relation entre un **type d'interface** et un **type de CO**, dans laquelle les **CO** de ce **type de CO** prennent en charge les références d'interface du **type d'interface** provenant de l'environnement du **CO**. (Défini par TINA.)

4.51 terminaison: fin d'une **opération** invoquée. (Défini par RM-ODP.)

4.52 paramètre de signal: élément d'un **signal** servant à acheminer une information; se rapporte à un **type de données**. (Défini par SDL.)

4.53 port utilisé: **port** au niveau duquel des **références d'interface** peuvent être enregistrées. (Défini par CCM.)

5 Métamodèle

Un **métamodèle** définit des concepts de modélisation à utiliser pour la construction de modèles dans un domaine particulier. Dans la présente Recommandation, le **métamodèle** est un **métamodèle** conforme à l'architecture métaobjet (MOF, *meta-object facility*). Il est décrit au moyen de diagrammes de classes UML et sa sémantique est donnée en langage naturel. Des règles de grammaticalité sont ajoutées lorsque c'est nécessaire. L'architecture MOF est la norme que l'OMG (groupe de gestion d'objets) a adoptée pour la métamodélisation. Elle définit un cadre générique pour la description et la représentation des méta-informations.

La norme MOF définit une architecture à quatre niveaux, illustrée sur la Figure 2:

- au niveau M3 se trouve un **méta-métamodèle** unique (le modèle MOF) qui définit les concepts fondamentaux nécessaires à la description orientée objet de quelque **métamodèle** que ce soit. Les constructions fondamentales sont les suivantes: classe, association, type de données, attribut de classe et héritage de classe;
- au niveau M2 se trouvent les **métamodèles** (langages). Un **métamodèle** définit les abstractions nécessaires à la construction des modèles. Il est décrit sur la base des constructions fondamentales du niveau M3 (en pratique, la syntaxe abstraite d'un **métamodèle** est fournie sous la forme d'un ensemble de diagrammes de classes). Exemples de **métamodèle**: UML, CWM (entrepôt de données) et le **métamodèle** CCM;
- au niveau M1 se trouvent les *modèles*. Ceux-ci sont décrits sur la base de l'un des **métamodèles** définis au niveau M2. Exemple de modèle: modèle de niveau réseau en UML qui définit ce qu'est un chemin;
- au niveau M0 se trouvent les données, qui sont des instances d'un modèle du niveau M1. Exemple de données: liste d'enregistrements représentant des chemins.

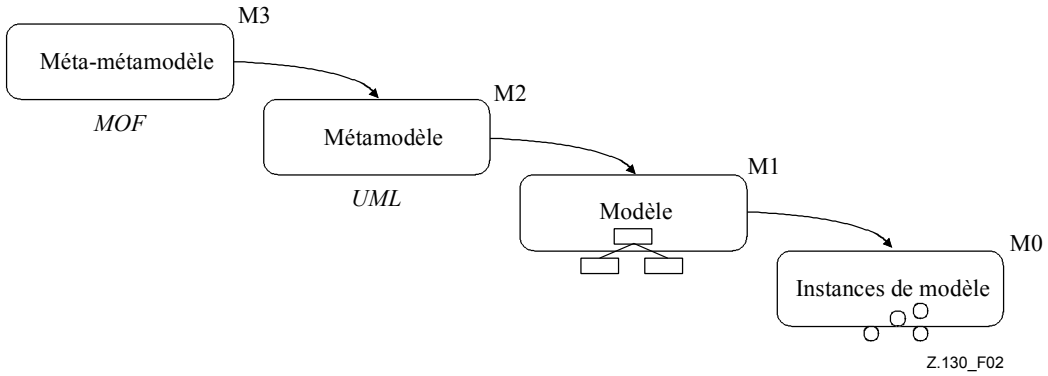


Figure 2/Z.130 – Niveaux MOF

Non seulement la norme MOF définit les constructions fondamentales servant à la description orientée objet des **métamodèles**, mais elle normalise aussi les **types d'interface** qui peuvent être utilisés au niveau d'entités de modèle (interfaces IDL OMG). Par ailleurs, la norme XMI [6] associée normalise la manière dont un modèle peut être externalisé dans un format de flux XML [12]. Le vocabulaire XML utilisé pour l'externalisation dépend uniquement des entités du **métamodèle**.

5.1 Définitions et conventions

Le présent paragraphe définit les concepts du modèle MOF (**méta-métamodèle**) qui servent à définir les concepts de l'eODL (**métamodèle**). Voir Figure 3, sur laquelle est présentée la notation utilisée aux fins de visibilité des attributs et des opérations. Cette notation est employée de manière cohérente sur toutes les figures UML. Pour plus d'informations, on se reportera au [4].

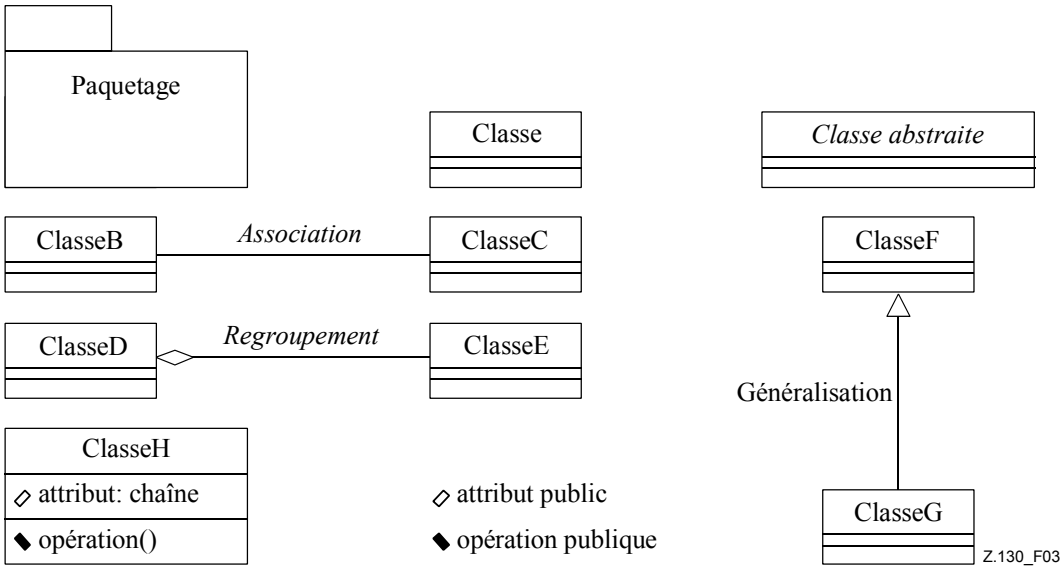


Figure 3/Z.130 – Notation UML pour les concepts MOF

5.1.1 ***Classe et objet***: une classe décrit un ensemble d'objets qui possèdent tous les mêmes caractéristiques de classe. Elle est utilisée pour la classification et sert de concept fondamental pour la construction. Un objet est une instance d'une certaine classe.

5.1.2 ***Généralisation***: une généralisation est une relation unidirectionnelle entre deux classes, qui associe la classe spéciale et la classe générale. La classe spéciale hérite de toutes les caractéristiques de la classe générale.

5.1.3 ***Association***: une association est une relation entre deux classes. S'il existe des instances des deux classes, une association peut être instanciée entre elles, mais ce n'est pas obligatoire. L'association est navigable d'un objet à l'autre.

5.1.4 ***Regroupement***: un regroupement est une relation directe entre classes, telle que les instances des classes regroupées (parties) sont considérées comme étant contenues dans des instances de la classe conteneur (c'est-à-dire la classe regroupante). Sa sémantique est celle d'une relation de composition (relation "has a"). Il existe deux types de regroupement – fort et faible – qui sont fonction du cycle de vie des parties et de leur conteneur. Dans la présente Recommandation, il n'est question que de regroupement fort, utilisé pour désigner une composition/décomposition.

Pour définir le **métamodèle** de l'eODL, on utilise la notation UML pour les concepts MOF (voir Figure 4). Les contraintes et les règles de grammaticalité qui font partie de ce **métamodèle** et qui sont essentielles pour sa sémantique sont énoncées sous forme de texte anglais. Les diagrammes UML représentés sur les figures des paragraphes qui suivent ne montrent que des

parties du **métamodèle** eODL complet. Pour pouvoir comprendre la sémantique, il faut lire le texte explicatif et notamment la description des contraintes qui figure dans ce texte. Les contraintes qui sont déjà décrites dans le cadre du **métamodèle** de l'IDL ne sont pas mentionnées à nouveau dans la présente Recommandation. On se reportera au [7].

L'Annexe D contient la référence à la représentation en XML du **métamodèle** complet, y compris toutes les contraintes.

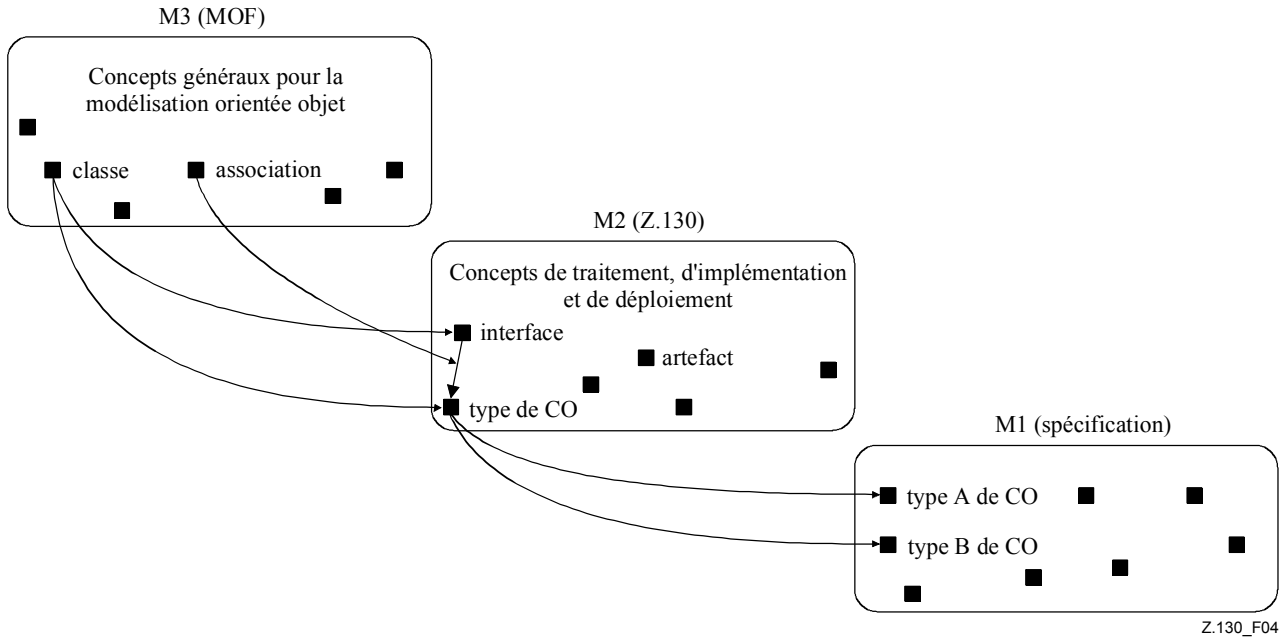


Figure 4/Z.130 – Concepts orientés objet

5.2 Dénomination et visibilité

Des règles de dénomination et de visibilité sont définies afin de pouvoir identifier de manière non ambiguë les éléments d'un modèle.

Le modèle dans son ensemble constitue un domaine de visibilité de noms. Chaque entité qui constitue un nouveau domaine de visibilité est une instance de la métaclasse abstraite *Container*. Les éléments contenus dans un domaine de visibilité sont des instances de la métaclasse abstraite *Contained*. Définir les métaclasses *Container* et *Contained* comme étant abstraites implique que toutes les instances sont des instances de métaclasses non abstraites dérivées. La relation *Container*-*Contained* est fréquemment utilisée dans le **métamodèle**.

Chaque entité dénommée (instance de la métaclasse *Contained*) possède un identificateur correspondant au nom. L'identificateur est un attribut de la métaclasse *Contained*. Les identificateurs de deux entités dénommées différentes qui appartiennent au même *Container* (definedIn pointe sur le même élément de modèle) doivent être différents.

Pour qu'un modèle puisse avoir des domaines de visibilité purs, on introduit la métaclasse *ModuleDef*. Celle-ci fait partie du **métamodèle** du CORBA-IDL sur lequel le **métamodèle** de la présente Recommandation est fondé. Il s'agit d'une métaclasse concrète, qui peut être instanciée. Elle n'a pas d'autres propriétés.

Chaque instance de *Container* constitue un espace de noms. La généralisation de *Contained* à *Container* exprime la capacité à imbriquer des domaines de visibilité de noms (voir Figure 5).

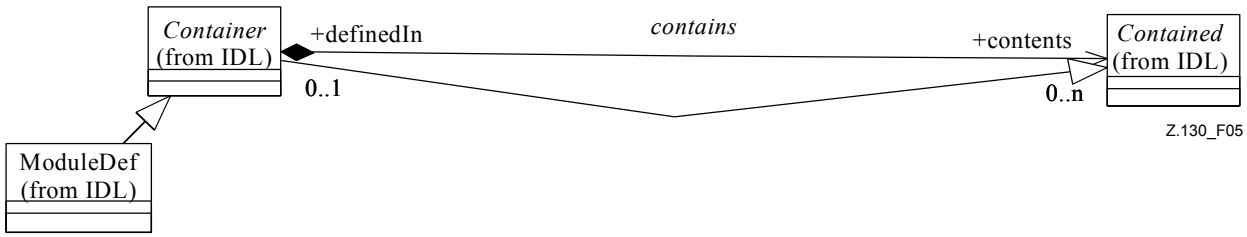


Figure 5/Z.130 – Dénomination et visibilité

5.3 Concepts de traitement

5.3.1 Concepts CORBA-IDL utilisés

Le **métamodèle** de l'eODL est fondé sur le **métamodèle** du CORBA-IDL, ce qui permet d'utiliser les concepts de modélisation suivants: **types de données**, **opérations**, **attributs**, **anomalies** et **types d'interface**.

Tous ces concepts de modélisation permettent de définir des modules de base pour des spécifications de traitement. Une spécification de traitement a notamment pour but de définir les signatures des **objets de traitement (CO)** au niveau de leurs **ports**. Comme la modélisation exposée dans la présente Recommandation est fondée sur les types, les **types de données** sont essentiels pour décrire ces signatures.

5.3.1.1 Types de données, types d'interface, opérations, attributs, anomalies

Dans les modèles, les **types de données** sont des instances de métaclasses dérivées de la métaclasse abstraite *IDLType*, d'où l'inclusion de la totalité du système de types de données du CORBA-IDL. L'utilisation de la métaclasse abstraite *IDLType* garantit la possibilité de changer de système de types de données afin de ne pas verrouiller la présente Recommandation. La Figure 6 montre un sous-ensemble du **métamodèle** du CORBA-IDL concernant les **types de données**. Ce **métamodèle** ainsi que sa description figurent dans le [7].

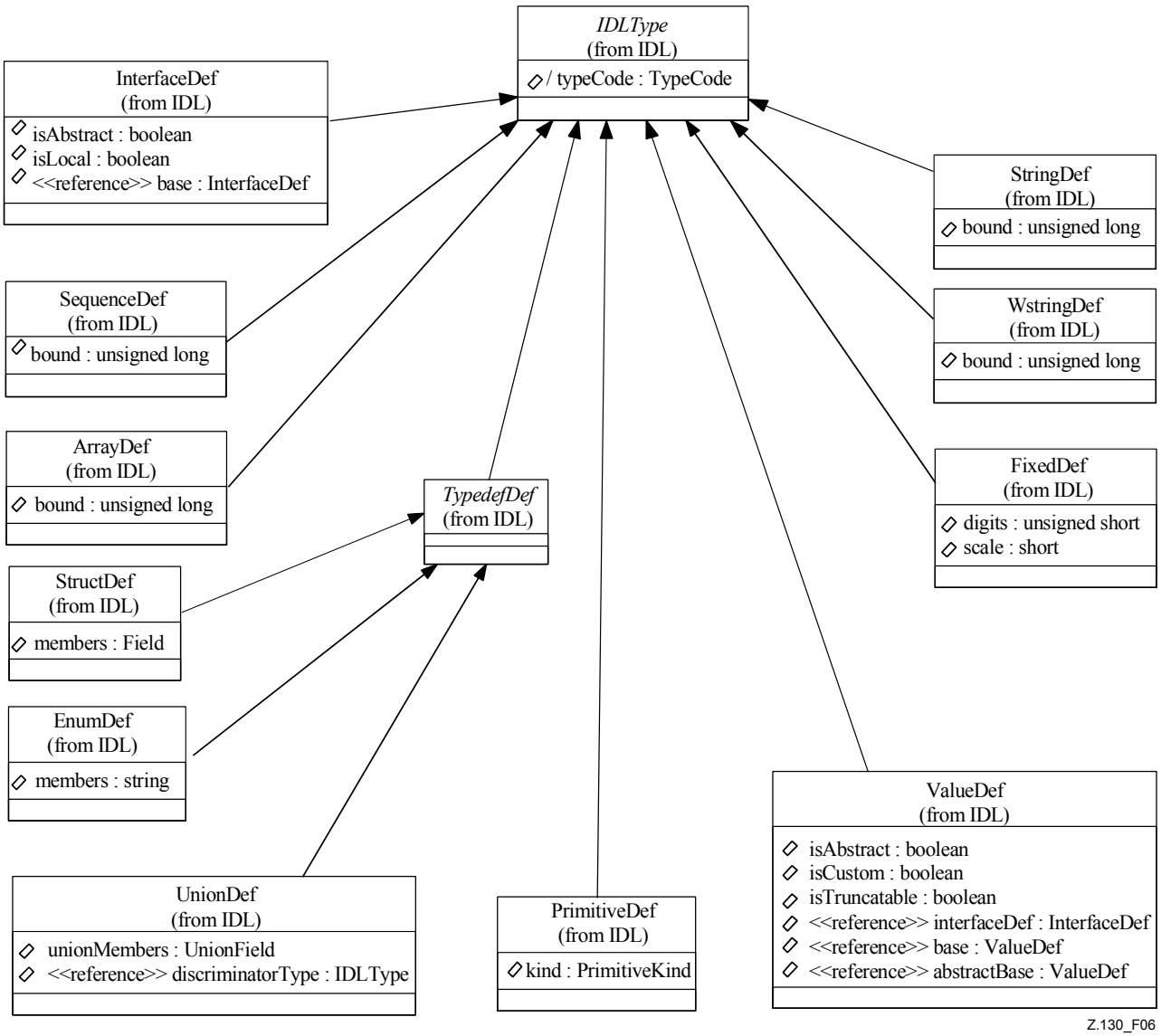


Figure 6/Z.130 – Types de données

Le système de types de données de CORBA contient tous les **types de données** primitifs couramment utilisés tels que *long, float, char, string*, etc. Il existe par ailleurs des concepts permettant de décrire des **types de données** structurés tels que *array, struct, sequence, union*, etc. Le **type de données** *value* est un type additionnel servant à passer des objets utilisant un objet par une sémantique de valeur. Tout comme les classes dans les langages de programmation, une instance de type *value* peut regrouper des **attributs** et des **opérations**.

Pour pouvoir utiliser des définitions de type de données autres que celles qui sont incluses dans CORBA, on introduit l'élément ExternType. Il s'agit d'une spécialisation du concept TypedefDef de CORBA. L'attribut *identifier* renvoie ici à une définition du type de données fournie en externe. Voir Figure 7.

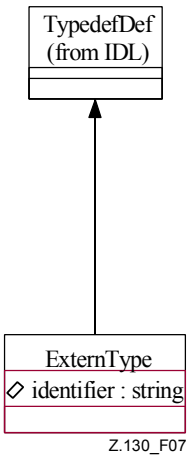


Figure 7/Z.130 – Type Extern

Les **attributs**, **opérations**, **anomalies** et **paramètres** sont des concepts nécessaires pour la définition d'interactions de type opération. Une **opération** définie dans un modèle contient une liste de paramètres, un type en vue d'une éventuelle valeur de retour et une liste d'échecs de terminaison modélisés par des **anomalies**. Les **anomalies** acheminent des informations et sont définies de la même façon que le **type de données** *StructDef*. Elles contiennent une liste de membres pour ces informations. Un **attribut** est une notation abrégée employée pour la modélisation des **opérations** servant à obtenir la valeur d'une variable dénommée d'un certain type (*get*) ou à fixer cette valeur (*set*). Par ailleurs, des **anomalies** peuvent être ajoutées aux **attributs**. Dans ce cas, on fait la distinction entre les **anomalies** susceptibles de se produire pendant une **opération** *set* pour cet attribut et celles qui se produisent pendant une **opération** *get*. Le **métamodèle** pour les **anomalies** est représenté sur la Figure 8.

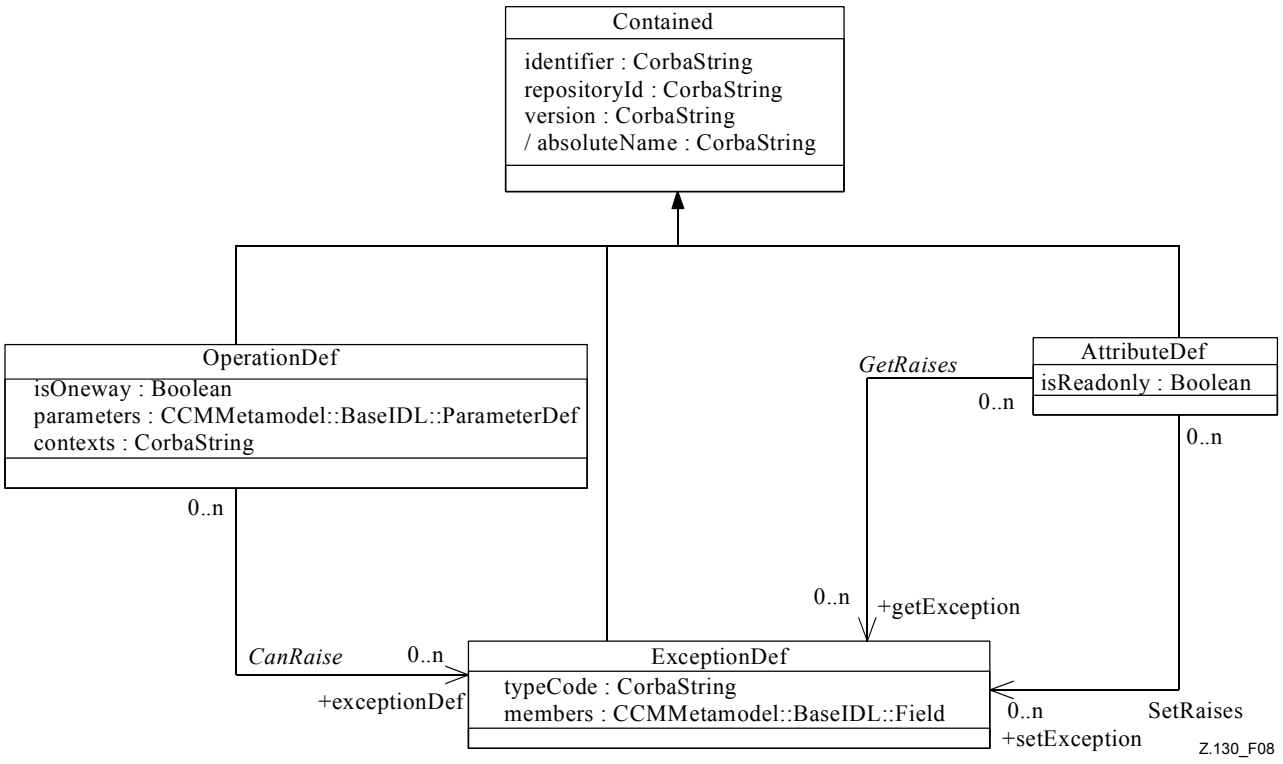


Figure 8/Z.130 – Anomalies

Les métaclasses *OperationDef*, *AttributeDef*, *ParameterDef* et *ExceptionDef* sont issues du **métamodèle** du CORBA-IDL et sont décrites en détail ici. Les **métamodèles** pour les **attributs** et les **opérations** sont représentés ci-dessous (Figures 9 et 10).

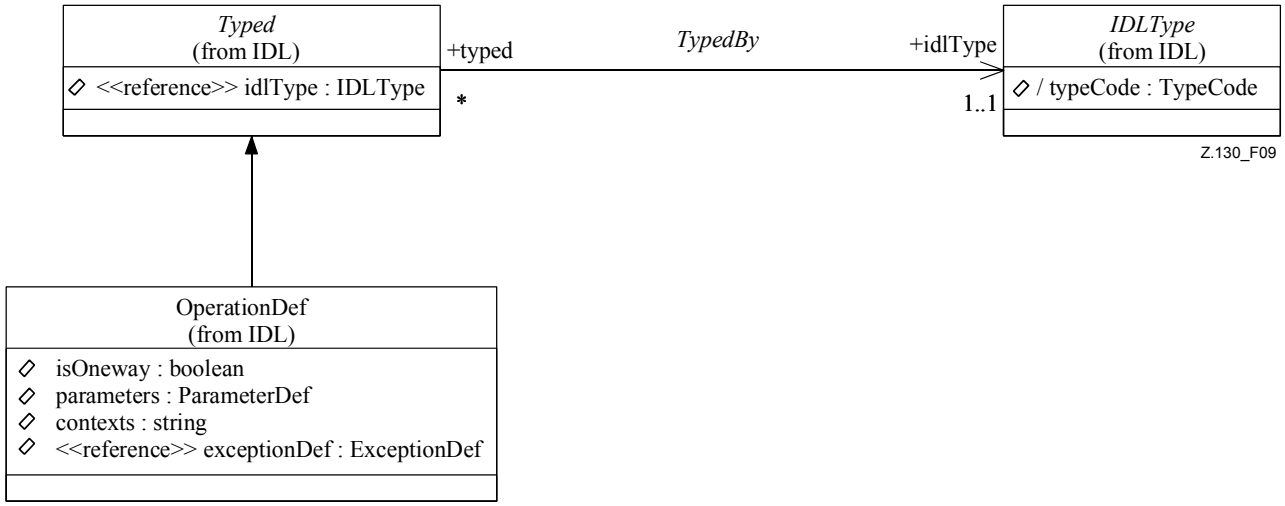


Figure 9/Z.130 – Opérations

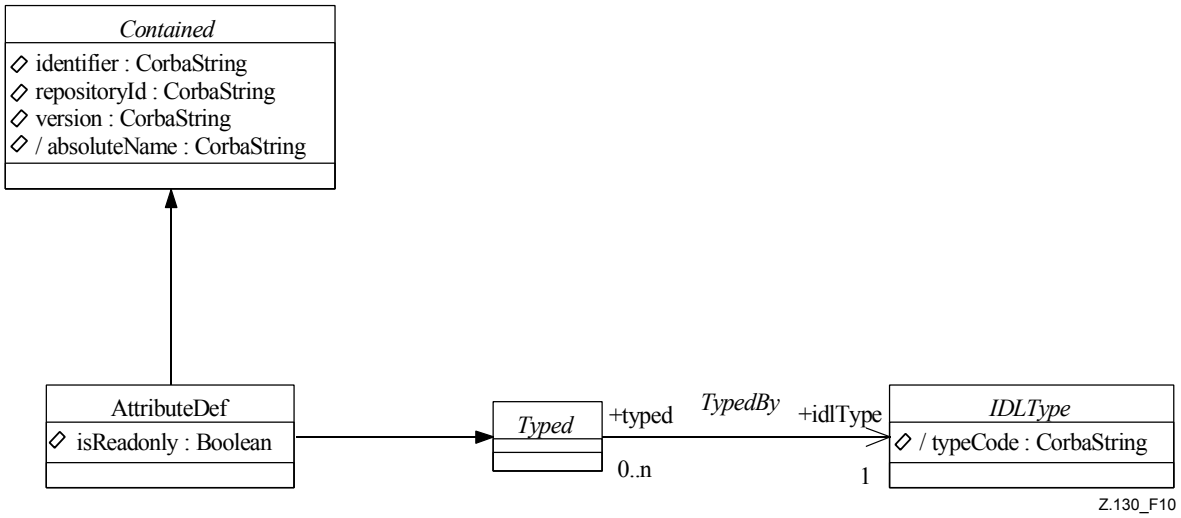


Figure 10/Z.130 – Attributs

Les **types d'interface** servent à définir des signatures pour les **interactions** possibles dans un système. Le concept de **type d'interface** est déjà connu en OMG IDL, où il est dénommé interface. En OMG IDL, les interfaces ne regroupent que des **éléments d'interaction** de type opération. Autrement dit, ce sont des conteneurs pour des **attributs** et des **opérations**. Voir Figure 11.

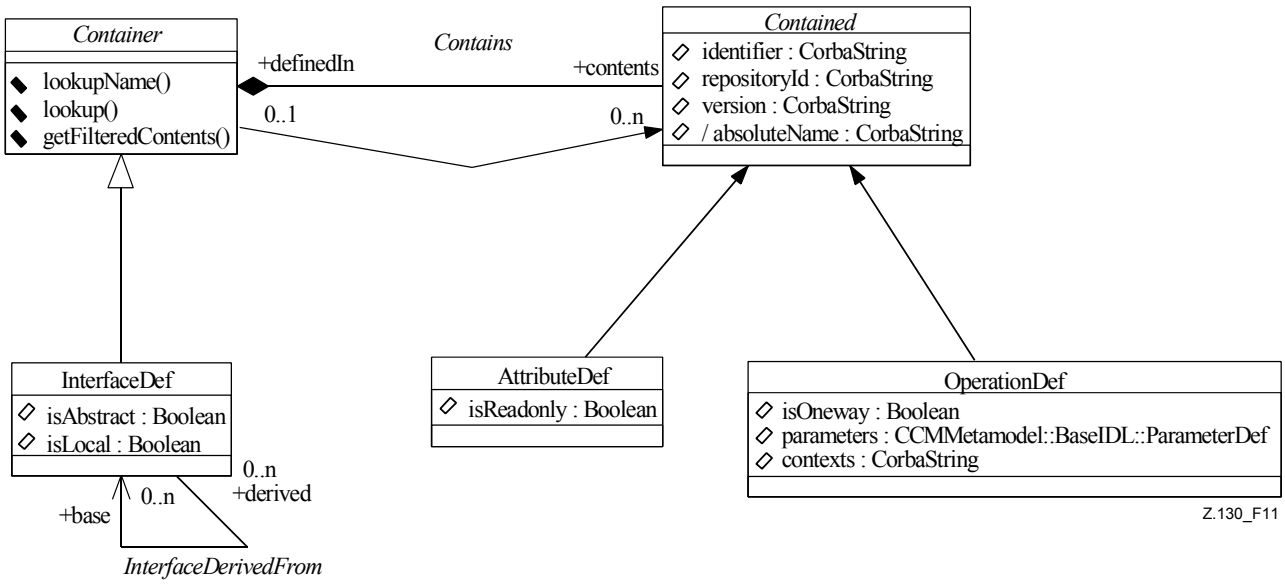


Figure 11/Z.130 – Interfaces d'opération

5.3.2 Signal et paramètres de signal

Métamodèle

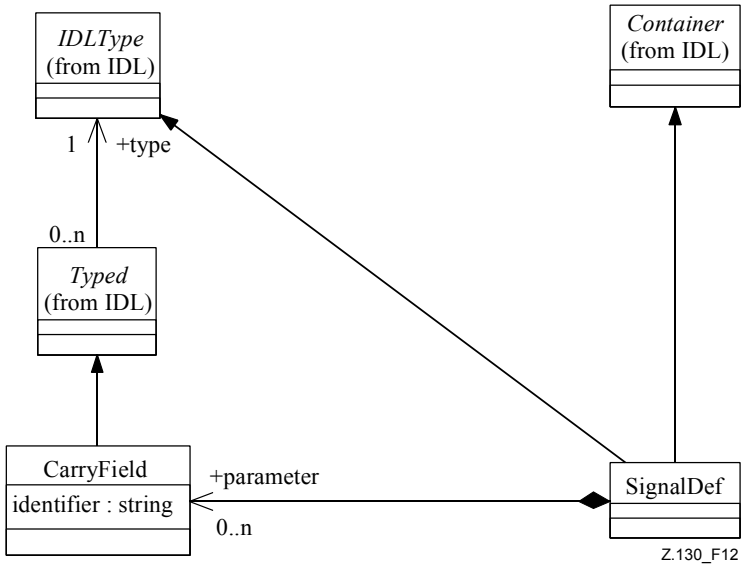


Figure 12/Z.130 – Signal et paramètre de signal

Sémantique

Pour améliorer les concepts de modélisation offerts par le CORBA-IDL et pour inclure d'autres types d'interaction dans la modélisation des systèmes répartis, d'autres concepts de modélisation sont nécessaires (voir Figure 12).

On utilise des **signaux** pour modéliser une interaction de type **signal**, c'est-à-dire l'échange asynchrone de messages découplés entre entités de système. Les **signaux** acheminent des informations. Ils sont modélisés sous forme d'instances de la métaclasse *SignalDef*. Les informations acheminées (appelées **paramètres de signal**) sont modélisées sous forme d'instances de *CarryField*, renvoyant chacune à une instance de *ValueDef*, qui est un **type de données** spécial

du CORBA-IDL. Chaque **paramètre de signal** peut être identifié par un nom dans le contexte de la définition du **signal**. Les noms doivent être uniques.

La définition d'un **signal** spécifie la structure et les propriétés des informations qui sont acheminées dans une interaction concrète de type signal entre entités de système. Elle n'est pas encore associée à un **type d'interface**. Les définitions de **signal** peuvent être réutilisées dans différentes définitions de **type d'interface**. Elles jouent un rôle identique à celui qui est joué par les **types de données** pour la définition des **opérations**. Ce sont des modules pour l'interaction de type signal.

Dans les modèles, on ne trouve des définitions de **signal** que dans les **espaces de noms** qui sont des modules ou qui constituent l'**espace de noms** global formé par la spécification proprement dite.

Les types IDL utilisés pour spécifier les paramètres de **signaux** doivent être des instances du type *value*.

5.3.3 Type de média, média, ensemble de médias

Métamodèle

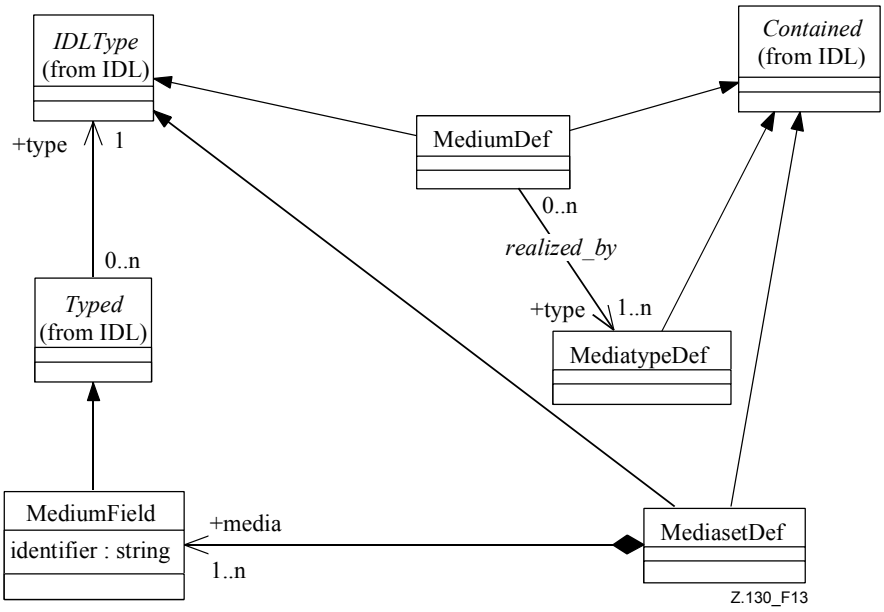


Figure 13/Z.130 – Média, type de média, ensemble de médias

Sémantique

En plus des interactions de type **opération** et **signal**, l'échange de **médias continus** constitue un type d'interaction important dans les systèmes répartis. Les concepts de modélisation pour ce type d'interaction doivent être définis. Pour cela, on procède, dans la présente Recommandation, exactement de la même façon que pour les interactions de type **opération** et **signal**. Il faut d'abord définir les modules de base pour les **éléments d'interaction**. Ce sont **média**, **ensemble de médias** et **type de média** (voir Figure 13).

Le concept d'ensemble de **médias** sert à modéliser l'interaction de type **médias continus**. Dans le **métamodèle**, ce concept est défini au moyen de la classe *MediasetDef*. Des instances de cette classe regroupent des instances de la classe *MediumDef* sur une liste dénommée sur laquelle chaque élément a le type *MediumField*. Le concept de **média** sert à modéliser un seul flux de données atomique entre deux entités. Un **média** a la signification d'informations multimédias (films ou séquences audio, par exemple). L'échange nécessite la présence de formats de codage, décodage et transmission, modélisés par des instances de la classe *MediatypeDef*. Un **média** peut être réalisé par un ou plusieurs **types de médias**. Les **ensembles de médias** sont nécessaires pour modéliser

l'échange simultané de deux médias différents (comme les données vidéo et audio d'un film) pour lesquels les interactions doivent avoir les mêmes propriétés.

Dans les modèles, on ne trouve des définitions de médias, **ensembles de médias** et **types de média** que dans les espaces de noms qui sont des modules ou qui constituent l'espace de noms global formé par la spécification proprement dite.

5.3.4 Consommer et produire

Métamodèle

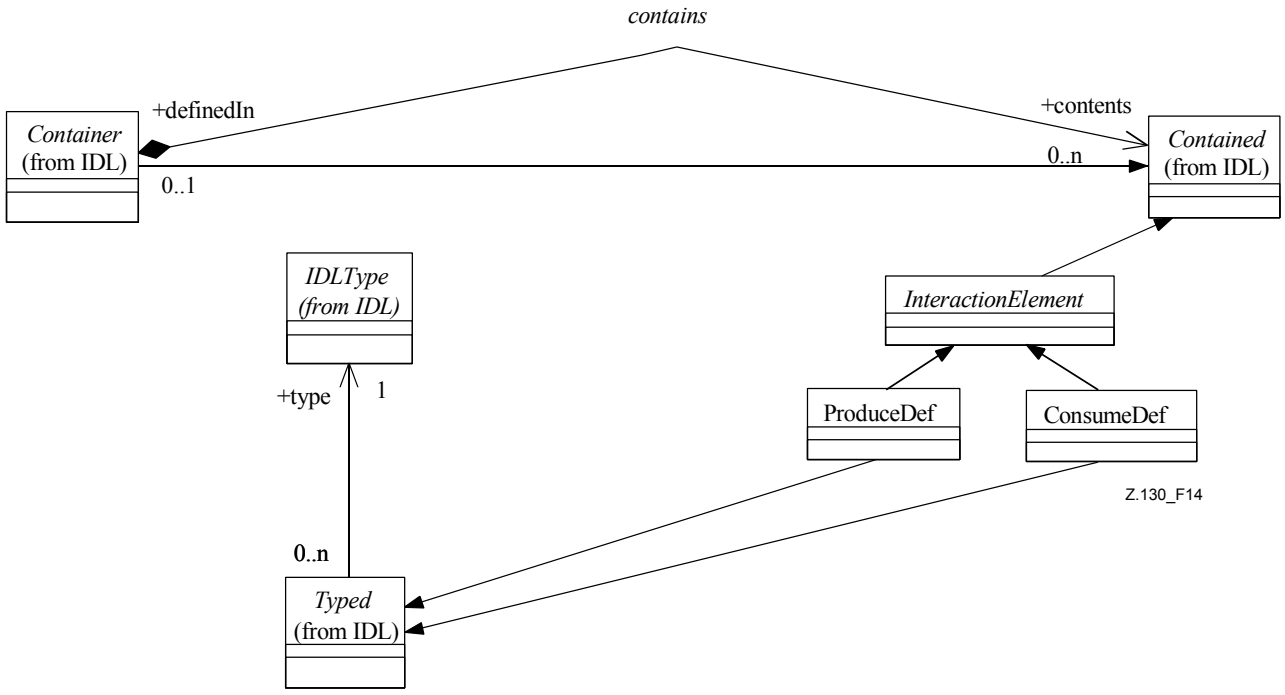


Figure 14/Z.130 – Consommer et produire

Sémantique

Des **signaux** seront échangés entre entités fonctionnelles au **stade de l'exécution** via leurs **interfaces**. C'est pour cette raison que les **types d'interface** offrent les signatures nécessaires. Dans le cas de la communication d'un **signal**, la signature contient le type (définition du **signal**), un nom pour l'**élément d'interaction** et une indication précisant si le **signal** est produit ou consommé via cette interface. Ainsi, les concepts **consommer** et **produire** sont des **éléments d'interaction** et sont utilisés pour modéliser la consommation ou la production de **signaux** (voir Figure 14). Ce sont les éléments de l'interaction de type **signal** au même titre que les **opérations** et les **attributs** sont les éléments de l'interaction de **type opération**. Comme tout **élément d'interaction**, les éléments **consommer** et **produire** sont des éléments identifiables. Ils sont inclus dans le **métamodèle** avec la définition des métaclasses *ConsumeDef* et *ProduceDef*, qui héritent de la métaclasse abstraite *InteractionElement*, qui hérite à son tour de *Contained*. *ConsumeDef* et *ProduceDef* sont des sous-classes de Typed. Cet héritage sert à établir la relation avec *SignalDef*, adressée par les éléments **produire** et **consommer**.

Il est à noter que *SignalDef* est une sous-classe de *IDLType*. L'instance de *IDLType* associée à une définition **produire** ou **consommer** doit nécessairement être un **signal**. Le concept de **signal** est défini comme étant une spécialisation de *IDLType* à cette fin.

5.3.5 Puits et source

Métamodèle

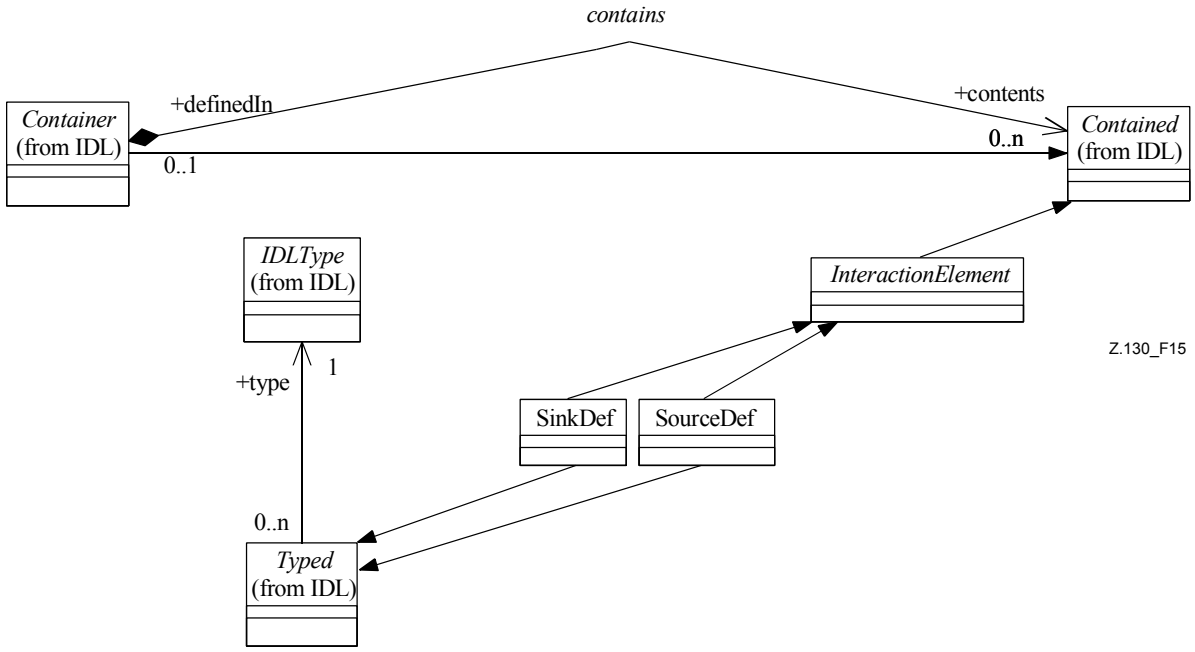


Figure 15/Z.130 – Puits et source

Sémantique

Pour compléter la liste des **éléments d'interaction** possibles qui sont utilisés pour spécifier des signatures de **type d'interface**, il faut définir des **éléments d'interaction** pour les interactions de type **média continu**. De manière analogue aux interactions de type **signal**, ces interactions sont caractérisées par les informations qui sont échangées en cas d'interaction au **stade de l'exécution** (**ensemble de médias**), un identificateur et le sens de la communication, c'est-à-dire si l'interface est un puits ou une source par rapport à l'interaction. Les concepts **puits** et **source** sont donc des **éléments d'interaction** servant à modéliser la consommation ou la production d'**ensembles de médias** (voir Figure 15). Ce sont les éléments de l'interaction de type **média continu** au même titre que les **opérations** et les **attributs** sont les éléments de l'interaction de type **opération**. Comme tout **élément d'interaction**, le **puits** et la **source** sont des éléments identifiables. Ils sont inclus dans le **métamodèle** avec la définition des métaclasses *SinkDef* et *SourceDef*, qui héritent de la métaclasse abstraite *InteractionElement*, qui hérite à son tour de *Contained*. *SinkDef* et *SourceDef* sont des sous-classes de *Typed*. Cet héritage sert à établir la relation avec *MediasetDef*, adressée par les éléments **puits** et **source**.

Il est à noter que *MediasetDef* est une sous-classe de *IDLType*. Le seul *IDLType* concret qu'il est permis d'associer à un **puits** ou à une **source** est **type de média**.

5.3.6 Type d'interface

Métamodèle

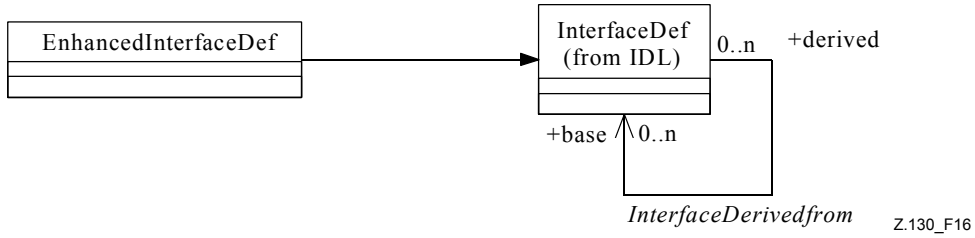


Figure 16/Z.130 – Type d'interface

Sémantique

Le concept **type d'interface** sert à spécifier un sous-ensemble d'**interactions** potentielles de **CO** de différents **types de CO**. Les **types d'interface** regroupent les **éléments d'interaction** des types d'interaction **opération**, **signal** et **média continu**. La sémantique d'un **type d'interface** est donc étendue par rapport au modèle RM-ODP: les **types d'interface** fournissent un contexte commun pour des **éléments d'interaction** de *différents* types d'interaction. Les clients qui ont une référence à une telle **interface** ont la possibilité d'utiliser tous les **éléments d'interaction** quel que soit le type d'interaction utilisé. Il appartient à l'environnement d'exécution de traiter cet aspect.

Dans le **métamodèle**, le **type d'interface** est une instance de la classe *EnhancedInterfaceDef* (voir Figure 16). Comme le **métamodèle** de l'IDL contient déjà une signification du regroupement des **éléments d'interaction opération**, la classe *EnhancedInterfaceDef* hérite de *InterfaceDef*. Il s'ensuit que les règles d'héritage et les contraintes de *InterfaceDef* s'appliquent aussi à *EnhancedInterfaceDef*.

Par ailleurs, les **types d'interface** sont des conteneurs pour les **éléments d'interaction produire**, **consommer**, **puits** et **source**.

5.3.7 Types de CO, relations prendre en charge et requérir

Métamodèle

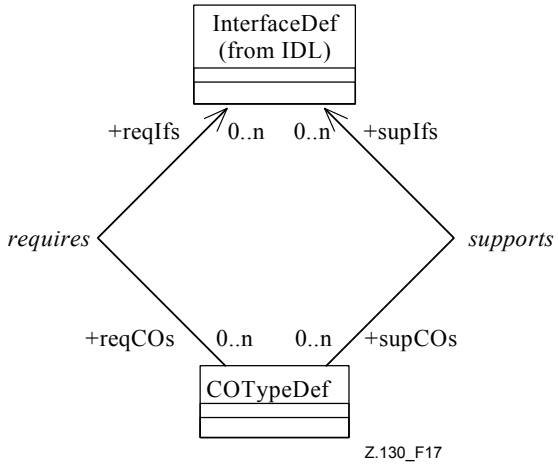


Figure 17/Z.130 – Types de CO, prendre en charge et requérir

Sémantique

Le concept de **type de CO** sert à spécifier la décomposition fonctionnelle d'un système. Les instances d'un **type de CO** (des **CO**) sont des entités autonomes qui interagissent; elles encapsulent

état et comportement. Les CO interagissent avec leur environnement via des **interfaces** bien définies, spécifiées au moyen du concept de **type d'interface** décrit plus haut.

Un **type de CO** peut prendre en charge ou requérir un **type d'interface**. Prendre en charge un **type d'interface** signifie que les **CO** de ce **type de CO** offrent des **interfaces** de ce **type d'interface**. Requérir un **type d'interface** signifie que les **CO** de ce **type de CO** utilisent des **interfaces** de ce **type d'interface**. Un **type de CO** est une instance de la classe *COTypeDef* du **métamodèle**. Les étiquettes *supports* (prendre en charge) et *requires* (requérir) identifient les associations entre *COTypeDef* et *InterfaceDef* (voir Figure 17).

Pour pouvoir accéder aux **CO** au **stade de l'exécution**, *COTypeDef* est dérivé de *InterfaceDef* (voir Figure 17). Ainsi, des instances peuvent être configurées au moyen d'attributs qui sont définis par ce **type de CO**. Il est important de noter qu'un **type de CO** ne peut contenir que des **éléments d'interaction** du type attribut. Aucun autre **élément d'interaction** n'est permis. Par ailleurs, les relations d'héritage entre **types de CO** et **types d'interface** ne peuvent pas être mélangées: les **types de CO** ne peuvent hériter que de **types de CO** et les **types d'interface** que de **types d'interface**.

5.3.8 Définition de port provisionné et de port utilisé

Métamodèle

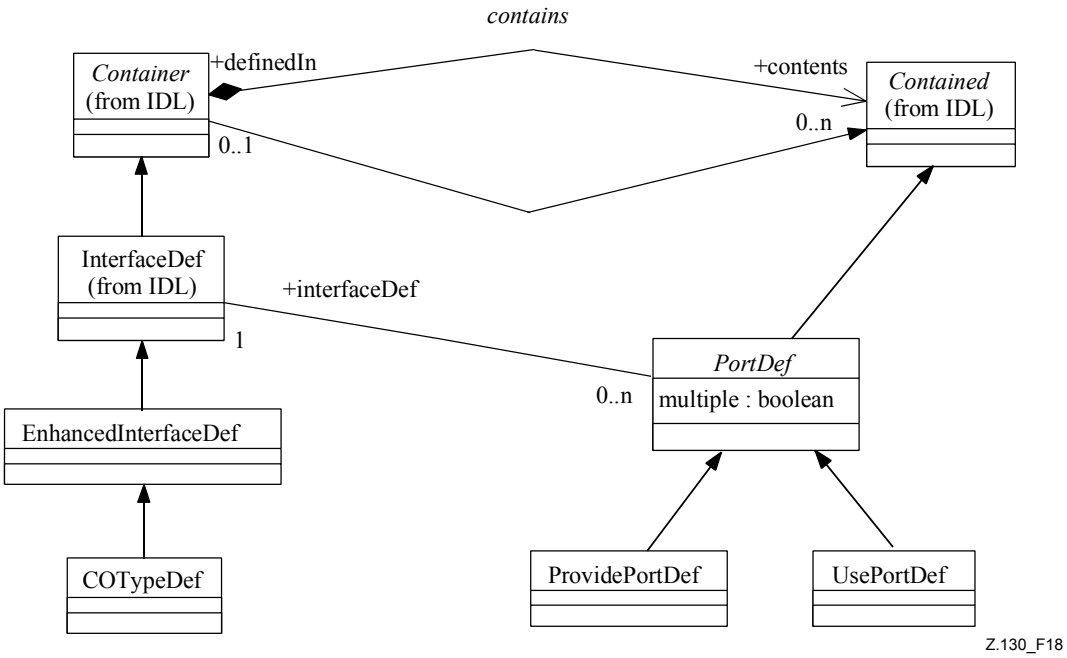


Figure 18/Z.130 – Port provisionné et port utilisé

Sémantique

Les **CO** sont les entités fonctionnelles d'un système réparti qui est spécifié selon la présente Recommandation. Ils communiquent via leurs **interfaces** prises en charge et requises. Toutefois, la configuration des systèmes répartis pose toujours problème, notamment en ce qui concerne la façon d'obtenir et d'échanger des **références d'interface**, constituant un préalable à toute **interaction**. C'est pourquoi la présente Recommandation introduit le concept de **port** comme étant un point d'interaction dénommé, au niveau duquel soit la référence d'une **interface** prise en charge par un **CO** peut être obtenue soit la référence d'une **interface** utilisée peut être enregistrée au **stade de l'exécution**.

Les concepts de **port provisionné** et de **port utilisé** servent à modéliser les **ports** d'un **type de CO**, qui sont utilisés par l'environnement pour obtenir une référence à une **interface** (**port provisionné**)

ou pour stocker une référence à une **interface** (**port utilisé**) sur la base d'un nom. Avec les concepts **prendre en charge** et **requérir**, on ne peut exprimer que l'offre ou l'utilisation potentielle de **types d'interface** dans le contexte d'un **type de CO**, mais pas les mécanismes concrets permettant à l'environnement d'un **CO** d'accéder à ces contextes d'interaction. Les concepts **port provisionné** et **port utilisé** sont définis comme étant des instances des classes *ProvidePortDef* et *UsePortDef*. Les deux classes héritent de la classe abstraite *PortDef*. La classe *PortDef* hérite de *Contained*, ce qui signifie qu'une instance de *COTypeDef* peut contenir des définitions de **port provisionné** et de **port utilisé**. La définition d'un **port provisionné** ou d'un **port utilisé** est toujours associée à une définition d'**interface** (voir Figure 18).

Les définitions de **port** ne sont autorisées que dans le cadre de définitions de **type de CO**. Une **interface** pour laquelle un **port provisionné** est défini est automatiquement une **interface** prise en charge. Une **interface** pour laquelle un **port utilisé** est défini est automatiquement une **interface** requise.

5.4 Concepts d'implémentation

5.4.1 Artefact et schéma d'instanciation

Métamodèle

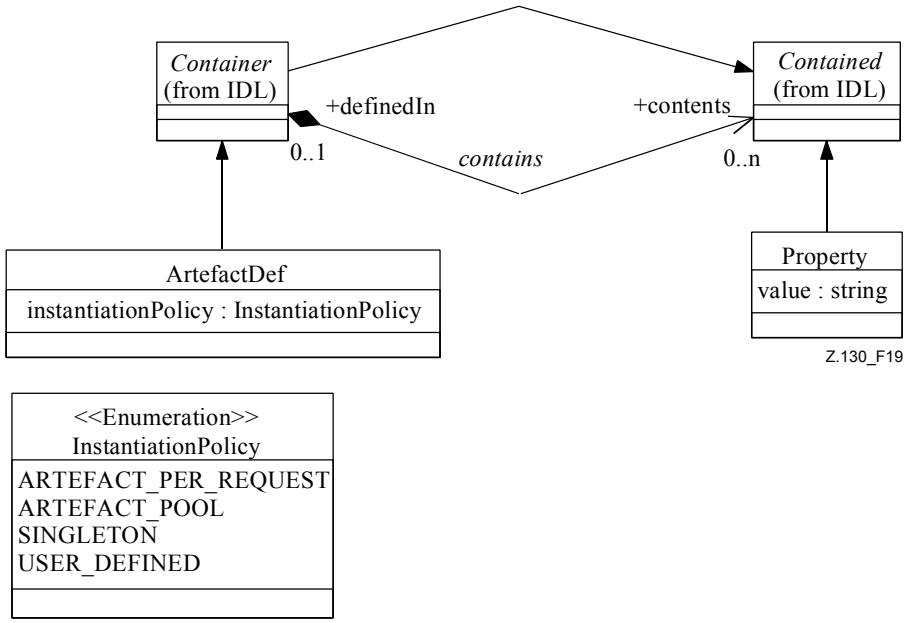


Figure 19/Z.130 – Artefact et schéma d'instanciation

Sémantique

Le concept d'**artefact** sert à décrire un contexte de langage de programmation (par exemple une classe d'un langage de programmation orienté objet) dans un modèle. Les instances du concept **artefact** réalisent le comportement de **CO**. Elles constituent donc la logique commerciale des types de CO. Les relations entre les **artefacts** et les parties comportementales des **CO** sont définies par des associations entre **artefacts** et **éléments d'interaction** de **types d'interface**. Les contextes de langage de programmation, qui sont modélisés par des instances du concept **artefact** seront instanciés au **stade de l'exécution** pour traiter, par exemple, des invocations d'**opération**, des données d'entrée de **signal** ou des données de **média continu**. Les politiques à utiliser pour l'instanciation sont spécifiées par des instances du concept de schéma d'instanciation. Les schémas autorisés sont donnés sur la Figure 19. D'autres schémas pourront être ajoutés en cas de besoin. La séparation des concepts **artefact** et **type de CO** offre une grande souplesse pour la conception d'une application répartie:

- le point de vue externe (la manière dont l'environnement interagit avec un **CO**) est séparé du point de vue interne (la manière dont le comportement d'un **CO** est assuré);
- on peut utiliser des arbres d'héritage différents pour les points de vue externe et interne;
- les définitions existantes de comportement et d'**interface** peuvent être réutilisées et ce, indépendamment l'une de l'autre.

Le concept d'**artefact** est exprimé dans le **métamodèle** par une instance de la classe *ArtefactDef*. Le concept de schéma d'instanciation est modélisé sous la forme d'un attribut de cette classe de type énumération, les éléments énumérés étant tels qu'indiqués sur la Figure 19.

5.4.2 Relation implémenter

Métamodèle

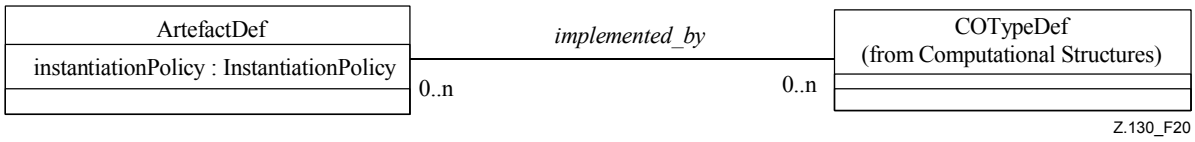


Figure 20/Z.130 – Relation implémenter

Sémantique

Les instances du concept **artefact** décrivent la réalisation du comportement attendu d'**éléments d'interaction** de **types d'interface** qui sont pris en charge ou requis par des **types de CO**. Comme décrit plus haut, le point de vue externe et le point de vue interne d'un **CO** sont complètement séparés l'un de l'autre. Toutefois, il faut décrire la relation entre eux afin de pouvoir choisir le comportement correct lorsqu'un **CO** intervient dans une certaine interaction. Le modèle indique donc quel ensemble d'**artefacts** offre le comportement à utiliser pour les **CO** d'un certain **type de CO**. Cette relation est définie par une association *implemented_by* entre *COTypeDef* et *ArtefactDef* dans le **métamodèle** (voir Figure 20).

5.4.3 Elément d'implémentation

Métamodèle

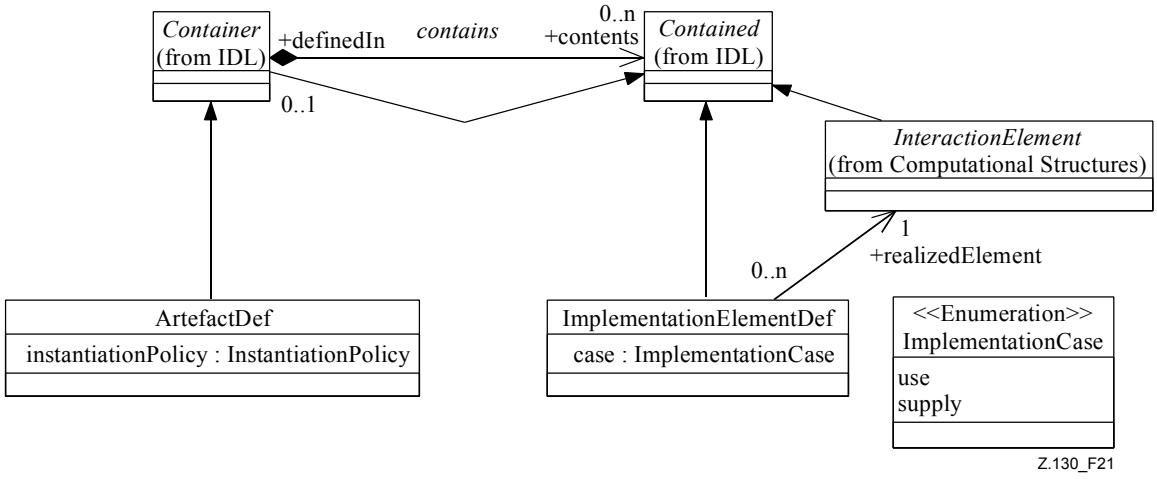


Figure 21/Z.130 – Elément d'implémentation

Sémantique

Dans le contexte d'une instance d'**artefact**, le concept d'**élément d'implémentation** sert à préciser qu'une partie comportementale d'un **artefact** (par exemple une méthode d'une classe, qui est modélisée par une instance du concept **artefact**) réalise un **élément d'interaction** particulier d'un **type d'interface**. Ce concept est nécessaire pour fournir davantage de détails concernant la relation *implemented_by* décrite ci-dessus. *implemented_by* spécifie qu'un **artefact** contribue au comportement d'un **CO** sans préciser quelle est la partie de l'**artefact** qui est responsable de telle ou telle partie du comportement du **CO**. Ces détails sont fournis avec les **éléments d'implémentation** de l'**artefact**. Ces informations sont nécessaires pour associer le comportement à une interaction au **moment de l'exécution**.

Le concept d'**élément d'implémentation** est spécifié sous la forme d'une instance de la classe *ImplementationElementDef*. Cette classe hérite de *Contained*, des instances de *ImplementationElementDef* peuvent être contenues dans des instances d'**artefacts** (voir Figure 21). L'attribut *case* définit le sens d'*implémentation* d'un **élément d'implémentation**: soit l'utilisation soit l'offre du comportement peut être **réalisée** par un **élément d'implémentation** par rapport à un certain **élément d'interaction**.

5.5 Concepts de déploiement

5.5.1 Composant logiciel et dépendance de composant

Métamodèle

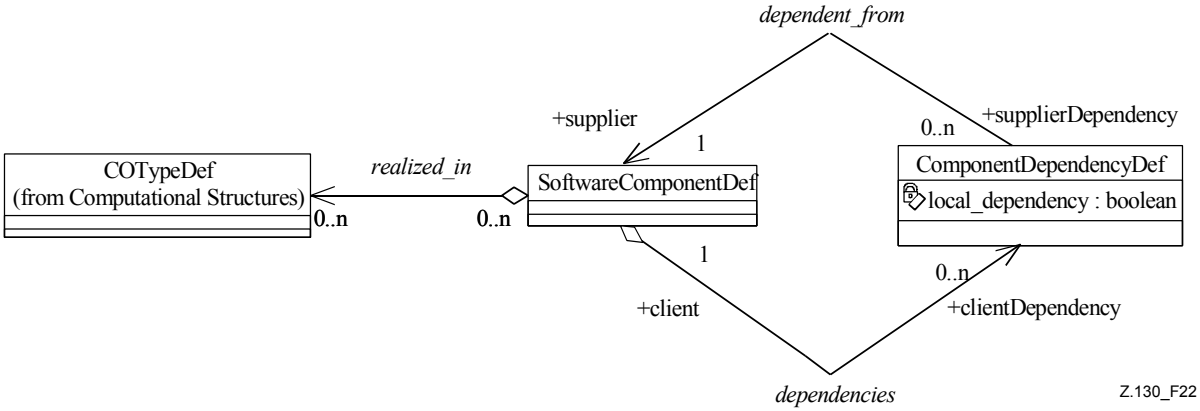


Figure 22/Z.130 – Composant logiciel et dépendance de composant

Sémantique

Le concept de **composant logiciel** représente le logiciel réel dans le modèle de conception. Il identifie une entité de **déploiement** et permet une description plus détaillée au moyen de propriétés. Un **composant logiciel** peut – mais ce n'est pas une obligation – **réaliser** un nombre arbitraire de **types de CO**. Il contient donc des séquences d'instructions qui, lorsqu'elles sont exécutées au niveau d'un nœud, instancient des **CO**, autrement dit fournissent le comportement, l'état et l'identité de **CO**. Dans le **métamodèle**, le concept est introduit par la métaclasse *SoftwareComponentDef*. Le concept de relation réaliser permet d'indiquer quels sont les **types de CO** qui sont **réalisés** par un certain **composant logiciel**. Il est introduit dans le **métamodèle** par une association entre les métaclasses *COTypeDef* et *SoftwareComponentDef* (voir Figure 22).

Certains **composants logiciels** peuvent avoir besoin d'autres **composants logiciels** pour pouvoir être exécutés correctement. Pour cela, on définit la métaclasse *SoftwareDependencyDef* dans le modèle. Cette métaclasse contient l'attribut *local_dependency* qui indique si le **composant logiciel** requis doit être disponible localement. Une *SoftwareComponentDef* peut contenir un nombre

arbitraire de *SoftwareDependencyDef,* chaque *SoftwareDependencyDef* étant associée à une autre *SoftwareComponentDef* indiquant le logiciel requis.

5.5.2 Assemblage et configuration initiale

Métamodèle

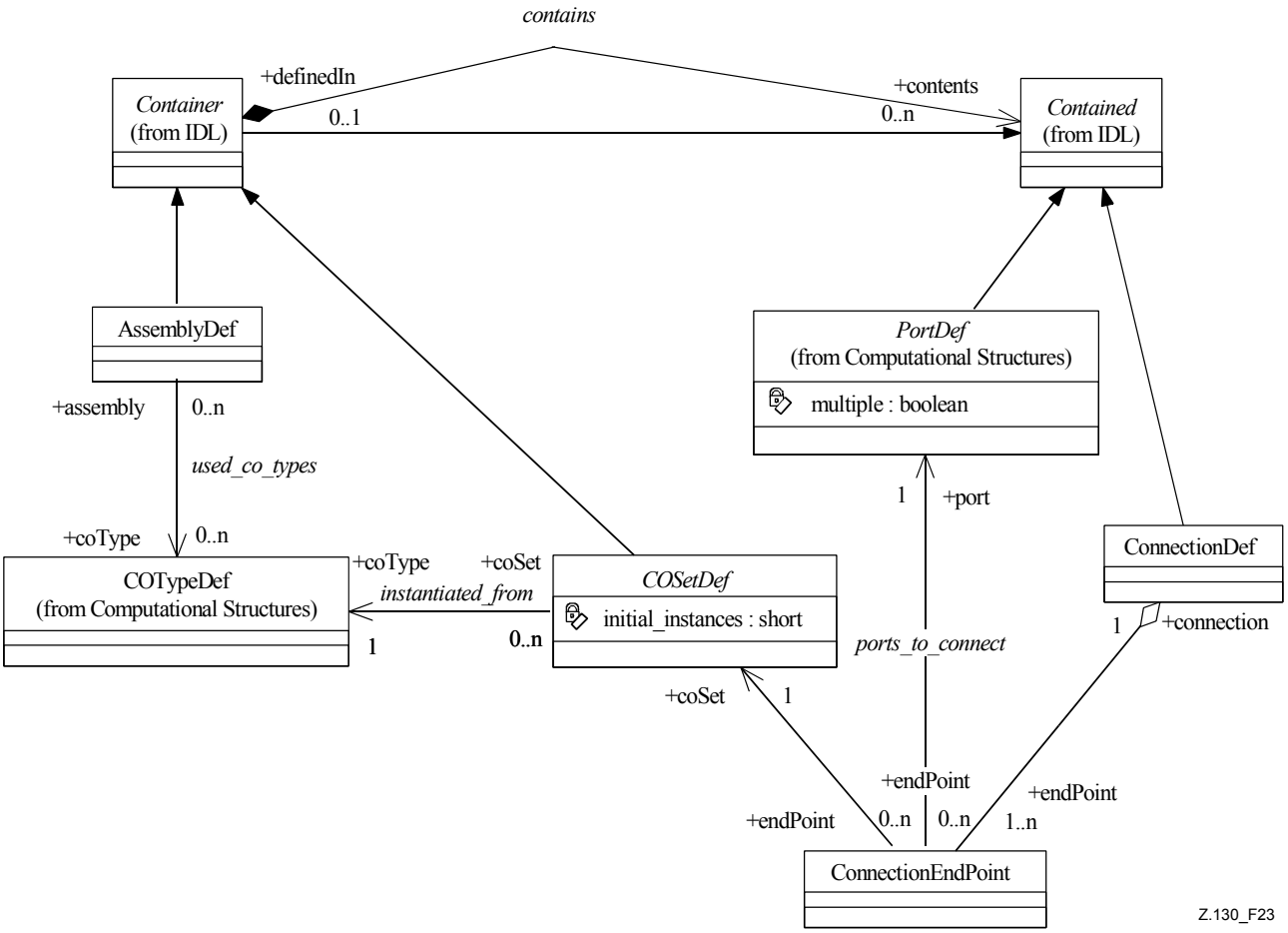


Figure 23/Z.130 – Assemblage et configuration initiale

Sémantique

Le concept d'**assemblage** sert à modéliser des systèmes logiciels en spécifiant les **types de CO** qui interviennent dans un système donné ainsi que la **configuration initiale** de ce système. La **configuration initiale** est la configuration qui est établie au début de l'exécution du système logiciel; elle est constituée des **CO initiaux** et de leurs **connexions initiales**. Dans le **métamodèle**, le concept d'**assemblage** est modélisé par la métaclasse *AssemblyDef*. Les **types de CO** sont associés grâce à l'introduction d'une association entre les métaclasses *AssemblyDef* et *COTypeDef* (voir Figure 23).

La métaclasse *COSetDef* contenue dans le **métamodèle** permet de modéliser les **CO initiaux**. Une *COSetDef* définit la création d'un nombre arbitraire d'instances du **type de CO** associé. Le nombre est déterminé par l'attribut *initial_instances*. Une *COSetDef* est contenue dans une *AssemblyDef*.

La métaclasse *ConnectionDef* contenue dans le **métamodèle** permet de modéliser les **connexions initiales**. Une **connexion** est établie entre des ports des **CO** participants par l'échange de **références d'interface** des **CO**. Ces références sont obtenues à partir d'un **CO** dont le **type de CO** a une définition de **port provisionné** et sont transférées à un **CO** dont le type a une définition de **port utilisé**. Dans le **métamodèle**, une *ConnectionDef* est constituée d'un ensemble de

ConnectionEndPoint. Un *ConnectionEndPoint* est associé à une *PortDef* d'une *COTypeDef* et à une *COSetDef*. Chaque **CO** d'une *COSetDef* associée à un *ConnectionEndPoint* est connecté à chaque **CO** de chaque *COSetDef* associée aux autres *ConnectionEndPoints* regroupés dans la même *ConnectionDef*.

5.5.3 Propriétés et contraintes

Métamodèle

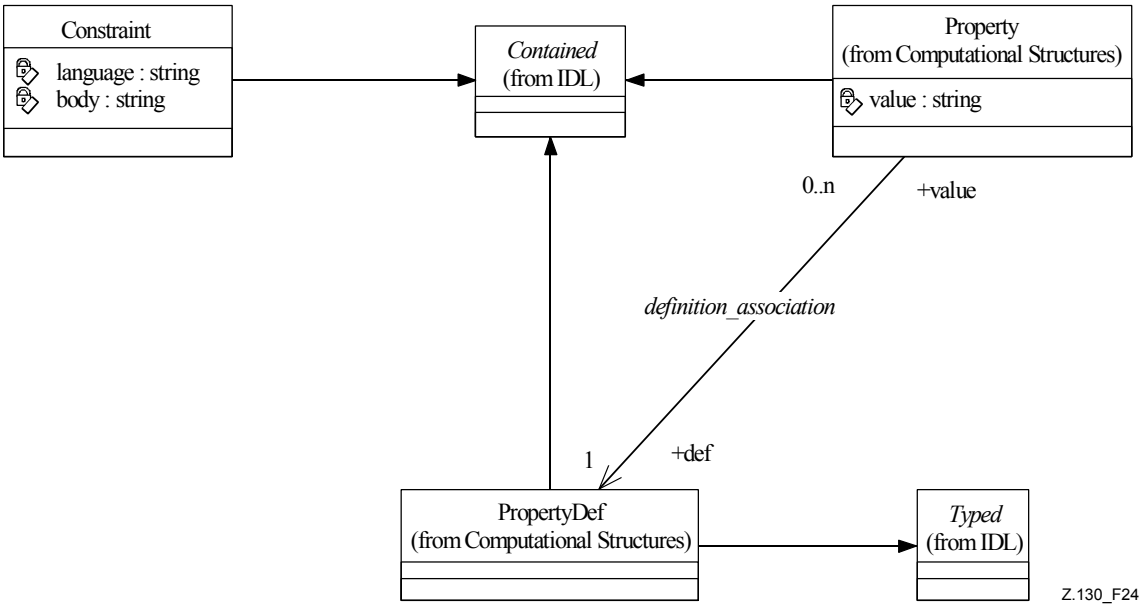


Figure 24/Z.130 – Propriétés et contraintes

Sémantique

Le concept de **propriétés**, qui peut être attaché à certains éléments du modèle, est représenté dans le **métamodèle** par les métaclasses *PropertyDef* et *Property* (voir Figure 24). Dans le **métamodèle**, une distinction est faite entre la définition d'une propriété et sa valeur. Une *Property* possède une valeur contenue dans l'attribut *value* de type chaîne tandis qu'une *PropertyDef* possède la spécification de **type de données** pour la valeur, telle qu'elle est héritée de la métaclasse *Typed*. Par exemple, un **type de CO** peut définir les propriétés nécessaires pour sa configuration, tandis qu'un **CO** peut définir les valeurs de propriété appropriées.

Le concept de **contrainte** est représenté dans le **métamodèle** par la métaclasse *Constraint*. Il a deux attributs. L'attribut *language* détermine le langage dans lequel la contrainte est écrite et doit être utilisé pour l'évaluation et l'attribut *body* contient la représentation effective de la contrainte sous forme de chaîne. Le choix du langage pour les contraintes est laissé à l'utilisateur, il sort du cadre de la présente Recommandation. Ainsi il est possible de choisir n'importe quel langage approprié et la sémantique des contraintes n'est pas définie ici, elle est laissée aux outils de traitement. Un **CO** peut définir un ensemble de contraintes pour exprimer les combinaisons permises de valeurs de propriété. Un **assemblage** peut définir des contraintes d'emplacement commun pour les composants en cours d'exécution. Les attributs ou définitions de propriété des références sont qualifiés par leurs noms.

5.6 Concepts de l'environnement cible

Pour spécifier la répartition et le **déploiement**, on modélise des environnements cibles réels et des environnements cibles logiques et on projette les entités logiques sur des nœuds réels des

environnements. Dans certains cas, un outil peut extraire automatiquement le modèle de l'environnement cible réel.

5.6.1 Environnement cible, nœud et liaison entre nœuds

Métamodèle

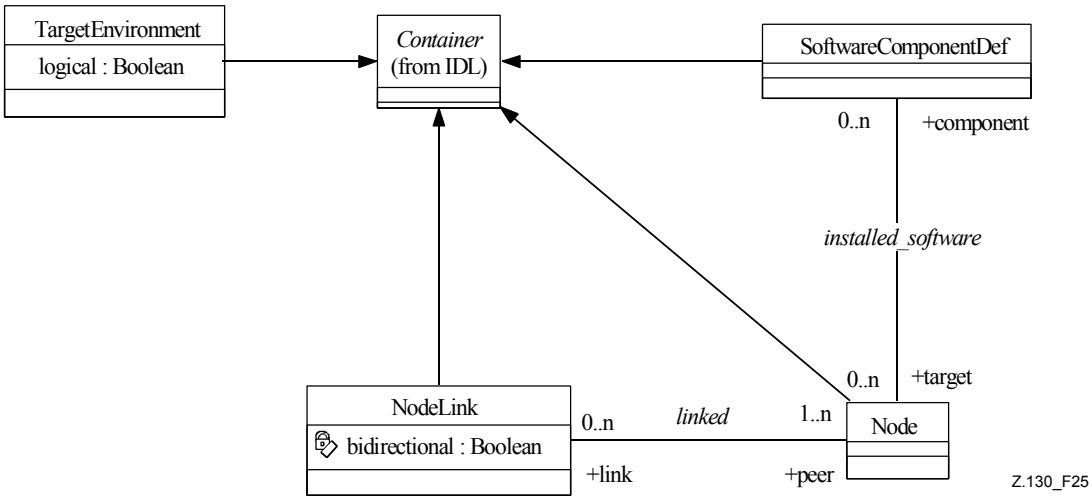


Figure 25/Z.130 – Environnement cible, nœud et liaison entre nœuds

Sémantique

Un environnement cible modélise un environnement d'exécution réparti physique, par exemple un réseau de télécommunication constitué de nœuds et de liaisons entre les nœuds. Dans le **métamodèle**, la métaclasse *TargetEnvironment* est un conteneur pour *Node* et *NodeLink* (voir Figure 25). L'attribut *logical* de *TargetEnvironment* sert à indiquer si le modèle correspond à un environnement existant ou à un environnement potentiel.

La métaclasse *Node* représente le concept de nœud, c'est-à-dire un élément de l'environnement cible qui est doté au moins d'un ou de plusieurs processeurs, d'une unité de mémoire et d'un système d'exploitation. Il est à noter qu'une machine physique donnée peut accueillir plusieurs *Node* logiques et non pas un seul. Un nœud comprend les logiciels installés (**composants logiciels**, compileurs, interpréteurs, etc.) et les matériels installés (représentés sous forme de pilotes). Les propriétés telles que le système d'exploitation ou le processeur sont décrites sous la forme des propriétés prédéfinies énumérées ci-dessous.

Le concept de liaisons entre nœuds est représenté par la métaclasse *NodeLink* comme étant une liaison physique entre deux nœuds ou plus (un bus partagé par exemple). L'attribut booléen *bidirectional* de *NodeLink* ne s'applique que dans le cas où la liaison entre nœuds est associée à deux nœuds exactement. Lorsque *bidirectional* est à faux, l'ordre des deux nœuds associés à la liaison entre nœuds est interprété comme suit: la première instance de *Node* correspond au nœud source, et la seconde au nœud de destination. L'association *linked* est donc ordonnée.

Node et *NodeLink* sont des conteneurs pour *PropertyDef* et *Property*. Autrement dit, des propriétés prédéfinies et des propriétés définies par l'utilisateur peuvent être attachées directement à des instances de nœud et à des instances de liaison entre nœuds.

5.6.1.1 Propriétés prédéfinies des nœuds et des liaisons entre nœuds

Node et *NodeLink* ont des propriétés prédéfinies (voir Tableau 1). A cette fin, les **types de données** implicites suivants sont introduits au moyen de l'IDL:

```
struct ProcessorType {
      string family;
      string type;
      integer frequency;
}
struct OSType {
      string name;
      string version;
}
```

Tableau 1/Z.130 – Propriétés prédéfinies

Entité	Nom de la propriété prédéfinie	Type	Description	Obligatoire
Node	Processeur	ProcessorType	Processeur situé au niveau du nœud	Oui
	Mémoire	Integer	Capacité mémoire maximale du nœud (en kilooctets)	Non
	Système d'exploitation	OSType	Identification du système d'exploitation du nœud	Oui
NodeLink	Largeur de bande	Integer	Débit maximal de la liaison (en kilooctets par seconde)	Non

5.6.2 Carte d'installation

Métamodèle

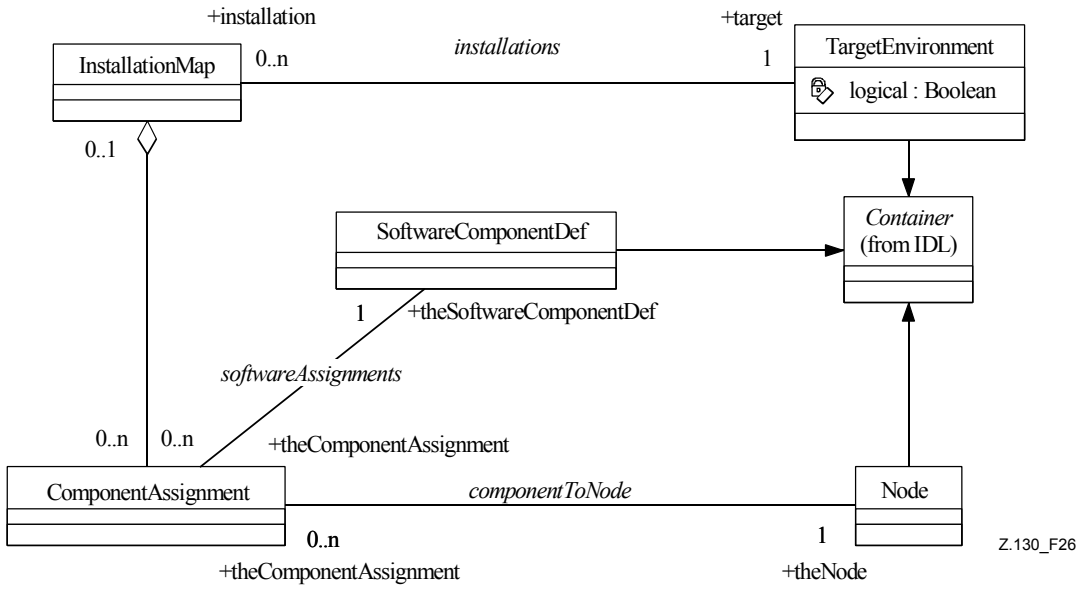


Figure 26/Z.130 – Carte d'installation

Sémantique

Une fois qu'un *environnement cible* est modélisé, des entités appropriées peuvent être attribuées à ses nœuds. Il existe deux types d'entités: les unités logicielles, représentant le logiciel nécessaire dans un nœud, et les ensembles de **CO**, représentant des instances concrètes des **types de CO**.

Une carte d'installation représente la répartition des implémentations dans un environnement cible. Elle est constituée d'un ensemble d'attributions d'installation, chacune associant un **composant logiciel** à un nœud. Une carte d'installation est fondée sur les nœuds d'un environnement cible. Elle est représentée par la métaclasse *InstallationMap* et les attributions d'installation sont représentées par la métaclasse *ComponentAssignment* (voir Figure 26).

5.6.3 Carte d'instanciation

Métamodèle

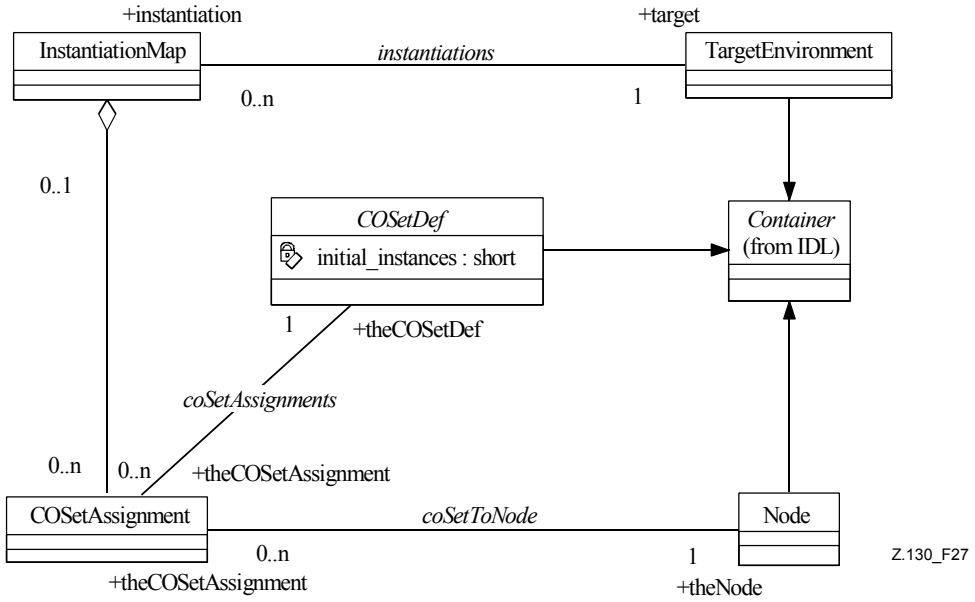


Figure 27/Z.130 – Carte d'instanciation

Sémantique

Une carte d'instanciation représente la répartition d'instances concrètes de **types de CO** dans un environnement cible. Elle est constituée d'un ensemble d'attributions d'instanciation, qui associent un ensemble de **CO** à un nœud de l'environnement cible. Une carte d'instanciation est fondée sur les **types de CO** définis dans le contexte d'un **assemblage** et sur les nœuds d'un environnement cible choisi. La métaclasse représentant le concept de carte d'instanciation est *InstantiationMap*. L'attribution de **CO** est représentée par la métaclasse *COSetAssignment* (voir Figure 27).

5.6.4 Plan de déploiement

Métamodèle

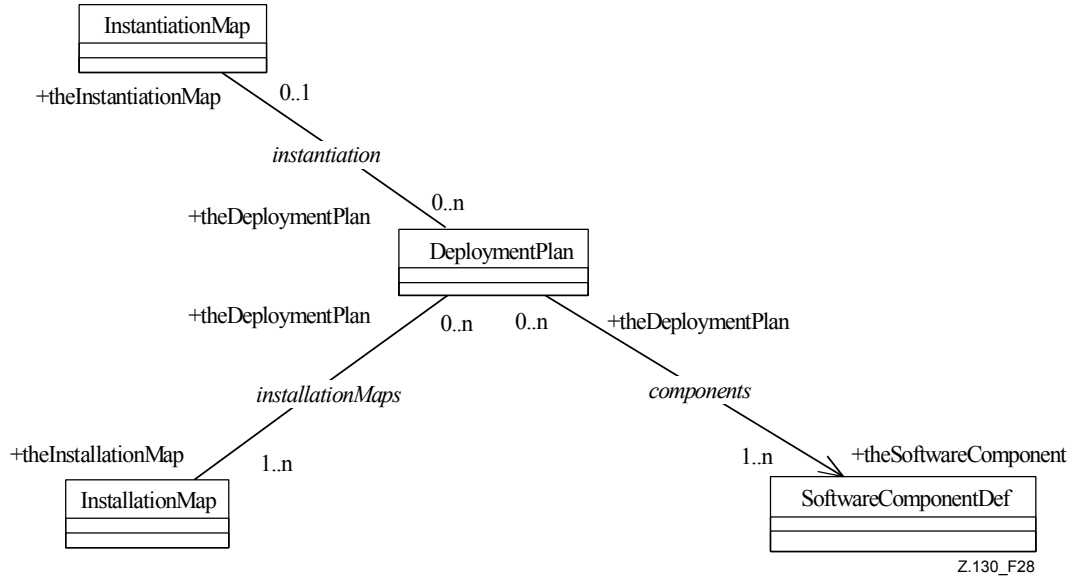


Figure 28/Z.130 – Plan de déploiement

Sémantique

Un plan de déploiement est défini par le choix d'un ou de plusieurs **composants logiciels**, contenant des implémentations de **types de CO**, une ou plusieurs cartes d'installation, déterminant l'endroit où les **composants logiciels** doivent être installés, et zéro ou une carte d'instanciation, déterminant l'endroit où des **CO** doivent être créés (voir Figure 28). La carte d'instanciation et toutes les cartes d'installation d'un modèle en eODL doivent être fondées sur le même environnement cible et sur le même **assemblage**.

6 Bibliographie

[11] Groupe de gestion d'objets: *OMG Unified Modeling Language Specification*, Version 1.3. Document formal/00-03-02 de l'OMG.

[12] Recommandation du W3C: *Extensible Markup Language (XML) 1.0 (Second edition)*.

[13] Groupe de gestion d'objets: *UML Profile for MOF*. Document ad/01-02-29 de l'OMG.

[14] Groupe de gestion d'objets: *Model Driven Architecture*. Document omg/00-11-05 de l'OMG.

[15] IEEE Std. 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*.

[16] SZYPERSKI (C.): *Component Software – Beyond Object-Oriented Programming*, Addison-Wesley, ISBN: 0-201-17888-5.

Annexe A

Syntaxe de l'eODL

A.1 Introduction

La présente annexe contient la notation textuelle de l'eODL dans le formalisme EBNF. La syntaxe des concepts provenant de la version 2.4.2 du CORBA-IDL de l'OMG est tirée du [5].

A.2 Conventions lexicales et base grammaticale

Dans les paragraphes qui suivent, on applique les définitions du § 3.1 (conventions lexicales de l'IDL) et du § 3.4 (grammaire de l'IDL) du document spécifiant la version 2.4.2 du CORBA IDL de l'OMG.

A.3 Point de vue de traitement

A.3.1 Espaces de noms, types de données, anomalies, opérations et attributs

Le modèle CMM est fondé sur le système de types de données du CORBA-IDL, le langage eODL est fondé sur le CORBA-IDL. Ces fondements permettent de projeter de façon canonique les types de données, les **espaces de noms** et les **anomalies** CMM en eODL. Les **éléments d'interaction de type opération** des modèles conformes au modèle CMM sont également mappés de façon canonique avec des **opérations** et des attributs du CORBA IDL.

A.3.2 Signaux et valeurs acheminées

L'une des extensions du modèle CMM par rapport au fondement conceptuel de l'UIT-ODL est l'introduction d'**éléments d'interaction de type signal**. Ces **éléments d'interaction** sont fondés sur la définition des **signaux**. En eODL, les **signaux** sont définis sur la base de la grammaire suivante:

```
<signal_dcl>      ::=   "signal" <identifier> "{" <member_list> "}"
<member_list>     ::=   <member>+
<member>          ::=   <type_spec> <declarators> ";"
```

A.3.3 Type de média, média et ensemble de médias

Le fondement conceptuel de l'UIT-ODL ne définit pas de sémantique pour les interactions de type flux. Le modèle CMM définit précisément la sémantique des interactions de type **média continu** qui remplacent l'interaction de type flux en UIT-ODL. La représentation en eODL des concepts **média**, **type de média** et **ensemble de médias** définis dans le modèle CMM est donnée par la grammaire suivante:

```
<mediaset_dcl>    ::=   "mediaset" <identifier> "{" <member_list> "}"
<mediatype_dcl>   ::=   "mediatype" <identifier>
<medium_dcl>      ::=   "medium" <identifier>  "(" <scoped_name> { "," <scoped_name> }* ")"
```

A.3.4 Types d'interface et éléments d'interaction

Dans le modèle CMM, un **type d'interface** combine les **éléments d'interaction** des différents types d'interaction dans un seul contexte d'interaction. Pour la représentation syntaxique de ce concept, la construction de l'interface est étendue par la notion d'éléments d'interaction **source**, **puits**, **produire** et **consommer** en plus des **éléments d'interaction de type opération**.

```
<interface>       ::=   <interface_dcl>
                  |     <forward_dcl>
<interface_dcl>   ::=   <interface_header> "{" <interface_body> "}"
<forward_dcl>     ::=   [ "abstract" ] "interface" <identifier>
```

```
<interface_header>    ::=    [ "abstract" ] "interface"  <identifier> [ <interface_inheritance_spec> ]
<interface_body>      ::=    <export> *
<export>              ::=    <type_dcl> ";"
                      |      <const_dcl> ";"
                      |      <except_dcl> ";"
                      |      <attr_dcl> ";"
                      |      <op_dcl> ";"
                      |       <produce_dcl> ";"
                      |      <consume_dcl> ";"
                      |       <source_dcl> ";"
                      |      <sink_dcl> ";"
<produce_dcl>         ::=    "produce" <scoped_name> <identifier>
<consume_dcl>         ::=    "consume" <scoped_name> <identifier>
<source_dcl>          ::=    "source" <scoped_name> <identifier>
<sink_dcl>            ::=    "sink" <scoped_name> <identifier>
```

A.3.5 Type d'objet de traitement

L'UIT-ODL possède déjà la notion de **type d'objet de traitement (type de CO)**. Compte tenu de la définition précise du point de vue configuration d'un **type de CO**, le concept de l'interface initiale est obsolète. La grammaire pour les **types de CO** issue de l'ITU-ODL est modifiée en eODL, elle est définie par les règles suivantes:

```
<object_template>           ::=    <object_template_header> "{" <object_template_export> "}"
<object_template_header>    ::=    "CO" <identifier> [ <object_inheritance_spec> ]
<object_inheritance_spec>   ::=    ":" <scoped_name> { "," <scoped_name> }*
<object_template_export>    ::=    <object_export>*
<object_export>             ::=    <export>
                            |      <reqrd_interf_templates> ";"
                            |      <suptd_interf_templates> ";"
                            |      <use_dcl> ";"
                            |      <provide_dcl> ";"
                            |      <implements_dcl> ";"
                            |      <state_def_dcl> ";"
                            |      <constraint_dcl> ";"
                            |      <property_list>
<reqrd_interf_templates>    ::=    "requires" <scoped_name> { "," <scoped_name> }*
<supd_interf_templates>     ::=    "supports" <scoped_name> { "," <scoped_name> }*
```

A.3.6 Propriété

Une propriété sert à définir les propriétés disponibles ou nécessaires des éléments de modèle. La notion de propriété est utilisée pour la définition des environnements cibles et des unités logicielles.

```
<property_list>       ::=    { <property_dcl> ";" }*
<property_dcl>        ::=    "property" <property_name> "=" <property_value>
<property_name>       ::=    <identifier>
<property_value>      ::=    <simple_property_value>
```

```
                        |       <structured_property_value>
                        |       <sequence_property_value>
<simple_property_value> ::=     <string_literal>
                        |       <integer_literal>
                        |       <boolean_literal>
<structured_property_value> ::= "{"  <property_assign>*  "}"
<sequence_property_value > ::=  "["  <property_value>*  "]"
<property_assign>       ::=     <property_name> "="  <property_value> ";"
<constraint_dcl>        ::=     "constraint" <identifier> "{" <constraint_body> "}"
<constraint_body>       ::=     "language" "=" <string_literal> "body" "=" <string_literal> ";"
```

A.3.7 Type externe

Un type externe sert à faire référence, au moyen d'un identificateur, à un type de données défini à l'extérieur.

```
<extern_type>           ::=     "extern" "type" <identifier> <string_literal>
```

A.4 Point de vue configuration

A.4.1 Ports

Dans le modèle CCM, le principal concept du point de vue configuration des **CO** est le concept de **port**. La notation pour les **ports** simples et dynamiques est définie par les règles suivantes:

```
<object_export>         ::=     <export>
                        |       <reqrd_interf_templates> ";"
                        |       <suptd_interf_templates> ";"
                        |       <use_dcl> ";"
                        |       <provide_dcl> ";"
                        |       <implements_dcl> ";"
                        |       <state_def_dcl> ";"
                        |       <constraint_dcl> ";"
                        |       <property_list>
<use_dcl>               ::=     "use" [ "multiple" ] <scoped_name> <identifier>
<provide_dcl>           ::=     "provide" [ "multiple" ] <scoped_name> <identifier>
```

A.5 Point de vue implémentation

A.5.1 Artefacts et éléments d'implémentation

Dans le modèle CCM, les **artefacts** sont les abstractions des constructions concrètes de langage de programmation qui implémentent le comportement des **CO**. La représentation des **artefacts** et des **éléments d'implémentation** en eODL est donnée par les règles suivantes:

```
<artefact>              ::=     <artefact_dcl>
                        |       <artefact_forward_dcl>
<artefact_forward_dcl>  ::=     "artefact"  <identifier>
<artefact_dcl>          ::=     <artefact_header> "{" <artefact_body> "}"
<artefact_header>       ::=     "artefact" <identifier> [ <artefact_inheritance_spec> ]
<artefact_inheritance_spec> ::= ":" <scoped_name> { "," <scoped_name> }*
```

```
<artefact_body>              ::=  <impl_elem_dcl>*
<impl_elem_dcl>              ::=  <identifier> "implements" <impl_case_dcl> <scoped_name> ";"
<impl_case_dcl>              ::=  "supply" | "use"
```

A.5.2 Relation implémenter et politiques d'instanciation

La relation **implemented_by** précise quel est l'**artefact** qui est utilisé pour la réalisation d'un comportement de **types de CO**. Elle est définie en eODL dans le contexte des définitions de **type de CO**:

```
<object_export>              ::=  <export>
                             |    <reqrd_interf_templates> ";"
                             |    <suptd_interf_templates> ";"
                             |    <use_dcl> ";"
                             |    <provide_dcl> ";"
                             |    <implements_dcl> ";"
                             |    <state_def_dcl> ";"
                             |    <constraint_dcl> ";"
                             |    <property_list>
<implements_dcl>             ::=  "implemented" "by" <artefact_with_policy>
                                  { "," <artefact_with_policy> }*
<artefact_with_policy>       ::=  <scoped_name> [ "with" <instantiation_policy_dcl> ]
<instantiation_policy_dcl>   ::=  "ArtefactPool"
                             |    "ArtefactPerRequest"
                             |    "Singleton"
                             |    "UserDefined"
```

A.5.3 Types d'état

Dans de nombreux cas d'implémentation, une instance d'**artefact** doit pouvoir accéder aux informations d'état des **CO** qu'elle implémente. La définition des informations d'état des **CO** dans les **types de CO** est donnée par les règles suivantes:

```
<object_export>              ::=  <export>
                             |    <reqrd_interf_templates> ";"
                             |    <suptd_interf_templates> ";"
                             |    <use_dcl> ";"
                             |    <provide_dcl> ";"
                             |    <implements_dcl> ";"
                             |    <state_def_dcl> ";"
                             |    <constraint_dcl> ";"
                             |    <property_list>
<state_def_dcl>              ::=  "state" <scoped_name>
                                  [ "provided" "to" "(" <provided_to_dcl> ")" ]
<provided_to_dcl>            ::=  <scoped_name> { "," <scoped_name> }*
```

A.6 Point de vue déploiement

A.6.1 Composant logiciel

Une implémentation concrète d'un **type de CO** est représentée par la définition d'une unité logicielle. Dans le cadre d'une telle définition, plusieurs **types de CO** peuvent être **réalisés**. Les unités logicielles peuvent dépendre d'autres unités logicielles et/ou nécessiter d'autres propriétés et services de l'environnement d'exécution final.

```
<softwarecomponent_dcl>     ::=   <softwarecomponent_header>
                                  "{" <softwarecomponent_body> "}"
<softwarecomponent_header>        ::=   "softwarecomponent"  <identifier>
                                        "realizes"  <cotype_identifier_list>
<cotype_identifier_list>    ::=   <cotype_identifier> { ","  <cotype_identifier> }*
<softwarecomponent_body>    ::=   <softwarecomponent_stmt>*
<softwarecomponent_stmt>    ::=   "dependent"  "{"  <softwarecomponent_list>  "}" ";"
                            |     "requires"  "{"  <property_list>  "}"  ";"
<softwarecomponent_list>    ::=   <softwarecomponent_identifier>
                                  {  ","  <softwarecomponent_identifier>  }*
<softwarecomponent_identifier>    ::=   <scoped_name>
```

A.6.2 Assemblage

Un **assemblage** décrit un ensemble de composants interconnectés, mais pas leur répartition concrète dans un environnement de traitement réparti.

A.6.2.1 Définition d'un assemblage

La définition d'un **assemblage** contient les définitions de tous les ensembles d'instances appartenant à l'**assemblage** et les définitions de **connexion** relatives à ces instances.

```
<assembly_dcl>              ::=   <assembly_header> "{"  <assembly_body>  "}"
<assembly_header>           ::=   "assembly"  <identifier>
<assembly_body>             ::=   <assembly_stmt>*
<assembly_stmt>             ::=   <instance_set_dcl> ";"
                            |     <connect_dcl> ";"
                            |     <constraint_dcl> ";"
                            |     <property_list>
```

A.6.2.2 Définition d'un ensemble d'instances

La définition d'un ensemble d'instances décrit un ensemble non vide d'instances d'un **type de CO**.

```
<instance_set_dcl>          ::=   <identifier> [ "(" <integer_literal> ")" ] ":" <cotype_identifier>
<cotype_identifier>         ::=   <scoped_name>
```

A.6.2.3 Définition de connexion

Les **connexions** entre les instances et les ensembles d'instances sont exprimées au moyen de la définition de connexion. Ici, des **ports** sont interconnectés conformément à la définition de **type de CO**, l'un des **ports** faisant office de **source** et l'autre de **puits**.

```
<connect_dcl>               ::=   "connect"  [ <identifier> ]  "{"  <connection_list>  "}"
<connection_list>           ::=   { <connection> ";" } +
<connection>                ::=   <instance_set_identifier>  "."  <port_identifier> "="
```

```
                                <instance_set_identifier> "." <port_identifier>
<instance_set_identifier>   ::= <scoped_name>
<port_identifier>           ::= <scoped_name>
```

A.6.3 Définition d'une carte d'installation

La définition d'une carte d'installation décrit une attribution de **types de CO** aux nœuds d'un environnement cible. Lors d'une action d'installation ultérieure, on peut faire référence à cette définition et déclencher l'installation d'unités logicielles au niveau de **nœuds**.

```
<installation_map_dcl>      ::= <installation_map_header> "{" <installation_map_body> "}"
<installation_map_header>   ::= "installation" <identifier>
                                "uses" "environment" <environment_identifier>
<installation_map_body>     ::= <install_stmt> *
<install_stmt>              ::= <softwarecomponent_identifier> "->" <node_identifier> ";"
<environment_identifier>    ::= <scoped_name>
```

A.6.4 Définition d'une carte d'instanciation

La définition d'une carte d'instanciation décrit une attribution concrète d'ensembles d'instances d'un **assemblage** aux **nœuds** de l'environnement cible spécifié.

```
<instantiation_map_dcl>      ::= <instantiation_map_header> "{" <instantiation_map_body> "}"
<instantiation_map_header>   ::= "instantiation" <identifier> <instantiation_map_header_env>
                                 <instantiation_map_header_ass>
<instantiation_map_header_env>  ::=  "uses" "environment" <environment_identifier>
<instantiation_map_header_ass>  ::=  "uses" "assembly" <assembly_identifier>
<assembly_identifier>        ::= <scoped_name>
<instantiation_map_body>     ::= <assign_instance_stmt>*
<assign_instance_stmt>       ::= <instance_set_identifier_list> "->" <node_identifier> ";"
<instance_set_identifier_list> ::= <instance_set_identifier> { "," <instance_set_identifier> }*
```

A.6.5 Action de déploiement

Une action de déploiement est une séquence d'actions d'installation et d'instanciation à exécuter pendant le **déploiement**.

```
<deployment_action>          ::= "deploy" "{" <deployment_body>  "}" ";"
<deployment_body>            ::= "install" "{" <install_list> "}" ";"
                                 "instantiate" "{" <instantiation_list> "}" ";"
```

A.6.5.1 Action d'installation

Une action d'installation spécifie l'installation d'une unité logicielle au niveau d'un nœud d'exécution dans un environnement cible.

```
<install_list>               ::= <install_member>*
<install_member>             ::= <installation_map_identifier> ";"
                             |   <qualified_install_stmt>
<qualified_install_stmt>     ::= <softwarecomponent_identifier> "->"
                                 <environment_identifier> "." <node_identifier> ";"
<installation_map_identifier> ::= <scoped_name>
```

A.6.5.2 Action d'instanciation

Une action d'instanciation spécifie l'instanciation d'un ensemble de **CO** au niveau d'un nœud d'exécution dans un environnement cible.

```
<instantiation_list>            ::=  <instantiation_member>*
<instantiation_member>          ::=  <instantiation_map_identifier> ";"
                                |    <qualified_assign_instance_stmt> ";"
<instantiation_map_identifier>  ::=  <identifier>
<qualified_assign_instance_stmt> ::= <assembly_identifier> "." <instance_set_identifier>
                                     "->" <environment_identifier> "." <node_identifier>
```

A.7 Environnement cible

Un environnement cible est un environnement d'exécution possible pour des **assemblages**. Il représente la structure et les propriétés de cet environnement. Une syntaxe textuelle eODL peut contenir plusieurs spécifications d'environnement cible.

A.7.1 Définition d'un environnement

La définition d'un environnement décrit un environnement d'exécution possible en termes de **nœuds** disponibles et de **liaison** de communication.

```
<environment_dcl>       ::=  <environment_header> "{" <environment_body> "}"
<environment_header>    ::=  "environment" <identifier>
<environment_body>      ::=  <environment_stmt>+
<environment_stmt>      ::=  <node_dcl> ";"
                        |    <link_dcl> ";"
```

A.7.2 Définition d'un nœud

La définition d'un nœud représente un **nœud** d'exécution identifiable dans l'environnement cible, qui peut être la cible de l'installation de **types de CO** et de l'instanciation d'ensembles d'instances. Les propriétés présentes dans la définition d'un nœud caractérisent les fonctionnalités du nœud d'exécution.

```
<node_dcl>              ::=  "node" <identifier> "{" <property_list> "}"
```

A.7.3 Définition d'une liaison

Les liaisons de communication entre nœuds d'exécution de l'environnement cible sont représentées sous forme de définitions de liaison. Les propriétés présentes dans la définition d'une liaison correspondent aux caractéristiques et au type de liaison de communication.

```
<link_dcl>              ::=  <link_header>  "{" <link_body> "}"
<link_header>           ::=  "link" <identifier>
<link_body>             ::=  "node" <node_list> ";" <property_list> ";"
<node_list>             ::=  <node_identifier> { "," <node_identifier> }*
<node_identifier>       ::=  <scoped_name>
```

A.8 Syntaxe de l'eODL

Le présent paragraphe donne l'ensemble complet des règles de production de l'eODL. Il inclut également toutes les règles héritées de la syntaxe de base de la version 2.4.2 de l'IDL de l'OMG.

```
<specification>         ::=  <definition>+ [ <deployment_action> ]
<definition>        ::=      <type_dcl> ";"
```

```
                    |       <const_dcl> ";"
                    |       <except_dcl> ";"
                    |       <interface> ";"
                    |       <object_template> ";"
                    |       <artefact> ";"
                    |       <module> ";"
                    |        <value> ";"
                    |       <signal_dcl> ";"
                    |       <mediaset_dcl> ";"
                    |       <mediatype_dcl> ";"
                    |       <medium_dcl> ";"
                    |       <assembly_dcl> ";"
                    |       <softwarecomponent_dcl> ";"
                    |       <environment_dcl> ";"
                    |       <installation_map_dcl> ";"
                    |       <instantiation_map_dcl> ";"
<module>            ::=     "module" <identifier> "{" <definition> + "}"
<object_template>           ::=     <object_template_header> "{" <object_template_export> "}"
<object_template_header>    ::=     "CO" <identifier> [ <object_inheritance_spec> ]
<object_inheritance_spec>   ::=     ":" <scoped_name> { "," <scoped_name> }*
<object_template_export>    ::=     <object_export>*
<object_export>     ::=     <export>
                    |       <reqrd_interf_templates> ";"
                    |       <suptd_interf_templates> ";"
                    |       <use_dcl> ";"
                    |       <provide_dcl> ";"
                    |       <implements_dcl> ";"
                    |       <state_def_dcl> ";"
                    |       <constraint_dcl> ";"
                    |       <property_list>

<reqrd_interf_templates>    ::=     "requires" <scoped_name> { "," <scoped_name> }*
<supd_interf_templates>     ::=     "supports" <scoped_name> { "," <scoped_name> }*
<use_dcl>       ::= "use" [ "multiple" ] <scoped_name> <identifier>
<provide_dcl>   ::= "provide" [ "multiple" ] <scoped_name> <identifier>
<artefact>      ::= <artefact_dcl>
                |   <artefact_forward_dcl>
<artefact_forward_dcl>      ::=     "artefact"  <identifier>
<artefact_dcl>  ::=         <artefact_header> "{" <artefact_body> "}"
<artefact_header>           ::=     "artefact" <identifier> [ <artefact_inheritance_spec> ]
<artefact_inheritance_spec>         ::=     ":" <scoped_name> { "," <scoped_name> }*
<artefact_body>     ::=     <impl_elem_dcl>*
```

```
<impl_elem_dcl>     ::=     <identifier> "implements" <impl_case_dcl> <scoped_name> ";"
<impl_case_dcl>     ::=     "supply" | "use"
<implements_dcl>    ::=     "implemented" "by" <artefact_with_policy>
                            { "," <artefact_with_policy> }*
<artefact_with_policy>      ::=     <scoped_name> [ "with" <instantiation_policy_dcl> ]
<instantiation_policy_dcl>  ::=     "ArtefactPool"
                            |       "ArtefactPerRequest"
                            |       "Singleton"
                            |       "UserDefined"
<state_def_dcl>     ::=     "state" <scoped_name> [ "provided" "to" "(" <provided_to_dcl> ")" ]
<provided_to_dcl>   ::=     <scoped_name> { "," <scoped_name> }*
<interface>         ::=     <interface_dcl>
                    |       <forward_dcl>
<interface_dcl>     ::=     <interface_header> "{" <interface_body> "}"
<forward_dcl>       ::=     [ "abstract" ] "interface" <identifier>
<interface_header>  ::=     [ "abstract" ] "interface"  <identifier> [ <interface_inheritance_spec> ]
<interface_body>    ::=     <export> *
<export>        ::=     <type_dcl> ";"
                |       <const_dcl> ";"
                |       <except_dcl> ";"`
                |       <attr_dcl> ";"
                |       <op_dcl> ";"
                |       <produce_dcl> ";"
                |       <consume_dcl> ";"
                |       <source_dcl> ";"
                |       <sink_dcl> ";"
<produce_dcl>       ::=     "produce" <scoped_name> <identifier>
<consume_dcl>       ::=     "consume" <scoped_name> <identifier>
<source_dcl>        ::=     "source" <scoped_name> <identifier>
<sink_dcl>          ::=     "sink" <scoped_name> <identifier>
<interface_inheritance_spec> ::=    ":" <interface_name> { "," <interface_name> } *
<interface_name>            ::=     <scoped_name>
<scoped_name>               ::=     <identifier>
                            |       "::" <identifier>
                            |       <scoped_name> "::" <identifier>
<signal_dcl>        ::=     "signal" <identifier> "{" <member_list> "}"
<mediaset_dcl>      ::=     "mediaset" <identifier> "{" <member_list> "}"
<mediatype_dcl>     ::=     "mediatype" <identifier>
<medium_dcl>        ::=     "medium" <identifier> "(" <scoped_name> { "," <scoped_name> }* ")"
<value>         ::=     ( <value_dcl> | <value_abs_dcl> | <value_box_dcl> | <value_forward_dcl>)
<value_forward_dcl> ::=     [ "abstract" ] "valuetype" <identifier>
<value_box_dcl>     ::=     "valuetype" <identifier> <type_spec>
```

```
<value_abs_dcl>     ::=     "abstract"  "valuetype"  <identifier>  [  <value_inheritance_spec>  ]
                            "{" <export>* "}"
<value_dcl>      ::=  <value_header> "{" <value_element>* "}"
<value_header>  ::=         ["custom" ] "valuetype" <identifier> [ <value_inheritance_spec> ]
<value_inheritance_spec>    ::=     [ ":" [ "truncatable" ] <value_name>
                                    { "," <value_name> }* ]
                                    [ "supports" <interface_name>
                                    { "," <interface_name> }* ]
<value_name>    ::=  <scoped_name>
<value_element>     ::=     <export> | <state_member> | <init_dcl>
<state_member>      ::=     ( "public" | "private" ) <type_spec> <declarators> ";"
<init_dcl>    ::=  "factory" <identifier> "(" [ <init_param_decls> ] ")" ";"
<init_param_decls>  ::=     <init_param_decl> { "," <init_param_decl> }
<init_param_decl>   ::=     <init_param_attribute> <param_type_spec> <simple_declarator>
<init_param_attribute>      ::=     "in"
<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>
                 |          <octet_type>
<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"
<positive_int_const> ::=            <const_exp>
<type_dcl>  ::=         "typedef" <type_declarator>
                |       <struct_type>
                |       <union_type>
                |       <enum_type>
                |       "native" <simple_declarator>
<type_declarator>       ::=         <type_spec> <declarators>
<type_spec>         ::=  <simple_type_spec>
                    |       <constr_type_spec>
                    |       <extern_type>
<extern_type>           ::=         "extern" "type" <identifier> <string_literal>
<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>
                        |           <value_base_type>
<template_type_spec>            ::=     <sequence_type>
                                |       <string_type>
                                |       <wide_string_type>
                                |       <fixed_pt_type>
<constr_type_spec> ::=              <struct_type>
                        |           <union_type>
                        |           <enum_type>
<declarators>       ::=  <declarator> { "," <declarator> } *
```

```
<declarator>      ::=  <simple_declarator> | <complex_declarator>
<simple_declarator> ::=        <identifier>
<complex_declarator>           ::=     <array_declarator>
<floating_pt_type>   ::=       "float"
                     |         "double"
                     |         "long" "double"
<integer_type>   ::=  <signed_int> | <unsigned_int>
<signed_int>     ::=  <signed_short_int>
                 |    <signed_long_int>
                 |    <signed_longlong_int>
<signed_short_int>  ::=        "short"
<signed_long_int>   ::=        "long"
<signed_longlong_int>          ::=     "long" "long"
<unsigned_int>       ::=       <unsigned_short_int>
                     |         <unsigned_long_int>
                     |         <unsigned_longlong_int>
<unsigned_short_int>           ::=     "unsigned" "short"
<unsigned_long_int>            ::=     "unsigned" "long"
<unsigned_longlong_int>        ::=     "unsigned" "long" "long"
<char_type>      ::=  "char"
<wide_char_type>    ::=        "wchar"
<boolean_type>       ::=       "boolean"
<octet_type>     ::=  "octet"
<any_type>       ::=  "any"
<object_type>    ::=  "Object"
<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>
<enum_type>      ::=           "enum" <identifier> "{" <enumerator> { "," <enumerator> } * "}"
<enumerator>     ::=           <identifier>
<sequence_type>                ::=     "sequence" "<" <simple_type_spec> ","
```

```
                                <positive_int_const> ">" | "sequence" "<" <simple_type_spec> ">"
<string_type>     ::=  "string" "<" <positive_int_const> ">" | "string"
<wide_string_type>  ::=        "wstring" "<" <positive_int_const> ">" | "wstring"
<array_declarator>  ::=        <identifier> <fixed_array_size>+
<fixed_array_size>  ::=        "[" <positive_int_const> "]"
<attr_dcl>    ::=       [ "readonly" ] "attribute"
                        <param_type_spec> <simple_declarator> { "," <simple_declarator> }*
<except_dcl>      ::=  "exception" <identifier> "{" <member>* "}"
<op_dcl>      ::=  [ <op_attribute> ] <op_type_spec>
                   <identifier> <parameter_dcls> [ <raises_expr> ] [ <context_expr> ]
<op_attribute>          ::=        "oneway"
<op_type_spec>          ::=        <param_type_spec> |"void"
<parameter_dcls>        ::=        "(" <param_dcl> { "," <param_dcl> } * ")"|"(" ")"
<param_dcl>       ::=  <param_attribute> <param_type_spec> <simple_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>
                        |          <scoped_name>
<fixed_pt_type>         ::=        "fixed" "<" <positive_int_const> ","<positive_int_const> ">"
<fixed_pt_const_type>           ::=      "fixed"
<value_base_type>  ::=        "ValueBase"
<assembly_dcl> ::=  <assembly_header> "{"  <assembly_body>  "}"
<assembly_header> ::=        "assembly"  <identifier>
<assembly_body>   ::=        <assembly_stmt>*
<assembly_stmt>   ::=        <instance_set_dcl> ";"
                        |          <connect_dcl> ";"
                        |          <constraint_dcl> ";"
                        |          <property_list>
<instance_set_dcl>  ::=        <identifier> [ "(" <integer_literal> ")" ] ":" <cotype_identifier>
<cotype_identifier>  ::=       <scoped_name>
<connect_dcl>       ::=        "connect"  [ <identifier> ]  "{"  <connection_list>  "}"
<connection_list>   ::=        { <connection> ";" } +
<connection>        ::=        <instance_set_identifier>  "."  <port_identifier>
                               "=" <instance_set_identifier>  "."  <port_identifier>
<instance_set_identifier>       ::=      <scoped_name>
<port_identifier>       ::=        <scoped_name>
<softwarecomponent_dcl>     ::=      <softwarecomponent_header>
                                     "{" <softwarecomponent_body> "}"
```

```
<softwarecomponent_header>      ::=     "softwarecomponent"  <identifier>
                                        "realizes"  <cotype_identifier_list>
<cotype_identifier_list>        ::=     <cotype_identifier> { ","  <cotype_identifier> }*
<softwarecomponent_body>  ::=   <softwarecomponent_stmt>*
<softwarecomponent_stmt>  ::=   "dependent"  "{"  <softwarecomponent_list>  "}" ";"
                          |     "requires"  "{"  <property_list>  "}"  ";"
<softwarecomponent_list>  ::=   <softwarecomponent_identifier>
                                { ","  <softwarecomponent_identifier>  }*
<softwarecomponent_identifier>  ::=     <scoped_name>
<environment_dcl>  ::=      <environment_header> "{"  <environment_body>  "}"
<environment_header>        ::=   "environment"  <identifier>
<environment_body> ::=      <environment_stmt>+
<environment_stmt>  ::=     <node_dcl>  ";"
                       |    <link_dcl>  ";"
<node_dcl> ::=      "node"  <identifier>  "{"  <property_list>  "}"
<property_list>     ::=     { <property_dcl> ";" }*
<property_dcl>      ::=     <property_name>  "="  <property_value>
<property_name>     ::=     <identifier>
<property_value>    ::=     <simple_property_value>
                    |       <structured_property_value>
                    |       <sequence_property_value>
<simple_property_value>     ::=   <string_literal>
                    |       <integer_literal>
                    |       <boolean_literal>
<structured_property_value>  ::=   "{"  <property_assign>*  "}"
<sequence_property_value>  ::=   "["  <property_value>*  "]"
<constraint_dcl>    ::=     "constraint" <identifier> "{" <constraint_body> "}"
<constraint_body>   ::=     "language" "=" <string_literal> "body" "=" <string_literal> ";"
<property_assign>   ::=     <property_name> "="  <property_value> ";"
<link_dcl>      ::=  <link_header>  "{"  <link_body>  "}"
<link_header>   ::=  "link"  <identifier>
<link_body>     ::=  "node" <node_list>  ";" <property_list> ";"
<node_list>     ::=  <node_identifier>  {  ","  <node_identifier>  }*
<node_identifier>   ::=     <scoped_name>
<installation_map_dcl>      ::=   <installation_map_header>  "{" <installation_map_body> "}"
<installation_map_header>   ::=   "installation" <identifier>
                                  "uses" "environment" <environment_identifier>
<installation_map_body>     ::=   <install_stmt>*
<install_stmt>              ::=   <softwarecomponent_identifier> "->" <node_identifier> ";"
<environment_identifier>    ::=   <scoped_name>
<instantiation_map_dcl>     ::=   <instantiation_map_header> "{"  <instantiation_map_body>  "}"
<instantiation_map_header>  ::=   "instantiation" <identifier>
```

```
                                    <instantiation_map_header_env>
                                    <instantiation_map_header_ass>
<instantiation_map_header_env>    ::=    "uses"  "environment"  <environment_identifier>
<instantiation_map_header_ass>    ::=    "uses" "assembly" <assembly_identifier>
<assembly_identifier>          ::=    <scoped_name>
<instantiation_map_body>       ::=    <assign_instance_stmt>*
<assign_instance_stmt>         ::=    <instance_set_identifier_list> "->" <node_identifier> ";"
<instance_set_identifier_list> ::=    <instance_set_identifier> { "," <instance_set_identifier> }*
<deployment_action> ::=        "deploy" "{" <deployment_body> "}" ";"
<deployment_body>   ::=        "install" "{" <install_list> "}" ";"
                               "instantiate" "{" <instantiation_list> "}" ";"
<install_list>      ::=  <install_member>*
<install_member>    ::=        <installation_map_identifier>  ";"
                        |      <qualified_install_stmt>
<qualified_install_stmt>       ::=    <softwarecomponent_identifier>  "->"
                                      <environment_identifier>  "."  <node_identifier>  ";"
<installation_map_identifier>  ::=    <scoped_name>
<instantiation_list>  ::=      <instantiation_member>*
<instantiation_member>         ::=    <instantiation_map_identifier>  ";"
                               |      <qualified_assign_instance_stmt> ";"
<instantiation_map_identifier>        ::=      <identifier>
<qualified_assign_instance_stmt>   ::=      <assembly_identifier>  "."
                                            <instance_set_identifier>  "->"
                                            <environment_identifier>  "."  <node_identifier>
```

Annexe B

Mappage entre métamodèle et syntaxe

B.1 Introduction

La présente annexe décrit la relation entre le **métamodèle** eODL et la syntaxe textuelle concrète de l'eODL définie à l'Annexe A. La description est limitée aux concepts du **métamodèle** qui sont des extensions du **métamodèle** CORBA. La relation entre la syntaxe textuelle de la version 2.4.2 de l'IDL de l'OMG et le **métamodèle** CORBA de l'OMG est bien définie par l'OMG et n'est donc pas reproduite ici.

Le **métamodèle** est tel que spécifié au § 5 et il utilise la notation graphique qui y est présentée. La syntaxe textuelle correspondante est donnée au-dessous du graphe, suivie par une explication textuelle.

B.2 Signal et paramètre de signal

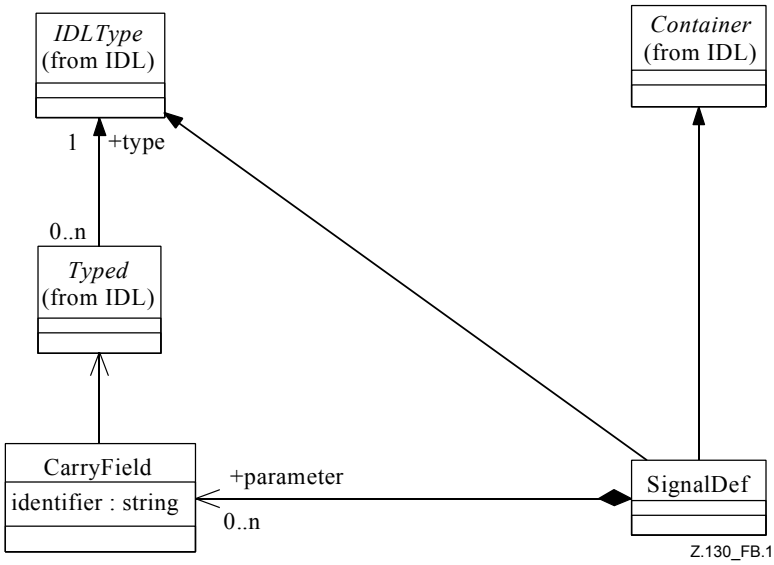


Figure B.1/Z.130 – Signal et paramètre de signal

Les productions **(1)**, **(2)** et **(3)** de la syntaxe concrète sont mappées avec les éléments *SignalDef/CarryField* du modèle (voir Figure B.1).

(1) <signal_dcl> ::= "signal" <identifier> "{" <member_list> "}"

(2) <member_list> ::= <member>+

(3) <member> ::= <type_spec> <declarators> ";"

Dans la production **(1)**, **<identifier>** est le nom de l'élément *SignalDef* du modèle. Tous les éléments *CarryField* qui participent à la relation de paramètre de ce *SignalDef* sont énumérés dans la production **<member_list> (2)**. Les productions **<member> (3)** de la syntaxe concrète correspondent aux éléments *CarryField* du modèle. Ici, **<type_spec>** correspond aux types du modèle, qui sont limités par le biais du concept Typed de l'IDL. Pour chaque déclarateur figurant dans **<declarators>**, il existe un élément *CarryField* du modèle.

B.3 Type de média, média, ensemble de médias

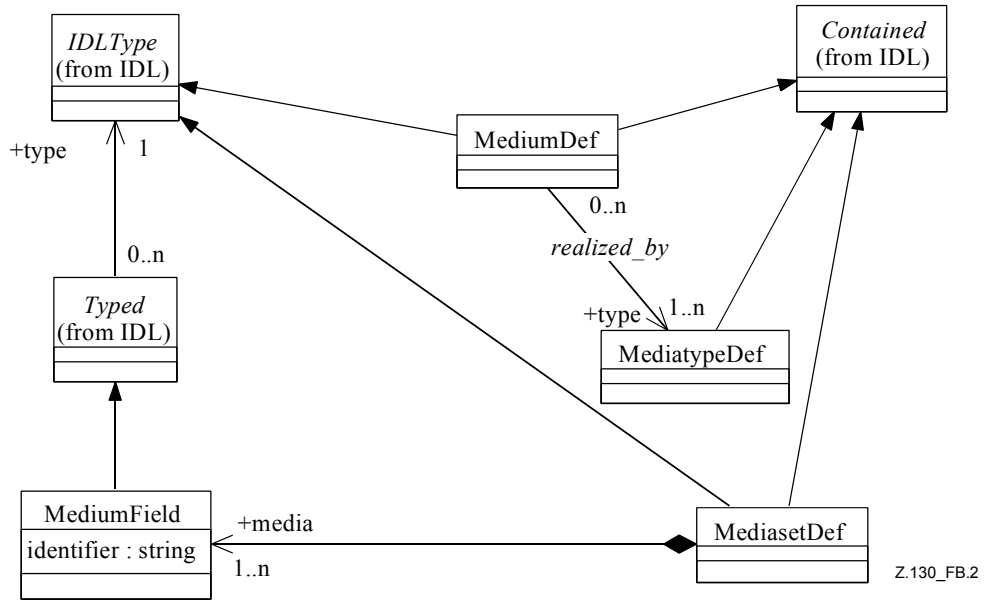


Figure B.2/Z.130 – Média, type de média, ensemble de médias

Les productions **(4)**, **(5)** et **(6)** de la syntaxe concrète sont mappées avec les éléments *MediasetDef*, *MediatypeDef* et *MediumDef* du modèle (voir Figure B.2).

(4) <mediaset_dcl> ::= "mediaset" <identifier> "{" <member_list> "}"

(5) <mediatype_dcl>::= "mediatype" <identifier>

(6) <medium_dcl> ::= "medium" <identifier>

La production **<mediatype_dcl> (5)** est représentée dans le modèle par l'élément *MediatypeDef*, **<identifier>** étant le nom qui correspond au concept Named de cet élément. Avec la production **<medium_dcl>**, la syntaxe concrète exprime les éléments *MediumDef*, **<identifier>** étant à nouveau le nom qui correspond au concept Named. Les **<scoped_name>** énumérés dans la production **(6)** doivent toujours se rapporter à un élément *MediatypeDef* et sont représentés dans le modèle par la relation *realized_by* avec l'élément *MediumDef*. Conformément à la production **(4)**, **<mediaset_dcl>** est représenté dans le modèle par l'élément *MediasetDef* avec **<identifier>** comme nom. Tous les éléments *MediumField* qui participent à la relation de média de cet élément *MediasetDef* sont énumérés dans la **<member_list>** de la production **(4)**. Concernant les **<member>** de la **<member_list>** de la production **(4)**, **<type_spec>** doit se rapporter à l'élément *MediatypeDef* et tous les déclarateurs figurant dans **<declarators>** doivent être simples.

B.4 Consommer et produire

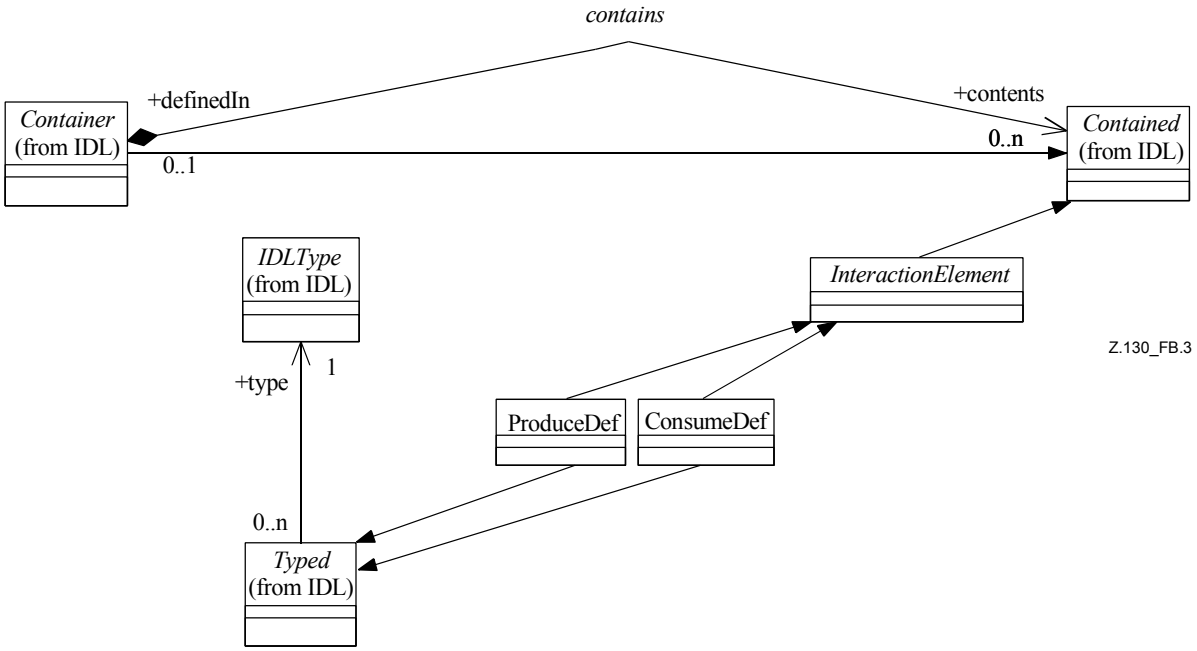


Figure B.3/Z.130 – Consommer et produire

Les productions **(7)** et **(8)** de la syntaxe concrète sont mappées avec les éléments ProduceDef/ConsumeDef du modèle (voir Figure B.3).

(7) <produce_dcl> ::= "produce" <scoped_name> <identifier>

(8) <consume_dcl> ::= "consume" <scoped_name> <identifier>

Dans les deux déclarations, **<scoped_name>** se rapporte à un élément *SignalDef*. Cette relation est traduite dans le modèle par la relation *Typed/IDLType* faisant intervenir *ProduceDef/ConsumeDef* par héritage et le *IDLType* est un *SignalDef* conformément à **<signal_dcl>**. *ProduceDef/ConsumeDef* sont tous deux des concepts dénommés issus du métamodèle et **<identifier>** figurant dans les productions **(7)/(8)** est mappé avec l'attribut de nom des éléments du modèle.

B.5 Puits et source

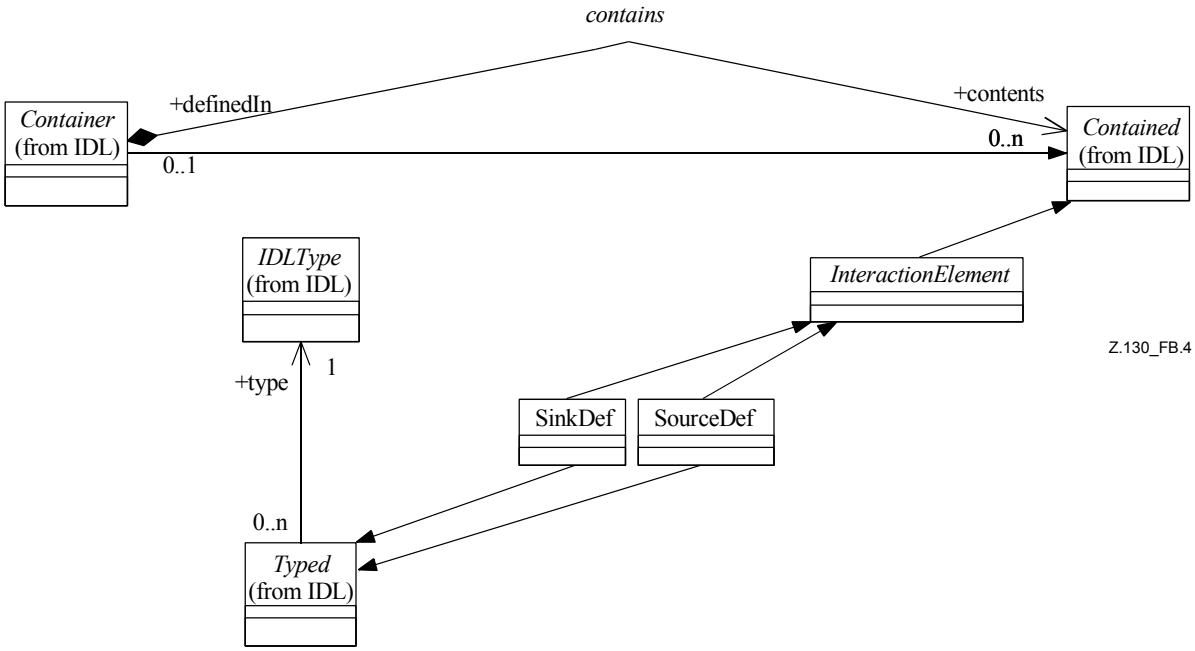


Figure B.4/Z.130 – Puits et source

Les productions **(10)** et **(9)** de la syntaxe concrète sont mappées avec les éléments *SinkDef/SourceDef* du modèle (voir Figure B.4).

(9) <source_dcl> ::= "source" <scoped_name> <identifier>

(10) <sink_dcl> ::= "sink" <scoped_name> <identifier>

Dans les deux déclarations, **<scoped_name>** se rapporte à un élément *MediasetDef*. Cette relation est traduite dans le modèle par la relation *Typed/IDLType* faisant intervenir *SinkDef/SourceDef* par héritage et le *IDLType* est un *MediasetDef* conformément à **<mediaset_dcl>**. *SinkDef/SourceDef* sont tous deux des concepts dénommés issus du **métamodèle** et **<identifier>** figurant dans les productions **(10)/(9)** est mappé avec l'attribut de nom des éléments du modèle.

B.6 Type d'interface

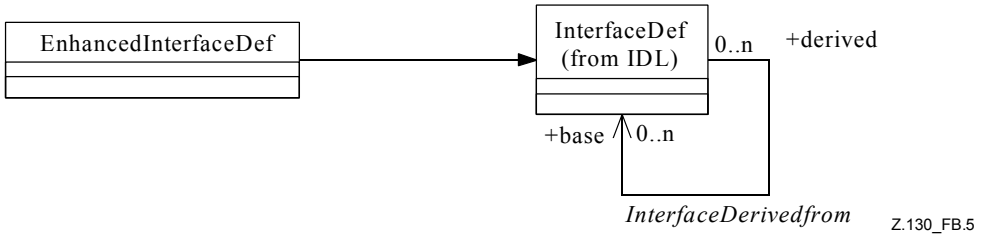


Figure B.5/.130 – Type d'interface

(11) <interface_dcl> ::= <interface_header> "{" <interface_body> "}"

(12) <interface_body> ::= <export> *

```
(13)  <export>          ::=  <type_dcl> ";"
                        |    <const_dcl> ";"
                        |    <except_dcl> ";"
                        |    <attr_dcl> ";"
                        |    <op_dcl> ";"
                        |    <produce_dcl> ";"
                        |    <consume_dcl> ";"
                        |    <source_dcl> ";"
                        |    <sink_dcl> ";"
```

La production **<interface_dcl> (11)** de la syntaxe concrète est mappée avec un élément *EnhancedInterfaceDef* du modèle. La production **<interface_body> (12)** est traitée de la même façon qu'en IDL. Par comparaison avec *InterfaceDef,* la production **<export> (13)** peut aussi contenir les éléments **<produce_dcl>, <consume_dcl>, <source_dcl>** et **<sink_dcl>**, dont le mappage est défini ci-dessus. Si **<interface_body>** ne contient pas ce type de nouveaux éléments, la production **<interface_dcl>** est mappée avec un élément *InterfaceDef* ordinaire (voir Figure B.5).

B.7 Types de CO, prendre en charge et requérir

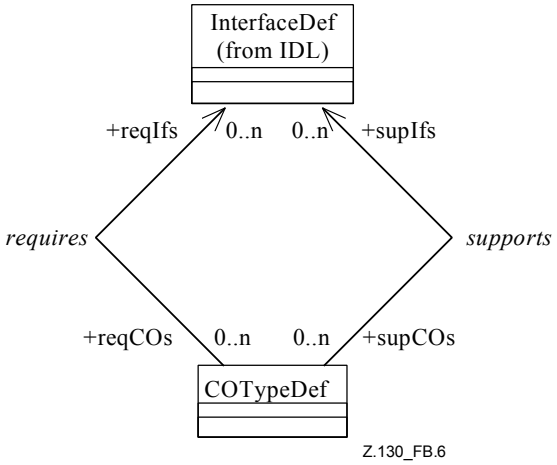


Figure B.6/Z.130 – Prendre en charge et requérir

Les productions **(14)**, **(15)** et **(16)** de la syntaxe concrète sont mappées avec l'élément *COTypeDef* du modèle (voir Figure B.6).

```
(14)  <object_template>          ::=  <object_template_header> "{" <object_template_export> "}"
(15)  <object_template_header>::=  "CO" <identifier> [ <object_inheritance_spec> ]
(16)  <object_inheritance_spec>::=  ":" <scoped_name> { "," <scoped_name> }*
(17)  <object_template_export> ::=  <object_export>*
(18)  <object_export>            ::=  <export>
                                 |    <reqrd_interf_templates> ";"
                                 |    <suptd_interf_templates> ";"
                                 |    <use_dcl> ";"
                                 |    <provide_dcl> ";"
                                 |    <implements_dcl> ";"
                                 |    <state_def_dcl> ";"
```

| | **<constraint_dcl> ";"**

| | **<property_list>**

(19) **<reqrd_interf_templates> ::= "requires" <scoped_name> { "," <scoped_name> }***

(20) **<supd_interf_templates> ::= "supports" <scoped_name> { "," <scoped_name> }***

Les productions **(14)**, **(15)** et **(16)** sont mappées avec un élément *COTypeDef* du modèle, **<identifier>** dans la production **(15)** étant le nom qui correspond au concept Named de *COTypeDef*. En tant que spécialisation de *InterfaceDef*, tous les *COTypeDef* qui sont en relation d'héritage avec le *COTypeDef* courant sont énumérés dans la production **(16)**. Les productions **(17)** et **(18)** expriment le concept de contenance pour ce *COTypeDef*. L'utilisation de **<export>** dans la production **(18)** est traitée de la même façon que pour *InterfaceDef* en IDL. En outre, **<reqrd_interf_templates>** et **<supd_interf_templates>** sont des éléments que *COTypeDef* peut contenir. Dans les productions **(19)** et **(20)**, **<scoped_name>** doit se rapporter à des éléments *InterfaceDef* ou *EnhancedInterfaceDef* du modèle. Ces éléments d'interface sont en relation requérir ou prendre en charge avec l'élément *COTypeDef* contenant.

B.8 Port provisionné et port utilisé

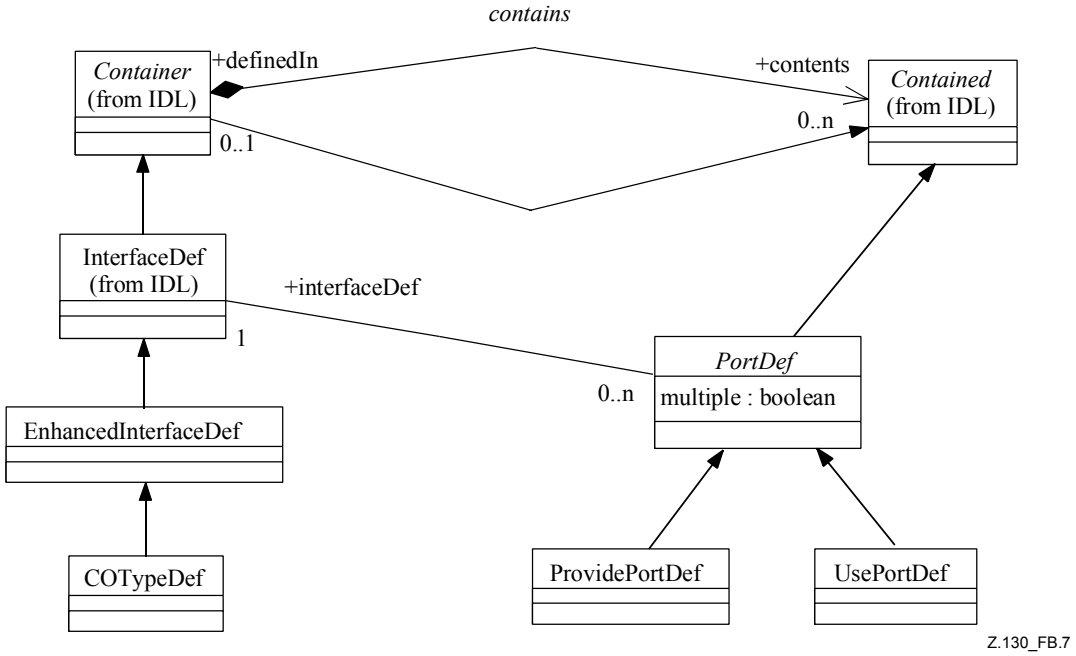


Figure B.7/Z.130 – Port provisionné et port utilisé

Les productions **(21)/(22)** de la syntaxe concrète sont mappées avec les éléments *UsePortDef/ProvidePortDef* du modèle (voir Figure B.7).

(21) **<use_dcl> ::= "use" ["multiple"] <scoped_name> <identifier>**

(22) **<provide_dcl> ::= "provide" ["multiple"] <scoped_name> <identifier>**

Les **ports utilisés** et **provisionné** sont exprimés au moyen des productions **(21)** et **(22)**. Ils se traduisent dans le modèle par les éléments *UsedPortDef* et *ProvidePortDef*. Dans les deux productions, **<scoped_name>** doit se rapporter à des éléments *InterfaceDef* ou *EnhancedInterfaceDef* du modèle. Si "multiple" est utilisé dans la syntaxe concrète, le champ booléen multiple figurant dans le *PortDef* courant est à Vrai.

B.9 Artefact et schéma d'instanciation

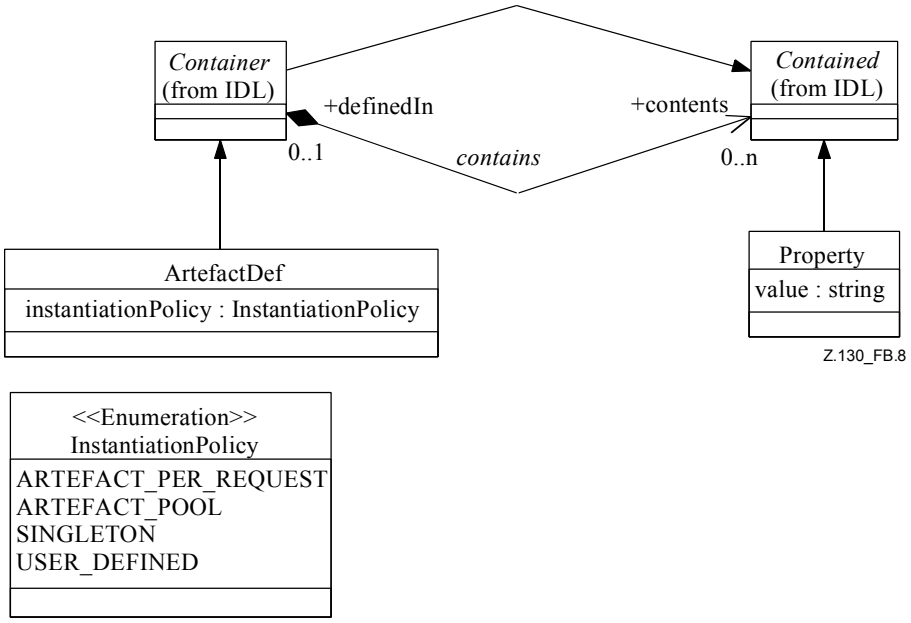


Figure B.8/Z.130 – Artefact et schéma d'instanciation

(23)	**<artefact>**	**::=**	**<artefact_dcl>**
		|	**<artefact_forward_dcl>**
(24)	**<artefact_forward_dcl>**	**::=**	**"artefact" <identifier>**
(25)	**<artefact_dcl>**	**::=**	**<artefact_header> "{" <artefact_body> "}"**
(26)	**<artefact_header>**	**::=**	**"artefact" <identifier> [<artefact_inheritance_spec>]**
(27)	**<artefact_inheritance_spec>**	**::=**	**":" <scoped_name> { "," <scoped_name> }***
(28)	**<artefact_body>**	**::=**	**<impl_elem_dcl>***

Les productions **(23)** et **(24)** permettent de se conformer à la syntaxe de l'IDL, **<identifier>** utilisé dans la production **(24)** ayant un *ArtefactDef* correspondant avec ce nom. Les productions **(25)**, **(26)** et **(27)** sont en relation avec un élément *ArtefactDef* du modèle, **<identifier>** dans la production **(26)** étant le nom qui correspond au concept Named. Tous les **<scoped_name>** énumérés dans la production **(27)** doivent se rapporter à des éléments *ArtefactDef* du modèle qui sont en relation d'héritage avec le *ArtefactDef* courant. Comme le montre la production **(28)**, seul *ImplementationElementDef* peut être contenu dans *ArtefactDef* (voir Figure B.8).

B.10 Relation implémenter

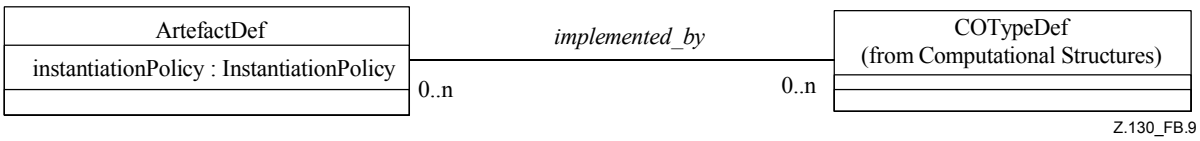


Figure B.9/Z.130 – Relation implémenter

(29)	**<implements_dcl>**	**::=**	**"implemented" "by" <artefact_with_policy>**
			{ "," <artefact_with_policy> }*
(30)	**<artefact_with_policy>**	**::=**	**<scoped_name> ["with" <instantiation_policy_dcl>]**

(31) **<instantiation_policy_dcl>** **::=** **"ArtefactPool"**

| **"ArtefactPerRequest"**

| **"Singleton"**

| **"UserDefined"**

Les productions **(29)** et **(30)** sont en relation avec un élément *ArtefactDef* du modèle (voir Figure B.9). Elles expriment la relation *implemented_by* du modèle. Dans la production **(30)**, **<scoped_name>** doit uniquement se rapporter à un élément *ArtefactDef*. La production **(31)** de la syntaxe concrète exprime le champ instantiationPolicy de l'élément *ArtefactDef* contenant. Les mots clés se rapportent directement aux éléments d'énumération du champ.

B.11 Elément d'implémentation

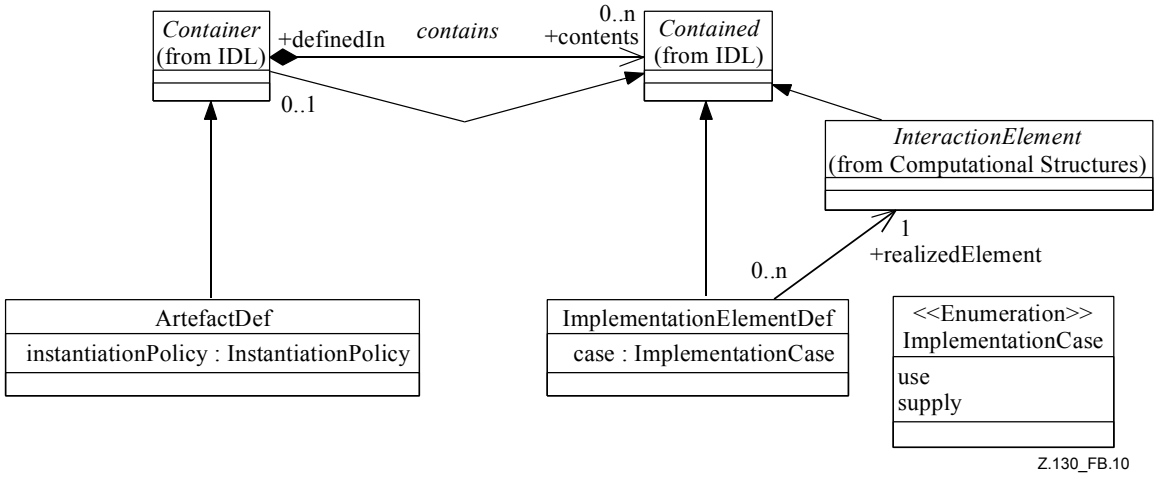


Figure B.10/Z.130 – Elément d'implémentation

(32) **<impl_elem_dcl>** **::=** **<identifier> "implements" <impl_case_dcl> <scoped_name> ";"**

(33) **<impl_case_dcl>** **::=** **"supply" | "use"**

Les productions **(32)** et **(33)** sont en relation avec un élément *ImplementationElementDef* du modèle (voir Figure B.10). Dans la production **(32)**, **<identifier>** est le nom qui correspond au concept Named. *ImplementationElementDef* peut uniquement être contenu dans *ArtefactDef*. La production **(33)** de la syntaxe concrète exprime le champ case de l'élément *ImplementationElementDef* contenant. Les mots clés se rapportent directement aux éléments d'énumération du champ.

B.12 Composant logiciel

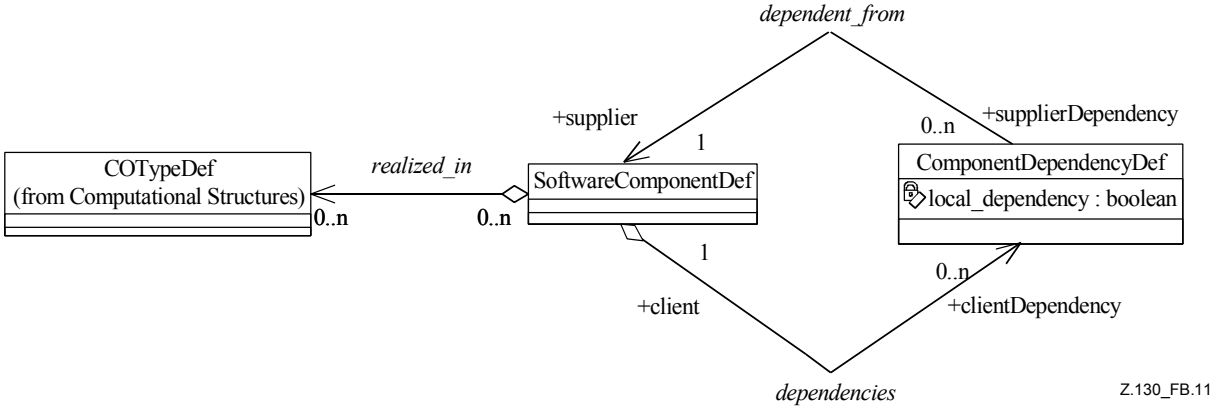


Figure B.11/Z.130 – Composant logiciel

La production **(34)** de la syntaxe concrète est mappée avec l'élément *SoftwareComponentDef* du modèle (voir Figure B.11).

(34) **<softwarecomponent_dcl>** **::=** **<softwarecomponent_header>**
"{" <softwarecomponent_body> "}"

(35) **<softwarecomponent_header>** **::=** **"softwarecomponent" <identifier> "realizes"**
<cotype_identifier_list>

(36) **<cotype_identifier_list>** **::=** **<cotype_identifier> { "," <cotype_identifier> }***

(37) **<softwarecomponent_body>** **::=** **<softwarecomponent_stmt>***

(38) **<softwarecomponent_stmt>** **::=** **"dependent" "{" <softwarecomponent_list> "}" ";"**
| **"requires" "{" <property_list> "}" ";**

(39) **<softwarecomponent_list>** **::=** **<softwarecomponent_identifier>**
{ "," <softwarecomponent_identifier> }*

Les productions **(34)**, **(35)** et **(36)** sont en relation avec un élément *SoftwareComponentDef* du modèle, **<identifier>** figurant dans la production **(35)** étant le nom qui correspond au concept Named. Tous les **<scoped_name>** énumérés dans la production **(36)** doivent se rapporter à des éléments *COTypeDef* du modèle qui sont en relation *realized_in* avec le *SoftwareComponentDef* courant. Les productions **(37)**, **(38)** et **(39)** sont en relation avec un élément *SoftwareComponentDef* du modèle. Tous les **<scoped_name>** énumérés dans la production **(36)** doivent se rapporter à des éléments SoftwareComponentDef du modèle.

Le **<softwarecomponent_identifier>** est un **<scoped_name>** qui se rapporte uniquement à un élément *SoftwareDependencyDef* du modèle.

B.13 Assemblage et configuration initiale

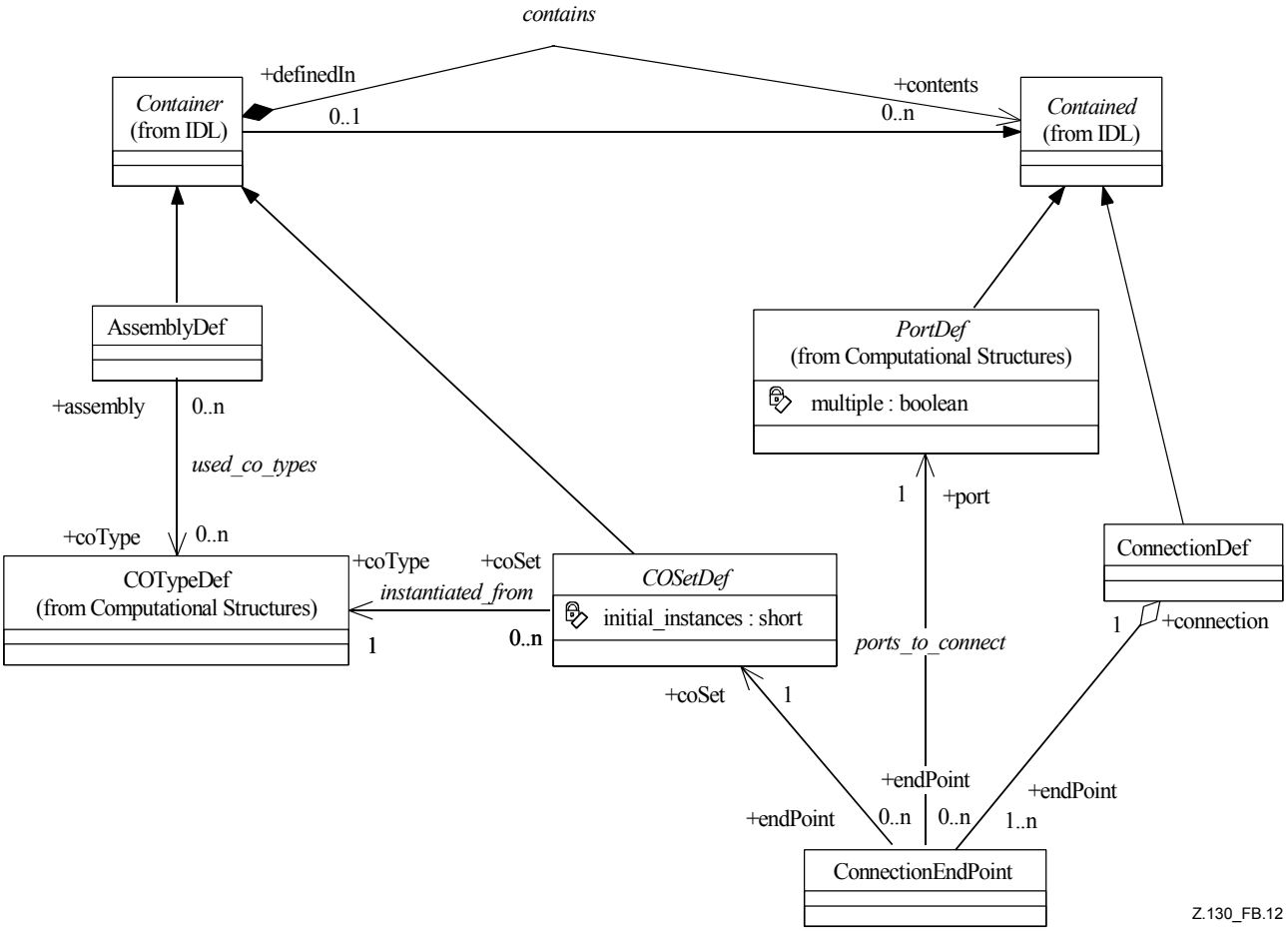


Figure B.12/Z.130 – Assemblage et configuration initiale

(40)	**<assembly_dcl>**	**::=**	**<assembly_header> "{" <assembly_body> "}"**
(41)	**<assembly_header>**	**::=**	**"assembly" <identifier>**
(42)	**<assembly_body>**	**::=**	**<assembly_stmt>***
(43)	**<assembly_stmt>**	**::=**	**<instance_set_dcl>";"**
		|	**<connect_dcl> ";"**
		|	**<constraint_dcl> ";"**
		|	**<property_list>**
(44)	**<instance_set_dcl>**	**::=**	**<identifier> ["(" <integer_literal> ")"] ":" <cotype_identifier>**
(45)	**<connect_dcl>**	**::=**	**"connect" [<identifier>] "{" <connection_list> "}"**
(46)	**<connection_list>**	**::=**	**{ <connection> ";" } +**
(47)	**<connection>**	**::=**	**<instance_set_identifier> "." <port_identifier>**
			"=" <instance_set_identifier> "." <port_identifier>

Les productions **(40)** et **(41)** sont en relation avec un élément *AssemblyDef* du modèle (voir Figure B.12), **<identifier>** figurant dans la production **(41)** étant le nom qui correspond au concept Named. Tous les **<scoped_name>** énumérés dans la production **(44)** doivent se rapporter à des éléments *COTypeDef* du modèle qui sont en relation *realized_in* avec le *SoftwareComponentDef* courant. Les **<cotype_identifier>, <instance_set_identifier>** et **<port_identifier>** figurant dans les

productions **(45)**, **(46)** et **(47)** sont des **<scoped_name>** qui se rapportent uniquement à des éléments *COTypeDef*, *InstanceSetDef* et *PortDef* du modèle.

B.14 Contraintes et propriétés

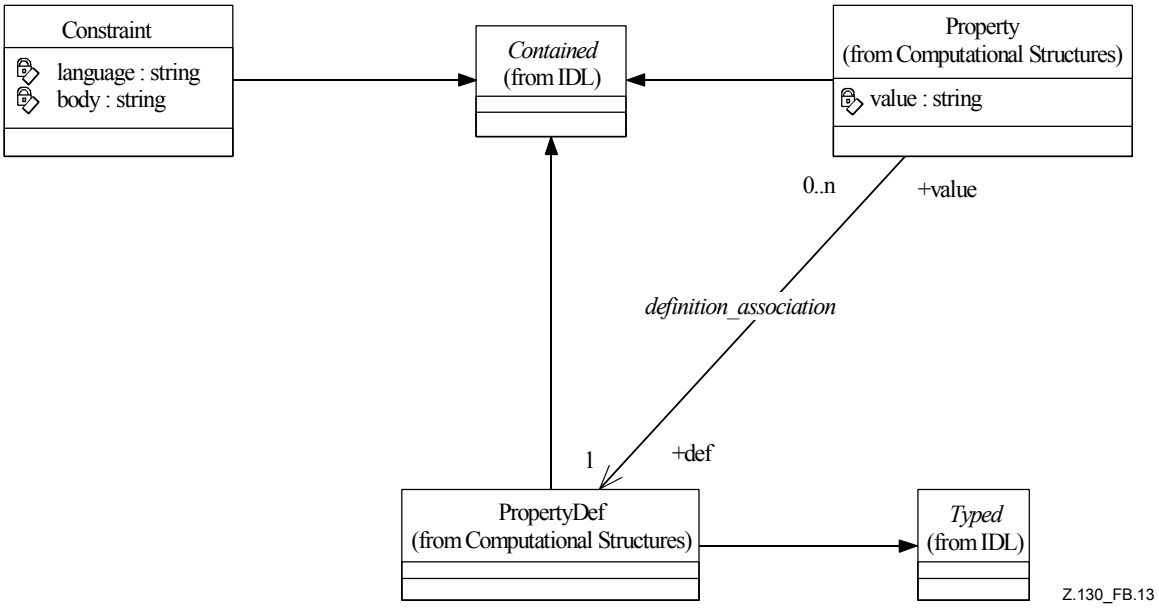


Figure B.13/Z.130 – Contraintes et propriétés

(48) **<property_dcl>** **::=** **"property" <property_name> "=" <property_value>**

(49) **<property_name>** **::=** **<identifier>**

(50) **<property_value>** **::=** **<simple_property_value>**
| **<structured_property_value>**
| **<sequence_property_value>**

(51) **<simple_property_value>** **::=** **<string_literal>**
| **<integer_literal>**
| **<boolean_literal>**

(52) **<structured_property_value>::=** **"{" <property_assign>* "}"**

(53) **<sequence_property_value>::=** **"[" <property_value>* "]"**

(54) **<property_assign>** **::=** **<property_name> "=" <property_value> ";"**

(55) **<constraint_dcl>** **::=** **"constraint" <identifier> "{" <constraint_body> "}"**

(56) **<constraint_body>** **::=** **"language" "=" <string_literal> "body" "=" <string_literal> ";"**

Les productions **(48)** et **(49)** sont en relation avec un élément *PropertyDef* du modèle (voir Figure B.13), **<identifier>** dans la production **(49)** étant le nom qui correspond au concept Named. La production **(54)** est mappée avec un élément *Property* du modèle. Les productions **(50)**, **(51)**, **(52)** et **(53)** de la syntaxe textuelle concrète sont utilisées pour fournir des valeurs pour le *champ* value.

La production **(55)** est en relation avec un élément *Constraint* du modèle, **<identifier>** figurant dans la production **(55)** étant le nom qui correspond au concept Named. La production **(56)** fournit des valeurs pour les champs *language* et *body* de l'élément *Constraint*.

B.15 Environnement cible, nœud et liaison entre nœuds

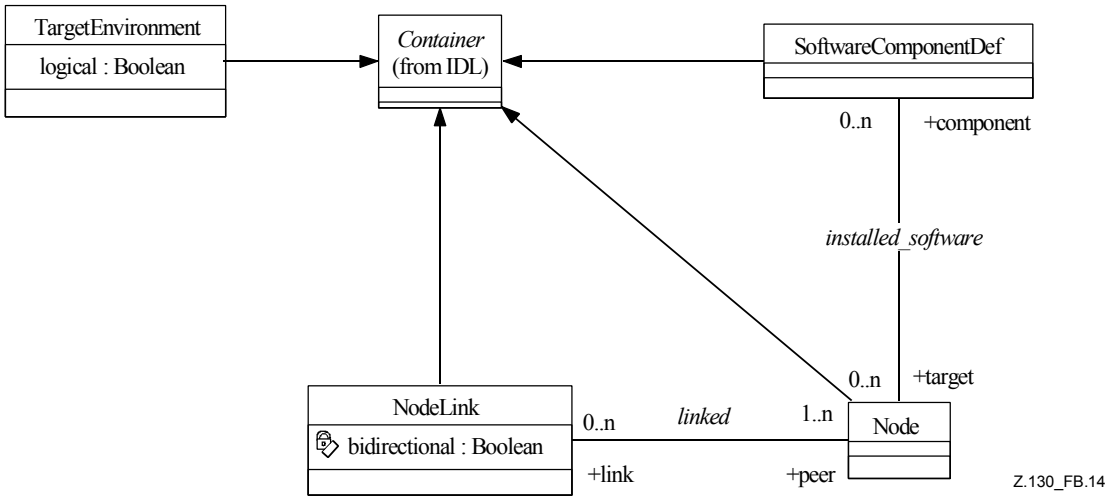


Figure B.14/Z.130 – Environnement cible, nœud et liaison entre nœuds

(57)	**<environment_dcl>**	**::=**	**<environment_header> "{" <environment_body> "}"**
(58)	**<environment_header>**	**::=**	**"environment" <identifier>**
(59)	**<environment_body>**	**::=**	**<environment_stmt>+**
(60)	**<environment_stmt>**	**::=**	**<node_dcl> ";"**
		|	**<link_dcl> ";"**
(61)	**<node_dcl>**	**::=**	**"node" <identifier> "{" <property_list> "}"**
(62)	**<link_dcl>**	**::=**	**<link_header> "{" <link_body> "}"**
(63)	**<link_header>**	**::=**	**"link" <identifier>**
(64)	**<link_body>**	**::=**	**"node" <node_list> ";" <property_list>**
(65)	**<node_list>**	**::=**	**<node_identifier> { "," <node_identifier> }+**

Les productions **(57)**, **(58)**, **(59)** et **(60)** sont en relation avec un élément *TargetEnvironment* du modèle (voir Figure B.14), **<identifier>** figurant dans la production **(58)** étant le nom qui correspond au concept Named. La production **(61)** est mappée avec un élément *Node* du modèle, **<identifier>** figurant dans la production **(61)** étant le nom qui correspond au concept Named. Les productions **(62)**, **(63)** et **(64)** sont en relation avec un élément *NodeLink* du modèle, **<identifier>** figurant dans la production **(63)** étant le nom qui correspond au concept Named. **<node_identifier>** figurant dans la production **(65)** est un **<scoped_name>** qui doit se rapporter à un élément *Node* du modèle.

B.16 Carte d'installation

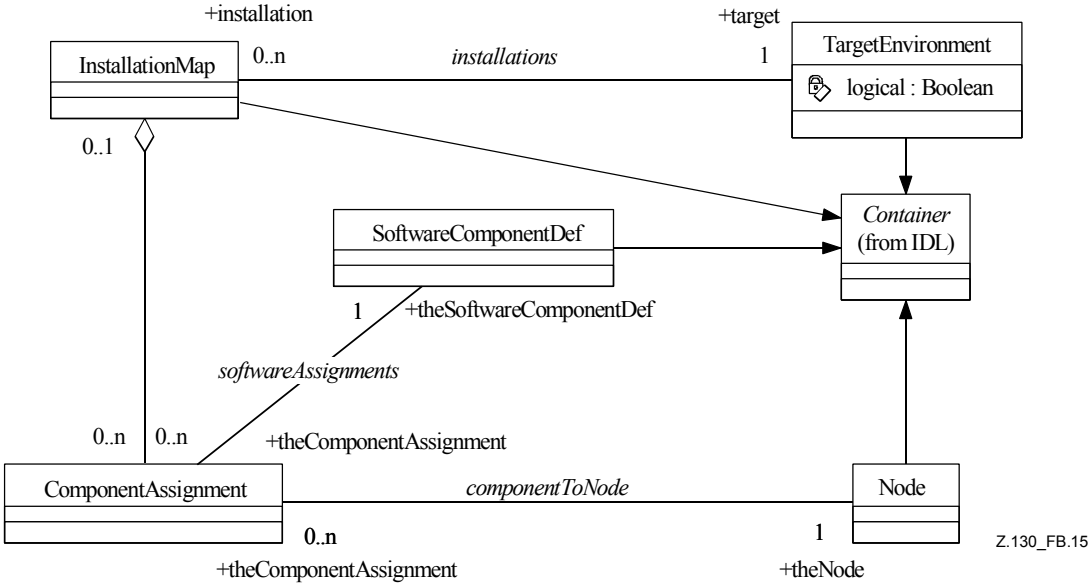


Figure B.15/Z.130 – Carte d'installation

(66) **<installation_map_dcl>** **::=** **<installation_map_header>**
"{" <installation_map_body> "}"

(67) **<installation_map_header>** **::=** **"installation" <identifier> "uses" "environment"**
<environment_identifier>

(68) **<installation_map_body>** **::=** **<install_stmt>***

(69) **<install_stmt>** **::=** **<softwarecomponent_identifier> "->" <node_identifier> ";"**

Les productions **(66)**, **(67)** et **(68)** sont en relation avec un élément *InstallationMap* du modèle (voir Figure B.15), **<identifier>** figurant dans la production **(67)** étant le nom qui correspond au concept Named. **<environment_identifier>** figurant dans la production **(67)** est un **<scoped_name>** qui se rapporte uniquement à un élément *TargetEnvironment* du modèle. **<softwarecomponent_identifier>** et **<node_identifier>** figurant dans la production **(69)** sont des **<scoped_name>** qui se rapportent uniquement à des éléments *SoftwareComponentDef* et *Node* du modèle.

B.17 Carte d'instanciation

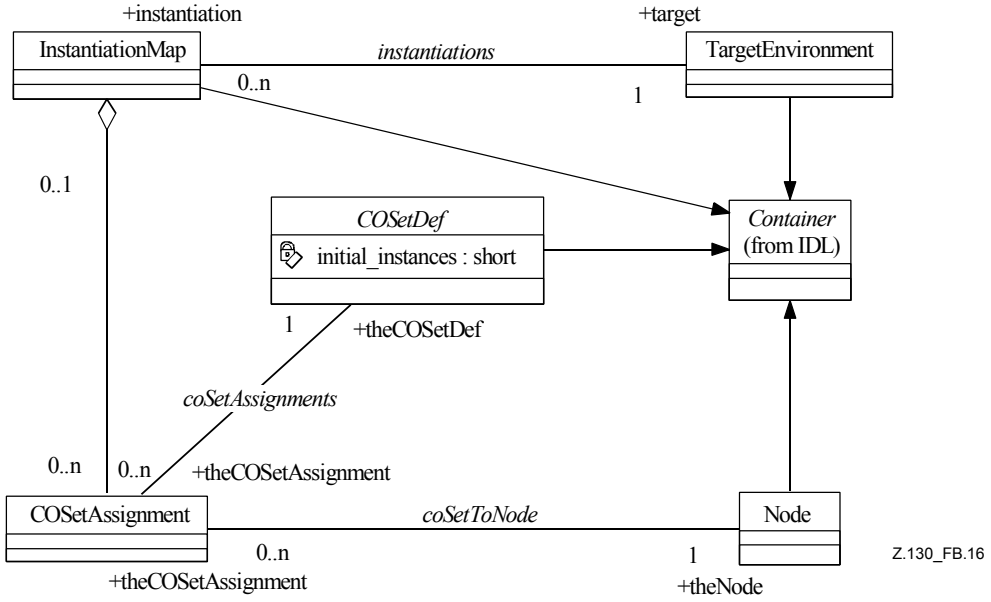


Figure B.16/Z.130 – Carte d'instanciation

(**70**)	**<instantiation_map_dcl>**	**::=**	**<instantiation_map_header>**
			"{" <instantiation_map_body> "}"
(71)	**<instantiation_map_header>**	**::=**	**"instantiation" <identifier>**
			<instantiation_map_header_env>
			<instantiation_map_header_ass>
(72)	**<instantiation_map_header_env>**	**::=**	**"uses" "environment" <environment_identifier>**
(73)	**<instantiation_map_header_ass>**	**::=**	**"uses" "assembly" <assembly_identifier>**
(74)	**<instantiation_map_body>**	**::=**	**<assign_instance_stmt>***
(75)	**<assign_instance_stmt>**	**::=**	**<instance_set_identifier_list> "->"**
			<node_identifier> ";"
(76)	**<instance_set_identifier_list>**	**::=**	**<instance_set_identifier>**
			{ "," <instance_set_identifier> }*

Les productions **(70)**, **(71)**, **(72)**, **(73)** et **(74)** sont en relation avec un élément *InstantiationMap* du modèle (voir Figure B.16), **<identifier>** figurant dans la production **(71)** étant le nom qui correspond au concept Named. **<environment_identifier>** figurant dans la production **(72)** et **<assembly_identifier>** figurant dans la production **(73)** sont des **<scoped_name>** qui se rapportent uniquement à des éléments *TargetEnvironment* et *AssemblyDef* du modèle. **<node_identifier>** figurant dans la production **(75)** est un **<scoped_name>** qui se rapporte uniquement à un élément *Node* du modèle, cet élément étant contenu dans *TargetEnvironment* qualifié par la production **(72)**. Les **<instance_set_identifier>** figurant dans la production **(76)** sont des **<scoped_name>** qui se rapportent uniquement à des éléments *COSetDef* du modèle, ces éléments étant contenus dans *AssemblyDef* qualifié par la production **(73)**.

B.18 Plan de déploiement

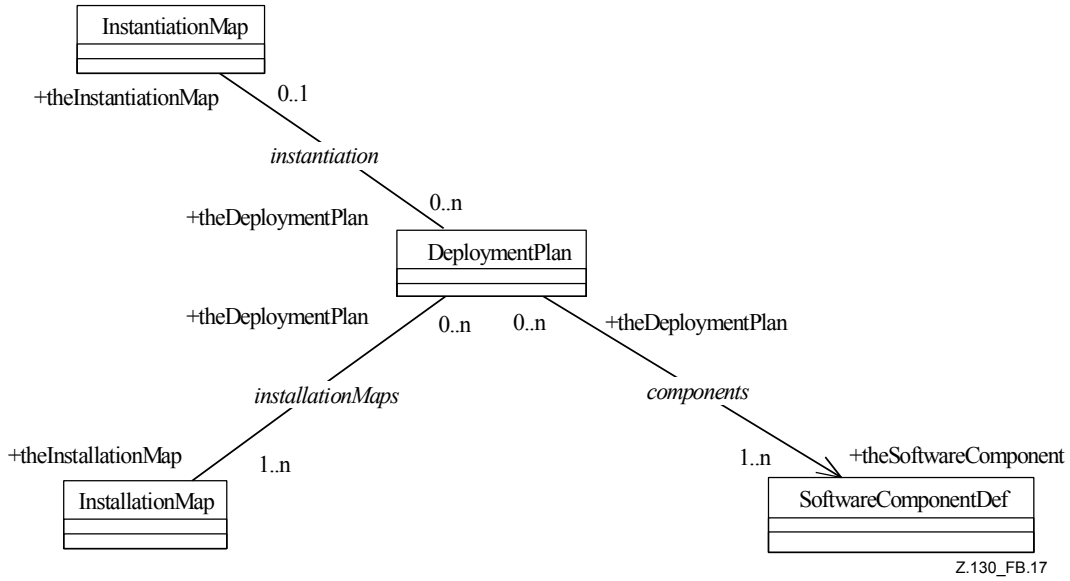


Figure B.17/Z.130 – Plan de déploiement

(77) **<deployment_action>** **::=** **"deploy" "{" <deployment_body> "}" ";"**

(78) **<deployment_body>** **::=** **"install" "{" <install_list> "}" ";" <instantiation_action>+**

(79) **<install_list>** **::=** **<install_member>***

(80) **<install_member>** **::=** **<installation_map_identifier> ";"**

(81) **<instantiation_action>** **::=** **"instantiate" <instantiation_map_identifier> ";"**

La production **(77) <deployment_action>** de la syntaxe concrète est mappée avec l'élément *DeploymentPlan* du modèle (voir Figure B.17). Les listes qui figurent dans le corps de **<deployment_action>**, construites par les productions **(78)**, **(79)**, **(80)** et **(81)**, expriment les relations *InstallationMaps* et instantiation du modèle. **<instantiation_map_identifier>** et **<installation_map_identifier>** sont des **<scoped_name>** qui se rapportent à des éléments *InstallationMap* et *InstantiationMap* du modèle. Chacun des identificateurs énumérés correspond à une relation dans le modèle.

B.19 Type externe

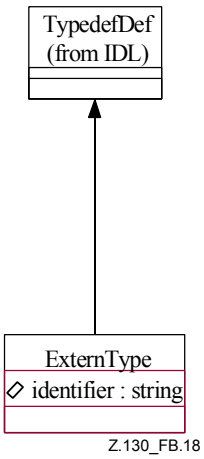


Figure B.18/Z.130 – Type externe

(82) <extern_type> ::= "extern" "type" <identifier> <string_literal>";"

La production **(82) <extern_type>** de la syntaxe concrète est mappée avec l'élément *ExternType* du modèle (voir Figure B.18). La valeur de **<string_literal>** est mappée directement avec l'attribut *identifier* de l'élément du modèle. **<identifier>** figurant dans la production **(82)** est mappé avec l'attribut de nom du concept *Contained*.

Annexe C

Mappage avec le SDL-2000

C.1 Introduction

Le langage de définition d'objet étendu spécifié par l'UIT (UIT-eODL) permet de décrire des systèmes répartis orientés composant. La présente annexe expose un mappage de l'UIT-eODL avec le SDL-2000. Ce mappage permet de générer automatiquement un code SDL-2000 sur la base d'une description donnée en UIT-eODL. On utilise la représentation textuelle (SDL/PR) du SDL-2000 [8].

Le mappage de l'UIT-eODL avec le SDL-2000 permet aux utilisateurs de générer automatiquement un squelette SDL-2000 sur la base d'un modèle eODL donné. Elle prend en charge la quasi-totalité des concepts de traitement et d'implémentation. La seule **anomalie** importante est la non-prise en charge du concept de **média continu**. Par ailleurs, les concepts de **déploiement** et d'environnement cible ne sont pas pris en charge.

Les types de l'eODL sont mappés avec les types appropriés du SDL-2000. Les concepts qui définissent des aspects comportementaux des types sont mappés avec des agents SDL implémentés automatiquement. A partir d'un modèle eODL complet, un outil de mappage génère un squelette SDL-2000. L'utilisateur doit implémenter la logique commerciale et peut utiliser les **CO** définis en SDL-2000 comme modules pour un système SDL-2000.

Le mappage vise à prendre en charge autant de concepts que possible même si cela passe par la production de structures quelque peu compliquées en SDL-2000. L'héritage multiple des **CO** est par exemple pris en charge au prix d'une plus grande complexité des structures et d'une moins grande efficacité du comportement, car le SDL-2000 ne prend pas en charge l'héritage multiple pour les types d'agent.

C.2 Paquetage eodl

Le mappage de l'UIT-eODL avec le SDL-2000 définit le paquetage SDL `eodl`. Ce paquetage contient les définitions des **types de données** qui sont utilisés par les modèles générés à partir de modèles eODL ainsi que les définitions des types dits prédéfinis. Le paquetage `eodl` figure en totalité au § C.10.

Les règles lexicales de la notation textuelle de l'eODL sont héritées du CORBA-IDL de l'OMG. En CORBA-IDL, les identificateurs non qualifiés commencent par un caractère alphabétique suivi par un nombre quelconque de caractères alphabétiques, chiffres ou soulignés, les caractères alphabétiques étant les caractères de 'A' à 'Z' majuscules et minuscules. Par ailleurs, dans les identificateurs eODL, les caractères majuscules et minuscules sont équivalents.

Conformément aux règles lexicales du SDL-2000, tous les identificateurs eODL (non qualifiés) sont des identificateurs SDL-2000.

Les identificateurs qualifiés sont abordés dans le paragraphe C.4 sur les "noms visibles".

C.3 Structure

Chaque fichier en UIT-eODL est mappé avec deux paquetages SDL:

- `<name>_interface` (appelé "paquetage d'interface");
- `<name>_definition` (appelé "paquetage de définition")

où `<name>` est le nom de la spécification en UIT-eODL (par exemple le nom du fichier). Voir Figure C.1.

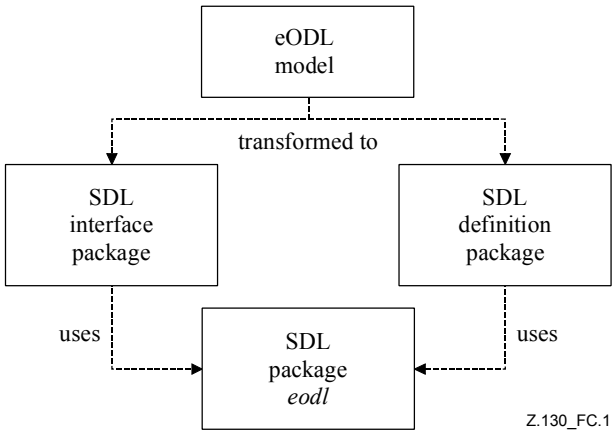


Figure C.1/Z.130 – Structure de transformation d'un modèle eODL en SDL

Le paquetage d'interface contient toutes les informations intéressant à la fois le côté client et le côté serveur du système, à savoir:

- les définitions des types de données;
- les définitions des constantes;
- les définitions d'interface représentant les **interfaces** ordinaires et les **interfaces** des **CO**.

Le paquetage de définition contient les squelettes pour le côté serveur du système, à savoir:

- les types de bloc représentant les **types de CO**;
- les types de processus représentant les **artefacts**.

C.4 Noms visibles

La qualification est un concept qui existe aussi bien en eODL qu'en SDL-2000. Le mappage est donc canonique: les noms qualifiés en eODL sont mappés avec des noms qualifiés en SDL.

C.5 Mappage des concepts de traitement

C.5.1 Modules

Un module eODL est un conteneur pour tous les autres éléments eODL et ouvre un espace de noms. Ce concept est mappé avec le concept de paquetage du SDL.

Le paquetage SDL avec lequel un module eODL est mappé peut être contenu soit dans le paquetage d'interface SDL soit dans le paquetage de définition SDL soit dans les deux paquetages, suivant les entités qui figurent dans le module eODL.

C.5.2 Définitions de type

La construction eODL `typedef` attribue un nom (différent) à un type donné. Elle est mappée avec la construction `syntype` du SDL.

C.5.3 Types de données prédéfinis

C.5.3.1 Types de données pour les nombres entiers

Les types de données eODL `unsigned short`, `unsigned long` et `unsigned long long` sont mappés avec des sortes SDL définies dans le paquetage `eodl`, comme suit:

- le type eODL `unsigned short` est mappé avec une sorte `ushort` de type `Integer` qui va de 0 à $2^{16} – 1$;
- le type eODL `unsigned long` est mappé avec une sorte `ulong` de type `Integer` qui va de 0 à $2^{32} – 1$;
- le type eODL `unsigned long long` est mappé avec une sorte `ulong_long` de type `Integer` qui va de 0 à $2^{64} – 1$.

Les types de données eODL `signed short`, `signed long` et `signed long long` sont mappés avec des sortes SDL, comme suit:

- le type eODL `signed short` est mappé avec une sorte `short` de type `Integer` qui va de $–2^{15}$ à $2^{15} – 1$;
- le type eODL `signed long` est mappé avec une sorte long de type `Integer` qui va de $–2^{31}$ à $2^{31} – 1$;
- le type eODL `unsigned long long` est mappé avec une sorte `long_long` de type `Integer` qui va de $–2^{63}$ à $2^{63} – 1$.

C.5.3.2 Types de données pour les nombres à virgule flottante

Les types eODL à virgule flottante `float`, `double` et `long double` sont mappés avec la sorte SDL prédéfinie `Real`. Il est à noter que `Real` n'est pas conforme à la norme IEEE 754 [15] (voir la Rec. UIT-T Z.100) alors que les types eODL à virgule flottante le sont.

C.5.3.3 Types de données pour les caractères

Le type eODL `char` est mappé avec la sorte SDL prédéfinie `Character`. Le type eODL `wchar` est mappé avec la sorte SDL prédéfinie `Natural`.

C.5.3.4 Type de données booléen

Le type eODL `boolean` est mappé avec la sorte SDL `Boolean` et les constantes `TRUE` et `FALSE` de ce type sont respectivement mappées avec les littéraux `true` et `false` de cette sorte.

C.5.3.5 Type de données octet

Le type eODL `octet` est mappé avec la sorte SDL `Octet`.

C.5.3.6 Type de données quelconque

Le type eODL `any` est mappé avec la sorte SDL `Any`. Il est à noter que la sémantique est identique dans les deux langages (voir le § 3.10.1.7 de la version 2.4.2 de l'IDL de l'OMG).

C.5.3.7 Identification de type au moyen de TypeCode

L'attribut `typeCode` d'une instance de *IDLType* dans le métamodèle de l'eODL n'a pas de correspondant en SDL. Toutefois, le type `TypeCode` est mappé avec la sorte SDL `TypeCode` dans le paquetage SDL. La sorte `TypeCode` est un type de données abstrait. Autrement dit, il n'y a pas de valeur de ce type de données conformément à ce mappage. Néanmoins, aucune restriction n'est imposée aux implémentations quant à l'utilisation d'un type de données concret dérivé.

C.5.4 Types de données construits

C.5.4.1 Enumérations

Les énumérations en eODL sont mappées avec des types de données valeur SDL ne contenant que des littéraux.

C.5.4.2 Structures

Les structures en eODL sont mappées avec des types de données valeur SDL ayant un constructeur de type de données structure publique. Les membres d'une structure eODL sont les champs du type de données SDL.

C.5.4.3 Unions

Les unions en eODL sont mappées avec des types de données valeur imbriquée SDL. Le type de données valeur extérieure en SDL a un constructeur de type de données structure publique comportant deux champs:

1) un champ `tag` qui représente le discriminateur de l'union eODL;
2) un champ `union` qui représente l'union proprement dite.

Le champ union a pour type de données valeur `<name-of-eODL-union>_union`. Ce type est déclaré dans le cadre du type d'union SDL extérieur. Il a un constructeur de type de données choix public et sa liste de choix représente les membres de l'union eODL.

C.5.4.4 Matrices

Les matrices en eODL sont mappées avec une sorte SDL qui hérite de la sorte SDL prédéfinie `Vector`.

Les matrices multidimensionnelles sont mappées avec un type de données valeur SDL qui prend en charge les opérateurs `Make`, `Modify` et `Extract`.

C.5.5 Types de valeur

Les types de valeur en eODL sont mappés avec des types de données objet SDL ayant un constructeur de type de données structure. Tous les membres d'un type de valeur eODL sont les champs du type de données SDL.

Si un type de valeur eODL est déclaré abstrait, le type de données objet SDL correspondant est également déclaré abstrait et il n'a pas de constructeur de type de données.

L'héritage simple de types de valeur en eODL est mappé avec l'héritage simple de types de données en SDL. L'héritage multiple est autorisé pour les types de valeur abstraits en eODL. Pour accomplir un héritage multiple en SDL, on copie la déclaration d'opération des types de données de base vers le type de données dérivé.

Les types de valeur en eODL peuvent avoir des fabriques (éléments d'initialisation). Celles-ci sont mappées avec des opérateurs SDL qui retournent une valeur du type de données. Si une et une seule fabrique est déclarée, l'opérateur `Make` est automatiquement implémenté par l'appel de cette fabrique. Dans les autres cas, il n'est pas implémenté exactement.

Si le type de valeur eODL prend en charge une interface, les **opérations** de cette interface deviennent des opérations du type de données SDL. Tout attribut d'une interface prise en charge est mappé avec un couple d'opérations get/set et un champ acheminant l'attribut.

Les types de valeur encadré sont mappés de la même manière que les types de valeur (concrets).

C.5.6 Types de données paramétrés

C.5.6.1 Séquences

Les séquences en eODL sont mappées avec la sorte SDL `Vector` si elles sont limitées ou avec la sorte SDL `Array` dans les autres cas.

C.5.6.2 Chaînes de caractères

Le type eODL `string` est mappé avec la sorte SDL prédéfinie `Charstring`.

Le type eODL `wstring` est mappé avec la sorte SDL `wstring` dans le paquetage `eodl`. Cette sorte hérite de `String<Natural>`. Une opération est par ailleurs ajoutée par rapport à `Charstring` pour convertir une valeur `Charstring` en valeur `wstring`.

C.5.6.3 Nombres à virgule fixe

Le paquetage SDL `eodl` contient un type `fixedpt` qui est paramétré par des **paramètres** de largeur et d'échelle. Tous deux sont de la sorte `Natural`. La sémantique des paramètres est identique à celle des paramètres en eODL.

Le type `fixedpt` définit les opérateurs "+", "-", "*", "/" qui font une addition, une soustraction, une multiplication et une division entre deux valeurs de type `fixedpt` et renvoient une valeur de type `fixedpt`. Par ailleurs, l'opérateur "=" compare deux valeurs de type `fixedpt` et renvoie une valeur de type `Boolean`: vrai si les deux opérandes représentent le même nombre et faux si les deux opérandes représentent des nombres différents.

La méthode `toReal` permet de convertir une valeur `fixedpt` en valeur `Real`. Il est possible de construire une valeur `fixedpt` à partir d'une valeur `Real` grâce à l'opérateur `Make`.

C.5.7 Constantes, littéraux de types de données et expressions constantes

C.5.7.1 Constantes

Les constantes eODL sont mappées avec des synonymes SDL.

C.5.7.2 Littéraux de types de données

Le présent paragraphe énumère les littéraux de types de données qui ne sont pas mappés avec des sortes SDL prédéfinies.

C.5.7.2.1 Littéraux de types d'entier

Les littéraux entiers décimaux et hexadécimaux sont pris en charge en SDL: les littéraux entiers décimaux sont pris en charge par la sorte SDL `Integer` et les littéraux entiers hexadécimaux peuvent être écrits sous forme de littéraux de type SDL `Bitstring` puis convertis en `Integer`. Les littéraux entiers octaux doivent être convertis en littéraux décimaux ou hexadécimaux.

C.5.7.2.2 Littéraux de types de caractère

Un littéral de type eODL `char` est mappé de telle sorte qu'il forme un littéral autorisé de la sorte SDL prédéfinie `Character`. Si le littéral eODL est un caractère dont la valeur est supérieure à 127, il ne peut pas être mappé. Il est recommandé d'utiliser le type `wchar` à la place.

Un littéral de type eODL `wchar` est mappé de telle sorte qu'il forme un littéral `Natural` autorisé.

C.5.7.2.3 Littéraux de chaînes de caractères

Un littéral de type eODL `string` est mappé de telle sorte qu'il forme un littéral autorisé de la sorte SDL prédéfinie `Charstring`. Si le littéral eODL contient un caractère dont la valeur est supérieure à 127, il ne peut pas être mappé. Il est recommandé d'utiliser le type `wstring` à la place.

Un littéral de type eODL `wstring` est mappé de telle sorte qu'il forme un littéral autorisé de la sorte SDL `wchar`. Cela inclut la possibilité de convertir un littéral `Charstring` en valeur SDL `wstring`.

C.5.7.2.4 Littéraux de type de nombre à virgule fixe

Les valeurs à virgule fixe ne peuvent être créées que par la conversion de nombres réels.

C.5.7.3 Expressions constantes

Les expressions constantes en eODL sont mappées avec des expressions constantes en SDL. Les expressions dans les deux langages doivent représenter les mêmes valeurs.

C.5.8 Types de signal

Les types de signal sont mappés avec des types de données valeur.

C.5.9 Anomalies

L'UIT-eODL et le SDL prennent tous deux en charge des **anomalies**. La seule différence est que les **anomalies** SDL ne nomment pas de membres. Les **anomalies** sont définies dans le paquetage d'interface.

C.5.10 Interfaces et éléments d'interaction

C.5.10.1 Interfaces

Une interface eODL englobe:

- des **opérations**;
- des flux de **signaux**;
- des interactions de type **média continu** (flux);
- des attributs.

Les interactions de type **média continu** n'ont pas de correspondant en SDL.

Une interface eODL `I` est mappée avec les interfaces SDL `exported_I` et `imported_I`. Celles-ci sont définies dans un paquetage SDL `I` dans le paquetage d'interface. L'interface `exported_I` contient:

- toutes les **opérations**;
- les **signaux** consommés.

tandis que l'interface `imported_I` contient les **signaux** produits.

Ce mappage est tel que le type d'interface `exported_<interface-name>` contient tout ce dont un client de l'interface a besoin pour invoquer un service que l'interface représente.

Elle est détaillée dans les paragraphes qui suivent.

C.5.10.2 Eléments d'interaction opération

Une opération eODL est mappée avec une procédure distante qui est déclarée en SDL dans l'interface `exported_<interface name>`. Les deux concepts prennent en charge au moins un type de retour, des paramètres qui peuvent être in, out ou inout ainsi que des **anomalies**. Les instances du concept "contexte" n'ont pas de correspondant.

C.5.10.3 Eléments d'interaction de type signal

Si une interface eODL déclare un **élément d'interaction** pour la consommation d'un **signal** `A`, l'interface SDL correspondante `exported_<interface-name>` déclare l'utilisation du **signal** `A`.

Si une interface eODL déclare un **élément d'interaction** pour la production d'un **signal** `A`, l'interface SDL correspondante `imported_<interface-name>` déclare l'utilisation du signal `A`.

C.5.10.4 Eléments d'interaction attribut

Les attributs en eODL sont mappés avec un couple de procédures distantes set/get dans l'interface SDL `exported_<interface-name>`. Si l'attribut est en lecture seule, la procédure distante set n'est pas générée.

C.5.10.5 Références d'interface

Un type d'interface SDL définit implicitement une sorte spéciale `PId`. Une instance de cette sorte `PId` sert de référence d'interface. Par ailleurs, ce concept assure une sécurité: un client qui possède un `PId` peut envoyer des signaux à un agent qui implémente cette interface et invoquer des procédures distantes au niveau de cet agent. Le type d'interface SDL qui sert de type de référence d'interface est `exported_<interface name>`.

C.5.10.6 Héritage

L'UIT-eODL et le SDL prennent en charge tous les deux l'héritage multiple d'interfaces. Le mappage est donc canonique.

C.5.11 Objets de traitement

En eODL, les **CO** encapsulent état et comportement. Ils offrent des interfaces (sous la forme de références d'interface) à l'environnement et peuvent utiliser des (références d') interfaces d'autres **CO**.

En SDL, un **CO** est un agent de type de processus. Chaque **CO** a trois portes appelées `initial` (porte entrante), `provides` et `uses`. Ces portes prennent en charge plusieurs interfaces (voir les paragraphes qui suivent). Les trois portes sont connectées à l'environnement du processus car elles constituent l'interface entre le **CO** et l'environnement.

Le type de processus du **CO** est défini dans le paquetage de définition.

Pour créer des processus de **CO**, une fabrique de **CO** est définie. Elle est représentée sous la forme d'un type de processus défini dans le même domaine de visibilité que le type de processus du **CO**. Le **type de CO** proprement dit et sa fabrique sont des types de processus définis dans un type d'agent de type bloc. Ce bloc est appelé SDL component. Il est à noter que le `SDL component` défini dans ce mappage est un concept du point de vue de traitement tandis que le *component* défini dans la présente Recommandation est un concept du point de vue **déploiement**. Un composant SDL représente un type de bloc avec une interface bien définie.

C.5.11.1 Interface d'un CO

Mis à part l'offre et l'utilisation d'interfaces, un **type de CO** expose sa propre interface. Celle-ci est constituée de trois composants: l'interface définie par l'utilisateur, l'interface d'identification de composant implicite et l'interface de configuration.

Le **métamodèle** définit un **type de CO** comme étant un type d'interface et limite les **éléments d'interaction** contenus à des instances de *AttributeDef*. Lorsque le **type de CO** est considéré uniquement comme une interface, on l'appelle "interface définie par l'utilisateur".

Pour l'identification des **CO**, chaque **type de CO** a une clé d'attribut implicite en lecture seule dont le type prédéfini est `ComponentKey`. Cet attribut est le seul **élément d'interaction** qui figure dans l'interface prédéfinie `ComponentBase`.

L'interface `ComponentBase` définit la procédure d'accès à la clé d'un **CO**. Cette clé implémente son identité et elle doit être unique au moins par rapport au composant SDL qui contient le **CO**. Le paquetage `eodl` contient la déclaration d'une procédure externe `compute_co_key` qui retourne une telle clé unique. Une implémentation de ce mappage doit fournir une implémentation de cette procédure ou générer un code qui calcule une clé unique par une autre méthode.

L'interface de configuration définit des procédures qui prennent en charge les opérations de **port**. Elle est présentée plus en détail dans le paragraphe sur le mappage des ports. L'interface de configuration hérite de l'interface prédéfinie `ConfigBase`, laquelle contient la déclaration des opérations de port génériques.

Toutes ces interfaces sont définies dans le paquetage d'interface. Elles sont contenues dans un paquetage portant le nom du **CO**. L'interface définie par l'utilisateur est appelée `<CO-name>_attributes`. Elle hérite de l'interface prédéfinie `ComponentBase`. L'interface de configuration est appelée `<CO-name>_config` et elle hérite de l'interface prédéfinie `ConfigBase`. On définit enfin l'interface `<CO-name>` qui hérite simplement de `<CO-name>_attributes` et `<CO-name>_config`.

L'interface `<CO-name>` est prise en charge par la porte `initial`. On définit un canal allant de l'environnement au type de processus de **CO**.

C.5.11.2 Interfaces prises en charge

La porte provides fait référence à toutes les interfaces que le **CO prend en charge**. C'est par le biais de cette porte que les clients peuvent utiliser le **CO**. L'`interface` SDL `exported_<name>` est prise en charge dans le sens allant de l'environnement au type de processus et l'interface SDL `imported_<name>` est prise en charge dans l'autre sens.

C.5.11.3 Interfaces requises

La porte `uses` fait référence à toutes les interfaces que le **CO requiert** et utilise. C'est par cette porte que le **CO** (dans le rôle d'un client) peut utiliser d'autres **CO**. L'interface SDL `imported_<name>` est prise en charge dans le sens allant de l'environnement au type de processus et l'interface SDL `exported_<name>` est prise en charge dans l'autre sens.

C.5.11.4 Héritage

Comme le SDL ne prend pas en charge l'héritage multiple, celui-ci est réalisé par délégation, comme suit:

1) l'interface définie par l'utilisateur du **type de CO** et l'interface de configuration du **type de CO** héritent des interfaces correspondantes du ou des super **CO**;
2) la porte `provides` prend en charge toutes les interfaces qui sont prises en charge par le ou les super **CO**. Elle prend également en charge toutes les interfaces qui sont prises en charge par le **type de CO** proprement dit;
3) la porte `uses` prend en charge toutes les interfaces qui sont requises par le ou les super **CO**. Elle prend également en charge toutes les interfaces qui sont requises par le **type de CO** proprement dit.

En ce qui concerne les aspects de comportement de l'héritage multiple, voir le § C.7.2.5.

C.5.11.5 Fabriques de CO

A chaque **type de CO** est associée une fabrique de **CO** `<CO-name>_factory`. Celle-ci implémente l'interface de fabrique, qui est définie dans un paquetage `<CO-name>_factory`, qui est lui-même défini dans le paquetage dans lequel les interfaces du **CO** sont définies. L'interface de fabrique hérite de l'interface prédéfinie `CoFactoryBase`.

L'interface `CoFactoryBase` contient les procédures suivantes. La procédure `get_co_type` retourne le nom entièrement qualifié du **type de CO**. Le caractère de qualification est le point. La procédure `generic_create` instancie le **type de CO** associé. La procédure `list_cos` retourne une liste de valeurs de ComponentKey des **CO** instanciés. La procédure `resolve_co` prend une valeur de ComponentKey et retourne le **CO** associé.

L'interface de fabrique déclare une procédure `create_<CO-name>` qui crée le **type de CO** associé. L'agent de fabrique possède une porte `factory` qui prend en charge l'interface `CoFactoryBase` dans le sens entrant.

C.5.11.6 Encapsulation du type de CO et de la fabrique de CO – composant SDL

Le **type de CO** et le type de fabrique de **CO** sont tous deux définis dans un type de bloc `<CO-name>_CO`, appelé composant SDL. Chaque composant SDL contient un ensemble d'instances `factory` de type `<CO-name>_factory` et un ensemble d'instances `cos` de type `<CO-name>`. L'ensemble d'instances `factory` contient exactement une instance. L'ensemble d'instances `cos` ne contient initialement aucune instance et ne spécifie pas de restriction quant au nombre maximal d'instances. Les portes `initial`, `provides` et `uses` du **type de CO** sont dupliquées par le composant SDL de même que la porte `factory` du type de fabrique, et ces portes sont raccordées par des canaux.

Le composant SDL est défini dans le paquetage de définition. Il représente un **type de CO**.

C.6 Mappage des concepts du point de vue configuration

C.6.1 Ports provisionnés

Le concept de **ports provisionnés** en eODL est un mécanisme permettant de distribuer les références d'interface qui sont offerts par un **CO** aux clients de ce **CO**.

Ce concept est mappé avec un ensemble de procédures distantes qui sont déclarées dans l'interface de configuration du **CO**.

Un **port provisionné** `foo` de type `bar` est mappé avec la procédure distante `provide_foo` qui retourne une référence (`PId`) à bar. Si l'attribut de `foo` vaut **single**, chaque appel de `provide_foo` doit retourner le même `PId`. Si l'attribut de `foo` vaut **multiple**, l'utilisateur doit lui-même implémenter la sémantique (voir le paragraphe C.7.4.3 sur la gestion des **ports**).

L'interface de configuration hérite de l'interface prédéfinie `ConfigBase`. Cette interface déclare une procédure `provide`, qui a un argument de type chaîne. Le **paramètre** effectif utilisé dans l'appel de la procédure désigne un **port**. Si ce port existe, une référence est retournée sous forme de `PId`. Si le **port** n'existe pas, une **anomalie** `NoSuchPort` est générée. L'**anomalie** `NoSuchPort` est prédéfinie.

C.6.2 Ports utilisés

Le concept de **port utilisé** en eODL est un mécanisme qui permet à un **CO** de stocker les références d'interface d'autres **CO**.

Ce concept est mappé avec un ensemble de procédures distantes qui sont déclarées dans l'interface de configuration du **CO**.

Un **port utilisé** `foo` de type `bar` est mappé avec la procédure distante `link_foo` qui a comme **paramètre** une référence à `bar`. Si l'attribut de `port` vaut `single` et qu'une référence est déjà stockée au niveau de ce **port**, l'**anomalie** prédéfinie `AlreadyConnected` est générée. Par ailleurs, une procédure distante `unlink_foo` est déclarée, qui supprime la référence stockée au niveau du **port** `foo`. Si aucune référence n'est stockée au niveau de `foo`, l'**anomalie** prédéfinie `NotConnected` est générée. Si l'attribut du **port** vaut **multiple**, une séquence de références est stockée. L'**anomalie** `AlreadyConnected` n'est jamais générée.

L'interface prédéfinie `ConfigBase`, dont l'interface de configuration hérite, déclare une procédure `link` et une procédure `unlink`. Ces procédures peuvent être utilisées de façon générique pour stocker ou supprimer une référence au niveau d'un **port utilisé**. Comme leur homologue pour les **ports provisionnés**, elles ont un argument de type chaîne qui désigne le nom du **port**. A nouveau, s'il n'existe pas de **port** avec le nom désigné, une **anomalie** `NoSuchPort` est générée. La procédure générique de connexion a un `PId` comme deuxième argument, elle le stocke au niveau du **port**

désigné si c'est possible ou génère une anomalie `AlreadyConnected` dans le cas contraire. Elle utilise la même sémantique que la procédure de connexion propre au **port** pour décider si le stockage de la référence est possible. La procédure générique de déconnexion génère une **anomalie** `NotConnected` si aucune référence n'est stockée au niveau du **port** désigné.

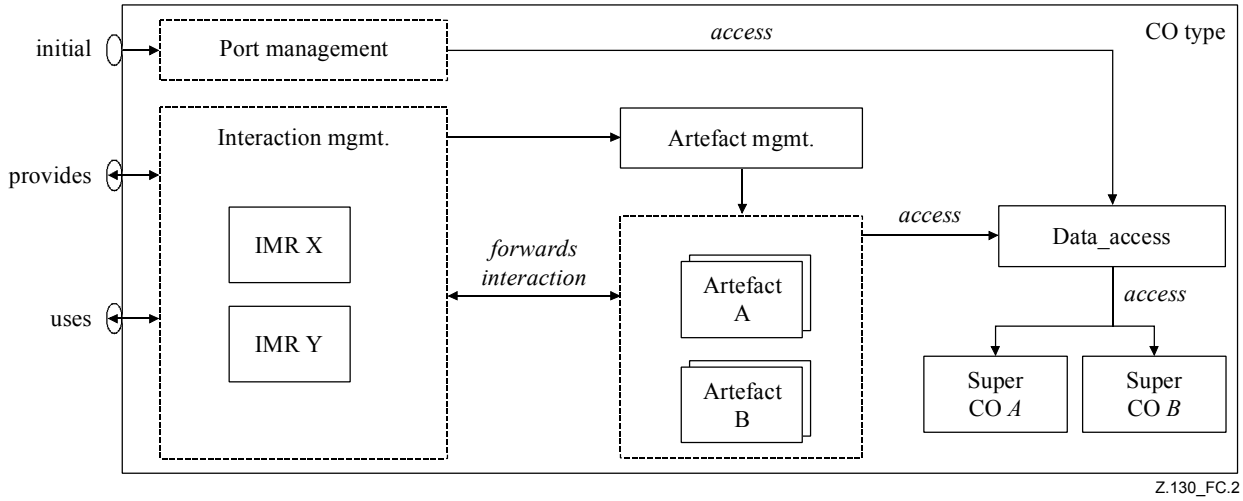
C.6.3 Service de dénomination SDL

Pour qu'un **CO** puisse découvrir dynamiquement les autres composants SDL, on définit un service de dénomination, implémenté par le type de processus `SDL_Component_Register_Type` dans le paquetage `eodl`. Chaque système SDL obtenu à partir d'un modèle eODL doit avoir un ensemble d'instances `SDL_Component_Registry` qui contient exactement une instance de type `SDL_Component_Register_Type`.

Dès qu'un composant SDL est instancié, il s'enregistre au moyen de la procédure exportée `register_SDLComponent`. Tout **CO** peut interroger le service de dénomination au moyen de `query_SDLComponent`. Pour consulter un composant SDL, la clé à utiliser est le nom entièrement qualifié avec le point comme caractère de qualification. La procédure d'interrogation retourne une référence à une instance de composant SDL du type demandé. Cette référence permet à tout client de demander des **CO** à la fabrique du composant SDL.

C.7 Mappage des concepts d'implémentation

La Figure C.2 illustre la structure interne d'un processus SDL représentant un **type de CO** sous la forme d'un aperçu avec différentes représentations SDL optionnelles.



IMR Interaction management representation

Figure C.2/Z.130 – Processus SDL représentant un type de CO

Un rectangle représente un processus SDL. Un rectangle en pointillés représente un concept et non un processus. Par exemple, la "gestion des interactions" représente le concept de gestion des interactions et contient des processus concrets de gestion des interactions, qui sont des processus SDL.

La Figure C.3 présente un exemple concret de type de processus de **CO** en notation graphique SDL. Les procédures `get_key`, `provide_SamplePort` et `provide` constituent la gestion des ports. Les processus `interaction_interfaceX` et `interaction_interfaceY` correspondent à "IMR X" et "IMR Y" sur la Figure C.2. Les ensembles d'instances de processus `A` et `B` sont des instances d'**artefacts**. L'ensemble d'instances `base` correspond à l'encadré "super CO A" sur la Figure C.2.

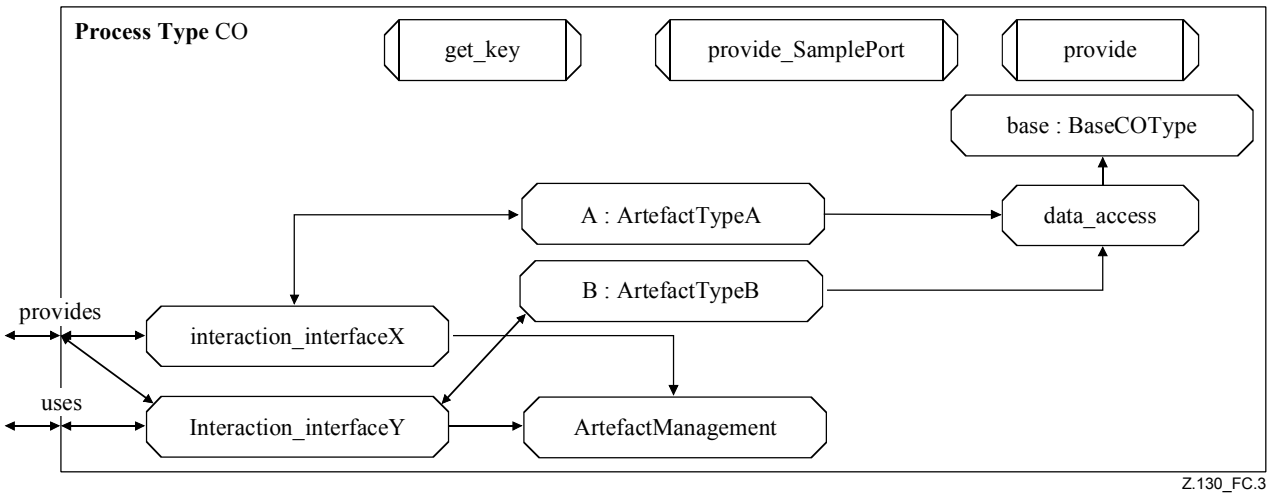


Figure C.3/Z.130 – Exemple de type de processus de CO

C.7.1 Accès aux données

Les données d'un CO sont stockées dans un processus `data_access` qui implémente les procédures *get/set* pour permettre aux artefacts d'accéder à ces données. Les données sont constituées de références utilisées par la gestion des ports et la gestion des interactions.

Un paquetage `<CO-name>_data` est déclaré dans le paquetage de définition afin de prendre en charge l'accès aux données en fonction d'un type. Il contient une interface `internal_data` qui déclare toutes les procédures *get/set*.

C.7.2 Artefacts et éléments d'implémentation

Les **artefacts** sont des constructions de langage de programmation qui contiennent des **éléments d'implémentation**. En SDL-2000, ils sont mappés avec des types de processus référencés qui sont définis dans les paquetages de définition. Les **artefacts** sont instanciés sous la forme d'un ensemble d'instances dans le type de processus de **CO** qu'ils implémentent.

Pour l'accès aux données d'un **CO**, on définit un canal allant de l'ensemble d'instances d'**artefact** au processus `data_access`. Ce canal achemine tous les appels de procédure destinés au processus `data_access` (d'accès aux données).

Les **éléments d'implémentation** associent les **artefacts** et les **éléments d'interaction** que l'**artefact** implémente. Ils n'ont pas de représentation en SDL.

L'implémentation d'un **élément d'interaction** dépend du cas d'implémentation. Il existe deux cas d'implémentation:

- fourniture;
- utilisation.

Le Tableau C.1 qui suit donne la sémantique associée aux types d'élément d'interaction et aux cas d'implémentation.

Tableau C.1/Z.130 – Sémantique associée aux types d'élément d'interaction et aux cas d'implémentation

Type d'élément d'interaction dans le modèle de conception	Définition de cas de l'élément d'implémentation dans le modèle de conception	Sémantique des éléments d'implémentation
opération/attribut	fourniture	Implémentation d'un comportement d'**opération**/fourniture d'**opérations** d'accès à l'attribut
opération/attribut	utilisation	Appel d'opération explicite possible
consommer	fourniture	Implémentation de la consommation de signal
consommer	utilisation	Implémentation de l'envoi de signal
produire	fourniture	Implémentation de l'envoi de signal
produire	utilisation	Implémentation de la consommation de signal

De façon analogue au mappage des interfaces décrit au § C.5.10 (Interfaces et éléments d'interaction), chaque interface eODL est en outre mappée dans le paquetage de définition avec un paquetage du même nom que l'interface contenant deux interfaces SDL: `exported_<interface-name>` et `imported_<interface-name>`. Le mappage des **éléments d'interaction** correspond exactement à la description figurant au § C.5.10 sauf pour les procédures et les signaux. Chaque procédure et chaque signal comportent un **paramètre** formel additionnel de type `PId`. Ce **paramètre** sert à acheminer des informations de l'émetteur ou du récepteur. On trouvera davantage d'informations au § C.7.4. Les types de signal auxquels ces interfaces font référence (comportant un **paramètre** formel additionnel de type `PId`) sont définis dans le paquetage de définition.

Les paragraphes qui suivent contiennent davantage de détails sur l'implémentation des **éléments d'interaction**.

C.7.2.1 Implémentation d'une opération

Pour pouvoir implémenter un **élément d'interaction opération** d'une interface, l'**artefact** doit contenir une procédure exportée qui implémente la procédure définie dans l'interface SDL `exported_<interface-name>` de l'interface correspondante figurant dans le paquetage de définition (voir le paragraphe d'avant). Il implémente donc une procédure qui contient un **paramètre** additionnel de type `PId`. L'**artefact** peut utiliser ce **paramètre** pour obtenir des informations sur l'émetteur d'origine de l'appel de la procédure.

C.7.2.2 Appel d'une opération à partir d'un artefact

Les appels d'opération destinés à d'autres **CO** sont réalisés comme suit: l'**artefact** envoie un appel de procédure à la représentation de gestion des interactions qui implémente l'interface `imported_<interface-name>` de l'interface qui contient l'opération à appeler. Cette procédure est définie dans le paquetage de définition et contient un **paramètre** formel additionnel de type `PId`. Le **paramètre** `PId` effectif sert à désigner le récepteur de l'appel de la procédure. La représentation de gestion des interactions est chargée de transmettre l'appel de procédure à son récepteur.

C.7.2.3 Envoi d'un signal

De manière analogue à l'appel d'une procédure, un **artefact** n'envoie pas de signal directement à son récepteur, mais à la représentation de gestion des interactions. Le signal contient un **paramètre** formel additionnel qui désigne le récepteur du signal et il est défini dans le paquetage de définition. La représentation de gestion des interactions est chargée de transmettre le signal à son récepteur.

C.7.2.4 Consommation d'un signal

Pour pouvoir implémenter la consommation d'un signal, l'**artefact** doit implémenter une unité de traitement du signal qui accepte le signal correspondant défini dans le paquetage de définition.

C.7.2.5 Héritage d'artefacts

L'héritage multiple est permis pour les **artefacts**. Etant donné que l'héritage multiple n'est pas admis en SDL pour les types d'agent, il est réalisé par délégation. Les **artefacts** de base sont contenus dans l'**artefact** dérivé et les portes correspondantes de chaque **artefact** de base et de l'artefact dérivé sont raccordées par un canal. Tous les appels de procédure et tous les signaux allant vers des **éléments d'implémentation** qui ne sont pas redéfinis sont transmis directement de l'environnement de l'**artefact** dérivé à l'**artefact** de base approprié. Si un certain **élément d'implémentation** est redéfini dans l'**artefact** dérivé, **cet élément** est défini dans l'**artefact** dérivé.

C.7.3 Gestion des artefacts et schéma d'instanciation

La gestion des **artefacts** est chargée de créer et de gérer les instances d'un **artefact**. Un **CO** possède un ensemble d'instances `artefact_<CO-name>` pour chaque **artefact** qui implémente le **CO**. Toutefois, le schéma d'instanciation qui figure dans le modèle précise quelle est l'instance d'**artefact** qui est utilisée dans une interaction. La gestion des **artefacts** implémente le schéma d'instanciation.

La gestion des **artefacts** est réalisée sous la forme d'un processus `artefactmanagement` contenu dans le type de processus de **CO**. L'ensemble d'instances de ce processus contient exactement une instance. Pour chaque type d'artefact qui implémente le **CO** donné, il existe une procédure distante exportée `get_artefact_<artefact-name>` qui retourne la référence à une instance de ce type d'**artefact**. Cette procédure est implémentée automatiquement. Son implémentation dépend du schéma d'instanciation à utiliser:

- *un artefact par demande*: à chaque appel de procédure, une nouvelle instance est créée et la référence à cette instance est retournée;
- *pool d'artefacts*: un nombre limité d'instances (un "pool") est créé et la référence à l'une de ces instances est retournée;
- *singleton*: il existe une seule instance de l'**artefact** et la référence à cette instance est retournée;
- *défini par l'utilisateur*: comme la sémantique est définie par l'utilisateur, la procédure ne peut pas être implémentée automatiquement. En revanche, une procédure référencée est déclarée et l'utilisateur doit lui-même l'implémenter.

C.7.4 Représentation de gestion des interactions

Dans le mappage avec le SDL, la représentation de gestion des interactions sert d'intermédiaire entre les **éléments d'implémentation** et l'environnement d'un **CO**. Chaque représentation d'un **élément d'interaction** traite à la fois les interactions entrantes et les interactions sortantes (se rapportant au **CO**).

Chaque interface qui est requise ou prise en charge par un **CO** est représentée par un processus dans le cadre du type de processus du **CO**. Ce processus implémente chaque **élément d'interaction** en transmettant la demande d'interaction:

- de l'environnement du **CO** à l'**élément d'implémentation** approprié, le schéma d'instanciation de l'**artefact** étant respecté grâce à l'utilisation de la représentation de gestion des **artefacts**;
- de l'**élément d'implémentation** à l'environnement du **CO**.

Il existe une et une seule instance de processus par interface. Toutefois, l'existence d'un **port multiple** de ce type constitue une exception à cette règle. Dans ce cas, il existe plusieurs instances. Le nombre concret dépend de l'implémentation. Par ailleurs, pour chaque processus de gestion des

interactions, il existe une variable implicite interne au **CO** `<process-name>_reference` qui comporte le `PId` de ce processus. Si la variable doit comporter plusieurs références d'instance, on utilise une chaîne de `PId`.

C.7.4.1 Représentation de gestion des interactions implémentant d'un CO avec l'environnement

En ce qui concerne les interactions avec l'environnement, la représentation de gestion des interactions implémente

- tous les appels d'**opération** (uniquement pour les interfaces requises);
- tous les signaux susceptibles d'être envoyés.

Elle prend en charge l'interface `imported_<interface-name>` du paquetage de définition dans le sens entrant (à partir des **artefacts**) et l'interface `imported_<interface-name>` du paquetage de définition dans le sens sortant (vers l'environnement).

Une **opération** (plus précisément: l'utilisation d'une **opération**) est implémentée de la façon suivante:

1) la valeur de `sender` est sauvegardée dans une variable temporaire;
2) la référence à une instance d'**artefact** est acquise grâce à l'appel de la procédure appropriée de la représentation de gestion des **artefacts**;
3) la procédure est appelée (la valeur sauvegardée de `sender` est ajoutée sur la liste des paramètres) avec la référence à l'instance d'**artefact** comme destination;
4) si des **anomalies** sont déclarées pour l'**opération**, elles doivent être obtenues et générées à nouveau.

L'envoi d'un signal est implémenté de la façon suivante: lorsque la représentation de l'**élément d'interaction** reçoit le signal spécifié (avec un **paramètre** additionnel spécifiant la destination), elle envoie le signal spécifié (sans le **paramètre** additionnel) à la destination.

C.7.4.2 Représentation de gestion des interactions de l'environnement avec un CO

En ce qui concerne les interactions dans ce sens, la représentation de gestion des interactions implémente:

- toutes les **opération**s déclarées au niveau de l'interface; et
- tous les **signaux** produits et consommés.

Elle prend en charge l'interface `exported_<interface-name>` du paquetage d'interface dans le sens entrant (à partir de l'environnement) et l'interface `exported_<interface-name>` du paquetage de définition dans le sens sortant (vers les **artefacts**).

Une **opération** (plus précisément: la fourniture d'une **opération** à l'environnement) est implémentée de la façon suivante. Lorsque l'appel de procédure est reçu, la référence à une instance d'**artefact** est acquise par l'appel de la gestion des **artefacts**. Un appel de procédure pour cette instance est ensuite lancé, la valeur de la variable **`sender`** étant fournie sous la forme d'un **paramètre** additionnel. Si l'**opération** a un **paramètre** de retour, la valeur de retour est retournée à l'émetteur. Si une **opération** est générée, la représentation de gestion des interactions est à nouveau mise à contribution.

La consommation d'un **signal** est implémentée de la façon suivante: lorsque la représentation de l'**élément d'interaction** reçoit le signal provenant de l'environnement, elle envoie le signal correspondant (avec un **paramètre** additionnel qui contient la valeur de la variable `sender`) à l'**artefact**.

C.7.4.3 Gestion des ports

La gestion des ports est chargée:

- de créer des représentations d'**élément d'interaction**;
- de gérer les références aux interfaces;
- d'implémenter des opérations d'accès propres à un **port**;
- d'implémenter des opérations génériques d'accès au niveau des **ports**.

Toutes ces attributions sont implémentées automatiquement.

C.7.4.4 Représentation de gestion des ports

La gestion des ports est implémentée par plusieurs procédures exportées et par la transition de départ de la machine à états finis du processus du **type de CO**.

C.7.4.5 Création de représentations de gestion des interactions

La gestion des ports est chargée de créer des processus représentant la gestion des interactions. Cette création est réalisée lors de la transition de départ du processus du **type de CO**. Après la création des processus de gestion des interactions, la gestion des ports stocke les références à chaque processus dans le processus d'accès aux données au moyen de la variable `interaction_<interface-name>`.

C.7.4.6 Gestion des références aux interfaces

Pour chaque **port**, il existe une variable implicite `port_<port-name>` dont la portée est limitée au **CO**. Le type de cette variable est soit le type de référence de l'interface associée au port (lorsque l'attribut du **port** vaut **single**) soit une chaîne de `PId` (lorsque l'attribut du **port** vaut **multiple**). Les opérations au niveau des **ports** manipulent directement ces variables internes.

C.7.4.7 Implémentation d'opérations d'accès propres à un port

Les opérations particulières sont implémentées de la façon suivante.

Port provisionné simple: cette opération retourne la référence qui est stockée dans la variable interne.

Port provisionné multiple: une référence doit être choisie parmi un ensemble de références. Pour procéder à ce choix, la procédure `choose_provide_<port-name>` (définie dans le processus de gestion des ports) est appelée. Cette procédure doit être implémentée par l'utilisateur et elle est donc référencée.

Port utilisé simple: l'opération link stocke une référence donnée dans le processus d'accès aux données au moyen de la variable `port_<port-name>`. Si une référence y est déjà stockée, une **anomalie** `AlreadyConnected` est générée. L'opération `unlink` attribue la valeur `Null` à la variable dont la portée est limitée au **CO**. Si la valeur `Null` lui est déjà attribuée, l'opération génère une **anomalie** `NotConnected`.

Port utilisé multiple: l'opération `link` stocke une référence donnée dans une séquence de références. Pour déterminer l'endroit exact où la référence doit être placée dans la séquence, l'utilisateur doit implémenter la procédure `choose_link_<port-name>` (définie dans le processus de gestion des ports) qui prend la référence et doit la stocker quelque part dans la séquence. De même, l'opération de déconnexion appelle la procédure `choose_link_<port-name>` pour supprimer une référence appropriée.

C.8 Omission du comportement généré automatiquement

Pour le mappage eODL-SDL présenté jusqu'ici, on suppose que l'utilisateur souhaite uniquement implémenter la logique commerciale. Toutefois, si l'utilisateur souhaite générer un code de

production en dehors du SDL, il voudra peut-être se passer du code généré automatiquement et implémenter lui-même le code.

Pour cela, il est prévu, dans le mappage, une option permettant d'omettre le code généré automatiquement. Autrement dit, les entités suivantes ne sont pas générées:

- la gestion des **artefacts** et les ensembles d'instances d'**artefact**;
- les représentations de gestion des interactions;
- la gestion des ports.

En résumé, le type de processus du **CO** ne contient aucune entité. L'utilisateur peut implémenter le type de processus du **CO** de la façon dont il le souhaite.

C.9 Concepts de l'eODL non mappés

Les concepts eODL suivants ne sont pas mappés dans le SDL-2000:

- identification d'**exécution** de type quelconque;
- interaction de type **média continu**;
- **composants logiciels**;
- **assemblages**;
- contraintes et propriétés;
- concepts de l'environnement cible;
- plan de déploiement.

C.10 Paquetage eodl prédéfini

Le présent paragraphe expose l'ensemble du contenu du paquetage `eodl`.

```
package eODL;
  syntype unsigned_short = Integer constants (0:65535);
  endsyntype;

  syntype unsigned_long = Integer constants (0:4294967295);
  endsyntype;

  syntype unsigned_long_long = Integer constants (0:18446744073709551615);
  endsyntype;

  syntype short = Integer constants (-32768:32767);
  endsyntype;

  syntype long = Integer constants (-2147483648:2147483647);
  endsyntype;

  syntype long_long = Integer constants
    (9223372036854775808:9223372036854775807);
  endsyntype;

  syntype char = Character endsyntype;
  syntype wchar = Natural endsyntype;

  syntype float = Real endsyntype;
  syntype double = Real endsyntype;
  syntype long_double = Real endsyntype;

  value type wstring
    inherits String < wchar >;
  endvalue type wstring;
```

```
value type wstring_bounded < synonym length Natural >
   inherits Vector < wchar, length>;
endvalue type wstring_bounded;

abstract value type TypeCode;
endvalue type;

value type fixedpt < synonym Width Natural; synonym scale Natural >;
  struct
    private unscaled_int Integer;
  operators
    Make ( Real ) -> this fixedpt;
    Make ( Integer ) -> this fixedpt;
     "+"  ( this fixedpt, this fixedpt ) -> this fixedpt;
     "-"  ( this fixedpt, this fixedpt ) -> this fixedpt;
     "*"  ( this fixedpt, this fixedpt ) -> this fixedpt;
     "/"  ( this fixedpt, this fixedpt ) -> this fixedpt;
     "="  ( this fixedpt, this fixedpt ) -> Boolean;
     ">"  ( this fixedpt, this fixedpt ) -> Boolean;
  methods
    toReal -> Real;

  OPERATOR Make (r Real ) -> this fixedpt  {
    DCL retVal this fixedpt;
    r := r * power(10,scale);
    retVal.unscaled_int := fix(r);
    return retVal;
  }
  OPERATOR Make (n Integer ) -> this fixedpt  {
    DCL retVal this fixedpt;
    retVal.unscaled_int := n * power(10,scale);
    return retVal;
  }
  OPERATOR  "+"  (a this fixedpt, b this fixedpt ) -> this fixedpt {
    DCL retVal this fixedpt;
    retVal.unscaled_int := a.unscaled_int + b.unscaled_int;
    return retVal;
  }
  OPERATOR  "-"  (a this fixedpt, b this fixedpt ) -> this fixedpt {
    DCL retVal this fixedpt;
    retVal.unscaled_int := a.unscaled_int - b.unscaled_int;
    return retVal;
  }
  OPERATOR  "*"  (a this fixedpt, b this fixedpt ) -> this fixedpt {
    DCL retVal this fixedpt,
        t Real;
    t := float(a.unscaled_int * b.unscaled_int);
     t := t / float(power(10,2*scale));
    retVal.unscaled_int := Make( t );
    return retVal;
  }
  OPERATOR  "/"  (a this fixedpt, b this fixedpt ) -> this fixedpt {
    DCL retVal this fixedpt,
         t Real;
    t := float(a.unscaled_int)/float(a.unscaled_int);
    retVal.unscaled_int := Make( t );
    return retVal;
  }
  OPERATOR  "="  (a this fixedpt, b this fixedpt ) -> Boolean {
    return a.unscaled_int = b.unscaled_int;
  }
  OPERATOR  ">"  (a this fixedpt, b this fixedpt ) -> Boolean {
    return a.unscaled_int > b.unscaled_int;
  }
```

```
  METHOD toReal -> Real {
    return float(unscaled_int)/float(power(10,scale));
  }
endvalue type;

package ComponentModel;
  value type ComponentKey;
     struct
       the_key string;
     methods
       virtual equal (this ComponentKey) -> Boolean;
  endvalue type;
  interface ComponentBase;
    procedure get_key -> ComponentKey;
  endinterface ComponentBase;

  procedure generate_CO_key -> ComponentKey external;

  value type ComponentKeySeq
    inherits String < ComponentKey >;
  endvalue type;
  interface CoFactoryBase;
    procedure get_co_typ -> string;
    procedure generic_create -> ComponentBase;
    procedure resolve_CO (ComponentKey) -> ComponentBase;
    procedure list_cos -> ComponentKeySeq;
  endinterface CoFactoryBase;

  exception NotConnected;
  interface ConfigBase;
    procedure provide(in string) -> PId
         raise NoSuchPort;
    procedure link(in string, in PId)
         raise AlreadyConnected, NoSuchPort;
    procedure unlink(in string, in ComponentBase)
         raise NotConnected, NoSuchPort;
  endinterface ConfigBase;

endpackage ComponentModel;

interface SDLComponent_Registry_IF;
  procedure register_SDLComponent(in string, in PId);
  procedure query_SDLComponent(in string) -> PId;
endinterface;

process type SDLComponent_Registry_Type;
  gate registry in with SDLComponent_Registry_IF;
  channel nodelay
    from env  to this via registry;
  endchannel;

  value type registry_store
    inherits Array < string, Pid >;
  endvalue type registry_store;

  dcl store registry_store := Make;

  exported as <<package eodl>>register_SDLComponent
  procedure register_SDLComponent(in key string, in item Pid);
    start;
    task registry_store := Modify(store,key,item);
    return;
  endprocedure;
```

```
    exported as <<package eodl>>query_SDLComponent
    procedure register_SDLComponent(in key string) ->  Pid
           raise InvalidIndex;
      dcl retval Pid;
      start;
      task retval := Extract(store,key);
      return retval;
    endprocedure;

  endprocess type SDLComponent_Registry_Type;


endpackage eODL;
```

Annexe D

Représentation en XML du métamodèle eODL

Le **métamodèle** a été défini sur la base de l'UML. Sa représentation en XML conformément au [6] de l'OMG sur le XMI, destinée à être lue par des outils, constitue l'Annexe D. Les données effectives figurent dans le paquetage logiciel "Z.130 Annex D.xml".

NOTE – Le paquetage logiciel Z.130 Annexe D.xml est disponible gratuitement en téléchargement depuis la base UIT-T des langages formels à l'adresse http://www.itu.int/ITU-T/formal-language/xml/database/itu-t/z/z130/2003/.

Appendice I

Exemple: dîner des philosophes

I.1 Introduction

Le présent appendice a pour objet de donner un exemple d'utilisation de l'eODL pour la conception, l'implémentation et le **déploiement** d'un système réparti.

Le problème du dîner des philosophes a été décrit pour la première fois en 1965 par Edsger W. Dijkstra. Il s'agit d'un modèle et d'une méthode universelle servant à tester et comparer les théories sur l'attribution des ressources. Dijkstra comptait l'utiliser pour créer plus facilement un système d'exploitation stratifié, en créant une machine qui pouvait être considérée comme un automate entièrement déterministe.

Un nombre configurable de philosophes (processus) sont assis autour d'une table ronde; un nombre limité de fourchettes (ressources) se trouvent sur la table. Les philosophes effectuent des actions – penser, manger et dormir. Ils n'ont besoin d'aucune ressource pour penser ou dormir, mais ils ont besoin de deux fourchettes chacun pour manger, une pour la main gauche et une pour la main droite. Avant de commencer à manger, un philosophe essaie donc d'obtenir les deux fourchettes qui se trouvent de chaque côté de lui, qui doivent être disponibles. Autrement dit, deux philosophes voisins ne peuvent pas manger en même temps.

Un observateur reçoit une notification chaque fois qu'un philosophe change d'activité, c'est-à-dire chaque fois qu'un philosophe commence à manger, commence à penser ou commence à dormir. Il reçoit aussi une notification lorsqu'un philosophe a faim.

I.2 Description

Dans ce problème, un ensemble limité de processus partagent un ensemble limité de ressources, chacune ne pouvant être utilisée que par un processus à la fois, d'où d'éventuelles situations de blocage.

Les ensembles limités de processus et de ressources et les interactions dynamiques entre eux constituent un système réparti. La tâche consiste à répartir les implémentations des ressources et des processus dans le réseau cible. En outre, les ressources doivent être raccordées aux processus.

Le scénario donné à titre d'exemple inclut trois **types de CO** différents:

- philosophe;
- fourchette;
- observateur.

Les différentes phases sont alors les suivantes:

Phase de conception

- Définition d'un modèle des éléments de l'exemple, comprenant des **types de CO**, des ports et des interfaces.
- Définition d'un modèle de la structure d'implémentation.

Phase d'implémentation

- Implémentation des **artefacts** conformément au modèle (fourniture de la logique commerciale).
- Génération de **composants logiciels** conformément au modèle.
- Définition d'un modèle de la structure initiale du système (**configuration initiale**) par la définition d'un assemblage.
- Mise en paquetage des **composants logiciels** et des informations connexes relatives au modèle pour pouvoir expédier l'implémentation aux clients.

Phase d'intégration

- Remise du paquetage à un client.
- Modélisation de l'environnement cible des locaux du client.
- Détermination d'une attribution appropriée des **composants logiciels** du paquetage dans l'environnement cible.
- Installation des **composants logiciels** attribués au niveau des nœuds cibles identifiés.
- Etablissement de la **configuration initiale** par interconnexion de tous les **CO initiaux** conformément à la **configuration initiale**.

Au paragraphe I.3, l'exemple du "dîner des philosophes" est spécifié en eODL. Dans la spécification, trois **types de CO** sont définis:

- le type d'objet o_Philosopher représente un philosophe;
- le type d'objet o_Fork représente une fourchette;
- le type d'objet o_Observer représente un observateur.

Le paragraphe I.4 contient le modèle SDL déduit du modèle eODL conformément aux règles de mappage données à l'Annexe C. Seuls les concepts des points de vue traitement, configuration et implémentation sont mappés étant donné que les concepts du point de vue déploiement n'ont pas de correspondant. Le modèle SDL est constitué de deux paquetages principaux:

- le paquetage d'interface SDL phil_interface;

- le paquetage de définition SDL phil_definition.

I.3 Exemple en eODL

```
module DiningPhilosophers {
     CO o_Philosopher;
     CO o_Fork;

     interface i_Fork;
     interface i_Philosopher;
     interface i_Observer;

     exception ForkNotAvailable {};
     exception NotTheEater {};

     enum e_ForkState {
          UNUSED,
          USED,
          WASHED
     };

     enum e_Pstate {
          EATING,
          THINKING,
          SLEEPING,
          DEAD,
          CREATED,
          HUNGRY
     };

     interface i_Fork {
        void obtain_fork ( in o_Philosopher eater )
        raises ( ForkNotAvailable );
        void release_fork ( in o_Philosopher eater ) raises ( NotTheEater );
     };

     artefact a_ForkImpl {
          obtain_fork implements supply i_Fork::obtain_fork;
          release_fork implements supply i_Fork::release_fork;
     };

     CO o_Fork {
          supports i_Fork;
          provide i_Fork fork;
          implemented by a_ForkImpl with ArtefactPool(2);
     };

     interface i_Philosopher {
          void set_name ( in string name);
     };

     artefact a_PhilosopherImpl {
          set_name_impl implements supply i_Philosopher::set_name;
          pstate_impl implements use i_Observer::pstate;
     };

     CO o_Philosopher {
          implemented by a_PhilosopherImpl with Singleton;
          supports i_Philosopher;
          requires i_Fork, i_Observer;
          use I_observer observer;
          use i_Fork left;
```

```
          use i_Fork right;
     };

     valuetype Pstate {
          public e_PState state;
          public string name;
          public i_Philosopher philosoph;
          factory create (
               in e_PState state,
               in string name,
               ini_Philosopher philo);
     };

     signal PhilosopherState {
          PState carry_pstate;
     };

     interface i_Observer {
          consume PhilosopherState pstate;
     };

     artefact a_Observer {
          pstate_Impl implements supply i_Observer::pstate;
     };

     CO o_Observer {
          implemented by a_Observer with Singleton;
          supports i_Observer;
          provide i_Observer observer;
     };
};
softwarecomponent Philosopher
realizes o_Philosopher, o_Observer
{
     requires {
          property os =  [
                              { name = "WINNT"; version = "4,0,0,0"; },
                              { name = "WIN98"; }
          ];
     };
};

softwarecomponent Fork
realizes o_Fork;
{
     requires {
          property os =  [
                              { name = "WINNT"; version = "4,0,0,0"; },
                              { name = "WIN98"; }
          ];
     };
};

assembly ass1 {
     p (3) : o_Philosopher;
     f1 : o_Fork;
     f2 : o_Fork;
     o : o_Observer;
     connect c1 {
          p.left  = f1.fork;
          p.right = f2.fork;
     };
```

```
     connect c2 { o.observer = p.observer; };
};

environment myenv_1 {
     node n1 {
          property os =  { name = "WINNT"; version = "4,0,0,0"; };
          property memory = 256;
     };

     node n2 {
          property os =  { name = "WINNT"; version = "4,0,0,0"; };
          property memory = 128;
     };

     link l1 { node n1, n2; };
};

installation install1
uses environment myenv_1 {
     Philosopher ->n2;
     Fork ->n1;
};

instantiation instantiate1
uses environment myenv_1
uses assembly ass1 {
     p,  o  -> n2;
     f1, f2 ->  n1;
};

deploy {
     install { install1; };
     instantiate { instantiate1; };
};
```

I.4 Exemple en SDL-2000

```
use eODL;
/* /--------------------------------\ */
/*     data types and interface       */
/*     needed by clients              */
package phil_interface;

package DiningPhilosophers;

  /* exceptions */
  exception ForkNotAvailable;
  exception NotTheEater;

  /* enumerations */
  value type e_ForkState;
    literals
      UNUSED, USED;
  endvalue type;

  value type e_ForkState;
    literals
      EATING, THINKING, SLEEPING,
      DEAD, CREATED, HUNGRY;
  endvalue type;

  /* interface i_Fork */
  package i_Fork;
```

```
    /* declaration of exported procedures */
    interface exported_i_Fork;
      procedure obtain_fork(in o_Philosopher)
             raise ForkNotAvailable;
      procedure release_fork(in o_Philosopher)
               raise NoTheEater;
    endinterface;

     /* declaration of consumed signals */
    interface imported_i_Fork;
         /* no consumed signals declared */
    endinterface;
  endpackage;

  /* definition of CO type o_Fork */
  use i_Fork;
  package o_Fork;

    /* contains attributes defined in CO type o_Fork */
    interface o_Fork_attributes
        inherits <<package eODL/package ComponentModel>>ComponentBase;
    endinterface;

    /* port operations */
    interface o_Fork_config
        inherits <<package eODL/package ComponentModel>>ConfigBase adding;
      /* provided port "fork" */
      procedure provide_fork -> exported_i_Fork;
    endinterface;

    /* combine config and attributes interfaces */
    interface o_Fork
      inherits o_Fork_attributes, o_Fork_config;
    endinterface;

    /* declaration of interface of CO factory */
    package factory;
      interface o_Fork_factory
       inherits <<package eODL/package ComponentModel>>CoFactoryBase adding;
        procedure create_o_Fork -> o_Fork;
      endinterface;
    endpackage;

  endpackage o_Fork;

package i_Philosopher;

    interface exported_i_Philosopher;
      procedure set_name(in string);
    endinterface;

    interface imported_i_Philosopher;
    endinterface;

  endpackage;

  use i_Philosopher;
  use o_Philosopher;
  object type Pstate;
    struct
      public eodl_state e_PState; /* state -> eodl_state ! */
      public name string;
       public philosoph exported_i_Philosopher;
    operators
```

```
       /* create -> eodl_create) */
      eodl_create(e_PState, string, exported_i_Philosopher) -> Pstate;
      make(e_PState, string, exported_i_Philosopher) -> PState;

    operator eodl_create(eodl_state e_PState,
                         name string,
                         philo exported_i_Philosopher) {
      dcl retval Pstate;
       retval.eodl_state := eodl_state;
      retval.name := name;
      retval.philosoph := philo;
       return retval;
    }
    operator make(eodl_state e_PState,
                  name string,
                  philo exported_i_Philosopher) {
       return eold_create(eodl_state,name,philo);
    }
  endobject type;

  use i_Philosopher;
  package o_Philosopher;

    /* contains attributes defined in CO type o_Fork */
    interface o_Philosopher_attributes
        inherits <<package eODL/package ComponentModel>>ComponentBase;
    endinterface;

    interface o_Philosopher_config
     inherits <<package eODL/package ComponentModel>>ConfigBase adding;
      procedure link_observer(exported_i_Observer) raise AlreadyConnected;
      procedure link_left(exported_i_Fork) raise AlreadyConnected;
      procedure link_right(exported_i_Fork) raise AlreadyConnected;
      procedure unlink_observer raise NotConnected;
      procedure unlink_left raise NotConnected;
      procedure unlink_right raise NotConnected;
    endinterface;

    interface o_Philosopher
         inherits o_Philosopher_attributes, o_Philosopher_config;
    endinterface;

    use eODL / package ComponentModel;
    package factory;
      interface o_Fork_factory
         inherits <<package eODL/package ComponentModel>>CoFactoryBase adding;
          procedure create_o_Philosopher -> o_Philosopher;
      endinterface;
    endpackage factory;

  endpackage;

  signal PhilosopherState(PState);

package i_Observer;

    interface exported_i_Observer;
      use PhilosopherState;
    endinterface;

    interface imported_i_Observer;
    endinterface;

  endpackage;
```

```
  /* CO o_Observer */
  use i_Observer;
  package o_Observer;

    interface o_Observer_attributes
        inherits <<package eODL/package ComponentModel>>ComponentBase adding;
    endinterface;

    interface o_Observer_config
        inherits <<package eODL/package ComponentModel>>ConfigBase adding;
    endinterface;

    interface o_Observer inherits o_Observer_attributes, o_Observer_config;
    endinterface;

    package factory;
      interface o_Observer_factory
       inherits <<package eODL/package ComponentModel>>CoFactoryBase adding;
        procedure create_o_Observer -> o_Observer;
      endinterface;
    endpackage;

  endpackage;

endpackage DiningPhilosophers;

endpackage phil_interface;
/* \--------------------------------/ */

/* /--------------------------------\ */
/*     implementation package         */
use eODL;
package phil_definition;

package DiningPhilosophers;


  /* used to define operations implemented or */
  /* used by artefacts */
  package i_Fork;
    /* operations implemented by artefacts */
    interface exported_i_Fork;
      procedure obtain_fork(in o_Philosopher, in Pid)
             raise ForkNotAvailable;
      procedure release_fork(in o_Philosopher, in Pid)
                raise NoTheEater;
    endinterface;

    /* operations used by artefacts */
    interface imported_i_Fork;
    endinterface;
  endpackage;


  /* state attributes */
  package o_Fork_data;

    interface internal_data;
      procedure get_port_fork -> exported_i_Fork;
      procedure get_interaction_i_Fork -> exported_i_Fork;
      procedure set_port_fork(exported_i_Fork);
      procedure set_interaction_i_Fork(exported_i_Fork);
```

```
  endinterface;
endpackage;

signallist a_ForkImpl_in =
     procedure <<package phil_definition/package DiningPhilosophers/
                  package i_Fork>>obain_fork,
      procedure <<package phil_definition/package DiningPhilosophers/
                  package i_Fork>>release_fork;

/* artefact: referenced definition */
use a_PhilosopherImpl;
use o_Fork_state;
process type a_ForkImpl with
   use (a_ForkImpl_in);;
  referenced;

/* /--------------------------------\ */
/*     SDL component o_Fork_CO        */
use a_PhilosopherImpl;
use o_Fork_state;
use a_ForkImpl;
block type o_Fork_CO;

  /* gate definitions */
  gate factory
    in with <<package phil_interface/package DiningPhilosopher/
        package o_Fork/package factory>>o_Fork_factory;
  gate initial
    in with  (<<package phil_interface/
        package DiningPhilosopher/package o_Fork>>o_Fork);
  gate provides
    in with <<package phil_interface/
        package DiningPhilosopher/package i_Fork>>exported_i_Fork;
    out with <<package phil_interface/
        package DiningPhilosopher/package i_Fork>>imported_i_Fork;

  /* defines the factory process */
  process type o_Fork_factory;
    gate factory
      in with <<package phil_interface/package DiningPhilosopher/
       package o_Fork/package factory>>o_Fork_factory; /* ; zuviel  */
    channel nodelay
      from this via factory to env;
    endchannel;

    /* stores component keys here */
    dcl keys ComponentKeysSeq;

    /* creates a CO and returns a reference to it */
    exported as <<package phil_interface/package DiningPhilosopher/
        package o_Fork/package factory>>generic_create
    procedure generic_create -> ComponentBase;
   dcl key ComponentKey;
       start;
      create co_instance;
       task key := call get_key to offspring;
        task keys := Modify(keys, key, offspring);
      return offspring;
    endprocedure;

    /* creates a CO and returns a reference to it */
    exported as <<package phil_interface/package DiningPhilosopher/
        package o_Fork/package factory>>create_o_Fork
    procedure create_o_Fork -> o_Fork;
```

```
      dcl key ComponentKey;
      start;
    create co_instance;
      task key := call get_key to offspring;
      task keys := Modify(keys, key, offspring);
    return offspring;
  endprocedure;

  /* returns name of CO type */
  exported as <<package phil_interface/package DiningPhilosopher/
     package o_Fork/package factory>>get_co_type
  procedure get_co_type -> string;
    start;
    return 'DiningPhilosophers.o_Fork';
  endprocedure;

  /* returns reference to a CO */
   exported as <<package phil_interface/package DiningPhilosophers/
          package o_Fork/package factory>>resolve_co
   procedure resolve_co(in key ComponentKey) -> ComponentBase
        raise InvalidIndex;
      dcl returnValue Pid;
      dcl i Integer := 1;
      start;
      task returnValue := Extract(keys,key);
      return returnValue;
   endprocedure;

  /* returns a list of CO keys */
   exported as <<package phil_interface/package DiningPhilosophers/
          package o_Fork/package factory>>list_co
   procedure list_co -> ComponentKeySeq;
      start;
      return keys;
   endprocedure;

endprocess type o_Fork_factory;
/* \-------------------------------/ */

/* defines the CO type itself */
process type o_Fork;

  /* gates used for comm. with environment */
  gate initial
    in with  (<<package phil_interface/package DiningPhilosopher
          /package o_Fork>>o_Fork);
  gate provides
    in with <<package phil_interface/package DiningPhilosopher
          /package i_Fork>>exported_i_Fork; /* NOS */
    out with <<package phil_interface/package DiningPhilosopher
          /package i_Fork>>imported_i_Fork;

  /* /-------------------------------\ */
  /*     encapsules state variables     */
  process data_access(1,1);

    dcl reference_interaction_i_Fork exported_i_Fork;
    dcl reference_port_fork exported_i_Fork;
    dcl the_key string;


  exported as <<package philo_definition/package DiningPhilosopher/
          package o_Fork_data>>get_port_fork
  procedure get_port_fork -> exported_i_Fork;
```

```
    start;
    return reference_port_fork;
  endprocedure;
  exported as <<package philo_definition/package DiningPhilosopher/
          package o_Fork_data>>get_interaction_i_Fork
  procedure get_interaction_i_fork -> exported_i_Fork;
    start;
    return reference_interaction_i_Fork;
  endprocedure;
  exported as <<package philo_definition/package DiningPhilosopher/
          package o_Fork_data>>set_port_fork
  procedure set_port_fork(in ref exported_i_Fork);
    start;
    reference_port_fork := ref;
  endprocedure;
  exported as <<package philo_definition/package DiningPhilosopher/
          package o_Fork_data>>set_interaction_i_Fork
  procedure set_interaction_i_fork(in ref exported_i_Fork);
    start;
    reference_interaction_i_Fork := ref;
  endprocedure;

endprocess data_access;
/* \--------------------------------/ */

/* channel artefact <--> state_access */
channel nodelay
  from artefact_a_ForkImpl to data_accessor
    with internal_state;
endchannel;


/* in case of CO type inheritance an instance of this type */
/* is contained in an "eODL-inherited" type. The following */
/* gate is used to interface with the state_access process */
/* of the inherited type. */
gate data_accessor
  in with internal_data;
channel route_state_access nodelay
  from
    env via data_accessor to data_access;
endchannel;


/* /--------------------------------\ */
/*     manages artefact instances     */
process artefactmanagement(1,1);
  /* signals for delegation of artefact creations */
  signal create_a_ForkImpl_req;
  signal create_a_ForkImpl_res(Pid);
  /* Artefact-Pool and pointer in pool */
  dcl a_ForkImpl_seq PIdSeq := (. .);
  dcl a_ForkImpl_ptr Integer := 0;
  /* returns an artefact instance reference */
  /* implements a pool of size POOLSIZE */
  /* creates a new artefact if pool is not yet of */
  /* size POOLSIZE, otherwise returns the "next" */
  /* artefact. */
  exported procedure get_artefact_a_ForkImpl -> PId;
    dcl new_pid PId;
    start;
     decision length(a_ForkImpl_seq);
      (0:1):
        /* delegate creation of artefact */
```

```
          output create_a_ForkImpl;
            nextstate wait4response;
        else:
          task a_ForkImpl_ptr := a_ForkImpl_ptr+1;
          task { if (a_ForkImpl_ptr>length(a_ForkImpl_seq))
                 a_ForkImpl_ptr := 1;
               };
          return extract(a_ForkImpl_seq, a_ForkImpl_ptr);
      enddecision;
      /* delegation continued ... */
      state wait4response;
      input create_a_ForkImpl_res(new_pid);
      task a_ForkImpl_seq := a_ForkImpl_seq // new_pid;
      return offspring;
    endprocedure;
    start;
    nextstate wait_for_signal;
    /* create artefact instance for procedure */
    state wait_for_signal;
    input create_a_ForkImpl_req;
    create artefact_a_ForkImpl;
     output create_a_ForkImpl_res(offspring) to sender;
    nextstate -;
  endprocess;
  /* \-------------------------------/ */

  /* this is the interactionmanagementrepresentation */
  /* for interface i_fork. handles procedure calls   */
  /* from the environment */
  process interaction_i_Fork(0,1);
    exported as <<interface exported_i_Fork>>obtain_Fork
    procedure obtain_fork(in eater o_Philosopher)
            raise ForkNotAvailable;
      dcl p PId;
      start;
      task p := get_artefact_a_ForkImpl;
      call obtain_Fork(who, sender) to p;
      return;
        exceptionhandler defhandler;
         handle ForkNotAvailable;
         raise ForkNotAvailable;
      endexceptionhandler defhandler;
    endprocedure;
    exported as <<interface exported_i_Fork>>release_Fork
    procedure release_fork(in eater o_Philosopher)
      raise NotTheEater;
      dcl p PId;
      start;
      task p := get_artefact_a_ForkImpl;
      call obtain_Fork(who,sender) to p;
      return;
        exceptionhandler defhandler;
         handle NotTheEater;
         raise NotTheEater;
      endexceptionhandler defhandler;
    endprocedure;
  endprocess;

  /* portmanagement */
  exported as <<package phil_interface/
       package DiningPhilosopher>>provide_fork
  procedure provide_fork -> exported_i_Fork;
    start;
    return call provide_fork;
```

```
endprocedure;
exported as <<package philo_interface/
   package DiningPhilosopher/package o_Fork>>provide
procedure provide(s string) -> PId
     raise NoSuchPort;
  start;
  decision s;
     (='fork'): return call provide_fork;
    else: raise NoSuchPort;
  enddecision;
endprocedure;
exported as <<package philo_interface/
     package DiningPhilosopher/package o_Fork>>port_connect
procedure port_connect(s string) -> PId
   raise NoSuchPort,AlreadyConnected;
  start;
  raise NoSuchPort;
endprocedure;
exported as <<package philo_interface/
     package DiningPhilosopher/package o_Fork>>port_disconnect
procedure port_disconnect(s string) -> PId
    raise NoSuchPort,NotConnected;
  start;
  raise NoSuchPort;
endprocedure;
  /* get/set for internal variables */
  exported as <<package philo_definition/package DiningPhilosopher/
          package o_Fork_data>>get_key
  procedure get_key -> ComponentKey;
    start;
    return the_key;
  endprocedure;


/* computes key and instantiates        */
/* interactionmanagementrepresentations */
start;
task the_key := <<package eODL>>generate_key;
create interaction_i_Fork;
call set_interaction_i_Fork(offspring);
task reference_port_fork := offspring;
nextstate initial_state;

/* process instance set of artefact a_ForkImpl */
process  artefact_a_ForkImpl(0,2): a_ForkImpl;

/* connects artefact and imr */
channel interaction_i_fork_a_fork_impl nodelay
   from interaction_i_Fork to artefact_a_ForkImpl
      with procedure <<package phil_definition/package DiningPhilosophers
          /package i_Fork>>obtain_fork,
          procedure <<package phil_definition/package DiningPhilosophers
          /package i_Fork>>release_fork;
endchannel;

channel ch_i_Fork nodelay
   from env via provides to interaction_i_Fork
      with exported_i_Fork;
   from interaction_i_Fork to env via provides
      with imported_i_Fork;
endchannel;

 channel ch_initial nodelay
  from env via initial to this
```

```
        with config_o_Fork;
    endchannel;

  endprocess type o_Fork;
  /* \--------------------------------/ */

  /* instance sets for factory and co type */
  process factory(1,1): o_Fork_factory;
  process cos(0,): o_Fork;

  channel factory_to_co nodelay
    from factory to co via initial;
  endchannel;
  channel initial_to_env nodelay
    from env via initial to co via initial;
  endchannel;
  channel provides_to_env nodelay
    from cos via provides to env via provides;
    from env via provides to cos via provides;
  endchannel;
  channel uses_to_env nodelay
    from cos via uses to env via uses;
    from env via uses to cos via uses;
  endchannel;


endblock type o_Fork_CO;
/* \--------------------------------/ */

package i_Philosopher;

    interface exported_i_Philosoper;
      procedure set_name(in string, in Pid);
    endinterface;

    interface imported_i_Philosoper;
    endinterface;

  endpackage i_Philosopher;

  package o_Philosopher_data;

    interface internal_data;
      procedure get_port_observer -> i_Observer;
      procedure get_port_left -> i_Fork;
      procedure get_port_right -> i_Fork;
      procedure get_interaction_i_Fork -> imported_i_Fork;
      procedure get_interaction_i_Observer -> imported_i_Observer;
      procedure get_interaction_i_Philosopher -> exported_i_Philosopher;
      procedure set_port_observer(i_Observer);
      procedure set_port_left(i_Fork);
      procedure set_port_right(i_Fork);
      procedure set_interaction_i_Fork(imported_i_Fork);
      procedure set_interaction_i_Observer(imported_i_Observer);
      procedure set_interaction_i_Philosopher(exported_i_Philosopher);
    endinterface;

  endpackage;

  signallist a_PhilosopherImpl :=
    procedure <<package phil_definition/package DiningPhilosophers/
                    package i_Philosopher>>set_name;
```

```
process type a_PhilosopherImpl with
   use (a_PhilosopherImpl_in);;
  referenced;


block type o_Philosopher_CO;

  gate factory
    in with <<package phil_interface/package DiningPhilosophers/
            package o_Philosopher/package factory>>o_Philosopher_factory;
  gate initial
    in with  <<package phil_interface/package DiningPhilosophers/
            package o_Philosopher>>o_Philosopher;
  gate provides
    in with <<package phil_interface/package DiningPhilosophers/
            package i_Philosopher>>exported_i_Philosopher;
    out with <<package phil_interface/package DiningPhilosophers/
            package i_Philosopher>>imported_i_Philosopher;
  gate uses
    out with <<package phil_interface/package DiningPhilosophers/
            package i_Fork>>exported_i_Fork,
             <<package phil_interface/package DiningPhilosophers
            /package i_Observer>>exported_i_Observer;

  process type o_Philosopher_factory;
    gate factory
      in with <<package phil_interface/package DiningPhilosophers/
            package o_Philosopher/package factory>>o_Philosopher_factory;
    channel nodelay
      from this via factory to env;
     endchannel;

     dcl keys ComponentKeysSeq;

    exported as <<package phil_interface/package DiningPhilosophers/
            package o_Philosopher/package factory>>generic_create
     procedure generic_create -> ComponentBase;
      dcl key ComponentKey;
       start;
      create co_instance;
       task key := call get_key to offspring;
       task keys := keys // key;
      return offspring;
    endprocedure;

    exported as <<package phil_interface/package DiningPhilosophers/
            package o_Philosopher/package factory>>create_o_Philosopher
     procedure create_o_Philosopher -> o_Philosopher;
       dcl key ComponentKey;
       start;
      create co_instance;
       task key := call get_key to offspring;
       task keys := keys // key;
      return offspring;
    endprocedure;

    exported as <<package phil_interface/package DiningPhilosophers/
            package o_Philosopher/package factory>>get_co_type
    procedure get_co_type -> string;
      start;
      return 'o_Philosopher';
    endprocedure;

    /* returns reference to a CO */
```

```
   exported as <<package phil_interface/package DiningPhilosophers/
          package o_Fork/package factory>>resolve_co
   procedure resolve_co(in key ComponentKey) -> ComponentBase
      raise InvalidIndex;
      dcl returnValue Pid;
      dcl i Integer := 1;
      start;
      task returnValue := Extract(keys,key);
      return returnValue;
   endprocedure;

  /* returns a list of CO keys */
   exported as <<package phil_interface/package DiningPhilosophers/
          package o_Fork/package factory>>list_co
   procedure list_co -> ComponentKeySeq;
      start;
      return keys;
   endprocedure;

endprocess type;


process type o_Philosopher;
  gate initial
    in with  <<package phil_interface/package DiningPhilosophers/
          package o_Philosopher>>o_Philosopher;

  channel ch_initial nodelay
    from env via initial to this with o_Philosopher;
  endchannel;

  gate provides
    in with <<package phil_interface/package DiningPhilosophers/
          package i_Philosopher>>exported_i_Philosopher;
    out with <<package phil_interface/package DiningPhilosophers/
          package i_Philosopher>>imported_i_Philosopher;
  channel ch_provides nodelay
    from env via provides to interaction_i_Philosopher
          with exported_i_Philosopher;
    from interaction_i_Philosopher via provides to env
          with imported_i_Philosopher;
  endchannel;

  gate uses
    out with <<package phil_interface/package DiningPhilosophers/
          package i_Fork>>exported_i_Fork,
              <<package phil_interface/package DiningPhilosophers/
          package i_Observer>>exported_i_Observer;

  dcl the_key string;

  process data_access(1,1);
   dcl reference_interaction_i_Philosopher exported_i_Philosopher;
   dcl reference_interaction_i_Fork imported_i_Fork;
   dcl reference_interaction_i_Observer imported_i_Observer;
   dcl reference_port_observer exported_i_Observer := Null;
   dcl reference_port_left exported_i_Fork := Null;
   dcl reference_port_right exported_i_Fork := Null;

 exported as <<package philo_definition/package DiningPhilosophers/
          package o_Philosopher_data>>get_port_observer
    procedure get_port_observer -> i_Observer;
        start;
        return reference_port_observer;
```

```
     endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
          package o_Philosopher_data>>set_port_observer
    procedure set_port_observer(ref i_Observer);
  start;
  task reference_port_observer := ref;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
     package o_Philosopher_data>>get_port_left
    procedure get_port_left -> i_Fork;
  start;
  return reference_port_left;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
     package o_Philosopher_data>>set_port_left
    procedure set_port_left(ref i_Fork);
  start;
  task reference_port_left := ref;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
          package o_Philosopher_data>>get_port_right
    procedure get_port_right -> i_Fork;
  start;
  return reference_port_right;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
          package o_Philosopher_data>>set_port_right
    procedure set_port_right(ref i_Fork);
  start;
  task reference_port_right := ref;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
          package o_Philosopher_data>>get_interaction_i_Fork
procedure get_interaction_req_i_Fork -> imported_i_Fork;
  start;
  return reference_interaction_i_Fork;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
          package o_Philosopher_data>>set_interaction_i_Fork
procedure set_interaction_i_Fork(ref imported_i_Fork);
  start;
  task reference_interaction_i_Fork := ref;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
     package o_Philosopher_data>>get_interaction_i_Observer
    procedure get_interaction_i_Observer -> imported_i_Observer;
  start;
  return reference_interaction_i_Observer;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
     package o_Philosopher_data>>set_interaction_i_Observer
    procedure set_interaction_i_Observer(ref imported_i_Observer);
  start;
  task reference_interaction_i_Observer := ref;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
     package o_Philosopher_data>>get_interaction_i_Philosopher
    procedure get_interaction_i_Philosopher -> exported_i_Philosopher;
  start;
  return reference_interaction_i_Philosopher;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
     package o_Philosopher_data>>set_interaction_i_Philosopher
    procedure set_interaction_i_Philosopher(ref exported_i_Philosopher);
```

```
   start;
   task reference_interaction_i_Philosopher := ref;
 endprocedure;

  endprocess;


/* channel artefact <--> state_access */
channel nodelay
  from artefact_a_ForkImpl to data_accessor
    with internal_state;
endchannel;


/* in case of CO type inheritance an instance of this type */
/* is contained in an "eODL-inherited" type. The following */
/* gate is used to interface with the state_access process */
/* of the inherited type. */
gate data_accessor
  in with internal_data;
channel route_state_access nodelay
  from
    env via data_accessor to data_access;
endchannel;

  process artefactmanagement(1,1);
    signal create_a_PhilosopherImpl_req;
    signal create_a_PhilosopherImpl_res(Pid);
    dcl a_PhilosopherImpl_ptr PId := Null;
    exported procedure get_artefact_a_PhilosopherImpl;
      dcl new_pid PId;
      start;
       decision a_PhilosopherImpl_ptr;
        (Null):
           output create_a_PhilosopherImpl;
            nextstate wait4response;
        else:
          return a_PhilosopherImpl_ptr;
      enddecision;

      state wait4response;
       input create_a_PhilosopherImpl_res(new_pid);
      task a_PhilosopherImpl_ptr := new_pid;
      return offspring;
    endprocedure;
    start;
    nextstate wait_for_signal;
    state wait_for_signal;
    input create_a_PhilosopherImpl;
    create artefact_a_PhilosopherImpl;
    nextstate -;
  endprocess;

  channel nodelay
  from
    artefact_a_PhilosopherImpl
    to state_accessor
    with internal_state;
  endchannel;

  process interaction_i_Philosopher(0,1);
     exported as <<package phil_definition/package DiningPhilosophers/
          package i_Philosopher>>set_name
    procedure set_name(in name string);
```

```
    dcl p PId;
    start;
     task p := get_artefact_a_PhilosopherImpl;
     call set_name(name,sender) to p;
     return;
   endprocedure;
 endprocess;

process interaction_i_Fork(0,1);
   exported as <<package phil_definition/package DiningPhilosophers/
         package i_Fork>>release_fork
  procedure obtain_fork(in eater o_Philosopher, in server PId);
    start;
     call obtain_fork(eater) to server;
     return;
   endprocedure;

  exported as <<package phil_definition/package DiningPhilosophers/
    package i_Fork>>release_fork
  procedure release_fork(in eater o_Philosopher, in server PId);
    start;
     call obtain_fork(eater) to server;
     return;
   endprocedure;
 endprocess;

 process interaction_i_Observer(0,1);
    dcl carry_PhilosopherState PhilosopherState;
    dcl consumer PId;
    start;
    nextstate signal_handler;
    state signal_handler;
    input PhilosopherState(carry_PhilosopherState, consumer);
    output PhilosopherState(carry_PhilosopherState) to consumer;
    nextstate -;
 endprocess;


process artefact_a_PhilosopherImpl(0,1): a_PhilosopherImpl;

  exported as <<package philo_interface/package DiningPhilosophers/
         package o_Philosopher>>link_observer
  procedure link_observer -> exported_i_Observer;
     start;
      decision call get_port_observer;
     (Null): call set_port_observer(ref);
      else: raise AlreadyConnected;
      enddecision;
  endprocedure;
  exported as <<package philo_interface/package DiningPhilosophers/
         package o_Philosopher>>unlink_observer
  procedure unlink_observer -> exported_i_Observer;
     start;
      decision call get_port_observer;
     (Null): raise NotConnected;
      else: call set_port_observer(ref);
      enddecision;
  endprocedure;
  exported as <<package philo_interface/package DiningPhilosophers/
         package o_Philosopher>>link_left
  procedure link_left(ref imported_i_Fork)
         raise AlreadyConnected;
     start;
      decision call get_port_left;
```

```
    (Null): call set_port_left(ref);
     else: raise AlreadyConnected;
     enddecision;
  endprocedure;
  exported as <<package philo_interface/package DiningPhilosophers/
        package o_Philosopher>>unlink_left
  procedure unlink_left
        raise NotConnected;
    start;
     decision call get_port_left;
    (Null): raise NotConnected;
     else: call set_port_left(Null);
     enddecision;
  endprocedure;
  exported as <<package philo_interface/package DiningPhilosophers/
        package o_Philosopher>>link_right
  procedure link_right(ref imported_i_Fork)
        raise AlreadyConnected;
    start;
     decision call get_port_right;
    (Null): call set_port_right ( ref );
     else: raise AlreadyConnected;
     enddecision;
  endprocedure;
  exported as <<package philo_interface/package DiningPhilosophers/
        package o_Philosopher>>unlink_right
  procedure unlink_right
        raise NotConnected;
    start;
     decision call get_port_right;
    (Null): raise NotConnected;
     else: call set_port_right(Null);
     enddecision;
  endprocedure;
  exported as <<package philo_interface/package DiningPhilosophers/
        package o_Philosopher>>link_observer
  procedure link_observer(ref imported_i_Observer)
        raise AlreadyConnected;
    start;
     decision call get_port_observer;
    (Null): call set_port_observer(ref);
     else: raise AlreadyConnected;
     enddecision;
  endprocedure;
  exported as <<package philo_interface/package DiningPhilosophers/
        package o_Philosopher>>unlink_observer
  procedure unlink_observer
        raise NotConnected;
    start;
     decision call get_port_observer;
    (Null): raise NotConnected;
     else: call set_port_observer(Null);
     enddecision;
  endprocedure;
exported as <<package philo_interface/
     package DiningPhilosopher/package o_Philosopher>>provide
procedure provide(s string) -> PId
     raise NoSuchPort;
  start;
   enddecision;
endprocedure;
exported as <<package philo_interface/
     package DiningPhilosopher/package o_Philosopher>>port_connect
procedure link(ref Pid, s string)
```

```
      raise NoSuchPort,AlreadyConnected;
    start;
    decision s;
      (='observer'): call link_observer(ref);
      (='left'): return call link_left(ref);
      (='right'): return call link_right(ref);
      else: raise NoSuchPort;
    return ;
  endprocedure;
  exported as <<package philo_interface/
       package DiningPhilosopher/package o_Philosopher>>port_disconnect
  procedure unlink(s string) -> PId
      raise NoSuchPort,NotConnected;
    start;
    decision s;
      (='observer'): call unlink_observer;
      (='left'): return call unlink_left;
      (='right'): return call unlink_right;
      else: raise NoSuchPort;
    return ;
  endprocedure;

 start;
    task the_key := <<package eODL>>generate_key;
 create interaction_i_Fork;
    call set_interaction_i_Fork(offspring);
 create interaction_i_Philosopher;
 call set__interaction_i_Philosopher(offspring);
 create interaction_i_Observer;
 call set__interaction_i_Observer(offspring);
 nextstate initial_state;

  exported as <<package phil_interface/package DiningPhilosophers/
       package o_Philosophers>>get_key
  procedure get_key -> ComponentKey;
    start;
    return the_key;
  endprocedure;

endprocess type;

process factory(1,1): o_Fork_factory;
process cos(0,): o_Fork;


channel factory_to_co nodelay
  from factory to co via initial;
endchannel;

channel initial_to_env nodelay
  from co via initial to env via initial;
endchannel;

channel provides_to_env nodelay
  from co via provides to env via provides;
endchannel;

channel uses_to_env nodelay
  from co via uses to env via uses;
endchannel;

channel factory_to_env nodelay
  from env via factory to  factory;
endchannel;
```

```
  endblock type;

package o_Observer_data;
    interface internal_data;
      procedure get_interaction_i_Observer -> exported_i_Observer;
      procedure get_port_observer -> exported_i_Observer;
      procedure set_interaction_i_Observer(exported_i_Observer);
      procedure set_port_observer(exported_i_Observer);
    endinterface;
  endpackage;

  signal PhilosopherState(PState, Pid);

  signallist a_Observer_in =
       <<package phil_definition/package DiningPhilosophers>>PhilosopherState;

  /* artefact: referenced definition */
  process type a_ForkImpl with
     use (a_Observer_in);;
    referenced;



  block type o_Observer_CO;


    gate factory
      in with <<package phil_interface/package DiningPhilosophers/
              package o_Observer/package factory>>o_Observer_factory;
    gate initial
      in with  <<package phil_interface/package DiningPhilosophers/
              package o_Observer>>o_Observer;
    gate provides
      in with <<package phil_interface/package DiningPhilosophers/
              package i_Observer>>exported_i_Observer;
      out with <<package phil_interface/package DiningPhilosophers/
              package i_Observer>>imported_i_Observer;

    process type o_Observer_factory;
      gate factory
        in with <<package phil_interface/package DiningPhilosophers/
              package o_Observer/package factory>>o_Observer_factory;
      channel nodelay
        from this via factory to env;
       endchannel;

       dcl keys ComponentKeysSeq;

      exported as <<package phil_interface/package DiningPhilosophers/
              package o_Observer/package factory>>generic_create
       procedure generic_create -> ComponentBase;
        dcl key ComponentKey;
         start;
        create co_instance;
         task key := call get_key to offspring;
         task keys := keys // key;
        return offspring;
      endprocedure;

      exported as <<package phil_interface/package DiningPhilosophers/
              package o_Observer/package factory>>create_o_Observer
       procedure create_o_Observer -> o_Observer;
```

```
        dcl key ComponentKey;
        start;
       create co_instance;
        task key := call get_key to offspring;
        task keys := keys // key;
       return offspring;
     endprocedure;

     exported as <<package phil_interface/package DiningPhilosophers/
             package o_Observer/package factory>>get_co_type
     procedure get_co_type -> string;
       start;
       return 'o_Observer';
     endprocedure;

     exported as <<package phil_interface/package DiningPhilosophers/
             package o_Observer/package factory>>resolve_co
     procedure resolve_co(in key ComponentKey) raise InvalidIndex;
        dcl returnValue Pid;
        dcl i Integer := 1;
        start;
        task returnValue := Extract(keys,key);
        return returnValue;
     endprocedure;

     exported as <<package phil_interface/package DiningPhilosophers/
             package o_Observer/package factory>>list_co
     procedure list_co -> ComponentKeySeq;
        start;
        return keys;
     endprocedure;

  endprocess type;

  process type o_Observer;
    gate initial
      in with  <<package phil_interface/
          package DiningPhilosopher/package o_Observer>>o_Observer;
    gate provides
      in with <<package phil_interface/
         package DiningPhilosopher/package o_Observer>>exported_i_Observer;
      out with <<package phil_interface/package DiningPhilosopher/
          package o_Observer>>imported_i_Observer;


  dcl reference_interaction_i_Observer exported_i_Fork;
  dcl reference_port_observer exported_i_Fork;
  dcl the_key string;

  process data_access(1,1);
    exported as <<package philo_definition/
        package DiningPhilosopher/package o_Fork_data>>get_port_observer
    procedure get_port_fork -> exported_i_Observer;
      start;
      return reference_port_observer;
    endprocedure;
    exported as <<package philo_definition/
        package DiningPhilosopher/package
o_Fork_state>>get_interaction_i_Observer
    procedure get_interaction_i_Observer -> exported_i_Observer;
      start;
      return reference_interaction_i_Observer;
    endprocedure;
  endprocess;
```

```
      exported as <<package philo_definition/
         package DiningPhilosopher/package o_Fork_data>>set_port_observer
      procedure set_port_fork(in ref exported_i_Observer);
        start;
        reference_port_observer := ref;
      endprocedure;
      exported as <<package philo_definition/
         package DiningPhilosopher/package
o_Fork_state>>set_interaction_i_Observer
      procedure set_interaction_i_Observer(exported_i_Observer);
        start;
        reference_interaction_i_Observer := ref;
      endprocedure;
    endprocess;

    gate state_accessor
      in with internal_state;
    channel route_state_access nodelay
      from
        state_accessor via state_accessor to env;
    endchannel;
    channel nodelay
      from artefact_a_Observer to state_accessor
        with internal_state;
    endchannel;


    process artefactmanagement(1,1);
      signal create_a_Observer_req;
      signal create_a_Observer_res(Pid);
      dcl a_Observer Pid := Null;
      exported procedure get_artefact_a_Observer;
        dcl new_pid PId;
        start;
         decision a_ForkImpl;
          (Null):
             output create_a_ForkImpl;
              nextstate wait4response;
          else:
            return a_Observer;
        enddecision;

        state wait4response;
        input create_a_ForkImpl_res(new_pid);
        task a_ForkImpl_seq := a_ForkImpl_seq // new_pid;
        return offspring;
      endprocedure;
      start;
      nextstate wait_for_signal;
      state wait_for_signal;
      input create_a_Observerl;
      create artefact_a_Observer;
      nextstate -;
    endprocess;

    process interaction_i_Observer(0,1);
      dcl p PId;
       dcl e_PState pstate;
      start;
      nextstate wait4signal;
       state wait4signal;
      input PhilosopherState(pstate);
      task p := get_artefact_a_Observer;
      output PhilosopherState(pstate, sender) to p;
```

```
    nextstate -;
  endprocess;

process  artefact_a_Observer(0,1): a_Observer;

    exported as <<package philo_interface/
         package DiningPhilosopher/package o_Observer>>provide_observer
    procedure provide_observer -> exported_i_Observer;
      start;
      return call get_provide_observer to data_access;
    endprocedure;
    exported as <<package philo_interface/
         package DiningPhilosopher/package o_Observer>>provide
    procedure provide(s string) -> PId
         raise NoSuchPort;
      start;
      decision s;
       ('observer'): return reference_port_observer;
       else: raise NoSuchPort;
       enddecision;
    endprocedure;
    exported as <<package philo_interface/
         package DiningPhilosopher/package o_Observer>>port_connect
    procedure port_connect(s string) -> PId
        raise NoSuchPort,AlreadyConnected;
      start;
      raise NoSuchPort;
    endprocedure;
    exported as <<package philo_interface/
         package DiningPhilosopher/package o_Observer>>port_disconnect
    procedure port_disconnect(s string) -> PId
        raise NoSuchPort,NotConnected;
      start;
      raise NoSuchPort;
    endprocedure;


  start;
  task the_key := <<package eODL>>generate_key;
  create interaction_i_Observer;
  call set_interaction_i_Observer(offspring) to data_access;
  call set_port_observer(offspring) to data_access;
  nextstate initial_state;

  channel ch_provides nodelay
     from env via provides to interaction_i_Observer
        with exported_i_Observer;
     from interaction_i_Fork to env via provides
        with imported_i_Observer;
  endchannel;

   channel ch_initial_port nodelay
     from env via initial to portmanagement
        with config_o_Observer;
  endchannel;

   channel ch_initial nodelay
     from env via initial to this with imported_o_Observer;
     from this via initial to env with exported_o_Observer;
   endchannel;

  exported as <<package phil_interface/package DiningPhilosophers/
       package o_Observer>>get_key
  procedure get_key -> ComponentKey;
```

```
      start;
      return the_key;
    endprocedure;
  endprocess type;

  process factory(1,1): o_Observer_factory;
  process co(0,): o_Observer;

  channel factory_to_co nodelay
    from factory to co via initial;
  endchannel;

  channel initial_to_env nodelay
    from co via initial to env via initial;
  endchannel;

  channel provides_to_env nodelay
    from co via provides to env via provides;
  endchannel;

  channel uses_to_env nodelay
    from co via uses to env via uses;
  endchannel;


  endblock type;

endpackage;

endpackage DiningPhilosophers;

endpackage phil_definition;
```

Appendice II

Traitement de l'information et outils

II.1 Introduction

L'existence d'un **métamodèle** simple et complet constitue une base stable pour le développement de logiciel, même dans des domaines d'application complexes. Pour que la technique soit utilisable et notamment pour que les développeurs puissent l'utiliser facilement, des outils appropriés sont nécessaires. D'une manière générale, ces outils peuvent prendre en charge le processus de modélisation proprement dit, comme un éditeur et/ou un simulateur, ou ils peuvent couvrir plusieurs phases, comme l'implémentation et le **déploiement** sur des **plates-formes cibles**.

En raison de la grande diversité des tâches qui doivent être remplies par des outils traitant un modèle unique depuis le début de la spécification et jusqu'au **déploiement** et à l'instanciation, différents outils devraient être utilisés en chaîne. Il faut donc avoir un format d'échange d'un outil à l'autre. A cette fin, il existe au moins une notation normalisée par défaut. En raison de l'application de l'architecture MOF de l'OMG, on utilise, pour le **métamodèle**, une représentation en XML, qui peut alors servir de format d'échange entre les différents outils.

La définition concrète des outils et de leur fonctionnalité ne peut pas faire l'objet d'une normalisation. Même si, dans la pratique, les outils pourront être conçus arbitrairement et être constitués en réalité de chaînes d'outils, on considère, dans les paragraphes qui suivent, des outils pour des aspects particuliers. Les outils réels pourront couvrir plusieurs de ces aspects.

II.2 Outils de modélisation

Les outils se rapportant à la manipulation d'informations de modèle peuvent utiliser des modèles dans une représentation arbitraire, la seule restriction étant qu'il doit exister un mappage entre la représentation et le métamodèle. Ces représentations peuvent être aussi bien des langages de programmation que des notations graphiques, comme l'UML. N'importe quel outil qui prend en charge le traitement d'une telle représentation peut être utilisé, sans aucune restriction.

Comme le **métamodèle** couvre pratiquement tout le cycle de vie du logiciel, il existe diverses actions de modélisation possibles, chacune pouvant être effectuée au moyen d'outils. Un modèle peut être élargi progressivement par l'ajout d'informations additionnelles dans plusieurs itérations de modélisation à des dates différentes. Ainsi, il est possible d'utiliser une collection de **types de CO** existants pour spécifier un **assemblage** ultérieurement, et de donner ensuite une attribution concrète de cet **assemblage** sur une **plate-forme cible**. En général, les actions de modélisation sont les suivantes, dans l'ordre dans lequel elles sont effectuées:

- spécification de **types de CO**;
- spécification d'un **assemblage**;
- paquétisation de l'implémentation (des **composants logiciels**);
- modélisation de l'environnement;
- attribution des **composants logiciels** sur une **plate-forme cible**;
- instanciation des **types de CO** et de l'**assemblage**.

La spécification de **types de CO** conformément au **métamodèle** constitue la première étape. Après cela, il est possible de définir des assemblages. Chaque type peut être utilisé dans un nombre arbitraire d'assemblages. L'étape suivante consiste à définir l'implémentation pour un **assemblage** tout entier contenant le code d'implémentation à utiliser pour les **types de CO** regroupés en **composants logiciels**. La paquétisation d'implémentations peut être effectuée par des outils d'archivage, tels que zip. Après cela, le modèle conjointement avec le paquetage contenant l'implémentation de l'**assemblage** peuvent être déployés au niveau de plates-formes de clients. Pour déployer un **assemblage**, il faut d'abord déterminer la répartition des **CO**. Au cours de ce processus, le modèle est enrichi par des informations additionnelles, se rapportant principalement à l'environnement concret et aux besoins particuliers d'un client. On compare le modèle de l'environnement cible et le modèle de l'**assemblage** afin de trouver une attribution appropriée de chaque **CO** à un nœud de la plate-forme. Cela peut se faire manuellement, de préférence par l'administrateur du système, ou semi-automatiquement par une fonction automatique afin de déterminer une **configuration initiale** de l'**assemblage**. Dans l'un ou l'autre cas, les conditions relatives à chaque **CO** de l'**assemblage** dans l'environnement cible doivent être respectées. On ne spécifie pas comment obtenir le modèle de l'environnement. Ce modèle peut être obtenu directement à partir de l'environnement cible, auquel cas une architecture particulière est nécessaire. A la fin de l'étape d'attribution, le modèle contient des informations sur l'endroit où chaque **composant logiciel** doit être installé. L'instanciation des **types de CO** ou de l'assemblage peut faire partie du modèle. L'outil doit tenir à jour le modèle au cours de l'**exécution**.

II.3 Outils de génération

Les efforts à déployer pour implémenter manuellement les entités du modèle peuvent être considérablement réduits grâce au recours à des outils d'aide à l'implémentation. Un code peut être généré pour tous les langages d'implémentation pour lesquels il existe un mappage avec le **métamodèle**. En général, les informations suivantes peuvent être générées à partir du modèle:

- les squelettes des **types de CO**;
- le code pour les spécifications de la qualité de service.

Le code généré peut offrir aux implémenteurs un cadre qui leur sert de base pour implémenter les **types de CO**. Tant que le **métamodèle** ne contient aucun aspect comportemental, aucune implémentation de logique commerciale ne peut être générée automatiquement. Elle doit être insérée manuellement.

II.4 Outils de déploiement

Pour le déploiement y compris l'instanciation d'assemblages sur la **plate-forme cible** d'un client, il faut des outils appropriés, mais la plate-forme proprement dite joue également un rôle important. Les outils réels à utiliser pour le **déploiement** sont donc étroitement liés aux architectures concrètes de plate-forme avec lesquelles ils doivent interagir. Les outils ne peuvent pas être indépendants tant qu'il n'existe pas d'architecture de plate-forme normalisée, avec laquelle ils peuvent collaborer.

D'une manière générale, le processus de **déploiement** comprend plusieurs tâches allant de la détermination d'une répartition correcte à l'installation et à l'instanciation des logiciels. Les tâches communes du **déploiement** peuvent être exécutées par des outils:

- modélisation de l'environnement;
- attribution des **composants logiciels** sur une **plate-forme cible**;
- installation des **composants logiciels** sur une **plate-forme cible**;
- instanciation de l'**assemblage** et des **types de CO**;
- traitement des contraintes et des actions.

Les tâches consistant à modéliser l'environnement et à attribuer les **composants logiciels** sur une **plate-forme cible** ont déjà été mentionnées dans le contexte des outils de modélisation. Cela s'explique par le fait que ces tâches se traduisent en pratique par une extension du modèle. En réalité, ces tâches sont censées être exécutées dans la plupart des cas dans le cadre du **déploiement** d'un **assemblage**. Comme cela a déjà été mentionné, une plate-forme peut prendre en charge des outils permettant d'obtenir les informations de modélisation de l'environnement. Une fois que ces tâches sont réalisées et qu'une attribution correcte des **composants logiciels** sur une **plate-forme cible** a été déterminée, l'étape suivante consiste à télécharger et installer les logiciels sur le nœud spécifié. L'**assemblage** ou les **types de CO** peuvent ensuite être instanciés à l'aide d'un autre outil ou d'une autre plate-forme. Enfin, au stade de l'**exécution**, les contraintes et les actions contenues dans le modèle doivent être traitées d'une manière ou d'une autre.

La Figure II.1 illustre la chaîne d'outils allant de la modélisation au **déploiement**.

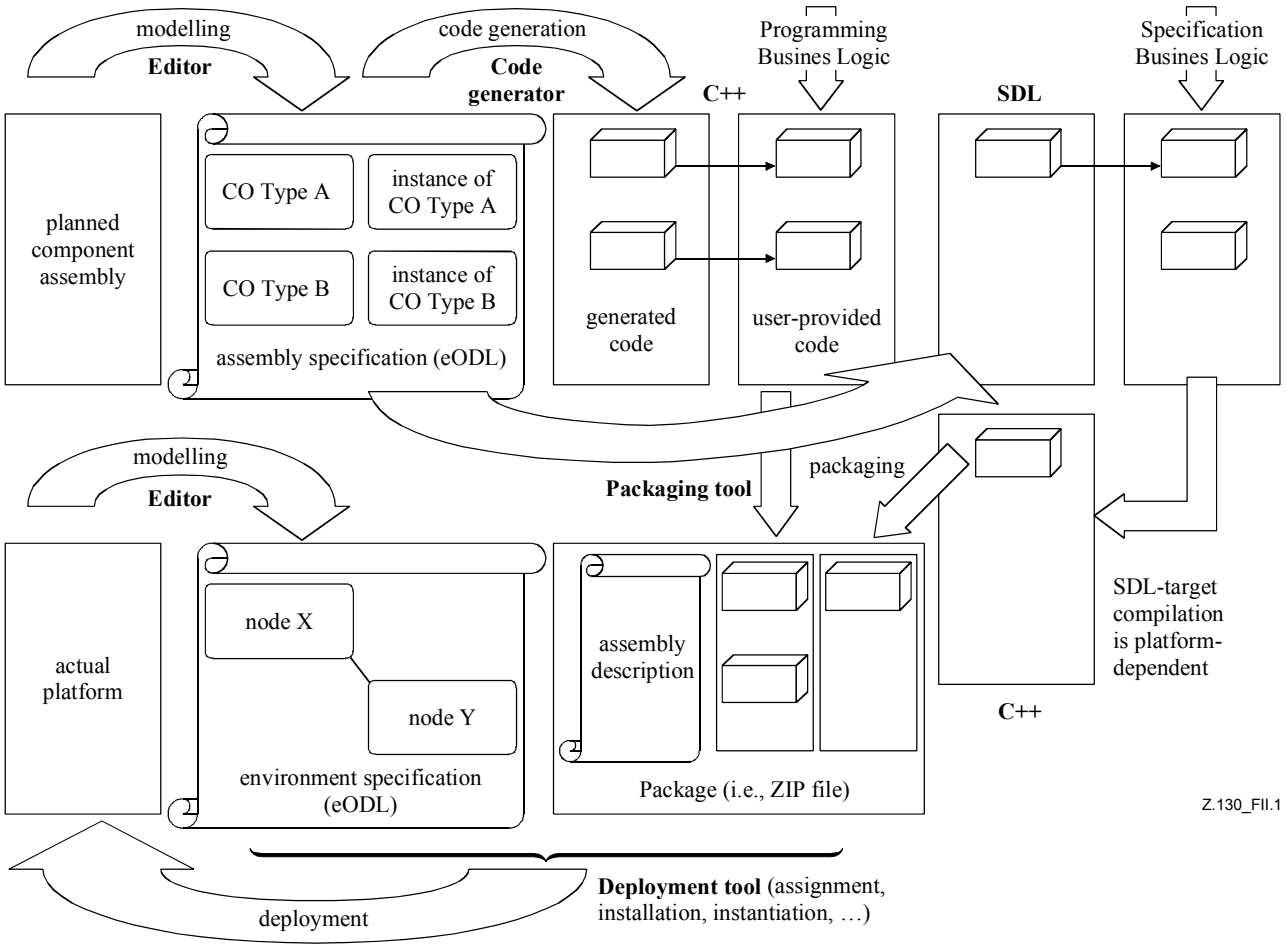


Figure II.1/Z.130 – Traitement de l'information

SÉRIES DES RECOMMANDATIONS UIT-T

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

Imprimé en Suisse
Genève, 2004