



UNION INTERNATIONALE DES TÉLÉCOMMUNICATIONS

**UIT-T**

SECTEUR DE LA NORMALISATION  
DES TÉLÉCOMMUNICATIONS  
DE L'UIT

**Z.100**

(11/99)

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

Techniques de description formelle – Langage de  
description et de spécification (SDL)

---

**SDL: langage de description et de spécification**

Recommandation UIT-T Z.100

(Antérieurement Recommandation du CCITT)

---

## RECOMMANDATIONS UIT-T DE LA SÉRIE Z

### LANGAGES ET ASPECTS INFORMATIQUES GÉNÉRAUX DES SYSTÈMES DE TÉLÉCOMMUNICATION

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

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

# RECOMMANDATION UIT-T Z.100

## SDL: LANGAGE DE DESCRIPTION ET DE SPÉCIFICATION

### Résumé

#### Domaine d'application-Objectifs

La présente Recommandation définit le langage de description et de spécification (SDL, *specification and description language*) destiné à être utilisé dans les spécifications et descriptions non ambiguës des systèmes de télécommunication. Le domaine d'application du SDL est précisé au paragraphe 1. La présente Recommandation constitue un manuel de référence du langage.

#### Couverture

Le SDL fournit des concepts pour la description du comportement et des données, ainsi que pour la structuration (notamment pour les grands systèmes). La description du comportement est fondée sur les machines à états finis étendus communiquant par message. La description des données est fondée sur les types de données, d'objets et de valeurs. Quant à la structuration, elle est fondée sur la décomposition hiérarchique et la hiérarchie des types. Ces fondements du SDL sont élaborés dans les principaux paragraphes correspondants de la présente Recommandation. La représentation graphique est une caractéristique propre du SDL.

#### Applications

Le langage SDL trouve son application au sein des organes de normalisation et dans l'industrie. Les principaux domaines d'application pour lesquels le SDL a été conçu sont précisés en 1.2, mais le SDL convient généralement à la description des systèmes réactifs. Ses applications couvrent une large gamme qui va de la description des exigences à l'implémentation.

#### Etat/Stabilité

La présente Recommandation constitue le manuel de référence complet du langage, appuyé par des directives d'utilisation données dans le Supplément 1. L'Annexe F donne une définition formelle des règles sémantiques du langage SDL. Le texte principal de la présente Recommandation est stable et doit être publié immédiatement pour répondre à des besoins commerciaux, mais des études ultérieures sont nécessaires pour compléter l'Annexe F. L'Appendice I indique l'état de la Recommandation Z.100, il convient de la mettre à jour en fonction de l'évolution des études ultérieures. Bien que d'autres extensions de langage soient prévues dans le futur, le SDL-2000 tel qu'il est défini dans la présente Recommandation devrait répondre aux besoins de la plupart des utilisateurs pendant quelques années. La version actuelle se fonde sur une grande expérience d'utilisation du SDL et sur des besoins récents des utilisateurs.

Le texte principal de la présente Recommandation est accompagné des annexes suivantes:

- Annexe A Index des non-terminaux.
- Annexe B Réservée pour une utilisation ultérieure – L'Annexe B (03/93) n'est pas valide.
- Annexe C Réservée pour une utilisation ultérieure – L'Annexe C (03/93) n'est plus en vigueur.
- Annexe D Données prédéfinies du SDL.
- Annexe E Réservée pour les exemples.

- Annexe F Définition formelle (études supplémentaires nécessaires pour le SDL 2000).
- Appendice I Etat de la Z.100, des documents et Recommandations associés.
- Appendice II Directives concernant la maintenance du SDL.
- Appendice III Conversion systématique du SDL 92 en SDL 2000.

UIT-T Z.100 a également un supplément publié séparément:

- Z.100 Suppl.1 SDL+ Méthodologie: utilisation de MSC et SDL (avec la notation ASN.1).

### **Travaux associés**

La Recommandation Q.65 décrit une méthode d'utilisation du SDL à l'intérieur des normes. La Recommandation Z.110 fournit une stratégie recommandée pour introduire une technique de description formelle telle que le langage SDL dans les normes. On trouvera sur le site <http://www.sdl-forum.org> des références à des textes supplémentaires concernant le SDL, ainsi que des informations sur son utilisation industrielle.

### **Historique**

Depuis 1976, le CCITT et l'UIT-T ont recommandé différentes versions du SDL. La présente version est une révision de la Recommandation Z.100 (03/93) et comprend l'Addendum 1 de la Recommandation Z.100 (10/96) et des parties de la Recommandation Z.105 (03/95).

Par comparaison avec le SDL défini dans la version de 1992, la version définie dans le présent texte a été étendue au domaine des données orientées objet, à l'harmonisation de plusieurs caractéristiques pour simplifier le langage, ainsi qu'à des caractéristiques permettant d'augmenter la capacité d'utilisation du SDL avec d'autres langages tels que ASN.1, ITU-T ODL (Z.130), CORBA et UML. D'autres modifications mineures ont été introduites, tout en veillant à ne pas rendre non valides les documents existants concernant le SDL; certaines modifications pourraient impliquer la mise à jour de certaines descriptions pour utiliser la présente version. Le paragraphe 1.5 donne les détails des modifications introduites.

### **Source**

La Recommandation UIT-T Z.100, révisée par la Commission d'études 10 de l'UIT-T (1997-2000), a été approuvée le 19 Novembre 1999 selon la procédure définie dans la Résolution n° 1 de la CMNT.

### **Mots clés**

Décomposition hiérarchique, machine d'état, orientation objet, représentation graphique, spécification fonctionnelle, technique de description formelle, technique de spécification, types de données.

## AVANT-PROPOS

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

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

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

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

## NOTE

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

## DROITS DE PROPRIÉTÉ INTELLECTUELLE

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

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

© UIT 2000

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

## TABLE DES MATIÈRES

	<b>Page</b>
1	Domaine d'application ..... 1
1.1	Objectifs..... 1
1.2	Application..... 1
1.3	Spécification d'un système ..... 2
1.4	Différences entre le SDL 88 et le SDL 92 ..... 3
1.5	Différences entre le SDL 92 et le SDL 2000 ..... 4
2	Références normatives ..... 5
3	Définitions ..... 6
4	Abréviations..... 7
5	Conventions ..... 7
5.1	Grammaires de SDL ..... 7
5.2	Définitions fondamentales ..... 8
5.2.1	Définition, type et instance ..... 8
5.2.2	Environnement..... 9
5.2.3	Erreur ..... 9
5.3	Présentation..... 9
5.3.1	Structuration du texte..... 9
5.3.2	Intitulés ..... 10
5.4	Métalangages ..... 12
5.4.1	Méta IV ..... 13
5.4.2	BNF ..... 14
5.4.3	Métalangage applicable à la grammaire graphique ..... 15
6	Règles générales..... 16
6.1	Règles lexicales..... 16
6.2	Macro ..... 21
6.2.1	Règles lexicales supplémentaires ..... 22
6.2.2	Définition de macro ..... 22
6.2.3	Appel de macro..... 23
6.3	Règles de visibilité, noms et identificateurs ..... 24
6.4	Texte informel..... 28
6.5	Règles applicables aux dessins ..... 28
6.6	Subdivision des dessins ..... 29
6.7	Commentaire..... 30
6.8	Extension de texte..... 30
6.9	Symbole de texte..... 31

	<b>Page</b>
7	Organisation des spécifications SDL ..... 31
7.1	Cadre d'application ..... 31
7.2	Paquetage ..... 32
7.3	Définition référencée ..... 36
8	Concepts structurels ..... 37
8.1	Types, instances et accès ..... 38
	8.1.1 Définitions de types structurels ..... 38
	8.1.2 Expression de type ..... 44
	8.1.3 Définitions fondées sur les types ..... 45
	8.1.4 Type abstrait ..... 48
	8.1.5 Références de type ..... 48
	8.1.6 Accès ..... 53
8.2	Paramètres de contexte ..... 56
	8.2.1 Paramètre de contexte de type d'agent ..... 57
	8.2.2 Paramètre de contexte d'agent ..... 58
	8.2.3 Paramètre de contexte de procédure ..... 58
	8.2.4 Paramètre de contexte de procédure distante ..... 58
	8.2.5 Paramètre de contexte de signal ..... 58
	8.2.6 Paramètre de contexte de variable ..... 59
	8.2.7 Paramètre de contexte de variable distante ..... 59
	8.2.8 Paramètre de contexte de temporisateur ..... 59
	8.2.9 Paramètre de contexte de synonyme ..... 59
	8.2.10 Paramètre de contexte de sorte ..... 60
	8.2.11 Paramètre de contexte d'exception ..... 60
	8.2.12 Paramètre de contexte de type d'état composite ..... 60
	8.2.13 Paramètre de contexte d'accès ..... 61
	8.2.14 Paramètre de contexte d'interface ..... 61
8.3	Spécialisation ..... 61
	8.3.1 Adjonction de propriétés ..... 61
	8.3.2 Type virtuel ..... 63
	8.3.3 Transition/sauvegarde virtuelle ..... 65
	8.3.4 Méthodes virtuelles ..... 66
	8.3.5 Initialisation virtuelle par défaut ..... 66
8.4	Associations ..... 66
9	Agents ..... 69
9.1	Système ..... 76
9.2	Bloc ..... 77
9.3	Processus ..... 78

	<b>Page</b>
9.4 Procédure .....	80
10 Communication.....	85
10.1 Canal .....	85
10.2 Connexion.....	87
10.3 Signal .....	88
10.4 Définition de liste de signaux .....	89
10.5 Procédures distantes.....	90
10.6 Variables distantes .....	94
11 Comportement .....	97
11.1 Départ.....	97
11.2 Etat .....	97
11.2.1 Etat de base.....	98
11.2.2 Application d'état composite .....	100
11.3 Entrée .....	101
11.4 Entrée prioritaire .....	103
11.5 Signal continu .....	104
11.6 Condition de validation.....	104
11.7 Sauvegarde.....	105
11.8 Transition implicite.....	106
11.9 Transition spontanée .....	106
11.10 Etiquette .....	107
11.11 Machine à états et état composite .....	108
11.11.1 Graphe d'état composite.....	109
11.11.2 Agrégat d'état.....	111
11.11.3 Point de connexion d'état.....	114
11.11.4 Connexion.....	115
11.12 Transition .....	116
11.12.1 Corps de transition.....	116
11.12.2 Termineur de transition.....	118
11.13 Action.....	123
11.13.1 Tâche .....	123
11.13.2 Création .....	124
11.13.3 Appel de procédure.....	126
11.13.4 Sortie.....	127
11.13.5 Décision .....	129
11.14 Liste d'instructions .....	132
11.14.1 Instruction composée .....	133



	<b>Page</b>
11.14.2 Actions et terminateurs de transition en tant qu'instructions .....	134
11.14.3 Expressions en tant qu'instructions.....	135
11.14.4 Instruction Si.....	135
11.14.5 Instruction de décision.....	136
11.14.6 Instruction de boucle.....	136
11.14.7 Instructions d'interruption et instructions étiquetées .....	139
11.14.8 Instruction vide .....	139
11.14.9 Instruction d'exception.....	139
11.15 Temporisateur .....	140
11.16 Exception .....	142
11.16.1 Gestionnaire d'exception.....	142
11.16.2 On-Exception.....	144
11.16.3 Traitement.....	146
12 Données .....	148
12.1 Définitions de données.....	149
12.1.1 Définition des types de données .....	152
12.1.2 Définition d'interface .....	152
12.1.3 Spécialisation des types de données .....	154
12.1.4 Opérations.....	156
12.1.5 Quelconque (Any).....	159
12.1.6 Pid et sortes de pid.....	161
12.1.7 Constructeurs de type de données.....	161
12.1.8 Comportement des opérations .....	167
12.1.9 Constructions de définition de données supplémentaires.....	170
12.2 Utilisation passive des données .....	178
12.2.1 Expressions.....	178
12.2.2 Littéral.....	181
12.2.3 Synonyme .....	181
12.2.4 Primaire étendu.....	182
12.2.5 Expression d'égalité .....	183
12.2.6 Expression conditionnelle.....	184
12.2.7 Application d'opération.....	184
12.2.8 Expression de vérification d'intervalle .....	187
12.3 Utilisation active des données.....	187
12.3.1 Définition de variable .....	188
12.3.2 Accès de variable.....	188
12.3.3 Affectation et tentative d'affectation.....	189
12.3.4 Expressions impératives .....	192
12.3.5 Appel de procédure renvoyant une valeur .....	195

	<b>Page</b>
13 Définition de système générique.....	196
13.1 Définition optionnelle.....	197
13.2 Chaîne de transition optionnelle.....	198
Annexe A – Index des non-terminaux.....	200
Annexe B – Réservee pour une utilisation future.....	220
Annexe C – Réservee pour une utilisation future.....	220
Annexe D – Données prédéfinies du SDL.....	220
D.1 Introduction.....	220
D.2 Notation.....	220
D.2.1 Axiomes.....	221
D.2.2 Equations conditionnelles.....	223
D.2.3 Egalité.....	223
D.2.4 Axiomes booléens.....	224
D.2.5 Terme conditionnel.....	224
D.2.6 Terme d'erreur.....	225
D.2.7 Littéraux non ordonnés.....	225
D.2.8 Equations de littéral.....	226
D.3 Paquetage Prédéfini (Predefined).....	227
D.3.1 Sorte booléenne (Boolean).....	227
D.3.2 Sorte de caractère (Character).....	227
D.3.3 Sorte de chaîne (String).....	228
D.3.4 Sorte Charstring.....	229
D.3.5 Sorte Entier (Integer).....	230
D.3.6 Syntype Naturel (Natural).....	231
D.3.7 Sorte Real.....	231
D.3.8 Sorte Tableau (Array).....	233
D.3.9 Vecteur.....	233
D.3.10 Sorte Powerset.....	234
D.3.11 Sort durée (Duration).....	235
D.3.12 Sorte temps (Time).....	236
D.3.13 Sorte Sac (Bag).....	236
D.3.14 Sortes Bit et Bitstring de l'ASN.1.....	237
D.3.15 Sortes Octet et Octetstring de l'ASN.1.....	240
D.3.16 Exceptions prédéfinies.....	241
Annexe E – Réservee aux exemples.....	241
Annexe F – Définition formelle.....	241
Appendice I – Etat de la Z.100, des documents et Recommandations associés.....	241

	<b>Page</b>
Appendice II – Directives concernant la maintenance du SDL.....	242
II.1 Maintenance du SDL .....	242
II.1.1 Terminologie .....	242
II.1.2 Règles de maintenance .....	243
II.1.3 Procédure de demande de modification.....	243
Appendice III – Conversion systématique de SDL 92 en SDL 2000 .....	246



## **Recommandation Z.100**

### **SDL: LANGAGE DE DESCRIPTION ET DE SPÉCIFICATION**

*(révisée en 1999)*

#### **1 Domaine d'application**

Le but poursuivi, en recommandant l'utilisation du langage de description et de spécification (SDL, *specification and description language*), est d'avoir un langage permettant de spécifier et de décrire sans ambiguïté le comportement des systèmes de télécommunication. Les spécifications et les descriptions faites à l'aide du SDL doivent être formelles dans ce sens qu'il doit être possible de les analyser et de les interpréter sans ambiguïté.

Les termes spécification et description sont utilisés dans le sens ci-après:

- a) la spécification d'un système est la description du comportement souhaité de celui-ci;
- b) la description d'un système est une description de son comportement réel, en d'autres termes de son implémentation pratique.

Une spécification du système, au sens large, est la spécification à la fois du comportement et d'un ensemble de paramètres généraux du système. Toutefois, le SDL ne vise qu'à décrire les aspects relatifs au comportement d'un système; les paramètres généraux concernant des propriétés telles que la capacité et le poids doivent être décrits à l'aide de techniques différentes.

NOTE – Etant donné qu'il n'est pas fait de distinction entre l'utilisation du SDL pour la spécification et son utilisation pour la description, le terme spécification dans la présente Recommandation est utilisé pour désigner à la fois le comportement souhaité et le comportement réel.

#### **1.1 Objectifs**

Les objectifs généraux qui ont été pris en compte lors de la définition du SDL sont de fournir un langage:

- a) facile à apprendre, à utiliser et à interpréter;
- b) permettant l'élaboration de spécifications dépourvues d'ambiguïté pour faciliter la passation de commandes, la soumission d'offres et la conception tout en laissant un certain nombre d'options ouvertes;
- c) extensible pour permettre un développement ultérieur;
- d) capable de prendre en charge différentes méthodologies de spécification et de conception de systèmes.

#### **1.2 Application**

La présente Recommandation constitue le manuel de référence du SDL. Le Supplément 1 de la Recommandation Z.100, tel que produit pendant la période d'études 1992-1996, constitue un document cadre de méthodologie et contient des exemples d'utilisation du SDL. L'Appendice I de la Recommandation Z.100, publié pour la première fois en mars 1993 contient également des directives concernant la méthodologie, bien que celles-ci ne permettent pas d'utiliser pleinement le potentiel du SDL.

Le domaine d'application principal du SDL est la spécification du comportement des systèmes en temps réel dans certains de leurs aspects, ainsi que la conception de ces systèmes. Les applications dans le domaine des télécommunications comprennent:

- a) le traitement des appels et des connexions (par exemple: écoulement, signalisation téléphonique, comptage aux fins de taxation, etc.) dans les systèmes de commutation;
- b) la maintenance et la relève des dérangements (par exemple: alarmes, relève automatique des dérangements, essais périodiques, etc.) dans les systèmes généraux de télécommunication;
- c) la commande de systèmes (par exemple: protection contre les surcharges, procédures de modification et d'extension, etc.);
- d) les fonctions d'exploitation et de maintenance, la gestion des réseaux;
- e) les protocoles de communication de données;
- f) les services de télécommunication.

Il va de soi que le SDL peut aussi servir à la spécification fonctionnelle du comportement d'un objet lorsque celui-ci peut être spécifié au moyen d'un modèle discret, c'est-à-dire un objet communiquant avec son environnement par messages discrets.

Le SDL est un langage particulièrement riche qui peut être utilisé à la fois pour des spécifications de haut niveau informelles (et/ou formellement incomplètes), des spécifications partiellement formelles et des spécifications détaillées. L'utilisateur doit choisir les parties appropriées du SDL en fonction du niveau de communication souhaité et de l'environnement dans lequel le langage sera utilisé. Selon l'environnement dans lequel une spécification est utilisée, certains éléments qui relèvent du simple bon sens pour l'émetteur et le destinataire de la spécification, ne seront pas explicités.

Ainsi, le SDL peut être utilisé pour:

- a) établir les besoins d'une installation;
- b) établir les spécifications d'un système;
- c) établir des Recommandations UIT-T ou autres normes similaires (internationales, régionales ou nationales);
- d) établir des spécifications de conception d'un système;
- e) établir des spécifications détaillées;
- f) décrire la conception d'un système (à la fois globalement et suffisamment dans le détail pour permettre l'implémentation directe);
- g) décrire les essais d'un système (en particulier en association avec MSC et TTCN).

L'organisation à laquelle appartient l'utilisateur peut choisir le niveau d'application du SDL qui convient.

### **1.3 Spécification d'un système**

Le SDL décrit un comportement de système sous la forme de stimulus/réaction, étant admis que les stimuli aussi bien que les réactions sont des entités discrètes et contiennent des informations. En particulier, la spécification d'un système est vue comme étant la séquence de réactions associée à une séquence de stimuli.

Le modèle de spécification d'un système est fondé sur la notion de machine à états finis étendus.

Le SDL fait appel à des concepts structurels qui facilitent la spécification des grands systèmes et des systèmes complexes. Il est ainsi possible de subdiviser la spécification d'un système en unités faciles à gérer qui peuvent être traitées et comprises de manière indépendante. La subdivision peut s'opérer en plusieurs étapes qui permettent d'obtenir une structure hiérarchique d'unités définissant le système à différents niveaux.

## 1.4 Différences entre le SDL 88 et le SDL 92

Le langage défini dans la version précédente de la présente Recommandation était une extension de la Recommandation Z.100 publiée dans le *Livre bleu* en 1988. On désigne par SDL 88 le langage défini dans le *Livre bleu* et par SDL 92 le langage défini dans la version précédente de la présente Recommandation. Tous les efforts avaient alors été entrepris pour faire du SDL 92 une pure extension du SDL 88 sans perdre la validité de la syntaxe ou changer la sémantique d'une utilisation existante du SDL 88. De plus, les améliorations n'ont été acceptées que sur la base des besoins exprimés par plusieurs organes membres de l'UIT-T.

Les extensions majeures furent apportées dans le domaine de la programmation orientée objet. Alors que le SDL 88 est fondé sur les objets dans son modèle sous-jacent, quelques constructions de langage furent ajoutées pour permettre au SDL 92 de prendre en charge de manière plus complète et uniforme le paradigme d'objet:

- a) paquetages;
- b) types de système, de bloc, de processus et de service;
- c) (ensemble d') instances de système, de bloc, de processus et de service fondées sur les types;
- d) paramétrisation des types au moyen de paramètres de contexte;
- e) spécialisation des types et redéfinitions de types et transitions virtuels.

Les autres extensions sont: les transitions spontanées, le choix non déterministe, le symbole d'entrée et sortie interne dans le SDL/GR pour la compatibilité avec les diagrammes existants, un opérateur **any**, non déterministe impératif, canal sans retard, appel de procédure distante, procédure retournant une valeur, entrée de champ de variable, définition d'opérateur, combinaison avec des descriptions de données externes, possibilités d'adressage externe en sortie, libre action dans les transitions, transition continue dans le même état avec la même priorité, connexions m:n de canaux et acheminement des signaux aux frontières de la structure. De plus, on a introduit un certain nombre de simplifications mineures de la syntaxe.

Il a été nécessaire dans de rares de cas de modifier le SDL 88 lorsque la définition du SDL 88 n'était pas consistante. Les restrictions et modifications introduites peuvent être effectuées par une procédure de traduction automatique. Cette procédure a été également nécessaire pour convertir les documents de SDL 88 en SDL 92 contenant des noms composés de mots qui sont des mots clés du SDL 92.

La sémantique de la construction **output** a été simplifiée entre le SDL 88 et le SDL 92; ceci peut nuire à la validité de certaines utilisations spéciales de **output** (lorsqu'il n'existe pas de clause **to** et qu'il existe plusieurs accès possibles pour le signal) dans les spécifications du SDL 88. Quelques propriétés de la propriété d'égalité des sorties ont été modifiées.

Quant aux constructions **d'import/export**, on a introduit une définition facultative de variable distante afin d'aligner l'export des variables avec l'introduction de l'export de procédures (procédures distantes). Cela a nécessité la modification des documents établis en SDL 88 qui contenaient des qualificatifs dans les expressions d'import ou introduisaient plusieurs noms importés dans la même portée avec différentes sorties. Dans les cas (rares) où il est nécessaire de qualifier les variables importées pour résoudre la résolution par contexte, la conversion du SDL 88 en SDL 92 consiste à introduire des définitions <remote variable definition> et d'effectuer la qualification avec l'identificateur du nom de variable distante introduit.

Pour la construction **view**, la définition de vue a été rendue locale par rapport au processus ou service visionnant. Cela a nécessité la modification des documents SDL 88 qui contenaient des qualificatifs dans les définitions de vues ou les expressions de vues. L'objectif de la conversion de SDL 88 en SDL 92 est d'éliminer ces qualificatifs. Ceci n'a pas modifié la sémantique des expressions de vue, étant donné qu'elle est décidée par leurs expressions pid (non changées).

La construction **service** a été définie en tant que concept primitif au lieu d'être une abréviation, sans étendre ses propriétés. L'utilisation de **service** n'a pas été affectée par cette modification car elle a été de toute manière utilisée comme concept primitif. La modification vise à simplifier la définition du langage et à l'aligner avec l'utilisation réelle ainsi qu'à réduire le nombre de restrictions sur le service dues aux règles de transformation du SDL 88. En conséquence à cette modification, l'acheminement de signal de service a été supprimé, les acheminements des signaux pouvant le remplacer. Il s'agissait là d'une modification conceptuelle mineure sans implications sur l'utilisation concrète (la syntaxe de l'acheminement des signaux du SDL 88 est identique à celle de l'acheminement des signaux dans le SDL 92).

La construction **priority output** a été supprimée du langage. Elle peut être remplacée par **output to self** à l'aide d'une procédure de traduction automatique.

Certaines définitions du SDL de base, comme la définition de **signal** ont été considérablement étendues. Il convient de noter que les extensions étaient facultatives mais qu'elles étaient utilisées pour tirer profit de la puissance offerte par les extensions orientées objet, par exemple pour utiliser la paramétrisation et la spécialisation des signaux.

Les mots clés du SDL 92 qui ne sont pas des mots clés du SDL 88 sont les suivants:

**any, as, atleast, connection, endconnection, endoperator, endpackage, finalized, gate, interface, nodelay, noequality, none, package, redefined, remote, returns, this, use, virtual.**

## 1.5 Différences entre le SDL 92 et le SDL 2000

Il a été délibérément décidé de ne pas modifier le SDL entre 1992 et 1996, de sorte qu'à l'issue de cette période, seul un nombre limité de modifications ont été effectuées sur le SDL. Ces modifications ont été publiées sous la forme de l'Addendum 1 de la Recommandation Z.100 (10/96) plutôt que de mettre à jour le document SDL 92. Bien que cette version du SDL ait été quelques fois appelée SDL 96, il ne s'agissait que de modifications mineures comparé aux modifications effectuées entre le SDL 88 et le SDL 92. Ces changements étaient les suivants:

- a) harmoniser les signaux avec des procédures distantes et des variables distantes;
- b) harmoniser les canaux et l'acheminement des signaux;
- c) ajouter des procédures et des opérations externes;
- d) permettre l'utilisation d'un bloc ou d'un processus comme un système;
- e) expressions d'état;
- f) permettre la présence de paquetages sur des blocs ou des processus;
- g) opérateurs sans paramètres.

Ces modifications, ainsi que certaines autres, ont été introduites dans la Z.100 pour fournir une version de SDL appelée SDL 2000. Dans la présente Recommandation, le langage défini dans la Z.100 (03/93) avec l'Addendum 1 de la Z.100 (10/96) est toujours appelé SDL 92.

Les avantages procurés par la stabilité du langage, entre 1992 et 1996, ont commencé à s'atténuer au regard de la nécessité de mettre le SDL à jour pour prendre en charge et améliorer la compatibilité avec d'autres langages qui sont fréquemment utilisés en combinaison avec le SDL. Les outils et techniques modernes ont également contribué à créer des logiciels issus plus directement de spécifications SDL, mais des avantages encore plus significatifs pourraient être obtenus en incorporant une meilleure prise en charge de cette utilisation du SDL. Bien que le SDL 2000 soit essentiellement une mise à jour du SDL 92, il a été convenu qu'un certain niveau d'incompatibilité avec le SDL 92 était justifié, sans cela en effet, le langage en résultant aurait été trop important, trop complexe et trop inconsistant. Le présent sous-paragraphe fournit des informations concernant ces modifications. La procédure permettant de transformer de manière systématique des descriptions de SDL 92 en SDL 2000 est présentée dans l'Appendice III.



Des modifications ont été appliquées dans certains domaines, avec pour objectif une simplification du langage et une adaptation pour de nouveaux domaines d'application:

- a) adaptation de conventions syntaxiques en fonction d'autres langages avec lesquels le SDL est utilisé;
- b) harmonisation des concepts de système, de bloc et de processus sur la base du concept "d'agent", et fusion du concept d'acheminement dans le concept de canal;
- c) descriptions d'interfaces;
- d) traitement des exceptions;
- e) prise en charge de la notation textuelle d'algorithmes au sein du SDL/GR;
- f) états composites;
- g) remplacement de la construction service par la construction state aggregation ;
- h) nouveau modèle pour les données;
- i) constructions pour prendre en charge l'utilisation de la notation ASN.1 avec le SDL précédent de la Recommandation Z.105 (03/95).

D'autres modifications ont été introduites: paquetages emboîtés, confinement direct de blocs et de processus dans des blocs, paramètres uniquement **out**.

Au niveau de la syntaxe, le SDL 2000 est en majuscules ou en minuscules. Les mots clés sont disponibles en deux orthographes: tout en majuscules ou tout en minuscules. Les mots clés du SDL 2000 qui ne sont pas des mots clés du SDL 92 sont les suivants:

**abstract, aggregation, association, break, choice, composition, continue, endexceptionhandler, endmethod, endobject, endvalue, exception, exceptionhandler, handle, method, object, onexception, ordered, private, protected, public, raise, value.**

Les mots clés suivants du SDL 92 ne sont pas des mots clés dans le SDL 2000:

**all, axioms, constant, endgenerator, endnewtype, endrefinement, endservice, error, fpar, generator, imported, literal, map, newtype, noequal, ordering, refinement, returns, reveal, reverse, service, signalroute, view, viewed.**

Quelques constructions du SDL 92 ne sont pas disponibles dans le SDL 2000: expressions de vues, générateurs, sous-structures de bloc, sous-structures de canal, affinage des signaux, définition axiomatique de données, macrodiagrammes. Ces constructions ont été peu (ou pas) utilisées et la surcharge nécessaire pour les conserver dans le langage et les outils ne se justifiait pas.

## 2 Références normatives

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

- Recommandation CCITT T.50 (1992), *Alphabet international de référence (ancien alphabet international n° 5 ou AIS) – Technologie de l'information – Jeux de caractères codés à 7 bits pour l'échange d'informations.*
- ISO/CEI 646:1991, *Technologies de l'information – Jeu ISO de caractères codés à 7 éléments pour l'échange d'informations.*

### 3 Définitions

De nombreux termes sont définis tout au long de la présente Recommandation, une liste de définitions dans le présent paragraphe ne ferait que répéter le texte de la Recommandation. Par conséquent, seuls quelques termes clés sont donnés dans le présent paragraphe.

- 3.1 **agent**: système, bloc ou processus qui contient une ou plusieurs machines à états finis étendus.
- 3.2 **bloc**: agent qui contient un ou plusieurs blocs ou processus concurrents et qui peut également contenir une machine à états finis étendus qui possède et traite des données à l'intérieur du bloc.
- 3.3 **corps**: graphe de la machine à états d'un agent, d'une procédure, d'un état composite ou d'une opération.
- 3.4 **canal**: voie de communication entre des agents.
- 3.5 **environnement**: tout ce qui se trouve autour du système et qui communique avec lui à la façon SDL.
- 3.6 **porte**: point de connexion pour la communication avec un type d'agent, qui, lorsque le type est instancié, détermine la connexion de l'instance d'agent avec d'autres instances.
- 3.7 **instance**: objet créé lorsqu'un type est instancié.
- 3.8 **objet**: terme utilisé pour des éléments de données qui sont des références à des valeurs.
- 3.9 **pid**: terme utilisé pour des éléments de données qui sont des références à des agents.
- 3.10 **procédure**: encapsulation d'une partie du comportement d'un agent, qui est définie à un endroit mais peut être appelée depuis plusieurs endroits à l'intérieur de l'agent. D'autres agents peuvent appeler une procédure distante.
- 3.11 **processus**: agent qui contient une machine à états finis étendus et qui peut contenir d'autres processus.
- 3.12 **signal**: le moyen de communication primaire s'effectue au moyen de signaux qui sont envoyés par l'agent émetteur et reçus par l'agent récepteur.
- 3.13 **sorte**: ensemble d'éléments de données qui possèdent des propriétés communes.
- 3.14 **état**: une machine à états finis étendus d'un agent est dans un certain état si elle attend un stimulus.
- 3.15 **stimulus**: événement qui peut entraîner un agent qui est dans un certain état à entrer dans une transition.
- 3.16 **système**: agent le plus extérieur qui communique avec l'environnement.
- 3.17 **temporisateur**: objet appartenant à un agent qui peut déclencher un stimulus de signal de temporisation à un moment spécifié.
- 3.18 **transition**: séquence d'actions exécutée par un agent jusqu'à ce qu'il entre dans un certain état.
- 3.19 **type**: définition qui peut être utilisée pour la création d'instances, et qui peut également être héritée et être spécialisée pour former d'autres types. Un type paramétré est un type qui possède des paramètres. En attribuant à ces paramètres différentes valeurs, on définit des types non paramétrés différents qui donnent, lorsqu'ils sont instanciés, des instances avec des propriétés différentes.
- 3.20 **valeur**: terme utilisé pour la classe de données qui est directement accessible. Les valeurs peuvent être transmises librement entre agents.

## 4 Abréviations

La présente Recommandation utilise les abréviations suivantes:

SDL/GR	représentation graphique du SDL ( <i>graphic representation form of SDL</i> )
SDL/PR	représentation textuelle du SDL ( <i>textual phrase representation form of SDL</i> )
SDL 2000	SDL tel qu'il est défini dans la présente Recommandation
SDL 92	SDL tel qu'il est défini dans la Z.100 (03/93) avec l'Addendum 1 (10/96)
SDL 88	SDL tel qu'il est défini dans la Z.100 (1988)

## 5 Conventions

Le texte du présent paragraphe n'est pas normatif. Il définit les conventions utilisées pour décrire le SDL. L'utilisation du SDL dans le présent paragraphe est uniquement illustrative. Les métalangages et conventions introduits ne sont utilisés que dans le but de décrire le SDL de manière non ambiguë.

### 5.1 Grammaires de SDL

Le SDL offre le choix entre deux formes syntaxiques différentes pour représenter un système: une représentation graphique (SDL/GR) et une représentation textuelle (SDL/PR). Comme ces formes sont toutes deux des représentations concrètes du même SDL, elles sont équivalentes. En particulier, elles sont toutes les deux équivalentes à une grammaire abstraite en ce qui concerne les concepts correspondants.

Un sous-ensemble du SDL/PR est commun avec le SDL/GR. Ce sous-ensemble est appelé grammaire textuelle commune.

Bien que le SDL puisse être écrit soit sous la forme SDL/PR soit sous la forme SDL/GR, ce langage a été conçu en gardant à l'esprit que le SDL/PR est rarement utilisé pour des utilisations telles que des échanges entre outils. Le format d'échange commun spécifié dans la Z.106 (10/96) diminue encore davantage l'utilisation du SDL/PR. La plupart des utilisateurs utilisent le SDL/GR.

La Figure 5-1 montre les relations qui existent entre le SDL/PR, le SDL/GR, les grammaires concrètes et la grammaire abstraite.

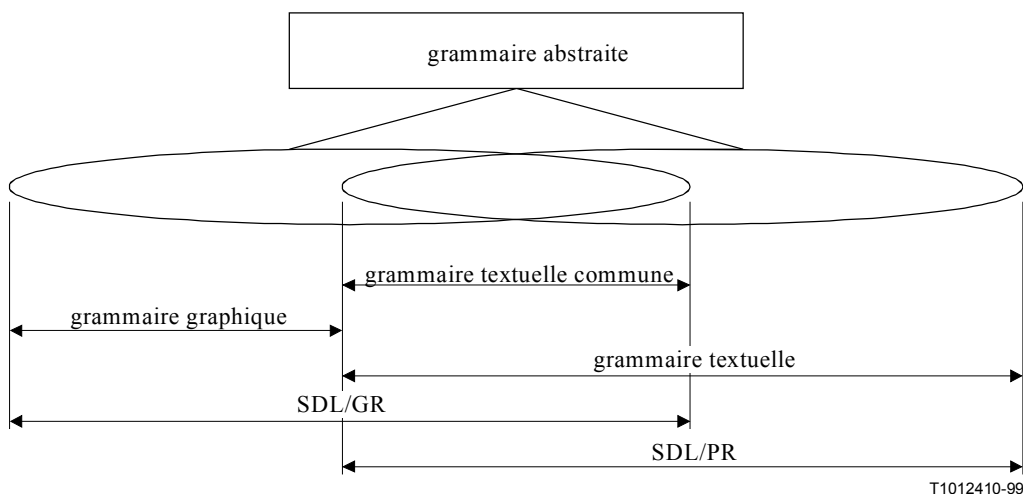


Figure 5-1/Z.100 – Grammaires de SDL

A chaque grammaire concrète, est associée sa propre syntaxe et les rapports qu'elle a avec la grammaire abstraite (c'est-à-dire la manière de la transformer en syntaxe abstraite). Avec cette méthode, la définition de la sémantique du SDL est unique; chacune des grammaires concrètes hérite de la sémantique par l'intermédiaire de ses relations avec la grammaire abstraite. Cette méthode permet également d'assurer l'équivalence entre le SDL/PR et le SDL/GR.

On dispose également d'une définition formelle du SDL qui définit la manière de transformer la spécification d'un système en syntaxe abstraite et comment interpréter une spécification donnée en termes de grammaire abstraite. La définition formelle est donnée dans l'Annexe F.

Il n'existe pas de syntaxe abstraite directement équivalente pour certaines constructions. Dans ce cas, un modèle est donné pour la transformation de la syntaxe concrète de ces constructions en syntaxe concrète d'autres constructions qui possèdent une syntaxe abstraite (directement ou indirectement par le biais d'autres modèles). Les éléments qui n'ont pas de correspondance avec la syntaxe abstraite (tels que les commentaires) n'ont aucune signification formelle.

## 5.2 Définitions fondamentales

La présente Recommandation fait appel à des conventions et à des concepts généraux; leurs définitions sont données dans les sous-paragraphes suivants.

### 5.2.1 Définition, type et instance

Dans la présente Recommandation, les concepts de type, d'instance et les relations qui existent entre elles, jouent un rôle fondamental. Le schéma et la terminologie utilisés sont explicités ci-après et illustrés par la Figure 5-2.

Dans le présent sous-paragraph, on introduit les règles sémantiques fondamentales des définitions de type, des définitions d'instance, des définitions des types paramétrés, de la paramétrisation, de la liaison des paramètres de contexte, de la spécialisation et de l'instanciation.

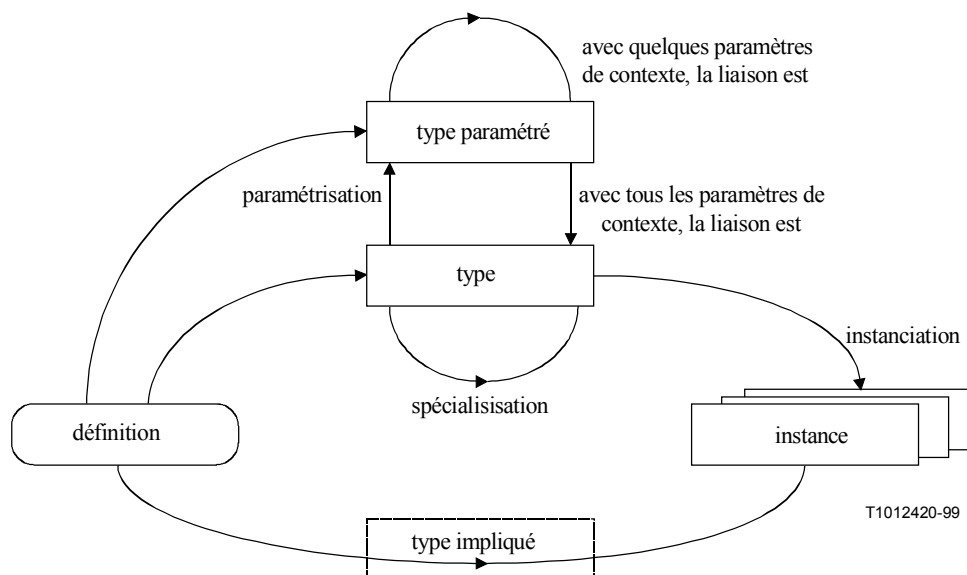


Figure 5-2/Z.100 – Le concept de type

Les définitions introduisent des entités nommées, qui sont soit des types soit des instances avec des types implicites. La définition d'un type définit toutes les propriétés qui lui sont associées. La définition d'un processus est un exemple de définition d'une instance. La définition d'un signal est un exemple de définition qui est une définition de type.

On peut instancier un nombre quelconque d'instances d'un type donné. Une instance d'un type particulier possède toutes les propriétés définies pour ce type. Une procédure constitue un exemple de type; elle peut être instanciée par des appels de procédure.

Un type paramétré est un type où certaines entités sont présentes comme paramètres de contexte formel. Un paramètre de contexte formel dans la définition d'un type a des contraintes. Les contraintes permettent l'analyse statique du type paramétré. La liaison de tous les paramètres d'un type paramétré conduit à un type ordinaire. L'exemple d'un type paramétré est la définition d'un signal paramétré où l'une des sortes convoyées par le signal est spécifiée par un paramètre de contexte de sorte formel; cela permet au paramètre d'être de différentes sortes dans différents contextes.

Une instance est définie soit directement soit en instanciant un type donné. L'exemple d'une instance est une instance d'un système, qui peut être définie par une définition du système, ou est une instantiation d'un type de système.

La spécialisation permet d'avoir un type (sous-type) fondé sur un autre type (son supertype) en ajoutant des propriétés à celles du supertype ou en redéfinissant des propriétés virtuelles de ce dernier. Une propriété virtuelle peut être restreinte afin de convenir à l'analyse des types généraux.

La liaison de tous les paramètres de contexte d'un type paramétré conduit à un type non paramétré. Il n'existe pas de relation supertype/sous-type entre un type paramétré et le type qui en découle.

NOTE – Pour éviter d'alourdir le texte, on peut s'abstenir d'utiliser le terme *instance*. Ainsi, pour exprimer "qu'une instance du système est interprétée ...", on écrira "un système est interprété ...".

## 5.2.2 Environnement

Les systèmes qui sont spécifiés en SDL réagissent d'après les stimuli échangés avec le monde extérieur. Ce monde extérieur est appelé environnement du système en cours de spécification.

On suppose qu'il y a une ou plusieurs instances d'agent dans l'environnement et, par conséquent, les stimuli circulant de l'environnement en direction du système ont des identités associées à ces instances d'agent. Ces agents ont des valeurs pids différentes des autres valeurs pid du système (voir 12.1.6).

Bien que le comportement du système soit non déterministe, il est supposé obéir aux contraintes imposées par la spécification du système.

## 5.2.3 Erreur

Une spécification de système est une spécification de système correcte en SDL seulement si elle répond aux règles syntaxiques et aux conditions statiques du SDL.

Lorsqu'une spécification SDL correcte est interprétée et qu'une condition dynamique se trouve violée, une erreur apparaît. Des exceptions prédéfinies (voir D.3.16) sont déclenchées, lorsqu'une erreur survient pendant l'interprétation d'un système. Si l'exception n'est pas traitée, le comportement ultérieur du système ne peut être obtenu à partir de la spécification.

## 5.3 Présentation

La présentation suivante est utilisée pour séparer les différents problèmes de langage selon chaque thème.

### 5.3.1 Structuration du texte

La présente Recommandation est structurée par thèmes, qui comportent des intitulés précédés éventuellement par une introduction; ces intitulés sont les suivants:

- a) *Grammaire abstraite* – Décrite par une syntaxe abstraite et des conditions statiques pour définitions bien formées.

- b) *Grammaire textuelle concrète* – Qui concerne à la fois la grammaire textuelle courante utilisée pour le SDL/PR et le SDL/GR et la grammaire uniquement utilisée pour le SDL/PR. Cette grammaire est décrite au moyen d'une syntaxe textuelle, de conditions statiques et de règles de définitions bien formées, concernant la syntaxe textuelle, et de la relation de la syntaxe textuelle avec la syntaxe abstraite.
- c) *Grammaire graphique concrète* – Décrite par la syntaxe graphique, les conditions statiques et les règles de définitions bien formées concernant la syntaxe graphique, la relation de cette syntaxe avec la syntaxe abstraite et quelques règles de dessin qui viennent en supplément (à celles indiquées en 6.5).
- d) *Sémantique* – Donne une signification à une construction, confère les propriétés de cette construction, la façon dont elle est interprétée et toutes conditions dynamiques qui doivent être remplies par la construction pour avoir un comportement correct au sens du SDL.
- e) *Modèle* – Donne le mappage pour les notations qui n'ont pas de syntaxe abstraite directe et qui sont exprimées en termes de constructions en syntaxe concrète différentes. Une notation qui est exprimée par d'autres constructions est appelée abréviation, et est considérée comme une syntaxe dérivée de la forme transformée.
- f) *Exemples*

Dans de rares cas (comme par exemple <SDL spécification>), un intitulé, *Grammaire concrète*, est utilisé lorsque les syntaxes textuelle et graphique sont fusionnées ou lorsque de nombreuses règles de grammaire devraient être répétées pour les cas textuels et graphiques. Le mécanisme de fusion du SDL/PR et du SDL/GR dans ces cas n'est pas défini dans la présente Recommandation.

### 5.3.2 Intitulés

L'introduction qui précède éventuellement les intitulés, est uniquement destinée à faciliter la compréhension du texte et non pas à compléter la présente Recommandation et à ce titre doit être considérée comme étant une partie officieuse de la présente Recommandation.

S'il n'existe pas de texte pour un intitulé, tout l'intitulé est omis.

La suite du présent sous-paragraphe décrit les autres formalismes particuliers utilisés dans chaque intitulé et les titres utilisés. Elle peut également être considérée comme un exemple de présentation typographique du premier niveau des intitulés définis ci-dessus, ce texte appartenant à l'introduction.

#### *Grammaire abstraite*

La notation en syntaxe abstraite est définie en 5.4.1.

L'absence de l'intitulé *grammaire abstraite* indique qu'il n'existe pas d'autres syntaxes abstraites pour le sujet traité et que la syntaxe concrète mappe la syntaxe abstraite définie par un autre paragraphe de texte numéroté.

On peut se référer à une des règles dans la syntaxe abstraite à partir de tout intitulé en maintenant le nom de la règle en italique.

Les règles dans la notation formelle peuvent être suivies par des paragraphes qui définissent les conditions qui doivent être satisfaites par une définition SDL bien formée et qui peuvent être vérifiées sans interprétation d'une instance. Les conditions statiques à ce niveau se réfèrent uniquement à la syntaxe abstraite. Les conditions statiques, qui ne concernent que la syntaxe concrète, sont définies postérieurement à la syntaxe concrète. La syntaxe abstraite, associée aux conditions statiques applicables à la syntaxe abstraite, définit la grammaire abstraite du langage.

#### *Grammaire textuelle concrète*

La syntaxe textuelle concrète est spécifiée dans la forme BNF élargie de la description de la syntaxe définie en 5.4.2.

La syntaxe textuelle est suivie par des paragraphes définissant les conditions statiques qui doivent être satisfaites dans un texte bien formé et qui peuvent être vérifiées sans interprétation d'une instance. Cela s'applique également aux conditions statiques, si elles existent, pour la grammaire abstraite.

Dans de nombreux cas, il y a une relation simple entre la syntaxe concrète et la syntaxe abstraite étant donné qu'une règle de la syntaxe concrète est simplement représentée par une seule règle dans la syntaxe abstraite. Lorsque le même nom est utilisé dans la syntaxe abstraite et dans la syntaxe concrète, afin d'indiquer qu'il représente le même concept, le texte précisant que "<x> dans la syntaxe concrète représente X dans la syntaxe abstraite" est implicite dans la description du langage et est souvent omis. Dans ce contexte, le cas est ignoré mais les sous-catégories sémantiques soulignées (voir 5.4.2) sont significatives.

La syntaxe textuelle concrète qui ne constitue pas une forme abrégée est une syntaxe textuelle concrète stricte. La relation entre la syntaxe textuelle concrète et la syntaxe abstraite est seulement définie pour la syntaxe textuelle concrète stricte.

La relation entre la syntaxe textuelle concrète et la syntaxe abstraite est omise si le sujet en cours de définition est une forme abrégée qui est modélisée par d'autres constructions SDL (voir le paragraphe *Modèle* ci-après).

#### *Grammaire graphique concrète*

La syntaxe graphique concrète est spécifiée dans la forme BNF élargie de la description de la syntaxe définie en 5.4.3.

La syntaxe graphique est suivie par des paragraphes définissant les conditions statiques qui doivent être satisfaites dans une SDL/GR bien formée et qui peuvent être vérifiées sans interprétation d'une instance. Cela s'applique également aux conditions statiques, si elles existent, pour la grammaire abstraite et aux conditions statiques pertinentes de la grammaire textuelle concrète.

La relation entre la syntaxe graphique concrète et la syntaxe abstraite est omise si le sujet en cours de définition est une forme abrégée qui est modélisée par d'autres constructions SDL (voir le paragraphe *Modèle* ci-après).

Dans de nombreux cas, il existe une relation simple entre les diagrammes de la grammaire graphique concrète et les définitions de la syntaxe abstraite. Lorsque le nom d'un non-terminal se termine dans la grammaire concrète par le mot "diagramme" et qu'il y a un nom dans la grammaire abstraite qui en diffère seulement parce qu'il se termine par le mot *définition*, les deux règles représentent la même notion. Par exemple, <system diagram> dans la grammaire concrète correspond à *System-définition* dans la grammaire abstraite.

L'expansion dans la syntaxe concrète provenant de constructions telles les définitions référencées (7.3) et les macros (6.2), doit être examinée avant la mise en correspondance entre la syntaxe concrète et la syntaxe abstraite. Ces expansions sont étudiées en détail dans l'Annexe F.

#### *Sémantique*

Les propriétés sont des relations entre différents concepts dans le langage SDL. Les propriétés sont utilisées dans les règles de bonne formation.

Un exemple de propriété est l'ensemble des identificateurs de signaux d'entrée valides d'un processus. Cette propriété est utilisée dans la condition statique "pour chaque *State-node*, tous les identificateurs *Signal-identifier* (dans l'ensemble des signaux d'entrée valides) apparaissent soit dans un ensemble *Save-signalset* soit dans un nœud *Input-node*".

Toutes les instances ont une propriété d'identité mais à moins que celle-ci soit formée d'une façon quelque peu inhabituelle, cette propriété d'identité est déterminée comme étant définie par le paragraphe général traitant des identités en 6.3. Cela n'est pas habituellement mentionné comme étant une propriété d'identité. Il n'est également pas nécessaire d'indiquer les sous-composantes d'une

définition contenues par la définition, étant donné que l'appartenance de telles sous-composantes est évidente à partir de la syntaxe abstraite. Par exemple, il est évident qu'une définition de bloc "a" englobé des processus et/ou des blocs.

Les propriétés sont statiques si elles peuvent être déterminées sans l'interprétation d'une spécification de système en SDL et sont dynamiques si l'interprétation de ce système est nécessaire pour déterminer la propriété.

L'interprétation est décrite de manière opérationnelle. Lorsqu'il y a une liste dans la syntaxe abstraite, cette liste est interprétée dans l'ordre donné. C'est-à-dire que la présente Recommandation décrit comment les instances sont créées à partir de la définition du système et comment celles-ci sont interprétées dans une "machine abstraite SDL". Les listes sont annotées dans la syntaxe abstraite au moyen des "\*" et "+" (voir 5.4.1 et 5.4.2).

Les conditions dynamiques sont des conditions qui doivent être satisfaites durant l'interprétation et qui ne peuvent être vérifiées sans interprétation. Les conditions dynamiques peuvent conduire à des erreurs (voir 5.2.3).

NOTE – Le comportement du système est produit par "l'interprétation" du SDL. Le terme "interprétation" est délibérément choisi (plutôt que de choisir un autre terme tel que "exécution") pour inclure à la fois l'interprétation mentale d'un humain et l'interprétation du SDL par un ordinateur.

### *Modèle*

Certaines constructions sont considérées comme étant une "syntaxe concrète dérivée" (ou une abréviation) pour d'autres constructions équivalentes en syntaxe concrète. Par exemple, l'omission d'une entrée pour un signal est une syntaxe concrète dérivée pour une entrée pour ce signal suivi par une transition nulle avec retour vers le même état.

Les propriétés d'une abréviation sont dérivées de la façon dont elle est modélisée en termes de (ou transformée en) concepts primitifs. Afin d'assurer une utilisation facile et sans ambiguïté des abréviations, et afin de réduire les effets de bord lorsque plusieurs abréviations sont associées, ces concepts sont transformés suivant un ordre spécifié défini dans l'Annexe F. L'ordre de transformation est également suivi dans les définitions des concepts du présent paragraphe.

Le résultat de la transformation d'une partie de texte en syntaxe concrète dérivée est généralement une autre partie de texte en syntaxe concrète dérivée ou une partie de texte en syntaxe concrète. Le résultat de la transformation peut également être vide. Dans ce dernier cas, le texte original est omis de la spécification. Lors de l'élaboration du modèle pour une syntaxe concrète, le méta-symbole ***transform*** peut être utilisé et correspond au résultat de la transformation de son argument. Par exemple:

<expression>-***transform***

correspond à la transformation de l'expression <expression>.

### *Exemples*

S'il existe un intitulé *Exemple(s)* qui contient un ou des exemples. D'autres exemples sont présentés dans l'Annexe E. Les exemples sont données pour information.

## **5.4 Métalangages**

Pour la définition des propriétés et des syntaxes du SDL, différents métalangages ont été utilisés en fonction des besoins particuliers.

La grammaire fournie dans la présente Recommandation a été rédigée pour aider à la présentation dans la Recommandation de manière à ce que les noms des règles aient un sens dans le contexte donné et puissent être utilisés dans le texte. Ceci signifie qu'il existe un certain nombre d'ambiguïtés



qui peuvent être levées par la réécriture systématique des règles de syntaxe ou l'application de règles sémantiques.

Dans ce qui suit, on trouvera une introduction aux métalangages; des références renvoyant à des livres ou à des publications particulières de l'UIT-T seront données au besoin.

### 5.4.1 Méta IV

Le sous-ensemble suivant du Méta IV est utilisé pour décrire la syntaxe abstraite du SDL.

Une définition dans la syntaxe abstraite peut être considérée comme étant un objet composite nommé (une arborescence) définissant un ensemble de sous-composantes.

Par exemple, la syntaxe abstraite pour la définition d'un canal est:

*Channel-path* :: *Originating-gate*  
*Destination-gate*  
*Signal-identifier-set*

qui définit le domaine de l'objet composite (arborescence) appelé *Channel-path*. Cet objet comporte trois sous-composantes qui, à leur tour, peuvent être des arborescences.

La définition en Méta IV

*Agent-identifier* = *Identifier*

indique qu'un identificateur *Agent-identifier* est un *Identifier* et ne peut par conséquent être syntaxiquement distingué des autres identificateurs.

Certains objets peuvent également être constitués par certains domaines élémentaires (non composites). Dans le cas du SDL, ces objets sont:

- a) des objets naturels  
exemple:

*Number-of-instance* :: *Nat* [*Nat*]

Le terme *Number-of-instances* désigne un domaine composite contenant une valeur naturelle obligatoire (*Nat*) et une valeur naturelle facultative (*[Nat]*) indiquant respectivement le nombre initial et le nombre maximal d'instances.

- b) objets de citation

Les objets de citation sont représentés par une séquence en caractères gras de majuscules et de chiffres.

Exemple:

*Channel-definition* :: *Channel-name*  
**[NODELAY]**  
*Channel-path-set*

Un canal peut ne pas être retardateur. Ceci est indiqué par une citation facultative **NODELAY**;

- c) marques

Le terme *Token* désigne le domaine des marques. Ce domaine peut être considéré comme étant composé d'un ensemble potentiellement infini d'objets atomiques distincts pour lesquels aucune représentation n'est requise.

Exemple:

*Name* :: *Token*

Un nom est un objet atomique tel que tout nom *Name* peut être distingué de tout autre nom;

d) objets non spécifiés

Un objet non spécifié désigne des domaines qui peuvent avoir une certaine représentation, mais pour lesquels la représentation n'intéresse pas la présente Recommandation.

Exemple:

*Informal-text* :: ...

*Informal-text* contient un objet qui n'est pas interprété.

Les opérateurs ci-après (constructeurs) dans la forme BNF (voir 5.4.2) sont également utilisés dans la syntaxe abstraite de Méta IV: "\*" pour désigner une liste pouvant être vide; "+" pour désigner une liste non vide; "|" pour représenter une alternative, et "[" "]" pour indiquer une option.

Les parenthèses sont utilisées pour regrouper les domaines qui présentent un rapport logique.

Enfin, la syntaxe abstraite utilise un autre opérateur de suffixe "-set" produisant un ensemble (collection non ordonnée d'objets distincts).

Exemple:

*Agent-graph* :: *Agent-start-node State-node-set*

Un graphe *Agent-graph* est constitué d'un nœud *Agent-start-node* et d'un ensemble de nœuds *State-nodes*.

#### 5.4.2 BNF

Dans le formalisme de Backus-Naur (BNF, *Backus-Naur form*) pour les règles lexicales, les terminaux sont <space> et les caractères imprimés en 6.1.

Dans la Backus-Naur Form pour les règles non lexicales, un symbole terminal est l'une des unités lexicales définies en 6.1 (<name>, <quoted operation name>, <character string>, <hex string>, <bit string>, <special>, <composite special> ou <keyword>). Dans les règles non lexicales, un symbole terminal peut être représenté par l'un des éléments suivants:

- a) un mot clé (tel qu'état);
- b) le caractère de l'unité lexicale si elle consiste en un seul caractère (tel que "=");
- c) le nom de l'unité lexicale (tel que <quoted operation name> ou <bit string>);
- d) le nom d'une unité lexicale <composite special> (tel que <implies sign>).

Pour éviter toute confusion avec la grammaire BNF, les noms d'unités lexicales <asterisk>, <plus sign>, <vertical line>, <left square bracket>, <right square bracket>, <left curly bracket> et <right curly bracket> sont toujours utilisés de préférence aux caractères équivalents. Il est à noter que la sémantique des deux terminaux spéciaux <name> et <character string> peut également être identique à celle définie ci-dessous.

Les crochets angulaires et le ou les mots entre crochets sont soit un symbole non-terminal ou l'une des unités lexicales. Les catégories syntaxiques sont les non-terminaux indiqués par un ou plusieurs mots compris entre des crochets angulaires. Pour chaque symbole non-terminal, une règle de production est donnée soit en grammaire textuelle concrète soit en grammaire graphique. Par exemple:

<block reference> ::=  
block <block name> referenced <end>

Une règle de production d'un symbole non-terminal consiste à placer le symbole non-terminal sur la partie gauche du symbole "::-=" et, sur la partie droite une ou plusieurs constructions constituées par un ou plusieurs symboles non-terminaux et éventuellement un ou plusieurs symboles terminaux. Par exemple, <block reference>, <block name> et <end> dans l'exemple ci-dessus sont des symboles non-terminaux; **block** et **referenced** sont des symboles terminaux.

Parfois, le symbole inclut une partie soulignée. Cette partie soulignée met en relief un aspect sémantique de ce symbole. Par exemple, <block name> est syntaxiquement identique à <name>, mais sur le plan sémantique, il spécifie que le nom doit être un nom bloc (block name).

A la partie droite du symbole " ::= ", il existe plusieurs possibilités de production de symboles non-terminaux, séparés par des barres verticales ( "|" ). Par exemple:

```
<diagram in package> ::=
| <package diagram>
| <package reference area>
| <type in agent area>
| <data type reference area>
| <signal reference area>
| <procedure reference area>
| <interface reference area>
| <create line area>
| <option area>
```

indique qu'un diagramme <diagram in package> est un diagramme <package diagram>, ou zone <package reference area>, ou zone <type in agent area>, ou zone <data type reference area>, ou zone <signal reference area>, ou zone <procedure reference area>, ou une zone <interface reference area>, ou zone <create line area> ou une zone <option area>.

Les éléments syntaxiques peuvent être regroupés au moyen d'accolades ("{" et "}"), analogues aux parenthèses du Méta IV (voir 5.4.1). Un groupe entre accolades peut contenir une ou plusieurs barres verticales, indiquant des éléments syntaxiques possibles. Par exemple,

```
<operation definitions> ::=
{ <operation definition>
| <textual operation reference>
| <external operation definition> }+
```

La répétition d'éléments syntaxiques ou de groupes entre accolades est indiquée au moyen d'un astérisque ("\*") ou du signe plus ("+"). Un astérisque indique que le groupe est facultatif et peut ultérieurement être répété un nombre quelconque de fois; un signe plus indique que le groupe doit être présent et peut être répété par la suite un nombre quelconque de fois. L'exemple ci-dessus indique que les définitions <operation definitions> contiennent au moins une définition <operation definition>, une référence <textual operation reference> ou une définition <external operation definition>, et peuvent contenir plus d'une de ces définitions et référence.

Si les éléments syntaxiques sont regroupés en utilisant des crochets ("[" et "]"), cela indique que le groupe est facultatif. Par exemple:

```
<valid input signal set> ::=
signalset [<signal list>] <end>
```

indique qu'un ensemble <valid input signal set> peut, mais pas nécessairement, contenir une liste <signal list>.

### 5.4.3 Métalangage applicable à la grammaire graphique

En ce qui concerne la grammaire graphique, le métalangage décrit en 5.4.2 est complété avec les métasympôles suivants:

- a) *contains*
- b) *is associated with*
- c) *is followed by*
- d) *is connected to*
- e) *set*

Le métasymbole *set* est un opérateur postfixé agissant sur les éléments syntaxiques placés à l'intérieur d'accolades et qui le précèdent immédiatement, il désigne un ensemble (non ordonné) d'éléments. Chaque élément peut être un groupe quelconque d'éléments syntaxiques, auquel cas, il faut le développer avant d'appliquer le métasymbole *set*.

Exemple:

{ <operation text area>\* <operation graph area> } *set*

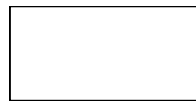
est un ensemble de zéro, d'une ou de plusieurs zones <operation text area> et d'une zone < operation graph area>. Le métasymbole *set* est utilisé lorsque la position des éléments syntaxiques reliés les uns aux autres dans le diagramme n'est pas pertinente et que les éléments peuvent être considérés dans n'importe quel ordre.

Tous les autres métasymboles sont des opérateurs infixes, ayant un symbole graphique non-terminal comme argument de gauche. L'argument de droite est soit un groupe d'éléments syntaxiques situés à l'intérieur d'accolades ou un seul élément syntaxique. Si le membre de droite d'une règle de production comporte un symbole graphique non-terminal comme premier élément et contient un ou plusieurs de ces opérateurs infixes, le symbole graphique non-terminal est alors l'argument de gauche de chacun de ces opérateurs infixes. Un symbole graphique non-terminal est un non-terminal se terminant par le mot "symbol".

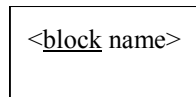
Le métasymbole *contains* indique que son argument de droite doit être placé à l'intérieur de son argument de gauche et, le cas échéant, à l'intérieur du symbole <text extension symbol> associé. Par exemple:

<block reference area> ::= <block symbol> *contains* <block name>

<block symbol> ::=



signifie



Le métasymbole *is associated with* indique que son argument de droite est logiquement associé avec son argument de gauche (comme s'il était "contenu" dans cet argument, l'association dépourvue d'ambiguïté est obtenue par des règles appropriées applicables aux dessins).

Le métasymbole *is followed by* signifie que son argument de droite suit (tant sur le plan logique que dans le dessin) son argument de gauche.

Le métasymbole *is connected to* signifie que son argument de droite est relié (tant sur le plan logique que dans le dessin) à son argument de gauche et implique un flow line symbol (voir 6.5).

## 6 Règles générales

### 6.1 Règles lexicales

Les règles lexicales définissent des unités lexicales. Les unités lexicales sont les symboles terminaux de la *grammaire textuelle concrète* et de la *grammaire graphique concrète*.

```

<lexical unit> ::=
    | <name>
    | <quoted operation name>
    | <character string>
    | <hex string>
    | <bit string>
    | <note>
    | <composite special>
    | <special>
    | <keyword>

<name> ::=
    <underline>* <word> {<underline>+ <word>}* <underline>*
    | {<decimal digit>}+ {<full stop>} <decimal digit>+ }*

<word> ::=
    {<alphanumeric>}+

<alphanumeric> ::=
    | <letter>
    | <decimal digit>

<letter> ::=
    <uppercase letter> | <lowercase letter>

<uppercase letter> ::=
    A | B | C | D | E | F | G | H | I | J | K | L | M
    | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<lowercase letter> ::=
    a | b | c | d | e | f | g | h | i | j | k | l | m
    | n | o | p | q | r | s | t | u | v | w | x | y | z

<decimal digit> ::=
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<quoted operation name> ::=
    <quotation mark> <infix operation name> <quotation mark>
    | <quotation mark> <monadic operation name> <quotation mark>

<infix operation name> ::=
    or | xor | and | in | mod | rem
    | <plus sign> | <hyphen>
    | <asterisk> | <solidus>
    | <equals sign> | <not equals sign>
    | <greater than sign> | <less than sign>
    | <less than or equals sign> | <greater than or equals sign>
    | <concatenation sign> | <implies sign>

<monadic operation name> ::=
    <hyphen> | not

<character string> ::=
    <apostrophe> { <general text character>
    | <special>
    | <apostrophe> <apostrophe>
    }* <apostrophe>

<apostrophe> <apostrophe> représente une <apostrophe> à l'intérieur d'une chaîne
<character string>.

<hex string> ::=
    <apostrophe> { <decimal digit>
    | a | b | c | d | e | f
    | A | B | C | D | E | F
    }* <apostrophe> { H | h }

<bit string> ::=
    <apostrophe> { 0 | 1
    }* <apostrophe> { B | b }

```

```

<note> ::=
                <solidus> <asterisk> <note text> <asterisk>+ <solidus>

<note text> ::=
                {
                |   <general text character>
                |   <other special>
                |   <asterisk>+ <not asterisk or solidus>
                |   <solidus>
                |   <apostrophe> }*

<not asterisk or solidus> ::=
                <general text character> | <other special> | <apostrophe>

<text> ::=
                { <general text character> | <special> | <apostrophe> }*

<general text character> ::=
                <alphanumeric> | <other character> | <space>

<composite special> ::=
                | <result sign>
                | <composite begin sign>
                | <composite end sign>
                | <concatenation sign>
                | <history dash sign>
                | <greater than or equals sign>
                | <implies sign>
                | <is assigned sign>
                | <less than or equals sign>
                | <not equals sign>
                | <qualifier begin sign>
                | <qualifier end sign>

<result sign> ::=
                <hyphen> <greater than sign>

<composite begin sign> ::=
                <left parenthesis> <full stop>

<composite end sign> ::=
                <full stop> <right parenthesis>

<concatenation sign> ::=
                <solidus> <solidus>

<history dash sign> ::=
                <hyphen> <asterisk>

<greater than or equals sign> ::=
                <greater than sign> <equals sign>

<implies sign> ::=
                <equals sign> <greater than sign>

<is assigned sign> ::=
                <colon> <equals sign>

<less than or equals sign> ::=
                <less than sign> <equals sign>

<not equals sign> ::=
                <solidus> <equals sign>

<qualifier begin sign> ::=
                <less than sign> <less than sign>

<qualifier end sign> ::=
                <greater than sign> <greater than sign>

<special> ::=
                <solidus> | <asterisk> | <other special>

```

<other special> ::=	<exclamation mark>	<number sign>	
	<left parenthesis>	<right parenthesis>	
	<plus sign>	<comma>	<hyphen>
	<full stop>	<colon>	<semicolon>
	<less than sign>	<equals sign>	<greater than sign>
	<left square bracket>	<right square bracket>	
	<left curly bracket>	<right curly bracket>	
<other character> ::=	<quotation mark>	<dollar sign>	<percent sign>
	<ampersand>	<question mark>	<commercial at>
	<reverse solidus>	<circumflex accent>	<underline>
	<grave accent>	<vertical line>	<tilde>
<exclamation mark>	::=	!	
<quotation mark>	::=	"	
<left parenthesis>	::=	(	
<right parenthesis>	::=	)	
<asterisk>	::=	*	
<plus sign>	::=	+	
<comma>	::=	,	
<hyphen>	::=	-	
<full stop>	::=	.	
<solidus>	::=	/	
<colon>	::=	:	
<semicolon>	::=	;	
<less than sign>	::=	<	
<equals sign>	::=	=	
<greater than sign>	::=	>	
<left square bracket>	::=	[	
<right square bracket>	::=	]	
<left curly bracket>	::=	{	
<right curly bracket>	::=	}	
<number sign>	::=	#	
<dollar sign>	::=	\$	
<percent sign>	::=	%	
<ampersand>	::=	&	
<apostrophe>	::=	'	
<question mark>	::=	?	
<commercial at>	::=	@	
<reverse solidus>	::=	\	
<circumflex accent>	::=	^	
<underline>	::=	_	
<grave accent>	::=	`	
<vertical line>	::=		
<tilde>	::=	~	

<keyword> ::=

<b>abstract</b>	<b>active</b>	<b>adding</b>
<b>aggregation</b>	<b>alternative</b>	<b>and</b>
<b>any</b>	<b>as</b>	<b>association</b>
<b>atleast</b>	<b>block</b>	<b>break</b>
<b>call</b>	<b>channel</b>	<b>choice</b>
<b>comment</b>	<b>composition</b>	<b>connect</b>
<b>connection</b>	<b>constants</b>	<b>continue</b>
<b>create</b>	<b>dcl</b>	<b>decision</b>
<b>default</b>	<b>else</b>	<b>endalternative</b>
<b>endblock</b>	<b>endchannel</b>	<b>endconnection</b>
<b>enddecision</b>	<b>endexceptionhandler</b>	<b>endinterface</b>
<b>endmacro</b>	<b>endmethod</b>	<b>endobject</b>
<b>endoperator</b>	<b>endpackage</b>	<b>endprocedure</b>
<b>endprocess</b>	<b>endselect</b>	<b>endstate</b>
<b>endsubstructure</b>	<b>endsyntype</b>	<b>endsystem</b>
<b>endtype</b>	<b>endvalue</b>	<b>env</b>
<b>exception</b>	<b>exceptionhandler</b>	<b>export</b>
<b>exported</b>	<b>external</b>	<b>fi</b>
<b>finalized</b>	<b>for</b>	<b>from</b>
<b>gate</b>	<b>handle</b>	<b>if</b>
<b>import</b>	<b>in</b>	<b>inherits</b>
<b>input</b>	<b>interface</b>	<b>join</b>
<b>literals</b>	<b>macro</b>	<b>macrodefinition</b>
<b>macroid</b>	<b>method</b>	<b>methods</b>
<b>mod</b>	<b>nameclass</b>	<b>nextstate</b>
<b>nodelay</b>	<b>none</b>	<b>not</b>
<b>now</b>	<b>object</b>	<b>offspring</b>
<b>onexception</b>	<b>operator</b>	<b>operators</b>
<b>optional</b>	<b>or</b>	<b>ordered</b>
<b>out</b>	<b>output</b>	<b>package</b>
<b>parent</b>	<b>priority</b>	<b>private</b>
<b>procedure</b>	<b>protected</b>	<b>process</b>
<b>provided</b>	<b>public</b>	<b>raise</b>
<b>redefined</b>	<b>referenced</b>	<b>rem</b>
<b>remote</b>	<b>reset</b>	<b>return</b>
<b>save</b>	<b>select</b>	<b>self</b>
<b>sender</b>	<b>set</b>	
<b>signal</b>	<b>signallist</b>	<b>signalset</b>
<b>size</b>	<b>spelling</b>	<b>start</b>
<b>state</b>	<b>stop</b>	<b>struct</b>
<b>substructure</b>	<b>synonym</b>	<b>syntype</b>
<b>system</b>	<b>task</b>	<b>then</b>
<b>this</b>	<b>timer</b>	<b>to</b>
<b>try</b>	<b>type</b>	<b>use</b>
<b>value</b>	<b>via</b>	<b>virtual</b>
<b>with</b>	<b>xor</b>	

<space> ::=

Les caractères dans les unités <lexical unit> et dans les notes <note>, ainsi que le caractère <space> et les caractères de commande sont définis par la version internationale de référence de l'Alphabet international de référence (Recommandation T.50). L'unité lexicale <space> représente le caractère espace T.50 SPACE (acronyme SP), qui (pour des raisons évidentes) ne peut être représenté.

<text> est utilisé dans une zone <comment area> lorsqu'il est équivalent à une chaîne <character string> et dans une zone <text extension area> lorsqu'il doit être considéré comme une séquence d'autres unités lexicales.

Les caractères de suppression T.50 delete sont totalement ignorés. Si un ensemble de caractères étendu est utilisé, les caractères non définis par la Recommandation T.50 ne peuvent apparaître qu'en



<text> dans une zone <comment area> ou une chaîne <character string> dans un commentaire <comment> ou une <note>.

Lorsqu'un caractère souligné <underline> est suivi d'un ou de plusieurs caractères <space>, tous ces caractères (y compris le <underline> souligné) sont ignorés, par exemple A\_ B correspond au même <name> que AB. Cet emploi de <underline> permet de répartir des unités <lexical unit> sur plus d'une ligne. Cette règle s'applique en priorité par rapport à toutes les autres règles lexicales.

Un caractère de commande (non-space) peut apparaître là où un <space> peut apparaître et a la même signification qu'un espace <space>.

L'occurrence d'un caractère de commande n'est pas significative dans un texte <informal text> ni dans une <note>. Pour construire une expression de chaîne contenant des caractères de commande, l'opérateur <concatenation sign> et les littéraux des caractères de commande doivent être utilisés. Tous les espaces d'une chaîne de caractères sont significatifs: une séquence d'espaces n'est pas interprétée comme un espace.

Un nombre quelconque d'espaces <space> peut être inséré avant ou après toute unité <lexical unit>. Des <space>s ou <note>s insérés n'ont aucune signification syntaxique mais il est parfois nécessaire d'utiliser un <space> ou une <note> pour séparer deux unités <lexical unit>.

Dans toutes les unités <lexical unit> sauf les mots clés, les lettres <letter> majuscules et minuscules sont distinctes. Par conséquent, AB, aB, Ab et ab représentent quatre mots <word> différents. Un mot clé tout en majuscules <keyword> a la même utilisation qu'un mot clé tout en minuscules <keyword> avec la même orthographe (quelle que soit la casse), mais une séquence de lettres majuscules et minuscules mélangées avec la même orthographe qu'un mot clé <keyword> représente un mot <word>.

Par souci de concision interne des Règles lexicales, de la *grammaire textuelle concrète* et de la *grammaire graphique concrète*, le mot clé <keyword> en minuscules correspond en tant que terminal au mot clé <keyword> en majuscules avec la même orthographe et peut être utilisé au même endroit. Par exemple, le terminateur de syntaxe concrète

**endblock**

représente les variantes lexicales

{ **endblock** | **ENDBLOCK** }

NOTE – Les caractères gras minuscules sont utilisés pour les mots clés dans la présente Recommandation. Il n'est pas obligatoire d'effectuer des distinctions au moyen d'attributs de police de caractères, mais cela peut être utile pour les lecteurs d'une spécification.

Une unité <lexical unit> se termine par le premier caractère, qui ne peut pas faire partie d'une unité <lexical unit> conformément à la syntaxe spécifiée ci-dessus. Si une unité <lexical unit> peut être à la fois un nom <name> et un mot clé <keyword>, alors c'est un mot clé <keyword>. Si deux noms <quoted operation name> ne diffèrent que par leur casse, c'est la sémantique du nom en minuscules qui s'applique, de sorte que (par exemple) l'expression "OR" (a, b) a la même signification que l'expression (a **or** b).

Des règles lexicales particulières s'appliquent dans un <macro body>.

## 6.2 Macro

Une définition de macro contient un ensemble de symboles graphiques ou d'unités lexicales, qui peuvent apparaître à un ou plusieurs endroits dans la grammaire textuelle concrète d'une spécification <sdl specification>. Chacun de ces endroits est indiqué par un appel de macro. Avant de pouvoir analyser une spécification <sdl specification>, chaque appel de macro doit être remplacé par la définition de macro correspondante.

## 6.2.1 Règles lexicales supplémentaires

```
<formal name> ::=  
    [<name>%] <macro parameter>  
    { [%<name>] %<macro parameter> }*  
    [%<name>]
```

## 6.2.2 Définition de macro

```
<macro definition> ::=  
    macrodefinition <macro name>  
        [<macro formal parameters>] <end>  
        <macro body>  
    endmacro [<macro name>] <end>  
  
<macro formal parameters> ::=  
    ( <macro formal parameter> { , <macro formal parameter> }* )  
  
<macro formal parameter> ::=  
    <name>  
  
<macro body> ::=  
    {<lexical unit> | <formal name>}*  
  
<macro parameter> ::=  
    <macro formal parameter>  
    | macroid
```

Les paramètres <macro formal parameter> doivent être distincts. Les paramètres <macro actual parameter> doivent correspondre un à un avec les paramètres <macro formal parameter> correspondants.

Le corps <macro body> ne doit pas contenir de mots clé **endmacro** ni **macrodefinition**.

Une définition <macro definition> contient des unités lexicales.

Le nom <macro name> est visible dans toute la définition du système quel que soit l'endroit où la définition de macro apparaît. Un appel de macro peut apparaître avant la définition de macro correspondante.

Une définition de macro peut contenir plusieurs appels de macro, mais une définition de macro ne peut pas s'appeler elle-même directement ou indirectement par l'intermédiaire d'appels de macro dans d'autres définitions de macro.

Le mot clé **macroid** peut être utilisé comme un paramètre formel de pseudo-macro à l'intérieur de chaque définition de macro. Aucun paramètre <macro actual parameter> ne peut lui être attribué, et il est remplacé par un nom <name> unique pour chaque développement de la définition de macro (à l'intérieur d'un développement le même nom <name> est utilisé pour chaque occurrence de **macroid**).

### Exemple

Un exemple de définition <macro definition> est donné ci-après:

```
macrodefinition Exam (alfa, c, s, a);  
    block alfa referenced;  
    channel c from a to alfa with s; endchannel c;  
endmacro Exam;
```

### 6.2.3 Appel de macro

```
<macro call> ::=
    macro <macro name> [<macro call body>] <end>
<macro call body> ::=
    ( <macro actual parameter> {, <macro actual parameter>}* )
<macro actual parameter> ::=
    <lexical unit>*
```

L'unité <lexical unit> ne peut être une virgule "," ou une parenthèse droite ")". Si aucun de ces caractères n'est requis dans un paramètre <macro actual parameter>, le paramètre <macro actual parameter> doit être une chaîne <character string>. Si le paramètre <macro actual parameter> est une chaîne <character string>, la valeur de la chaîne <character string> est utilisée lorsque le paramètre <macro actual parameter> remplace un paramètre <macro formal parameter>.

Un appel <macro call> peut apparaître à un endroit quelconque où une unité <lexical unit> est autorisée.

#### Modèle

Une spécification <sdl specification> peut contenir des définitions de macro et des appels de macro. Avant de pouvoir analyser une spécification <sdl specification>, tous les appels de macro doivent être développés. Le développement d'un appel de macro signifie qu'une copie de la définition de macro ayant le même nom <macro name> que celle qui est donnée dans l'appel de macro, est développée pour remplacer l'appel de macro. C'est-à-dire qu'une copie du corps de macro est créée, et que chaque occurrence des paramètres <macro formal parameter> de la copie est remplacée par les paramètres <macro actual parameter> correspondants de l'appel de macro, ensuite, les appels de macro dans la copie, le cas échéant sont développés. Tous les caractères pour-cent (%) dans les noms <formal name> sont supprimés lorsque les paramètres <macro formal parameter> sont remplacés par les paramètres <macro actual parameter>.

Il doit y avoir une correspondance biunivoque entre paramètre <macro formal parameter> et paramètre <macro actual parameter>.

#### Exemple

Un exemple d'appel <macro call> dans une partie d'une définition <block definition> est donné ci-après:

```
.....
block A referenced;
macro Exam (B, C1, S1, A);
.....
```

Le développement de cet appel de macro, utilisant l'exemple du 6.2.2, donne le résultat suivant:

```
.....
block A referenced;
block B referenced;
channel C1 from A to B with S1; endchannel C1;
.....
```

### 6.3 Règles de visibilité, noms et identificateurs

#### Grammaire abstraite

<i>Identifieur</i>	::	<i>Qualifieur Nom</i>
<i>Qualifieur</i>	=	<i>Path-item</i> +
<i>Path-item</i>	=	<i>Package-qualifieur</i>
		<i>Agent-type-qualifieur</i>
		<i>Agent-qualifieur</i>
		<i>State-type-qualifieur</i>
		<i>State-qualifieur</i>
		<i>Data-type-qualifieur</i>
		<i>Procedure-qualifieur</i>
		<i>Signal-qualifieur</i>
		<i>Interface-qualifieur</i>
<i>Package-qualifieur</i>	::	<i>Package-name</i>
<i>Agent-type-qualifieur</i>	::	<i>Agent-type-name</i>
<i>Agent-qualifieur</i>	::	<i>Agent-name</i>
<i>State-type-qualifieur</i>	::	<i>State-type-name</i>
<i>State-qualifieur</i>	::	<i>State-name</i>
<i>Data-type-qualifieur</i>	::	<i>Data-type-name</i>
<i>Procedure-qualifieur</i>	::	<i>Procedure-name</i>
<i>Signal-qualifieur</i>	::	<i>Signal-name</i>
<i>Interface-qualifieur</i>	::	<i>Interface-name</i>
<i>Package-name</i>	=	<i>Name</i>
<i>Agent-type-name</i>	=	<i>Name</i>
<i>Agent-name</i>	=	<i>Name</i>
<i>State-type-name</i>	=	<i>Name</i>
<i>Data-type-name</i>	=	<i>Name</i>
<i>Interface-name</i>	=	<i>Name</i>
<i>Name</i>	::	<i>Token</i>

#### Grammaire textuelle concrète

<identifieur> ::=	[<qualifieur>] <nom>
<qualifieur> ::=	<qualifieur début signe> <path item> { / <path item> } * <qualifieur fin signe>
<string name> ::=	<character string>
	<bit string>
	<hex string>
<path item> ::=	[<scope unit kind>] <nom>
<scope unit kind> ::=	<b>package</b>
	<b>system type</b>
	<b>system</b>
	<b>block</b>
	<b>block type</b>
	<b>process</b>
	<b>process type</b>
	<b>state</b>
	<b>state type</b>
	<b>procedure</b>
	<b>signal</b>
	<b>type</b>
	<b>operator</b>
	<b>method</b>
	<b>interface</b>

Les unités de portée sont définies par les symboles non-terminaux de la grammaire concrète. Les symboles appartenant à la grammaire textuelle concrète figurent dans la colonne de gauche; les symboles correspondants en grammaire graphique concrète figurent dans la colonne de droite.

<package definition>	<package diagram>
<agent definition>	<agent diagram>
<agent type definition>	<agent type diagram>
<procedure definition>	<procedure diagram>
<data type definition>	
<interface definition>	
<operation definition>	<operation diagram>
<composite state>	<composite state area>
<composite state type definition>	<composite state type diagram>
<sort context parameter>	
<signal definition>	
<signal context parameter>	
<compound statement>	<task area>

Une liste de définitions est associée à une unité de portée. Chaque définition définit une ou plusieurs entités appartenant à la même classe d'entité et ayant un nom associé. La liste contient les définitions <textual gate definition>, les paramètres <formal context parameter>, les paramètres <agent formal parameters> et les paramètres <formal variable parameters> contenus dans l'unité de portée.

Bien que les noms <quoted operation name> et les noms <string name> aient leur propre notation syntaxique, ils sont en réalité des <name> qui sont représentés dans la *syntaxe abstraite* par un *Name*. Dans ce qui suit, ils sont étudiés comme s'ils étaient syntaxiquement aussi des <name>.

Les entités peuvent être regroupées en classes d'entités. Les classes d'entités sont les suivantes:

- a) paquetages;
- b) agents (système, blocs, processus);
- c) types d'agent (types de système, types de bloc, types de processus);
- d) canaux, accès;
- e) signaux, temporisateurs, interfaces, types de données;
- f) procédures, procédures distantes;
- g) variables (y compris les paramètres formels), synonymes;
- h) littéraux, opérateurs, méthodes;
- i) variables distantes;
- j) sortes;
- k) types d'état;
- l) listes de signaux;
- m) exceptions.

Un paramètre contextuel formel est une entité appartenant à la même classe d'entités que les paramètres contextuels réels correspondants.

Une définition de référence est une entité après l'étape de transformation de la définition <referenced definition> (voir Annexe F).

Chaque entité est dite avoir son contexte de définition dans l'unité de portée qui la définit.

Les entités sont référencées au moyen d'identificateurs <identifieur>. Le <qualifieur> dans un <identifieur> spécifie uniquement le contexte de définition de l'entité.

Le qualifieur <qualifieur> fait référence à un supertype ou reflète la structure hiérarchique à partir du niveau du système ou d'ensemble vers le contexte de définition, et de manière telle que le niveau du système ou d'ensemble soit la partie textuelle la plus à gauche. Le nom *Name* d'une entité est

alors représenté par le qualificateur, le nom de l'entité et, uniquement pour les entités de classe h), par la signature (voir 12.1.7.1, 12.1.4). Toutes les entités de même classe doivent avoir des noms *Name* différents.

NOTE 1 – Par conséquent, il ne peut y avoir deux définitions dans la même unité de portée et appartenant à la même classe d'entités portant le même nom <name>. Les seules exceptions sont les opérations définies dans la même définition <data type definition>, sous réserve qu'elles aient au moins un argument <sort> ou un résultat <sort> différent.

Les <state name>, les <connector name> et les <gate name> qui se présentent dans les définitions de canaux, les <macro formal parameter> et les <macro name> ont des règles de visibilité particulières et ne peuvent pas être qualifiés. D'autres règles de visibilité particulières sont expliquées dans les paragraphes concernés.

NOTE 2 – Il n'existe pas de classe <scope unit kind> correspondant aux unités de portée définies par les schémas <task>, <task area> et <compound statement>. Il n'est donc pas possible de se référer aux identificateurs introduits dans une définition associée à ces unités de portée par des qualificateurs.

Une entité peut être référencée au moyen d'un <identifiant>, si cette entité est visible. Une entité est visible dans une unité de portée si:

- a) son contexte de définition se trouve dans cette unité de portée;
- b) l'unité de portée est une spécialisation et l'entité est visible dans le type de base;
  - 1) elle n'est pas protégée contre la visibilité par une construction spéciale définie en 12.1.9.3;
  - 2) le renommage de spécialisation des données n'a pas été appliqué (12.1.3);
  - 3) il ne s'agit pas d'un paramètre contextuel formel qui a déjà été relié à un paramètre contextuel réel (8.2);
- c) l'unité de portée possède une clause <package use clause> qui mentionne un <package> de sorte que:
  - 1) la liste <definition selection list> de la clause <package use clause> est omise ou le nom <name> de l'entité est mentionné dans une sélection <definition selection>;
  - 2) l'interface <package interface> de l'ensemble <package>, c'est-à-dire le contexte de définition de l'entité, a été omise ou le nom <name> de l'entité est mentionné dans l'interface <package interface>;
- d) l'unité de portée contient une définition <interface definition> qui correspond au contexte de définition de l'entité (voir 12.1.2);
- e) l'unité de portée contient une définition <data type definition> qui correspond au contexte de définition de l'entité et elle n'est pas protégée contre la visibilité par une construction spéciale définie en 12.1.9.3;
- f) l'entité est visible dans l'unité de portée qui définit cette unité de portée.

Il est permis d'omettre certains <path item> les plus à gauche ou bien tout le <qualifier> d'un <identifiant> si les <path item> omis peuvent être étendus uniquement à un <qualifier> complet.

Lorsque la partie <name> d'un <identifiant> désigne une entité qui n'appartient pas à la classe d'entités h), le <name> est associé à une entité qui a son contexte de définition dans l'unité de portée englobante la plus proche dans laquelle le <qualifier> de l'<identifiant> est le même que la partie la plus à droite du <qualifier> complet désignant cette unité de portée (résolution par conteneur). Si l'<identifiant> ne contient pas de <qualifier>, la nécessité de mise en correspondance des <qualifier> n'est pas applicable.

L'association d'un <name> à une définition au moyen de la résolution par conteneur se déroule selon les étapes suivantes, en débutant par l'unité de portée désignée par le <qualifier> partiel:

- a) s'il existe une entité unique dans une unité de portée avec le même <name> et la même classe d'entités, le <name> est associé à cette entité; autrement
- b) si l'unité de portée est une spécialisation, l'étape a) est répétée en boucle jusqu'à ce que le <name> puisse être associé à une entité; autrement
- c) si l'unité de portée possède une clause <package use clause> et qu'il existe une entité unique visible dans le <package>, le <name> est associé à cette entité; autrement
- d) si l'unité de portée a une définition <interface definition> et qu'il existe une entité unique visible dans la définition <interface definition>, le <name> est associé à cette entité; autrement
- e) la résolution par conteneur est tentée dans l'unité de portée qui définit l'unité de portée actuelle.

En fonction de la visibilité et de l'utilisation de qualificatifs, une clause <package use clause> associée à une unité de portée est considérée comme la définition d'un paquetage englobant directement l'unité de portée et est définie dans l'unité de portée à l'endroit où cette unité de portée est définie. Si l'<identifiant> ne contient pas de <qualifier>, une clause <package use clause> est considérée comme l'unité de portée englobante la plus proche à laquelle elle est associée et contient les entités visibles depuis le paquetage.

NOTE 3 – Dans la syntaxe concrète, les paquetages ne peuvent pas être définis à l'intérieur d'autres unités de portée. La règle ci-dessus n'est valable que pour les règles de visibilité qui s'appliquent aux paquetages. Par conséquent, il est possible de référencer des noms dans un paquetage au moyen de différents qualificatifs, un pour chaque clause <package use clause> comprise dans le paquetage.

Lorsque la partie <name> d'un <identifiant> désigne une entité appartenant à la classe d'entités h), l'association du <name> à une définition doit pouvoir être résolue par le contexte. La résolution par le contexte est tentée après la résolution par conteneur; c'est-à-dire que si un <name> peut être associé à une entité au moyen de la résolution par conteneur, cette association est utilisée, même si la résolution par le contexte pouvait également permettre d'associer ce <name> à une entité. Le contexte de résolution d'un <name> est une affectation <assignment> (si le <name> est apparu dans une <assignment>), une décision <decision> (si le <name> est apparu dans la <question> ou les réponses <answer> d'une <decision>) ou bien une <expression> qui ne fait partie d'aucune autre <expression>. La résolution par le contexte est effectuée comme suit:

- a) pour chaque nom <name> apparaissant dans le contexte, trouver l'ensemble d'identificateurs <identifiant>, de sorte que la partie du nom <name> soit visible, ayant le même nom <name> et le même qualificatif <qualifier> partiel, en tenant compte du renommage;
- b) construire le produit des ensembles d'identificateurs <identifiant> associés à chaque nom <name>;
- c) considérer uniquement les éléments dans le produit qui ne violent aucune contrainte de sorte statique, en tenant également compte des sortes dans les paquetages qui ne sont pas rendues visibles dans une clause <package use clause>. Chaque élément restant représente une association possible et statistiquement correcte entre des noms <name> dans l'<expression> et des entités;
- d) en fonction de la possibilité de polymorphisme dans les affectations <assignment> (voir 12.3.3), la sorte statique d'une <expression> peut être différente de la sorte statique d'une <variable>, il en va de même pour les affectations implicites dans les paramètres. Compter le nombre de ces défauts d'appariement pour chaque élément;
- e) comparer les éléments par paires en éliminant ceux qui présentent plus de défauts d'appariement;

- f) s'il reste plus d'un élément, tous les identificateurs <identifieur> non uniques doivent représenter la même signature *Dynamic-operation-signature*, autrement le <name> dans le contexte ne peut pas être associé à une définition.

Il est uniquement permis d'omettre la classe facultative <scope unit kind> dans un trajet <path item> si le <name> ou le nom <quoted operation name> détermine uniquement l'unité de portée.

Il n'existe pas de syntaxe abstraite correspondante pour la classe <scope unit kind> désignée par **operator** ou **method**.

Dans la grammaire textuelle concrète, le nom ou l'identificateur optionnel dans une définition après les mots clé de terminaison (**endsystem**, **endblock**, etc.) doit être syntaxiquement le même que le nom ou l'identificateur qui suit le mot clé commençant correspondant (respectivement **system**, **block**, etc.).

## 6.4 Texte informel

*Grammaire abstraite*

*Informal-text* :: ...

*Grammaire textuelle concrète*

<informal text> ::= <character string>

*Sémantique*

Lorsqu'un texte informel est utilisé dans une spécification, cela signifie qu'aucun élément sémantique du texte n'est défini par le SDL. La sémantique du texte informel peut être définie par d'autres moyens.

## 6.5 Règles applicables aux dessins

La taille des symboles graphiques est choisie par l'utilisateur.

Les frontières des symboles ne doivent ni se superposer, ni se couper. Font exception à cette règle les symboles de ligne, qui peuvent se couper. Il n'existe pas d'association logique entre les symboles qui se coupent. Les éléments suivants sont des symboles de ligne:

- <association symbol>
- <channel symbol>
- <create line symbol>
- <dashed association symbol>
- <dependency symbol>
- <flow line symbol>
- <solid association symbol>
- <solid on exception association symbol>
- <specialization relation symbol>

Le métasymbole *is followed by* indique un symbole <flow line symbol>.

Les symboles de ligne peuvent être constitués par un ou plusieurs segments de droite en trait plein.

Les flèches sont nécessaires chaque fois qu'un symbole <flow line symbol> entre dans un autre symbole <flow line symbol>, dans un symbole <out connector symbol> ou dans une zone <nextstate area>. Dans d'autres cas, ces flèches sont facultatives sur les symboles <flow line symbol>. Les symboles <flow line symbol> sont horizontaux ou verticaux.

On peut utiliser des images symétriques verticales des symboles <input symbol>, <output symbol>, <internal input symbol>, <internal output symbol>, <priority input symbol>, <raise symbol>, <handle symbol>, <comment symbol> et <text extension symbol>.



L'argument de la partie droite du métasymbole *is associated with* doit être plus proche de l'argument de gauche que tout autre symbole graphique. Les éléments syntaxiques de l'argument de droite doivent pouvoir être distingués les uns des autres.

Le texte situé à l'intérieur d'un symbole graphique doit être lu de la gauche vers la droite, en partant du coin supérieur gauche. La limite droite du symbole est interprétée comme un caractère de nouvelle ligne, indiquant le cas échéant que la lecture doit continuer au point le plus à gauche de la ligne suivante.

## 6.6 Subdivision des dessins

La définition qui suit concernant la subdivision ne fait pas partie de la *grammaire graphique concrète*, néanmoins, on utilise le même métalangage.

```

<page> ::=
    <frame symbol> contains
    { <heading area> <page number area> { <symbol> | <lexical unit> }* }

<heading area> ::=
    <implicit text symbol> contains <heading>

<heading> ::=
    <kernel heading> [<extra heading>]

<kernel heading> ::=
    [<virtuality>] [exported]
    <drawing kind> <drawing qualifier> | <drawing name>

<drawing kind> ::=
    package      | system [type] | block [type] | process [type]
    state [type] | procedure   | operator   | method

<extra heading> ::=
    une partie de l'en-tête de diagramme n'est pas contenue dans l'en-tête principal

<page number area> ::=
    <implicit text symbol> contains [<page number> [ (<number of pages>) ]]

<page number> ::=
    <literal name>

<number of pages> ::=
    <Natural literal name>

<symbol> ::=
    il peut s'agir de n'importe quel terminal défini par un nom de règle se terminant par "symbol".

```

La <page> est un non-terminal de départ, par conséquent, il n'est associé à aucune règle de production. Un dessin peut être subdivisé en un nombre de <page>, auquel cas le symbole <frame symbol> délimitant le dessin et l'en-tête <heading> du dessin est remplacé par un symbole <frame symbol> et un en-tête <heading> pour chaque <page>.

Un <symbol> est un symbole non-terminal graphique (voir 5.4.3).

Afin d'avoir une séparation nette entre la zone <heading area> et la zone <page number area>, le symbole <implicit text symbol> n'est pas matérialisé, il est sous-entendu. La zone <heading area> est placée dans le coin supérieur gauche du symbole <frame symbol>. La zone <page number area> est placée dans le coin supérieur droit du symbole <frame symbol>. L'en-tête <heading> et l'unité syntaxique dépendent du type de dessin.

Les en-têtes <extra heading> doivent être présentés sur la première page du dessin, mais ils sont facultatifs sur les pages suivantes. Les en-têtes <heading> et les classes <drawing kind> sont élaborés pour les dessins particuliers dans les paragraphes individuels de la présente Recommandation. La présente Recommandation ne définit pas les en-têtes <extra heading>.

<virtuality> désigne la virtualité du type défini par le diagramme (voir 8.3.2) et **exported** indique si une procédure donnée est exportée en tant que procédure distante (voir 10.5).

Les dessins de SDL/GR sont la zone <specification area>, le diagramme <package diagram>, le diagramme <agent diagram>, le diagramme <agent type diagram>, le diagramme <procedure diagram>, le diagramme <operation diagram>, la zone <composite state area> et le diagramme <composite state type diagram>.

## 6.7 Commentaire

Un commentaire est une notation qui représente des commentaires associés avec des symboles ou du texte.

Dans la *grammaire textuelle concrète*, on utilise deux formes de commentaires. La première forme est la <note>.

La syntaxe concrète de la deuxième forme est:


<end> ::= [ <comment> ] <semicolon>


<comment> ::= **comment** <character string>

<end> dans les zones <package text area>, <agent text area>, <procedure text area>, <composite state text area>, <operation text area> et dans la liste <statement list> ne doit pas contenir de commentaire <comment>.

Dans la *grammaire graphique concrète*, la syntaxe suivante est utilisée:

<comment area> ::= <comment symbol> **contains** <text>  
**is connected to** <dashed association symbol>

<comment symbol> ::= 

<dashed association symbol> ::= 

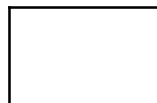
Une des extrémités du symbole <dashed association symbol> doit être reliée au milieu du segment vertical du symbole <comment symbol>.

Un symbole <comment symbol> peut être relié à tout symbole graphique au moyen d'un symbole <dashed association symbol>. Le symbole <comment symbol> est considéré comme un symbole fermé en complétant (par la pensée) le rectangle afin d'entourer le texte. Il contient le texte du commentaire se rapportant au symbole graphique.

NOTE – Le <text> dans la *grammaire graphique concrète* correspond à <character string> dans la *grammaire textuelle concrète* sans les <apostrophe>s de fermeture. Toute conversion entre les notations graphiques et textuelles entraîne le doublement ou le dédoublement des <apostrophe>s de fermeture.

## 6.8 Extension de texte

<text extension area> ::= <text extension symbol> **contains** <text>  
**is connected to** <solid association symbol>

<text extension symbol> ::= 

<solid association symbol> ::=

Une des extrémités du symbole <solid association symbol> doit être reliée au milieu du segment vertical du symbole <text extension symbol>.

Un symbole <text extension symbol> peut être relié à tout symbole graphique au moyen d'un symbole <solid association symbol>. Le symbole <text extension symbol> est considéré comme un symbole fermé en complétant (par la pensée) le rectangle.

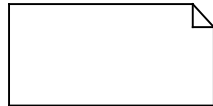
Le texte <text> contenu dans le symbole <text extension symbol> est la suite du texte dans le symbole graphique et est considéré comme contenu dans ce symbole et est par conséquent traité comme un nombre d'unités lexicales.

## 6.9 Symbole de texte

Le symbole <text symbol> est utilisé dans tout diagramme <diagram>. Le contenu dépend du diagramme.

*Grammaire graphique concrète*

<text symbol> ::=



## 7 Organisation des spécifications SDL

Un système SDL ne peut généralement pas être décrit de manière simple comme une partie unique et indépendante d'un texte ou à l'aide d'un diagramme unique. Le langage SDL permet donc la subdivision de la spécification et l'utilisation de SDL provenant de l'extérieur.

### 7.1 Cadre d'application

Une spécification <sdl specification> peut être décrite comme une <system specification> monolithique (éventuellement augmentée par une collection de <package>) ou comme une collection de <package> et de <referenced definition>. Un <package> permet l'utilisation des définitions dans différents contextes en "utilisant" le package dans ces contextes, c'est-à-dire dans des systèmes ou des paquetages qui peuvent être indépendants. Une <referenced definition> est une définition qui a été déplacée de son contexte de définition pour avoir un aperçu général à l'intérieur d'une description de système. Elle est "insérée" exactement à un endroit (le contexte de définition) au moyen d'une référence. Une zone <specification area> permet la représentation graphique de la relation entre la spécification <system specification> et les paquetages <package>.

*Grammaire abstraite*

*SDL-specification* :: [ *Agent-definition* ]  
*Package-definition-set*

*Concrete grammar*

<sdl specification> ::=

[<specification area>]  
{ <package> | <system specification> } <package>\* <referenced definition>\*

<system specification> ::=

<textual system specification> | <graphical system specification>

<package> ::=

<package definition> | <package diagram>

### Grammaire textuelle concrète

```
<textual system specification> ::=  
    <agent definition>  
    | <textual typebased agent definition>
```

### Grammaire graphique concrète

```
<graphical system specification> ::=  
    <agent diagram>  
    | <graphical typebased agent definition>  
  
<specification area> ::=  
    <frame symbol> contains  
    {  
    | <agent reference area>  
    | <graphical typebased agent definition>  
    [ is connected to {<graphical package use area>+ } set ]  
    }  
    {<package reference area>* } set}
```

### Sémantique

Une *spécification de SDL* présente la sémantique combinée de l'agent du système (s'il existe) avec les paquetages. Si aucun agent de système n'est spécifié, la spécification fournit un ensemble de définitions à utiliser dans d'autres spécifications.

### Modèle

Une spécification <system specification> étant une définition <process definition> ou une définition <textual typebased process definition> est une syntaxe dérivée pour une définition <system definition> ayant le même nom que le processus, contenant des canaux implicites et contenant la définition <process definition> ou la <textual typebased process definition> comme définition unique.

Une spécification <system specification> étant un diagramme <process diagram> ou une définition <graphical typebased process definition> est une syntaxe dérivée pour un diagramme <system diagram> ayant le même nom que le processus, contenant des canaux implicites et contenant le diagramme <process diagram> ou la définition <graphical typebased process definition> comme définition unique.

Une spécification <system specification> étant une définition <block definition> ou une définition <textual typebased block definition> est une syntaxe dérivée pour une définition <system definition> ayant le même nom que le bloc, contenant des canaux implicites et contenant la définition <block definition> ou la définition <textual typebased block definition> comme définition unique.

Une spécification <system specification> étant un diagramme <block diagram> ou une définition <graphical typebased block definition> est une syntaxe dérivée pour un diagramme <system diagram> ayant le même nom que le bloc, contenant des canaux implicites et contenant le diagramme <block diagram> ou la définition <graphical typebased block definition> comme définition unique.

## 7.2 Paquetage

Afin qu'une définition de type puisse être utilisée dans des systèmes différents, elle doit être définie comme une partie d'un *paquetage*.

Les définitions qui font partie d'un paquetage définissent des types, des générateurs de données, des listes de signaux, des spécifications distantes et des synonymes.

Les définitions à l'intérieur d'un paquetage sont rendues visibles à une autre unité de portée par une clause d'utilisation de paquetage (*package use clause*).

## Grammaire abstraite

*Package-definition* :: *Package-name*  
*Package-definition-set*  
*Data-type-definition-set*  
*Syntype-definition-set*  
*Signal-definition-set*  
*Exception-definition-set*  
*Agent-type-definition-set*  
*Composite-state-type-definition-set*  
*Procedure-definition-set*

## Grammaire textuelle concrète

<package definition> ::=  
    {<package use clause>}\*  
    <package heading> <end>  
    {<entity in package>}\*  
    **endpackage** [<package name>] <end>

<package heading> ::=  
    **package** [ <qualifier> ] <package name>  
    [<package interface>]

<entity in package> ::=  
    | <agent type definition>  
    | <agent type reference>  
    | <package definition>  
    | <package reference>  
    | <signal definition>  
    | <signal reference>  
    | <signal list definition>  
    | <remote variable definition>  
    | <data definition>  
    | <data type reference>  
    | <procedure definition>  
    | <procedure reference>  
    | <remote procedure definition>  
    | <composite state type definition>  
    | <composite state type reference>  
    | <exception definition>  
    | <select definition>  
    | <macro definition>  
    | <interface reference>  
    | <association>

<package reference> ::=  
    **package** [ <qualifier> ] <package name> **referenced** <end>

<package use clause> ::=  
    **use** <package identifier> [ / <definition selection list> ] <end>

<definition selection list> ::=  
    <definition selection> { , <definition selection> }\*

<definition selection> ::=  
    [<selected entity kind>] <name>

```

<selected entity kind> ::=
    | system type
    | block type
    | process type
    | package
    | signal
    | procedure
    | remote procedure
    | type
    | signallist
    | state type
    | synonym
    | remote
    | exception
    | interface

```

```

<package interface> ::=
    public <definition selection list>

```

Pour chaque identificateur <package identifier> mentionné dans une clause <package use clause>, un paquetage <package> correspondant doit exister. Ce paquetage peut faire partie d'une <sdl specification> ou peut être inclus dans un autre paquetage, ou bien il doit exister un mécanisme permettant d'accéder au <package> correspondant, comme s'il faisait partie de la spécification <sdl specification>. Ce mécanisme n'est pas défini dans la présente Recommandation.

Si le paquetage fait partie d'une spécification <sdl specification> ou s'il existe un mécanisme permettant d'accéder au <package> correspondant, il ne doit pas y avoir de <qualifier> dans le <package identifier>.

Si le paquetage <package> correspondant est contenu dans un autre paquetage, l'identificateur <package identifier> reflète la structure hiérarchique depuis le paquetage <package> le plus à l'extérieur jusqu'au paquetage <package> défini. Les <path items> les plus à gauche peuvent être omis.

L'identificateur <package identifier> doit désigner un paquetage visible. Tous les paquetages <package> dans le qualificatif <qualifier> de l'identificateur <package identifier> entièrement qualifié doivent être visibles. Un paquetage est visible s'il fait partie de la spécification <sdl specification> ou si son identificateur <identifier> est visible conformément aux règles de visibilité du SDL pour les <identifier>. Les règles de visibilité du SDL précisent qu'un <package identifier> peut être visible avec une <package use clause> et qu'un paquetage est visible dans la portée dans laquelle il est contenu. Cette portée s'étend également à la <package use clause> du paquetage conteneur.

De même, si la spécification <system specification> est omise dans une spécification <sdl specification>, il doit exister un mécanisme permettant d'utiliser les <package> dans d'autres spécifications <sdl specification>. Avant d'utiliser les <package> dans d'autres spécifications <sdl specification>, le modèle des macros et des définitions référencées est appliqué. La présente Recommandation ne définit pas ce mécanisme.

La **procédure** <selected entity kind> est utilisée pour sélectionner à la fois les procédures (normales) et les procédures distantes. Si une procédure normale et une procédure distante ont le même nom <name> donné, **procédure** désigne la procédure normale. Pour forcer la sélection <definition selection> à désigner la procédure distante, le mot clé **procédure** peut être précédé de **remote**.

Le mot clé **type** est utilisé pour sélectionner un nom de sorte et également un nom de type synonyme dans un paquetage. Le mot clé **remote** est utilisé pour la sélection d'une définition de variable distante.

## Grammaire graphique concrète

<package diagram> ::=

```

<frame symbol> contains
  { <package heading>
    { {<package text area>}*
      {<diagram in package>}* } set }
  [ is associated with <package use area> ]

```

<package use area> ::=

```

<text symbol> contains {<package use clause>}*

```

<package text area> ::=

```

<text symbol> contains
  {
  | <agent type reference>
  | <package reference>
  | <signal definition>
  | <signal reference>
  | <signal list definition>
  | <remote variable definition>
  | <data definition>
  | <data type reference>
  | <procedure definition>
  | <procedure reference>
  | <remote procedure definition>
  | <exception definition>
  | <select definition>
  | <macro definition>
  | <interface reference> }*

```

<diagram in package> ::=

```

| <package diagram>
| <package reference area>
| <type in agent area>
| <data type reference area>
| <signal reference area>
| <procedure reference area>
| <interface reference area>
| <create line area>
| <option area>

```

<package reference area> ::=

```

<package symbol> contains <package identifier>
  [ is connected to {<graphical package use area>+ } set ]

```

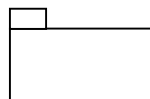
<graphical package use area> ::=

```

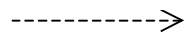
<dependency symbol> is connected to { <package diagram> | <package reference area> }

```

<package symbol> ::=



<dependency symbol> ::=



La zone <package use area> doit être placée en tête du symbole <frame symbol>. Le <qualifier> facultatif et le nom <package name> d'une zone <package reference area> doivent être contenus dans le rectangle inférieur du symbole <package symbol>.

Les zones <graphical package use area> pour une zone <package reference area> sont des spécifications partielles de la clause <package use clause> pour le diagramme <package diagram> ou du progiciel <package> ou de la spécification <system specification> pour une zone <package reference area> contenue dans une zone <specification area>, et ces zones doivent être cohérentes avec cette clause <package use clause>.

## Sémantique

La visibilité du nom d'une entité définie à l'intérieur d'un <package> est expliquée en 6.3.

Les signaux qui ne sont pas rendus visibles dans une clause **use**, peuvent faire partie d'une liste de signaux via un identificateur <signal list identifier> rendu visible dans une clause **use**. Ces signaux pourront alors affecter l'ensemble complet de signaux d'entrée valides d'un agent.

Si un nom dans une sélection <definition selection> désigne une sorte <sort>, la sélection <definition selection> désigne aussi implicitement le type de données ayant défini la sorte <sort> et tous les littéraux et les opérations définis par le type de données. Si un nom dans une sélection <definition selection> désigne un syntype, la sélection <definition selection> désigne aussi implicitement le type de données ayant défini l'identificateur <parent sort identifier> et tous les littéraux et les opérations définis par le type de données.

La classe <selected entity kind> dans la sélection <definition selection> désigne la classe d'entité du nom <name>. Toute paire de (<selected entity kind>, <name>) doit être distincte à l'intérieur d'une liste <definition selection list>. Pour une sélection <definition selection> dans une interface <package interface>, la classe <selected entity kind> peut être omise uniquement s'il n'y a pas d'autre nom ayant son occurrence de définition directement dans le <package>. Pour une sélection <definition selection> dans une clause <package use clause>, la classe <selected entity kind> peut être omise si et seulement si une seule et unique entité de ce nom est mentionnée dans n'importe quelle liste <definition selection list> pour le paquetage, ou si le paquetage n'a pas de liste <definition selection list> et contient directement une définition unique de ce nom.

## Modèle

Une définition <system definition> et toute <package definition> possèdent une clause <package use clause> implicite:

```
use Predefined;
```

où Predefined désigne un paquetage contenant les données définies selon l'Annexe D.

Si un paquetage est mentionné dans plusieurs clauses <package use clause> d'une définition <package definition>, cela correspond à une clause <package use clause> qui sélectionne l'union des définitions choisies dans les clauses <package use clause>.

## 7.3 Définition référencée

### Grammaire concrète

```
<referenced definition> ::=  
    <definition> | <diagram>
```

### Grammaire textuelle concrète

```
<definition> ::=  
    | <package definition>  
    | <agent definition>  
    | <agent type definition>  
    | <composite state>  
    | <composite state type definition>  
    | <procedure definition>  
    | <operation definition>  
    | <macro definition>
```



## Grammaire graphique concrète

```
<diagram> ::=  
    | <package diagram>  
    | <agent diagram>  
    | <agent type diagram>  
    | <composite state area>  
    | <composite state type diagram>  
    | <procedure diagram>  
    | <operation diagram>
```

Pour chaque <referenced definition>, à l'exception de <macro definition>, il doit y avoir une référence dans le <package> associé ou la spécification <system specification> associée. Les références textuelles et graphiques sont définies respectivement comme <... reference> et <... reference area> (par exemple <block reference> et <block reference area>).

Un <qualifier> et <name> facultatifs sont présents dans une définition <referenced definition> après le ou les mots clés initiaux. Pour chaque référence, il doit exister une définition <referenced definition> avec la même classe d'entité que la référence, dont le qualificateur <qualifier>, s'il est présent, désigne un trajet à partir de l'unité de portée englobant la référence jusqu'à la référence. Si deux définitions <referenced definition> de la même classe d'entité ont le même <name>, le <qualifier> de l'un des qualificateurs ne doit pas constituer la partie la plus à gauche de l'autre <qualifier> et aucun des <qualifier> ne peut être omis. Le <qualifier> doit être présent si la définition <referenced definition> est une définition <package definition>.

On ne peut pas spécifier un qualificateur <qualifier> après le ou les mots clés initiaux pour les définitions qui ne sont pas des définitions <referenced definition>.

### Modèle

Avant d'établir les propriétés d'une spécification <system specification>, chaque référence est remplacée par la définition <referenced definition> correspondante. Dans cette substitution, le <qualifier> de la définition <referenced definition> est éliminé. Si la définition <referenced definition> est un <diagram> référencé d'une <definition> ou si elle est une <definition> référencée d'un <diagram>, on considère que cette définition <referenced definition> est traduite à la grammaire appropriée pendant la substitution.

## 8 Concepts structurels

Le présent paragraphe introduit un certain nombre de mécanismes du langage, caractérisés par la modélisation des phénomènes spécifiques à l'application par des instances et la modélisation des concepts spécifiques à l'application par des types. Cela implique que le mécanisme d'héritage est destiné à représenter la généralisation et de spécialisation des concepts.

Les mécanismes du langage introduits fournissent:

- a) des définitions de types (pures) qui peuvent être définies à n'importe quel endroit dans un système ou un paquetage;
- b) des définitions d'instances fondées sur les types qui définissent des instances ou des ensembles d'instances conformément aux types;
- c) des définitions de types paramétrés qui sont indépendantes de la portée englobante au moyen de paramètres de contexte et qui peuvent être limitées à des portées spécifiques;
- d) la spécialisation des définitions de supertypes en des définitions de sous-types, en ajoutant des propriétés et en redéfinissant des types et transitions virtuels.

## 8.1 Types, instances et accès

On distingue dans les descriptions du SDL entre les définitions d'instances (ou ensembles d'instances) et les définitions de type. Le présent paragraphe spécifie (en 8.1.1) les définitions de types pour les agents ainsi que les spécifications des instances correspondantes (en 8.1.3); l'introduction d'autres types se trouve dans les procédures (9.4), les signaux (10.3), les temporisateurs (11.15), les sortes (12.1.1) et les interfaces (12.1.2). Une définition de type d'agent n'est connectée (par des canaux) à aucune instance; en revanche, les définitions de types d'agents introduisent des accès (8.1.6). Il s'agit des points de connexion sur les instances fondées sur les types pour les canaux.

Un type définit un ensemble de propriétés. Toutes les instances du type (5.2.1) ont cet ensemble de propriétés.

Une instance (ou un ensemble d'instances) a toujours un type, qui est implicite si l'instance n'est pas fondée de manière explicite sur un type. Par exemple, un diagramme de processus a un type implicite de processus anonyme équivalent.

### 8.1.1 Définitions de types structurels

Il s'agit des définitions de types pour les entités qui sont utilisées dans la structure d'une spécification. A l'opposé, les définitions de procédures sont également des définitions de types mais s'orientent plus sur le comportement que sur la structure.

#### 8.1.1.1 Types d'agent

Un type d'agent est un type de système, de bloc ou de processus. Lorsqu'un type est utilisé pour définir un agent, l'agent est du genre correspondant (système, bloc ou processus).

##### *Grammaire abstraite*

```
Agent-type-definition          :: Agent-type-name  
                                Agent-kind  
                                [ Agent-type-identif ]  
                                Agent-formal-parameter*  
                                Data-type-definition-set  
                                Syntype-definition-set  
                                Signal-definition-set  
                                Timer-definition-set  
                                Exception-definition-set  
                                Variable-definition-set  
                                Agent-type-definition-set  
                                Composite-state-type-definition-set  
                                Procedure-definition-set  
                                Agent-definition-set  
                                Gate-definition-set  
                                Channel-definition-set  
                                [ State-machine-definition ]  
  
Agent-type-identif           = Identif
```

##### *Grammaire textuelle concrète*

```
<agent type definition> ::=  
    <system type definition> | <block type definition> | <process type definition>
```

```

<agent type structure> ::=
    [<valid input signal set>]
    {
        {
            <entity in agent>
            <channel definition>
            <gate in definition>
            <agent definition>
            <agent reference>
            <textual typebased agent definition> }*
        [<state partitioning>]
        |
        {
            <entity in agent>
            <gate in definition> }*
        <agent type body> }

<type preamble> ::=
    [ <virtuality> | <abstract> ]

<agent type additional heading> ::=
    [<formal context parameters>] [<virtuality constraint>]
    <agent additional heading>

<agent type reference> ::=
    <system type reference>
    |
    <block type reference>
    |
    <process type reference>

<agent type body> ::=
    [ [<on exception>] <start> ]
    { <state> | <exception handler> | <free action> } *

```

### Grammaire graphique concrète

```

<agent type diagram> ::=
    <system type diagram> | <block type diagram> | <process type diagram>

<type in agent area> ::=
    <agent type diagram>
    |
    <agent type reference area>
    |
    <composite state type diagram>
    |
    <composite state type reference area>
    |
    <procedure diagram>
    |
    <procedure reference area>
    |
    <data type reference area>
    |
    <signal reference area>
    |
    <association area>

<agent type diagram content> ::=
    {
        <agent text area>*
        <type in agent area>*
        { <interaction area> | <agent type body area> } } set

<agent type body area> ::=
    <type state machine graph area>

<type state machine graph area> ::=
    {
        [<on exception association area>] [<start area>]
        { <state area> | <exception handler area> | <in connector area> }* } set

<agent type reference area> ::=
    {
        <system type reference area>
        |
        <block type reference area>
        |
        <process type reference area> }
    is connected to { <gate property area>* } set

```

La zone <package use area> doit être placée en tête du symbole <frame symbol>.

S'il existe une zone <agent type reference area> pour l'agent défini par un diagramme <agent type diagram>, les zones <gate property area> associées à la zone <agent type reference area> correspondent aux diagrammes <gate on diagram> associés au diagramme <agent type diagram>.

Aucune zone <gate property area> associée avec une zone <agent type reference area> ne doit contenir des <signal list item> qui ne sont pas contenus dans les diagrammes <gate on diagram> correspondants associés avec le diagramme <agent type diagram>.

### Sémantique

Une *Agent-type-definition* définit un type d'agent. Tous les agents d'un type de système donné ont les mêmes propriétés que celles qui sont définies pour ce type d'agent.

Les signaux mentionnés dans les sorties <output> de la machine à états d'un type d'agent doivent faire partie de l'ensemble complet des signaux d'entrée valides du type d'agent ou de la liste <signal list> à l'accès partant du type d'agent.

La définition d'un type d'agent implique la définition d'une interface dans la même portée du type d'agent (voir 12.1.2). La sorte de pid implicitement définie par cette interface est identifiée avec le *Agent-type-name* et elle est visible dans la même unité de portée que celle dans laquelle le type d'agent est défini.

NOTE – Etant donné que chaque type d'agent possède une interface définie implicitement avec le même nom, le type d'agent doit toujours avoir un nom différent de chaque interface définie explicitement et de chacun des agents (lesquels possèdent également des interfaces implicites) définis dans le même domaine de visibilité, afin d'éviter des collisions de noms.

Les propriétés définies dans une *Agent-type-definition* telle que l'ensemble *Procedure-definition-set*, l'ensemble *Agent-definition-set*, et l'ensemble *Gate-definition-set* détermine les propriétés de toute *Agent-definition* fondée sur le type et donc décrite au paragraphe 9.

### Modèle

Un agent possédant un corps <agent type body> ou une zone <agent type body area> est une abréviation pour un type d'agent ayant seulement une machine à états mais pas d'agents contenus. La machine à états est obtenue en remplaçant le corps <agent type body> ou la zone <agent type body area> par une définition d'état composite. Cette définition d'état composite a le même nom que le type d'agent et son graphe *State-transition-graph* est représenté par le corps <agent type body> ou par la zone <agent type body area>.

#### 8.1.1.2 Type de système

Une définition de type de système est une définition de type d'agent de niveau supérieur. Elle est désignée par le mot clé **system type**. Une définition de type de système ne doit pas être incluse dans une autre définition d'agent ou de type d'agent. Un type de système ne peut être ni abstrait ni virtuel.

#### Grammaire textuelle concrète

```
<system type definition> ::=
    <package use clause>*
    <system type heading> <end> <agent type structure>
    endsystem type [ [<qualifier>] <system type name>] <end>

<system type heading> ::=
    system type [<qualifier>] <system type name>
    <agent type additional heading>

<system type reference> ::=
    system type <system type identifier> <type reference properties>
```

Un paramètre <formal context parameter> de paramètres <formal context parameters> ne doit pas être un paramètre <agent context parameter>, <variable context parameter> ni un paramètre <timer context parameter>.

L'en-tête <agent type additional heading> dans une définition <system type definition> ou un diagramme <system type diagram> peut ne pas inclure des paramètres <agent formal parameters>.

## Grammaire graphique concrète

<system type diagram> ::=  
    <frame symbol> **contains** {<system type heading> <agent type diagram content> }  
    [ **is associated with** <package use area> ]

<system type reference area> ::=  
    <type reference area>

<system type symbol> ::=  
    <block type symbol>

La zone <type reference area> qui fait partie d'une zone <system type reference area> doit avoir un en-tête <graphical type reference heading> qui contient un nom <system type name>.

## Sémantique

Une définition <system type definition> définit un type de système.

### 8.1.1.3 Type de bloc

#### Grammaire textuelle concrète

<block type definition> ::=  
    <package use clause>\*  
    <block type heading> <end> <agent type structure>  
    **endblock type** [ [<qualifier>] <block type name> ] <end>

<block type heading> ::=  
    <type preamble>  
    **block type** [<qualifier>] <block type name>  
    <agent type additional heading>

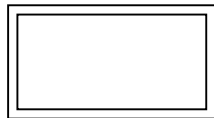
<block type reference> ::=  
    <type preamble>  
    **block type** <block type identifier> <type reference properties>

#### Grammaire graphique concrète

<block type diagram> ::=  
    <frame symbol> **contains** {<block type heading> <agent type diagram content> }  
    **is connected to** { { <gate on diagram>\* } **set** }  
    [ **is associated with** <package use area> ]

<block type reference area> ::=  
    <type reference area>

<block type symbol> ::=



La zone <type reference area> qui fait partie d'une zone <block type reference area> doit avoir un en-tête <graphical type reference heading> qui contient un nom <block type name>.

Le diagramme <gate on diagram> dans un diagramme <block type diagram> peut ne pas inclure d'identificateurs <external channel identifiers>.

Les diagrammes <gate on diagram> dans un diagramme <block type diagram> doivent être à l'extérieur du cadre du diagramme.

## Sémantique

Une définition <block type definition> définit un type de bloc.

### 8.1.1.4 Type de processus

#### Grammaire textuelle concrète

<process type definition> ::=  
    <package use clause>\*  
    <process type heading> <end> <agent type structure>  
    **endprocess type** [ [<qualifier>] <process type name>] <end>

<process type heading> ::=  
    <type preamble>  
    **process type** [<qualifier>] <process type name>  
    <agent type additional heading>

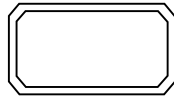
<process type reference> ::=  
    <type preamble>  
    **process type** <process type identifier> <type reference properties>

#### Grammaire graphique concrète

<process type diagram> ::=  
    <frame symbol> **contains** {<process type heading> <agent type diagram content> }  
    **is connected to** { {<gate on diagram>\* } **set** }  
    [ **is associated with** <package use area> ]

<process type reference area> ::=  
    <type reference area>

<process type symbol> ::=



La zone <type reference area> qui fait partie d'une zone <process type reference area> doit avoir un en-tête <graphical type reference heading> qui contient un nom <process type name>.

#### Sémantique

Une définition <process type definition> définit un type de processus.

Le diagramme <gate on diagram> dans un diagramme <process type diagram> peut ne peut inclure d'identificateurs <external channel identifiers>.

Le diagramme <gate on diagram> dans un diagramme <process type diagram> doit être à l'extérieur du cadre du diagramme.

### 8.1.1.5 Type d'état composite

#### Grammaire abstraite

*Composite-state-type-definition* ::= *State-type-name*  
    [ *Composite-state-type-identifier* ]  
    *Composite-state-formal-parameter*\*  
    *State-entry-point-definition-set*  
    *State-exit-point-definition-set*  
    *Gate-definition-set*  
    *Connect-node-set*  
    *Data-type-definition-set*  
    *Syntype-definition-set*  
    *Exception-definition-set*  
    *Composite-state-type-definition-set*  
    *Variable-definition-set*  
    *Procedure-definition-set*  
    [ *Composite-state-graph* | *State-aggregation-node* ]

*Composite-state-type-identifier* ::= *Identifier*

L'ensemble *Gate-definition-set* doit être vide sauf si l'état composite est utilisé en tant que définition *State-machine-definition*.

### Grammaire textuelle concrète

```

<composite state type definition> ::=
    {<package use clause>}*
    <composite state type heading> <end> substructure
    [<valid input signal set>]
    {<gate in definition>}*
    {<state connection points>}*
    {<entity in composite state>}*
    <composite state body>
endsubstructure state type [ [ <qualifier> ] <composite state type name> ] <end>
|
    {<package use clause>}*
    <state aggregation type heading> <end> substructure
    [<valid input signal set>]
    {<gate in definition>}*
    {<state connection points>}*
    {<entity in composite state>}*
    <state aggregation body>
endsubstructure state type [ [ <qualifier> ] <composite state type name> ] <end>

<composite state type heading> ::=
    [<virtuality>]
    state type [ <qualifier> ] <composite state type name>
    [<formal context parameters>] [<virtuality constraint>]
    [<specialization>]
    [<agent formal parameters>]

<state aggregation type heading> ::=
    [<virtuality>]
    state aggregation type [ <qualifier> ] <composite state type name>
    [<formal context parameters>] [<virtuality constraint>]
    [<specialization>]
    [<agent formal parameters>]

<composite state type reference> ::=
    <type preamble>
    state type <composite state type identifier> <type reference properties>

```

### Grammaire graphique concrète

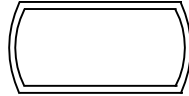
```

<composite state type diagram> ::=
    <frame symbol>
    contains {
        <composite state type heading>
        {
            {<composite state text area>}*
            {<type in composite state area>}*
            <composite state body area> }set }
    is associated with { <graphical state connection point>* }set
    is connected to { { <gate on diagram>* }set }
    [ is associated with <package use area> ]
|
    <frame symbol>
    contains {
        <state aggregation type heading>
        {
            {<composite state text area>}*
            {<type in composite state area>}*
            <state aggregation body area> }set }
    is associated with { <graphical state connection point>* }set
    is connected to { { <gate on diagram>* }set }
    [ is associated with <package use area> ]

<composite state type reference area> ::=
    <type reference area>
    is connected to { <gate property area>* }set

```

<composite state type symbol> ::=



La zone <package use area> doit être placée en tête du symbole <frame symbol>.

La zone <type reference area> qui fait partie d'une zone <composite state type reference area> doit avoir un en-tête <graphical type reference heading> qui contient un nom <composite state type name>.

Le diagramme <gate on diagram> dans un diagramme <composite state type diagram> peut ne pas inclure d'identificateurs <external channel identifiers>.

Les diagrammes <gate on diagram> dans un diagramme <composite state type diagram> doivent être à l'extérieur du cadre du diagramme

S'il existe une zone <composite state type reference area> pour un état composite défini par un diagramme <composite state type diagram>, les zones <gate property area> associées à la zone <composite state type reference area> correspondent aux diagrammes <gate on diagram> associés au diagramme <composite state type diagram>. Aucune zone <gate property area> associée à la zone <composite state type reference area> ne doit contenir d'éléments <signal list item> qui ne sont pas contenus dans le diagramme <gate on diagram> correspondant associé au diagramme <composite state type diagram>.

### Sémantique

Une définition *Composite-state-type-definition* définit un type d'état composite. Tous les états de composite d'un type d'état composite ont les mêmes propriétés que celles qui sont définies pour ce type d'état composite. La sémantique est définie en 11.11.

## 8.1.2 Expression de type

Une expression de type est utilisée pour définir un type en termes d'un autre type, comme défini par la spécialisation en 8.3.

### Grammaire textuelle concrète

<type expression> ::=

<base type> [<actual context parameters>]

<base type> ::=

<identifiant>

Les paramètres <actual context parameters> peuvent être spécifiés si et seulement si le type <base type> désigne un type paramétré. Les paramètres de contexte sont définis en 8.2.

A l'extérieur d'un type paramétré, celui-ci ne peut être utilisé que par référence à son identificateur <identifiant> dans l'expression <type expression>.

### Sémantique

Une expression <type expression> conduit à un type identifié par l'identificateur du type <base type> dans le cas où il n'existe pas de paramètres de contexte réels, ou à un type anonyme défini en appliquant les paramètres de contexte réels aux paramètres de contexte formels du type paramétré désigné par l'identificateur du type <base type>.

Si certains paramètres de contexte réels sont omis, le type reste paramétré.

En plus de satisfaire à toutes les conditions statiques de la définition désignée par le type <base type>, l'utilisation de l'expression <type expression> doit également satisfaire à toutes les conditions statiques sur le type résultant.



NOTE – L'utilisation d'une expression <type expression> peut violer les propriétés statiques dans l'un des cas suivants, par exemple:

- lorsqu'une unité de portée a des paramètres de contexte de signal ou des paramètres de contexte de temporisation, la condition qui constitue le stimulus d'un état donné doit être disjointe, selon les paramètres de contexte réels qui seront utilisés;
- lorsqu'une sortie dans une unité de portée fait référence à un accès, à un acheminement de signal ou à un canal, qui n'est pas défini dans le type ayant des accès le plus proche, l'instanciation de ce type entraîne une spécification erronée s'il n'existe pas de trajet de communication vers l'accès;
- lorsqu'une procédure contient des références à des identificateurs de signaux, des variables distantes et des procédures distantes, la spécialisation de cette procédure à l'intérieur d'un agent conduit à une spécification erronée si l'utilisation de tels identificateurs à l'intérieur de la procédure viole l'utilisation valide du processus;
- lorsque des types d'état sont instanciés en tant que parties du même agrégat d'états, l'état composite résultant est erroné si deux parties ou plus ont le même signal dans l'ensemble des signaux d'entrée;
- lorsqu'une unité de portée a un paramètre de contexte d'agent utilisé dans une action de sortie, l'existence d'un trajet de communication possible dépend du paramètre de contexte réel qui sera utilisé;
- lorsqu'une unité de portée a un paramètre de contexte de sorte, l'application du paramètre de contexte de sorte réel entraînera une spécification erronée si une attribution polymorphique sur une valeur est tentée dans le type spécialisé;
- si un paramètre formel d'une procédure ajoutée dans une spécialisation a un genre <parameter kind> "in/out" ou "out", un appel à un sous-type dans le supertype (en utilisant "this") conduira à un paramètre "in/out" ou "out" réel omis, c'est-à-dire à une spécification erronée;
- si un paramètre de contexte de procédure formel est défini dans une contrainte "atleast" et que le paramètre de contexte réel a ajouté un paramètre du genre <parameter kind> "in/out" ou "out", un appel au paramètre de contexte de procédure réel dans le type paramétré peut conduire à un paramètre "in/out" ou "out" réel omis, c'est-à-dire à une spécification erronée.

### *Modèle*

Si l'unité de portée contient <specialization> et si des paramètres <actual context parameter> quelconques sont omis dans l'expression <type expression>, les paramètres <formal context parameter> sont copiés (tout en conservant leur ordre) et insérés en tête des paramètres <formal context parameter> de l'unité de portée (s'il y en a). A la place des paramètres <actual context parameter>, les noms des paramètres <formal context parameter> correspondants sont insérés comme <actual context parameter>. Ces paramètres <actual context parameter> possèdent ainsi le contexte de définition de l'unité de portée courante.

### **8.1.3 Définitions fondées sur les types**

Une définition d'agent fondée sur le type définit un ensemble d'instances d'agents conformément au type désigné par l'expression <type expression>. Les entités définies possèdent les propriétés des types sur lesquelles elles sont fondées.

#### *Grammaire textuelle concrète*

```
<textual typebased agent definition> ::=  
    <textual typebased system definition>  
    |  
    <textual typebased block definition>  
    |  
    <textual typebased process definition>
```

Le type d'agent désigné par le type <base type> dans l'expression de type d'une définition <textual typebased agent definition> doit contenir une transition de départ non étiquetée dans sa machine à états.

### Grammaire graphique concrète

```
<graphical typebased agent definition> ::=
    <graphical typebased system definition>
    | <graphical typebased block definition>
    | <graphical typebased process definition>
<inherited agent definition> ::=
    <inherited block definition>
    | <inherited process definition>
```

Le type d'agent désigné par le type <base type> dans l'expression de type d'une définition <graphical typebased agent definition> ou d'une définition <inherited agent definition> doit contenir une transition de départ non étiquetée dans sa machine à états.

Dans une définition <graphical typebased agent definition>, les définitions <graphical gate definition> et les définitions <graphical interface gate definition> doivent être situées à l'extérieur du symbole <block symbol> ou du symbole <process symbol>.

#### 8.1.3.1 Définition de système fondée sur le type de système

##### Grammaire textuelle concrète

```
<textual typebased system definition> ::=
    <typebased system heading> <end>
<typebased system heading> ::=
    system <system name> <colon> <system type expression>
```

##### Grammaire graphique concrète

```
<graphical typebased system definition> ::=
    <block symbol> contains <typebased system heading>
```

##### Sémantique

Une définition de système fondée sur le type définit une définition *Agent-définition* de *Agent-kind* **SYSTEM** obtenue par transformation à partir d'un type de système.

##### Modèle

Une définition <textual typebased system definition> ou <graphical typebased system definition> est transformée en une définition <system definition> qui possède les définitions du type de système telles qu'elles sont définies par l'expression <system type expression>.

#### 8.1.3.2 Définition de bloc fondée sur le type de bloc

##### Grammaire textuelle concrète

```
<textual typebased block definition> ::=
    block <typebased block heading> <end>
<typebased block heading> ::=
    <block name> [<number of instances>] <colon> <block type expression>
```

##### Grammaire graphique concrète

```
<graphical typebased block definition> ::=
    <block symbol> contains { <typebased block heading> { <gate>* } set }
    is connected to { { <gate property area>* } set }
<inherited block definition> ::=
    <dashed block symbol> contains { <block identifier> { <gate>* } set }
    is connected to { { <gate property area>* } set }
```

<dashed block symbol> ::=



Les accès <gate> sont placés au voisinage de la frontière des symboles et associés avec le point de connexion aux canaux.

#### Sémantique

Une définition de bloc fondée sur le type définit des définitions *Agent-definitions* de *Agent-kind* **BLOCK** obtenues par transformation à partir d'un type de bloc.

#### Modèle

Une définition <textual typebased block definition> ou <graphical typebased block definition> est transformée en une définition <block definition> qui possède les définitions du type de bloc telles qu'elles sont définies par l'expression <block type expression>.

Une définition <inherited block definition> peut seulement apparaître dans une définition de sous-type. Elle représente le bloc défini dans le supertype de la définition du sous-type. Il est possible de spécifier des canaux supplémentaires connectés aux accès du bloc hérité.

### 8.1.3.3 Définition de processus fondée sur le type de processus

#### Grammaire textuelle concrète

<textual typebased process definition> ::=

**process** <typebased process heading> <end>

<typebased process heading> ::=

<process name> [<number of instances>] <colon> <process type expression>

#### Grammaire graphique concrète

<graphical typebased process definition> ::=

<process symbol> **contains** { <typebased process heading> { <gate>\* } **set** }  
**is connected to** { {<gate property area>\*} **set** }

<inherited process definition> ::=

<dashed process symbol> **contains** { <process identifier> { <gate>\* } **set** }  
**is connected to** { {<gate property area>\*} **set** }

<dashed process symbol> ::=



Les accès <gate> sont placés au voisinage de la frontière des symboles et associés avec le point de connexion aux canaux.

#### Sémantique

Une définition de processus fondée sur le type définit une définition *Agent-definition* avec *Agent-kind* **PROCESS** obtenue par transformation à partir d'un type de processus.

#### Modèle

Une définition <textual typebased process definition> ou <graphical typebased process definition> est transformée en une définition <process definition> qui possède les définitions du type de processus telles qu'elles sont définies par l'expression <process type expression>.

Une définition <inherited process definition> peut seulement apparaître dans une définition de sous-type. Elle représente le processus défini dans le supertype de la définition du sous-type. Il est possible de spécifier des canaux supplémentaires connectés aux accès du processus hérité.

### 8.1.3.4 Définition d'état composite fondée sur le type d'état composite

#### Grammaire textuelle concrète

<textual typebased state partition definition> ::=  
                  **state aggregation** <typebased state partition heading> <end>  
<typebased composite state> ::=  
          <state name> [<actual parameters>] <colon> <composite state type expression>

#### Sémantique

Une définition d'état composite fondée sur le type et une définition textuelle de partition d'état fondée sur le type définissent un état composite obtenu par transformation à partir d'un type d'état composite.

#### Modèle

Un état <typebased composite state> est transformé en un état <composite state> qui possède les définitions du type de l'état composite telles que définies par l'expression <composite state type expression>.

### 8.1.4 Type abstrait

#### Grammaire concrète

<abstract> ::=  
                  **abstract**

<abstract> fait partie de la définition de type. Voir 8.1.1.1, 8.1.5, 9.4, 10.3 et 12.1.1.

Un type est un type abstrait si sa définition contient <abstract>.

Une procédure abstraite ne peut pas être appelée.

Un agent fondé sur le type (voir 8.1.3) ne doit pas être spécifié en prenant pour type un type d'agent abstrait.

Un identificateur <operation identifier> ne doit pas représenter l'identificateur *Operation-identifier* pour l'opérateur Make défini pour un type de données abstrait.

Un identificateur <literal identifier> ne doit pas représenter un identificateur *Literal-identifier* désignant une signature *Literal-signature* d'un type de données abstrait.

Un type abstrait ne peut pas être instancié. Toutefois, un sous-type de type abstrait de données peut être instancié, s'il n'est pas lui-même abstrait.

### 8.1.5 Références de type

Les diagrammes de type et les définitions de type d'entité peuvent avoir des références de type. Une référence de type spécifie qu'un type est défini dans l'unité de portée de la définition ou du diagramme qui le contient (mais qui est entièrement décrit dans la définition référencée ou le diagramme référencé) et que les propriétés de ce type sont partiellement spécifiées dans la référence de type. La définition ou le diagramme référencé définit les propriétés du type alors que les références de type ne sont que des définitions partielles. Il est nécessaire que la spécification partielle faisant partie de la référence de type corresponde aux spécifications de la définition ou du diagramme de type. Une spécification partielle de variable, par exemple, peut fournir le nom de la variable mais pas sa sorte. Il doit exister une variable répondant à ce nom dans la définition référencée et, dans cette définition, une sorte doit être spécifiée.

La même définition de type peut avoir plusieurs références de type. Les références sans qualificateur doivent toutes se trouver dans la même unité de portée et la définition de type est insérée dans cette unité de portée.

## Grammaire textuelle concrète

```
<type reference properties> ::=  
    [ with { <attribute property> <end> }+ ]  
    [ with { <behaviour property> <end> }+ ]  
    referenced <end>
```

Si les propriétés <type reference properties> apparaissent dans les zones <package text area>, <agent text area>, <procedure text area>, <composite state text area>, et <operation text area>, ni les propriétés <attribute property> ni les propriétés <behaviour property> ne doivent être présentes.

```
<attribute property> ::=  
    <variable property>  
    | <field property>  
    | <signal parameter property>  
    | <interface variable property>
```

Chaque propriété <attribute property> fournit une spécification partielle des propriétés des variables ou des champs définis dans la définition de type qui est référencée par la référence de type. Les éléments de la propriété <attribute property> doivent être cohérents avec les propriétés correspondantes dans la définition de type référencée.

```
<variable property> ::=  
    [ <local> | <exported> ] <variable name> [<sort>]
```

```
<local> ::=  
    <hyphen>
```

```
<exported> ::=  
    <plus sign>
```

Une propriété <variable property> correspond à une définition <variable definition> dans un type d'agent, une procédure ou un type d'état composite. <local > indique une variable locale; <exported> indique une variable exportée. Le nom <variable name> et la sorte <sort>, s'ils sont présents, doivent être les mêmes que ceux présents dans la définition de variable correspondante.

```
<field property> ::=  
    [<symbolic visibility>] <field name> [<sort>]
```

```
<symbolic visibility> ::=  
    <private>  
    | <public>  
    | <protected>
```

```
<private> ::=  
    <hyphen> | private
```

```
<public> ::=  
    <plus sign> | public
```

```
<protected> ::=  
    <number sign> | protected
```

Une propriété <field property> correspond à un champ <field> dans un type de données. <private> (<public >, <protected>) correspond à une visibilité <visibility> privée (publique, protégée) dans le champ correspondant. Le nom <field name> et la sorte <sort>, s'ils sont présents, doivent être les mêmes que ceux présents dans la définition de champ correspondante.

```
<signal parameter property> ::=  
    <sort list>
```

Une propriété <signal parameter property> correspond à une liste de paramètres de signal dans une définition de signal. La liste <sort list> doit correspondre à la liste <sort list> dans la définition <signal definition item> de la définition de signal correspondante. Il ne doit y avoir qu'une seule propriété <signal parameter property> dans les propriétés <type reference properties> ou dans la zone <type reference area>.

<interface variable property> ::=  
     <remote variable name> [<sort>]

Une propriété <interface variable property> correspond à une définition <interface variable definition> dans une interface. La sorte <sort> doit être la même que la sorte <sort> dans la définition de variable d'interface.

<behaviour property> ::=  
     { [**operator**] <operation property> }  
     | { [**method**] <operation property> }  
     | { [**procedure**] <procedure property> }  
     | { [**signal**] <signal property> }  
     | { [**exception**] <exception property> }  
     | { [**timer**] <timer property> }  
     | { <interface use list> }

Une propriété <behaviour property> fournit une spécification partielle des propriétés des procédures et des opérations définies dans la définition de type qui est référencée par la référence de type. Cette spécification doit être cohérente avec les définitions correspondantes dans la définition de type correspondante.

<operation property> ::=  
     [<symbolic visibility>] <operation name>  
     <procedure signature>

Une propriété <operation property> correspond à une définition <operation definition> dans une référence de type d'objet ou de valeur. <private> (<public>, <protected>) correspond à une visibilité <visibility> privée (public, protected) dans la définition correspondante d'opérations. La liste des paramètres <formal parameter>, le résultat <result> et les <raises> dans la signature <procedure signature>, s'ils sont présents, doivent être respectivement les mêmes que les paramètres <formal parameter>, le résultat <result> et les <raises> dans la définition d'opération correspondante.

<procedure property> ::=  
     [ <local> | <exported> ] <procedure name>  
     <procedure signature>

Une propriété <procedure property> dans une référence de type d'agent correspond à une définition <procedure definition> dans le type d'agent. <local > indique une procédure locale; <exported > indique une procédure exportée. La liste des paramètres <formal parameter>, le résultat <result> et les <raises> dans la signature <procedure signature>, s'ils sont présents, doivent être respectivement les mêmes que les paramètres <procedure formal parameter>, le résultat <procedure result> et les <raises> dans la définition de procédure correspondante.

Une propriété <procedure property> dans une référence d'interface correspond à une définition <interface procedure definition> dans une interface. <local> ne doit pas être présent dans une référence d'interface. La signature <procedure signature>, si elle est présente, doit être la même que celle présente dans la définition de procédure d'interface correspondante.

<signal property> ::=  
     <signal name> [<sort list>]

Une propriété <signal property> dans une référence de type d'agent correspond à un signal traité comme une entrée dans le type d'agent.

Une propriété de signal dans une référence d'interface correspond à un élément <signal definition item> dans une définition <signal definition> dans une définition <interface definition>. La liste <sort list>, si elle est présente, doit être la même que celle présente dans l'élément <signal definition item> correspondant.

<exception property> ::=  
     <exception name> [<sort list>]

Une propriété <exception property> dans une référence de type correspond à un élément <exception definition item> dans la définition de type référencée par la référence de type. La liste <sort list>, si elle est présente, doit être la même que celle présente dans l'élément <exception definition item> correspondant.

```
<timer property> ::=
    <timer name> [<sort list>]
```

Une propriété <timer property> dans une référence de type correspond à un élément <timer definition item> dans la définition de type référencée par la référence de type. La liste <sort list>, si elle est présente, doit être la même que celle présente dans l'élément <timer definition item> correspondant.

Une liste <interface use list> correspond à une liste <interface use list> de la définition d'interface référencée par la référence de type. Chaque élément <signal list item> doit correspondre à un élément <signal list item> dans la liste <interface use list> de la définition d'interface référencée.

### Grammaire graphique concrète

```
<type reference area> ::=
    { <basic type reference area> | <iconized type reference area> }
    is connected to { <graphical package use area>* <specialization area>* } set
```

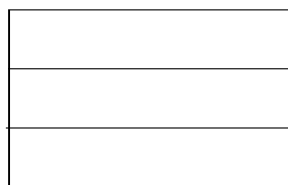
La zone <graphical package use area> pour une zone <type reference area> est une spécification partielle de la clause <package use clause> correspondante pour le diagramme de type et doit correspondre à celui-ci.

La zone <specialization area> doit être reliée avec la partie supérieure de la zone <basic type reference area> ou de la zone <iconized type reference area>, en utilisant l'extrémité du symbole <specialization relation symbol> qui ne possède pas de flèche. Il ne doit y avoir qu'une seule zone <specialization area> pour toutes les zones <type reference area>, excepté pour une référence d'interface.

La zone <specialization area> correspond à la <specialization> du type référencé. La référence <type reference area> associée doit correspondre au type <base type> dans l'expression <type expression> de la <specialization>. Les paramètres <actual context parameters> dans la zone <specialization area> doivent correspondre aux paramètres <actual context parameters> dans l'expression <type expression>.

```
<basic type reference area> ::=
    <class symbol>
    contains {
        <graphical type reference heading>
        <attribute properties area>
        <behaviour properties area> }
```

```
<class symbol> ::=
```



La position relative des deux lignes divisant le symbole <class symbol> en trois rectangles peut être différente de celle qui est représentée.

```

<graphical type reference heading> ::=
    { <type reference kind symbol> contains system | <system type symbol> }
    | { <system type type reference heading>
    | { <type reference kind symbol> contains block | <block type symbol> }
    | { <block type type reference heading>
    | { <type reference kind symbol> contains process | <process type symbol> }
    | { <process type type reference heading>
    | { <type reference kind symbol> contains state | <composite state type symbol> }
    | { <composite state type type reference heading>
    | { <type reference kind symbol> contains procedure | <procedure symbol> }
    | { <procedure type reference heading>
    | <type reference kind symbol> contains signal
    | <signal type reference heading>
    | { <type reference kind symbol> contains { value | object } | <data symbol> }
    | { <data type type reference heading>
    | { <type reference kind symbol> contains interface | <data symbol> }
    | { <interface type reference heading>

```

L'en-tête <graphical type reference heading> doit être placé dans le rectangle supérieur du symbole <class symbol> qui le contient.

```

<type reference heading> ::=
    <type preamble> [<qualifier>] <name> [<formal context parameters>]

```

Le préambule <type preamble> doit correspondre au préambule <type preamble> du type référencé, si la référence est virtuelle, le type référencé doit être virtuel. Si la référence est abstraite, le type référencé doit être abstrait.

Le paramètre <formal context parameters> correspond au paramètre <formal context parameters> du type référencé. La liste <formal context parameter list> doit correspondre à la liste <formal context parameter list> du type référencé.

```

<type reference kind symbol> ::=
    «    »

```

Le symbole <type reference kind symbol> est placé au-dessus ou à gauche du symbole <type reference heading>.

```

<data symbol> ::=

```

NOTE 1 – Le symbole <data symbol> est un rectangle sans aucun cadre visible. Ce qui signifie qu'un en-tête <graphical type reference heading> ne contenant pas de symbole <type reference kind symbol> contient en fait un symbole <data symbol>.

Le symbole <data symbol> correspond à une définition <data type definition> ou <interface definition>.

Si l'en-tête <graphical type reference heading> contient un symbole différent d'un symbole <type reference kind symbol>, celui-ci doit être placé dans le coin supérieur droit de l'en-tête <graphical type reference heading>.

```

<iconized type reference area> ::=
    <system type symbol> contains <system type type reference heading>
    | <block type symbol> contains <block type type reference heading>
    | <process type symbol> contains <process type type reference heading>
    | <composite state type symbol> contains <composite state type type reference heading>
    | <procedure symbol> contains <procedure type reference heading>

```

NOTE 2 – Il n'y a pas de zone <iconized type reference area> correspondant à des signaux, des interfaces, ni à des types d'objet ou de valeur

```

<attribute properties area> ::=
    { { <attribute property> <end> }* } set

```



La première propriété <attribute property> dans une zone <attribute properties area> doit être placée le plus haut possible dans le rectangle du milieu du symbole <class symbol> conteneur. Chaque propriété <attribute property> suivante doit être placée au-dessous de la précédente.

```
<behaviour properties area> ::=
    { { <behaviour property> <end> }* }set
```

La première propriété <behaviour property> dans une zone <behaviour properties area> doit être placée le plus haut possible dans le rectangle inférieur du symbole <class symbol> conteneur. Chaque propriété <behaviour property> suivante doit être placée au-dessous de la précédente.

### Modèle

Un en-tête <type reference heading> sans <qualifier> placé avant le nom <name> est une syntaxe dérivée dans laquelle l'entité identifiée par <qualifier> est le contexte englobant.

On considère qu'une référence de type dans laquelle l'entité identifiée par le <qualifier> de l'en-tête <type reference heading> est différente du contexte englobant est déplacée au contexte fourni par le qualificateur et, par conséquent, les règles de visibilité de ce dernier contexte s'appliquent.

Plusieurs références de type présentes dans le même contexte qui se rapportent à la même classe d'entité et qui ont le même qualificateur et le même nom sont équivalentes à une référence de type unique obtenue de ce contexte et pourvue de tous les éléments <attribute property> et <behaviour property> de toutes ces références.

Après réduction des références de type multiples, la référence de type, dans laquelle le <qualifier> de l'en-tête <type reference heading> est le même que celui du contexte englobant, est remplacée par le type référencé tel que défini au 7.3.

NOTE 3 – Le modèle pour les références de type dans lesquelles l'entité identifiée par le <qualifier> de l'en-tête <type reference heading> est différente du contexte englobant, signifie que le type référencé peut être un type autrement invisible dans la portée directement dans le contexte englobant.

### 8.1.6 Accès

Les accès sont définis dans les types d'agent (types de bloc, types de processus) ou dans les types d'états et représentent des points de connexion pour les canaux qui relient les instances de ces types (comme définis en 8.1.3) aux autres instances ou au symbole de cadre englobant.

Il est également possible de définir des accès dans des agents et des états composites et ceci correspond à une notation permettant de spécifier que l'entité considérée possède un point de connexion nommé.

#### Grammaire abstraite

```
Gate-definition          ::=  Gate-name
                             In-signal-identifier-set
                             Out-signal-identifier-set
Gate-name                =    Name
In-signal-identifier     =    Signal-identifier
Out-signal-identifier    =    Signal-identifier
```

#### Grammaire textuelle concrète

```
<gate in definition> ::=
    <textual gate definition> | <textual interface gate definition>
<textual gate definition> ::=
    gate <gate> [adding] <gate constraint> [ <gate constraint> ] <end>
<gate> ::=
    <gate name>
```

<gate constraint> ::=  
                                   { **out** [to <textual endpoint constraint>] | **in** [from <textual endpoint constraint>] }  
                                   [ **with** <signal list> ]

<textual endpoint constraint> ::=  
                                   [**atleast**] <identifieur>

<textual interface gate definition> ::=  
                                   **gate** { **in** | **out** } **with** <interface identifieur> <end>

**out** ou **in** indique la direction de la liste <signal list>: respectivement à partir du type ou vers celui-ci. Les types à partir desquels les instances sont définies doivent avoir une liste <signal list> contenue dans les contraintes <gate constraint>.

Un identificateur *In-signal-identifieur* représente un élément de la liste <signal list> vers l'accès. Un *Out-signal-identifieur* représente un élément de la liste <signal list> à partir de l'accès.

L'identificateur <identifieur> de la contrainte <textual endpoint constraint> doit indiquer une définition de type de la même classe d'entité que la définition du type dans laquelle l'accès est défini.

Un canal connecté à l'accès doit être compatible avec la contrainte de l'accès. Un canal est compatible avec la contrainte de l'accès si l'autre point extrémité du canal est un agent ou un état du type indiqué par l'identificateur <identifieur> dans la contrainte du point extrémité ou un sous-ensemble de ce type (s'il contient une contrainte <textual endpoint constraint> avec **atleast**) et si l'ensemble des signaux (s'il est spécifié) sur le canal est égal ou est inclus dans l'ensemble des signaux spécifié pour l'accès dans la direction respective.

Si le type désigné par <base type> dans une définition <textual typebased block definition> ou <textual typebased process definition> contient des canaux, la règle suivante s'applique: pour chaque combinaison (accès, signal, direction) définie par le type, celui-ci doit contenir au moins un canal mentionnant – pour la direction donnée – **env** et l'accès, il mentionne également le signal ou ne possède pas de liste <signal list> explicite associée. Dans ce dernier cas, il doit être possible d'en déduire que le canal peut acheminer le signal dans la direction donnée.

Si le type contient des canaux mentionnant des procédures distantes ou des variables distantes, une règle similaire s'applique

Lorsque deux contraintes <gate constraint> sont spécifiées, l'une doit être dans la direction opposée à l'autre et les contraintes <textual endpoint constraint> des deux contraintes <gate constraint> doivent être les mêmes.

**adding** peut seulement être spécifié dans une définition de sous-type et seulement pour un accès défini dans le supertype. Lorsque **adding** est spécifié pour un accès <gate>, toutes les contraintes <textual endpoint constraint> et les listes <signal list> sont des adjonctions aux contraintes <gate constraint> de l'accès dans le supertype.

Si la contrainte <textual endpoint constraint> est spécifiée pour un accès dans le supertype, l'identificateur <identifieur> d'une contrainte (ajoutée) <textual endpoint constraint> doit désigner le même type ou sous-type du type désigné dans la contrainte <textual endpoint constraint> du supertype.

L'identificateur <interface identifieur> d'une définition <textual interface gate definition> ne doit pas identifier l'interface qui est implicitement définie par l'entité à laquelle l'accès est connecté (voir 12.1.2).

*Grammaire graphique concrète*

<gate on diagram> ::=  
                                   <graphical gate definition> | <graphical interface gate definition>

<gate property area> ::=  
                                   <graphical gate definition> | <graphical interface gate definition>


```


<graphical gate definition> ::=
    { <gate symbol> | <inherited gate symbol> }
    is associated with { <gate> [ <signal list area> ] [ <signal list area> ] } set
    [ is connected to <endpoint constraint> ]

<endpoint constraint> ::=
    { <block symbol> | <process symbol> | <state symbol> }
    contains <textual endpoint constraint>

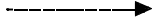
<graphical interface gate definition> ::=
    <gate symbol 1>
    is associated with <interface identifier>

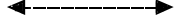
<gate symbol> ::=
    <gate symbol 1> | <gate symbol 2>

<gate symbol 1> ::=
    

<gate symbol 2> ::=
    

<inherited gate symbol> ::=
    <inherited gate symbol 1> | <inherited gate symbol 2>

<inherited gate symbol 1> ::=
    

<inherited gate symbol 2> ::=
    

```

Le diagramme <gate on diagram> se trouve à l'extérieur du cadre de diagramme.

Une définition <graphical gate definition> qui fait partie d'une zone <gate property area> ne doit pas contenir une contrainte <endpoint constraint>.

Les éléments de la zone <signal list area> sont associés aux directions du symbole d'accès.

Les zones <signal list area> et la contrainte <endpoint constraint> associées à un symbole <inherited gate symbol> sont considérées comme des adjonctions à celles de la définition de l'accès dans le supertype.

Un symbole <inherited gate symbol> ne peut apparaître que dans une définition de sous-type et représente un accès avec le même nom <gate name> spécifié dans le supertype de la définition du sous-type.

Il peut exister une zone <signal list area> pour chaque flèche sur le symbole <gate symbol>. Une zone <signal list area> doit être suffisamment proche de la flèche à laquelle elle est associée pour éviter toute ambiguïté. La flèche indique si la zone <signal list area> désigne une contrainte <gate constraint> **in** ou **out**.

### Sémantique

L'utilisation des accès dans les définitions de types correspond à l'utilisation des trajets de communication dans l'unité de portée englobante dans les spécifications d'instances (ou d'ensemble d'instances).

### Modèle

La définition <textual interface gate definition> et la définition <graphical interface gate definition> sont des abréviations respectivement pour une définition <textual gate definition> ou une définition <graphical gate definition>, elles ont le nom de l'interface comme nom <gate name> et l'identificateur <interface identifier> comme la contrainte <gate constraint> ou la zone <signal list area>.

## 8.2 Paramètres de contexte

Afin d'utiliser une définition de type dans différents contextes, à la fois à l'intérieur de la spécification d'un même système et des spécifications de systèmes différents, les types peuvent être paramétrés par des paramètres de contexte. Ces derniers sont remplacés par des paramètres de contexte réels, conformément en 8.1.2.

Les définitions de types suivantes peuvent avoir des paramètres de contexte formels: type de système, type de bloc, type de processus, procédure, signal, état composite, interface et type de données.

On peut fixer des contraintes pour les paramètres de contexte, c'est-à-dire des entités et des propriétés requises désignées par l'identificateur réel correspondant. Les paramètres de contexte à l'intérieur du type possèdent ces propriétés.

### *Grammaire textuelle concrète*

```
<formal context parameters> ::=
    <context parameters start> <formal context parameter list> <context parameters end>

<formal context parameter list> ::=
    <formal context parameter> {<end> <formal context parameter> }*

<actual context parameters> ::=
    <context parameters start> <actual context parameter list> <context parameters end>

<actual context parameter list> ::=
    [<actual context parameter>] {, [<actual context parameter> ] }*

<actual context parameter> ::=
    <identifiant> | <constant primary>

<context parameters start> ::=
    <less than sign>

<context parameters end> ::=
    <greater than sign>

<formal context parameter> ::=
    <agent type context parameter>
    | <agent context parameter>
    | <procedure context parameter>
    | <remote procedure context parameter>
    | <signal context parameter>
    | <variable context parameter>
    | <remote variable context parameter>
    | <timer context parameter>
    | <synonym context parameter>
    | <sort context parameter>
    | <exception context parameter>
    | <composite state type context parameter>
    | <gate context parameter>
    | <interface context parameter>
```

L'unité de portée d'une définition de type avec des paramètres de contexte formels définit les noms des paramètres de contexte formels. Ces noms sont alors visibles dans la définition du type et aussi dans la définition des paramètres de contexte formels.

Un paramètre <actual context parameter> ne doit pas être un primaire <constant primary> sauf s'il est utilisé pour paramètre de contexte synonyme. Un primaire <constant primary> est un primaire <primary> qui est une expression <constant expression> valable (voir 12.2.1).

Les paramètres de contexte formels ne peuvent être utilisés ni comme type <base type> dans l'expression <type expression> ni dans les contraintes **atleast** des paramètres <formal context parameters>.

Les contraintes sont spécifiées par des spécifications de contraintes. Une spécification de contrainte introduit l'entité du paramètre de contexte formel suivie soit par une signature de contrainte soit par une clause **atleast**. Une signature de contrainte introduit les propriétés directement suffisantes du paramètre de contexte formel. Une clause **atleast** indique que le paramètre de contexte formel doit être remplacé par un paramètre de contexte réel qui est du même type ou sous-type que le type identifié par la clause **atleast**. Les identificateurs suivants le mot clé **atleast** dans cette clause doivent identifier des définitions de type du genre d'entité du paramètre de contexte et ne doivent être ni des paramètres de contexte formels ni des types paramétrés.

Un paramètre de contexte formel d'un type donné doit être limité seulement à un paramètre de contexte réel du même genre d'entité qui satisfait aux contraintes du paramètre formel.

Le type paramétré peut seulement utiliser les propriétés d'un paramètre de contexte, données par la contrainte, à l'exception des cas énumérés en 8.1.2.

Un paramètre de contexte utilisant d'autres paramètres de contexte dans sa contrainte ne peut pas être limité avant les autres paramètres.

Les virgules de poursuite peuvent être omises dans les paramètres <actual context parameters>.

### *Sémantique*

Les paramètres de contexte formels d'une définition de type qui n'est ni une définition de sous-type ni définie par liaison des paramètres de contexte formels dans une expression <type expression> sont les paramètres spécifiés dans les paramètres <formal context parameters>.

Les paramètres de contexte d'un type sont limités dans la définition d'une expression <type expression> aux paramètres de contexte réels. Dans la liaison, les occurrences des paramètres de contexte formels à l'intérieur du type paramétré sont remplacées par les paramètres réels. Pendant la liaison des identificateurs contenus dans les paramètres <formal context parameter> aux définitions (c'est-à-dire la dérivation de leurs qualificatifs, voir 6.3), les autres définitions locales différentes des paramètres <formal context parameters> sont ignorées.

Les types paramétrés ne peuvent pas être des paramètres de contexte réels. Afin de permettre qu'une définition soit un paramètre de contexte réel, elle doit être du même genre d'entité que le paramètre formel et satisfaire à la contrainte du paramètre formel.

### *Modèle*

Si une unité de portée contient <specialization>, tout paramètre de contexte réel omis dans la <specialization> est remplacé par le paramètre correspondant <formal context parameter> du type <base type> dans l'expression <type expression> et ce paramètre <formal context parameter> devient un paramètre de contexte formel de l'unité de portée.

## **8.2.1 Paramètre de contexte de type d'agent**

### *Grammaire textuelle concrète*

<agent type context parameter> ::=  
          {**process type** | **block type**} <agent type name> [<agent type constraint>]

<agent type constraint> ::=  
          **atleast** <agent identifier> | <agent signature>

Un paramètre de type d'agent réel doit être un sous-type du type d'agent de contrainte (**atleast** <agent type identifier>) sans adjonction de paramètres formels aux paramètres du type de contrainte ou il doit être compatible avec la signature d'agent formel.

Une définition de type d'agent est compatible avec la signature d'agent formel si elle est du même genre et si les paramètres formels de la définition de type d'agent ont les mêmes sortes que les <sort> correspondantes des paramètres <agent signature>.

## 8.2.2 Paramètre de contexte d'agent

### Grammaire textuelle concrète

<agent context parameter> ::=  
    { **process** | **block** } <agent name> [<agent constraint>]  
  
<agent constraint> ::=  
    { **atleast** | <colon> } <agent type identifier> | <agent signature>  
  
<agent signature> ::=  
    <sort list>

Un paramètre d'agent réel doit identifier une définition d'agent. Son type doit être un sous-type du type de processus de contrainte (**atleast** <agent type identifier>) sans adjonction de paramètres formels aux paramètres du type de contrainte, ou il doit être le type désigné par l'identificateur <agent type identifier> ou il doit être compatible avec la signature <agent signature> formelle

Une définition d'agent est compatible avec la signature <agent signature> formelle si les paramètres formels de la définition de l'agent ont les mêmes sortes <sort> que la signature <agent signature> ou les paramètres <agent formal parameters> correspondants de la signature et si les deux définitions ont le même *Agent-kind*.

## 8.2.3 Paramètre de contexte de procédure

### Grammaire textuelle concrète

<procedure context parameter> ::=  
    **procedure** <procedure name> <procedure constraint>  
  
<procedure constraint> ::=  
    **atleast** <procedure identifier> | <procedure signature in constraint>  
  
<procedure signature in constraint> ::=  
    [ ( <formal parameter> { , <formal parameter> } \* ) ] [<result>]

Un paramètre de procédure réel doit identifier une définition de procédure qui est soit une spécialisation de la procédure de la contrainte (**atleast** <procedure identifier>) ou compatible avec la signature de procédure formelle.

Une définition de procédure est compatible avec la signature de procédure formelle si:

- a) les paramètres formels de la définition de la procédure ont les mêmes sortes que les paramètres correspondants de la signature, s'ils ont le même genre <parameter kind> et s'ils ont tous un résultat de la même sorte <sort> ou ne renvoient aucun résultat;
- b) chaque paramètre **in/out** et **out** de la définition de la procédure a le même identificateur <sort identifier> ou <syntype identifier> que le paramètre correspondant de la signature.

## 8.2.4 Paramètre de contexte de procédure distante

### Grammaire textuelle concrète

<remote procedure context parameter> ::=  
    **remote procedure** <procedure name> <procedure signature in constraint>

Un paramètre réel correspondant à un paramètre de contexte de procédure **remote** doit identifier une définition <remote procedure definition> ayant la même signature.

## 8.2.5 Paramètre de contexte de signal

### Grammaire textuelle concrète

<signal context parameter> ::=  
    **signal** <signal name> [<signal constraint>]  
    { , <signal name> [<signal constraint>] } \*

<signal constraint> ::=  
                                   **atleast** <signal identifier> | <signal signature>  
 <signal signature> ::=  
                                   <sort list>

Un paramètre de signal réel doit identifier une définition de signal qui est soit un sous-type du type de signal de la contrainte (**atleast** <signal identifier>) ou compatible avec la signature de signal formel.

### Sémantique

Une définition de signal est compatible avec une signature de signal formel si les sortes du signal sont les mêmes que les sortes de la liste de contraintes de sorte.

## 8.2.6 Paramètre de contexte de variable

### Grammaire textuelle concrète

<variable context parameter> ::=  
                                   **dcl** <variable name> { , <variable name> } \* <sort>  
   { , <variable name> { , <variable name> } \* <sort> } \*

Un paramètre réel doit être une variable ou un paramètre d'agent ou de procédure formel de la même sorte que la sorte de la contrainte.

## 8.2.7 Paramètre de contexte de variable distante

### Grammaire textuelle concrète

<remote variable context parameter> ::=  
                                   **remote** <remote variable name> { , <remote variable name> } \* <sort>  
   { , <remote variable name> { , <remote variable name> } \* <sort> } \*

Un paramètre réel doit identifier une définition <remote variable definition> de la même sorte.

## 8.2.8 Paramètre de contexte de temporisateur

### Grammaire textuelle concrète

<timer context parameter> ::=  
                                   **timer** <timer name> [<timer constraint>]  
   { , <timer name> [<timer constraint>] } \*  
 <timer constraint> ::=  
                                   <sort list>

Un paramètre réel de temporisateur doit identifier une définition de temporisateur compatible avec la liste de contraintes de sorte formelle. Une définition de temporisateur est compatible avec la liste de contraintes de sorte formelle si les sortes du temporisateur sont les mêmes sortes de la liste de contraintes de sorte.

## 8.2.9 Paramètre de contexte de synonyme

### Grammaire textuelle concrète

<synonym context parameter> ::=  
                                   **synonym** <synonym name> <synonym constraint>  
   { , <synonym name> <synonym constraint> } \*  
 <synonym constraint> ::=  
                                   <sort>

Un synonyme réel doit être une expression constante de la même sorte que la sorte de la contrainte.

## Modèle

Si le paramètre réel est une expression `<constant expression>` (plutôt qu'un identificateur `<synonym identifier>`), il existe une définition implicite d'un synonyme anonyme dans le contexte autour du type défini dans le paramètre de contexte.

### 8.2.10 Paramètre de contexte de sorte

#### Grammaire textuelle concrète

```
<sort context parameter> ::=
    [ { value | object } ] type <sort name> [<sort constraint>]

<sort constraint> ::=
    atleast <sort> | <sort signature>

<sort signature> ::=
    literals <literal signature> { , <literal signature> } *
    [ operators <operation signature in constraint> { , <operation signature in constraint> } * ]
    [ methods <operation signature in constraint> { , <operation signature in constraint> } * ]
    | operators <operation signature in constraint> { , <operation signature in constraint> } *
    | methods <operation signature in constraint> { , <operation signature in constraint> } *

<operation signature in constraint> ::=
    <operation name> [ ( <formal parameter> { , <formal parameter> } * ) ] [<result>]
    | <name class operation> [<result>]
```

Si la contrainte `<sort constraint>` est omise, la sorte réelle peut être n'importe quelle sorte. Autrement, une sorte réelle doit être soit un sous-type sans `<renaming>` de la sorte de la contrainte (**atleast** `<sort>`), ou compatible avec la signature de sorte formelle.

Une sorte est compatible avec la signature de sorte formelle si les littéraux de la sorte comprennent les littéraux de la signature de sorte formelle et les opérations définies par le type de données ayant introduit la sorte comprennent les opérations dans la signature de sorte formelle et ont la même signature.

La signature `<literal signature>` ne doit pas contenir de nombre `<named number>`.

## Modèle

Si le mot clé **value** est fourni et que la sorte réelle est une sorte d'objet, le paramètre réel est traité comme l'identificateur `<sort identifier>` de la **value** de la sorte étendue. Si le mot clé **object** est fourni et que la sorte réelle est une sorte de valeur, le paramètre réel est traité comme l'identificateur `<sort identifier>` de l'**object** de la sorte de référence.

### 8.2.11 Paramètre de contexte d'exception

#### Grammaire textuelle concrète

```
<exception context parameter> ::=
    exception <exception name> [<exception constraint>]
    { , <exception name> [<exception constraint>] } *

<exception constraint> ::=
    <sort list>
```

Un paramètre d'exception réel doit identifier une exception avec la même signature.

### 8.2.12 Paramètre de contexte de type d'état composite

#### Grammaire textuelle concrète

```
<composite state type context parameter> ::=
    state type <composite state type name> [<composite state type constraint>]
```



```

<composite state type constraint> ::=
    atleast <composite state type identifier> | <composite state type signature>

<composite state type signature> ::=
    <sort list>

```

Un paramètre de type d'état composite réel doit identifier une définition de type d'état composite. Son type doit être un sous-type du type d'état composite de contrainte (**atleast** <composite state type identifier>) sans adjonction de paramètres formels à ceux du type de contrainte ou il doit être compatible avec la signature de type d'état composite formel.

Une définition de type d'état composite est compatible avec la signature de type d'état composite formel si les paramètres formels de la définition de type d'état composite ont les mêmes sortes que les <sort> correspondantes de la contrainte <composite state type constraint>.

### 8.2.13 Paramètre de contexte d'accès

*Grammaire textuelle concrète*

```

<gate context parameter> ::=
    gate <gate> <gate constraint> [<gate constraint>]

```

Un paramètre d'accès réel doit identifier une définition d'accès. Sa contrainte d'accès vers l'extérieur contient tous les éléments mentionnés dans la liste <signal list> du paramètre de contexte d'accès formel correspondant. La contrainte d'accès formel vers l'intérieur doit contenir tous les éléments dans la liste <signal list> du paramètre d'accès réel.

### 8.2.14 Paramètre de contexte d'interface

*Grammaire textuelle concrète*

```

<interface context parameter> ::=
    interface <interface name> [<interface constraint>]
    { , <interface name> [<interface constraint>] } *

<interface constraint> ::=
    atleast <interface identifier>

```

Un paramètre d'interface réel doit identifier une définition d'interface. Le type de l'interface doit être un sous-type du type d'interface de la contrainte (**atleast** <interface identifier>).

## 8.3 Spécialisation

Un type peut être défini comme une spécialisation d'un autre type (le supertype), conduisant ainsi à un nouveau sous-type. Un sous-type peut avoir des propriétés supplémentaires aux propriétés du supertype et peut redéfinir des types et transitions locaux virtuels. A l'exception du cas d'interfaces, il y a un supertype au plus.

On peut imposer des contraintes aux types virtuels, c'est-à-dire, des propriétés de toute redéfinition que le type virtuel doit avoir. Ces propriétés sont utilisées pour garantir les propriétés de toute redéfinition. Les types virtuels sont définis en 8.3.2.

### 8.3.1 Adjonction de propriétés

*Grammaire textuelle concrète*

```

<specialization> ::=
    inherits <type expression> [adding]

```

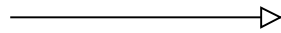
*Grammaire graphique concrète*

```

<specialization area> ::=
    <specialization relation symbol>
    [ is associated with <actual context parameters> ]
    is connected to <type reference area>

```

<specialization relation symbol> ::=



L'extrémité de la flèche du symbole <specialization relation symbol> pointe en direction de la référence <type reference area>. Le type relié à l'extrémité de la flèche est le supertype alors que l'autre type est le sous-type. Les références associées doivent être du même genre. La liaison associée de paramètres de contexte signifie que le supertype est une expression de type avec des paramètres de contexte réels.

L'expression <type expression> désigne le type de base. Le type de base est appelé supertype du type spécialisé, et le type spécialisé est appelé sous-type du type général. Toute spécialisation du sous-type est un sous-type du type général.

Si un type subT est un sous-type d'un (super) type T par une spécialisation (directement ou indirectement), alors:

- a) T ne doit pas englober subT;
- b) T ne doit pas être une spécialisation de subT;
- c) les définitions englobées par T ne doivent pas être des spécialisations de subT.

En cas de types d'agent, ces règles doivent également s'appliquer pour les définitions englobées dans T et, de plus, les définitions englobées directement ou indirectement par T ne doivent pas être des définitions fondées sur le type de subT.

La syntaxe concrète pour la spécialisation de types de données est présentée en 12.1.3.

### *Sémantique*

Le contenu qui résulte de la définition d'un type spécialisé avec des définitions locales est composé du contenu du supertype suivi par le contenu de la définition spécialisée. Cela implique que l'ensemble des définitions de la définition spécialisée est l'union des définitions données dans la définition spécialisée elle-même et celles du supertype. L'ensemble résultant des définitions doit obéir aux règles concernant les noms distincts, conformément au 6.3. Il existe toutefois trois exceptions à ces règles; il s'agit des cas suivants:

- a) une redéfinition d'un type virtuel est une définition avec le même nom que celui du type virtuel;
- b) un accès du supertype peut avoir une définition étendue (en termes de signaux acheminés et de contraintes de point d'extrémité) dans un sous-type; cela est spécifié par une définition d'accès avec le même nom que celui du supertype;
- c) si l'expression <type expression> contient des paramètres <actual context parameters>, toute occurrence du type <base type> de l'expression <type expression> est remplacée par le nom du sous-type;
- d) un opérateur du supertype n'est pas hérité si la signature de l'opérateur spécialisé diffère de celle de l'opérateur du type de base;
- e) un opérateur ou une méthode non virtuelle (c'est-à-dire une méthode qui n'est ni virtuelle ni redéfinie) du supertype n'est pas hérité(e) si un opérateur ou une méthode ayant une signature égale à celle de l'opérateur spécialisé ou de la méthode spécialisée est déjà présent(e) dans le sous-type.

Les paramètres de contexte formels du sous-type sont les paramètres de contexte formels non limités de la définition du supertype suivis par les paramètres de contexte formels du type spécialisé (voir 8.2).

Les paramètres formels d'un type d'agent spécialisé sont les paramètres formels du supertype d'agent, suivis par les paramètres formels ajoutés dans la spécialisation.

Les paramètres formels d'une procédure spécialisée sont les paramètres formels de la procédure avec les paramètres formels ajoutés dans la spécialisation. Si la procédure précédant la spécialisation a un résultat <procedure result>, les paramètres ajoutés dans la spécialisation sont insérés avant le dernier paramètre (le paramètre **out** pour le résultat), autrement, ils sont insérés après le dernier paramètre.

L'ensemble complet de signaux d'entrée valides d'un type d'agent spécialisé est l'union de l'ensemble complet de signaux d'entrée valides du type d'agent spécialisé et de l'ensemble complet de signaux d'entrée valides du supertype d'agent respectivement.

Le graphe résultant d'un type d'agent spécialisé, d'une définition de procédure ou d'un type d'état se compose du graphe de la définition de son supertype suivi par le graphe du type d'agent spécialisé, de la définition de procédure spécialisée ou du type d'état spécialisé.

Le graphe de transition d'état d'un type d'agent donné, d'une définition de procédure donnée ou d'un type d'état donné doit avoir au plus une transition de départ non étiquetée.

Une définition de signal spécialisé peut ajouter (en attachant) des sortes à la liste de sortes du supertype.

Une définition de type de données spécialisée peut ajouter des littéraux, des champs ou des choix dans les constructeurs de type hérités, elle peut ajouter des opérateurs et des méthodes et elle peut ajouter des initialisations par défaut et des assignations par défaut.

Les paramètres formels d'un type d'état composite spécialisé sont les paramètres formels du type d'état composite avec les paramètres formels ajoutés dans la spécialisation.

NOTE – Lorsqu'un accès dans un sous-type est une extension d'un accès hérité d'un supertype, le symbole <inherited gate symbol> est utilisé dans la syntaxe graphique concrète.

### 8.3.2 Type virtuel

Un type d'agent, une procédure ou un type d'état peuvent être spécifiés en tant que types virtuels lorsqu'ils sont définis localement dans un autre type (désigné comme le type *enclosing* ou type englobant). Un type virtuel peut être redéfini dans des spécialisations du type englobant.

#### *Grammaire textuelle concrète*

<virtuality> ::=

**virtual** | **redefined** | **finalized**

<virtuality constraint> ::=

**atleast** <identifieur>

<virtuality> et la contrainte <virtuality constraint> font partie de la définition de type.

Un type virtuel est un type ayant **virtual** ou **redefined** comme <virtuality>. Un type redéfini est un type ayant **redefined** ou **finalized** comme <virtuality>. Seuls des types virtuels peuvent être redéfinis.

Tout type virtuel s'associe avec une contrainte virtuelle qui est un identificateur <identifieur> du même genre d'entité que le type virtuel. Si une contrainte <virtuality constraint> est spécifiée, la contrainte de virtualité est l'identificateur <identifieur> contenu; autrement, la contrainte de virtualité est dérivée comme décrit ci-dessous.

Un type virtuel et ses contraintes ne peuvent pas avoir de paramètres de contexte.

Seuls les types virtuels peuvent avoir une contrainte <virtuality constraint> spécifiée.

Si <virtuality> est présente à la fois dans la référence et dans la définition référencée, les deux doivent être égales. Si le préambule <procedure preamble> est présent à la fois dans la référence de procédure et dans la définition référencée, les deux doivent être égaux.

Un type d'agent virtuel doit avoir exactement les mêmes paramètres formels, au moins les mêmes accès et interfaces avec au moins les mêmes définitions que ceux de sa contrainte. Un type d'état

virtuel doit avoir au moins les mêmes points de connexion d'état que sa contrainte. Une procédure virtuelle doit avoir exactement les mêmes paramètres formels que sa contrainte. Les restrictions concernant les arguments des opérateurs virtuels sont présentées en 8.3.4.

Si **inherits** et **atleast** sont utilisés tous les deux, le type hérité doit alors être identique au, ou être un, sous-type de la contrainte.

Dans le cas d'une contrainte implicite, une redéfinition impliquant **inherits** doit être un sous-type de la contrainte.

### *Sémantique*

Un type virtuel peut être redéfini dans la définition d'un sous-type du type englobant du type virtuel. Dans le sous-type, c'est la définition à partir du sous-type qui définit le type d'instances du type virtuel, et aussi lorsque le type virtuel est appliqué dans des parties du sous-type héritées du supertype. Un type virtuel qui n'est pas redéfini dans une définition de sous-type a la définition donnée dans la définition du supertype.

L'accès à un type virtuel au moyen de qualificateur désignant un des supertypes implique toutefois l'application de la (re)définition du type virtuel donné dans le supertype réel désigné par le qualificateur. Un type T dont le nom est caché dans un sous-type englobant par une redéfinition de T peut être rendu visible à travers la qualification avec un nom de supertype (c'est-à-dire, un nom de type dans une chaîne d'héritage). Le qualificateur se composera d'un seul élément de trajet, désignant le supertype particulier.

Un type virtuel ou redéfini qui n'a pas de <specialization> fourni de manière explicite spécifiée peut avoir une <specialization> implicite. La contrainte de virtualité et la <specialization> implicite éventuelle sont dérivées comme suit.

Pour un type virtuel V et un type redéfini R de V, les règles suivantes s'appliquent (toutes les règles sont appliquées dans l'ordre donné):

- a) si le type virtuel V n'a pas de contrainte <virtuality constraint>, la contrainte du type V (contrainte VC) est identique au type virtuel V et désigne le type V, autrement, la contrainte VC est identifiée par la contrainte <virtuality constraint> donnée avec le type V;
- b) si le type virtuel V n'a pas de <specialization> et que la contrainte VC est le type V, le type V n'a pas de spécialisation implicite;
- c) si le type virtuel V n'a pas de <specialization> et que la contrainte VC n'est pas le type V, la spécialisation implicite du type V (spécialisation VS) est identique à la contrainte VC;
- d) si la <specialization> du type virtuel V existe, la spécialisation VS doit être identique à, ou être un sous-type de la contrainte VC;
- e) si le type redéfini R n'a pas de contrainte <virtuality constraint>, la contrainte du type R (contrainte RC) est identique au type R, autrement, la contrainte RC est identifiée par la contrainte <virtuality constraint> donnée avec le type R;
- f) si le type redéfini R n'a pas de <specialization>, la spécialisation implicite du type R (spécialisation RS) est identique à la contrainte VC issue du type V, autrement, la spécialisation RS est identifiée par la <specialization> explicite donnée avec le type R;
- g) la contrainte RC doit être identique à, ou être un sous-type de la contrainte VC;
- h) la spécialisation RS du type R doit être la même ou être un sous-type de la contrainte RC;
- i) si le type R est un type virtuel (redéfini plutôt que finalisé), les mêmes règles s'appliquent pour R comme pour V.

Un sous-type d'un type virtuel est un sous-type du type original et pas d'une redéfinition possible.

### 8.3.3 Transition/sauvegarde virtuelle

Les transitions ou les sauvegardes d'un type de processus, d'un type d'état ou d'une procédure sont spécifiées en tant que transitions ou sauvegardes virtuelles au moyen du mot clé **virtual**. Les transitions ou sauvegardes virtuelles peuvent être redéfinies dans les spécialisations. Cela est indiqué respectivement par des transitions et des sauvegardes ayant le même couple (état, signal) et par le mot clé **redefined** ou **finalized**.

#### *Grammaire textuelle concrète*

Les syntaxes des transitions et sauvegardes virtuelles sont définies aux 9.4 (départ de procédure virtuelle), 10.5 (entrée et sauvegarde virtuelles de procédure distante), 11.1 (départ de processus virtuel), 11.2.2 (entrée virtuelle), 11.4 (entrée prioritaire virtuelle), 11.5 (signal continu virtuel), 11.7 (sauvegarde virtuelle), 11.9 (transition spontanée virtuelle) et 11.16.3 (traitement virtuel).

Les transitions ou sauvegardes virtuelles ne doivent pas apparaître dans les définitions (d'ensemble d'instances) d'agent ou dans les définitions d'état composite.

Un état ne doit pas avoir plus d'une seule transition spontanée virtuelle.

Une redéfinition dans une spécialisation marquée après le mot clé **redefined** peut être définie de manière différente dans d'autres spécialisations, tandis qu'une redéfinition marquée par le mot clé **finalized** ne doit pas avoir de nouvelles définitions dans d'autres spécialisations.

Une entrée ou une sauvegarde avec <virtuality> ne doit pas contenir <asterisk>.

#### *Sémantique*

La redéfinition de transitions/sauvegardes virtuelles correspond étroitement à la redéfinition des types virtuels (voir 8.3.2).

Une transition de départ virtuelle peut être redéfinie en une nouvelle transition de départ.

Une entrée prioritaire ou une transition d'entrée virtuelle peut être redéfinie en une nouvelle entrée prioritaire, une nouvelle transition d'entrée ou une nouvelle sauvegarde.

Une sauvegarde virtuelle peut être redéfinie en une entrée prioritaire, une transition d'entrée ou une sauvegarde.

Une transition spontanée virtuelle peut être redéfinie en une nouvelle transition spontanée.

Une transition traitée virtuelle peut être redéfinie en une nouvelle transition traitée virtuelle.

Une transition continue virtuelle peut être redéfinie en une nouvelle transition continue. La redéfinition est indiquée par le même couple (état, priorité), s'il est présent, que la transition continue redéfinie. Si plusieurs transitions continues virtuelles existent dans un état, chacune d'elles doit avoir une priorité distincte. Si une seule transition continue virtuelle existe dans un état, la priorité peut être omise.

Une transition d'une transition d'entrée de procédure distante virtuelle peut être redéfinie en une nouvelle transition d'entrée de procédure distante ou en une sauvegarde de procédure distante.

Une sauvegarde de procédure distante virtuelle peut être redéfinie en une transition d'entrée de procédure distante ou en une sauvegarde de procédure distante.

La transformation des transitions d'entrée virtuelles s'applique également aux transitions d'entrée de procédure distante virtuelle.

La transformation des transitions et sauvegardes virtuelles dans les états astérisques est élaborée dans l'Annexe F.

### 8.3.4 Méthodes virtuelles

Des méthodes de type de données sont spécifiées pour être des méthodes virtuelles au moyen du mot clé **virtual** dans <virtuality>. Les méthodes virtuelles peuvent être redéfinies dans les spécialisations. Cela est indiqué par des méthodes ayant le même nom <operation name> et avec le mot clé **redefined** ou **finalized** dans <virtuality>.

Si le type dérivé contient uniquement une signature <operation signature> mais pas de définition <operation definition>, <textual operation reference> ou <external operation definition> pour la méthode redéfinie, la signature de la méthode redéfinie est modifiée.

#### *Grammaire textuelle concrète*

La syntaxe des méthodes virtuelles est définie en 12.1.4.

Lorsqu'une méthode est redéfinie dans une spécialisation, sa signature doit être compatible en termes de sorte avec la signature correspondante dans le type base type et, de plus, si le *Result* dans la signature *Operation-signature* désigne une sorte A, le *Result* de la méthode redéfinie ne peut désigner qu'une sorte B telle que la sorte B est compatible avec la sorte A.

La redéfinition d'une méthode virtuelle ne doit pas modifier le genre <parameter kind> dans un quelconque <argument> de la signature héritée <operation signature>.

La redéfinition d'une méthode virtuelle ne doit pas ajouter <argument virtuality> dans un quelconque <argument> de la signature héritée <operation signature>.

#### *Sémantique*

Les méthodes virtuelles n'ont pas de contrainte <virtuality constraint> qui, uniquement dans ce cas, ne limite pas la redéfinition.

La redéfinition de méthodes virtuelles correspond étroitement à la redéfinition de types virtuels (voir 8.3.2).

### 8.3.5 Initialisation virtuelle par défaut

Le présent sous-paragraphe décrit l'initialisation virtuelle par défaut, tel que définie au 12.3.3.2.

L'initialisation par défaut d'instances de type de données est spécifiée pour être virtuelle au moyen du mot clé **virtual** dans <virtuality>. Une initialisation virtuelle par défaut peut être redéfinie dans les spécialisations. Cela est indiqué par une initialisation par défaut avec le mot clé **redefined** ou **finalized** dans <virtuality>.

Si le type dérivé ne contient pas d'expression <constant expression> dans son initialisation par défaut, il n'a alors pas d'initialisation par défaut.

#### *Grammaire textuelle concrète*

La syntaxe des initialisations par défaut virtuelles a été définie au 12.3.3.2.

#### *Sémantique*

La redéfinition d'une initialisation par défaut virtuelle correspond étroitement à la redéfinition des types virtuels (voir 8.3.2).

## 8.4 Associations

Une association exprime une relation binaire entre deux types d'entité qui ne sont pas nécessairement distincts. Les associations sont destinées à fournir des annotations structurées pour indiquer des propriétés supplémentaires des types auxquels sont liées les associations, dans un diagramme ou une définition contenant des références de type. La signification de ces propriétés n'est pas définie dans la présente Recommandation; c'est-à-dire, la signification peut être définie par une autre

Recommandation, norme, spécification commune ou par le sens commun. Un système SDL qui contient une association a la même signification et le même comportement (tels que définis dans la présente Recommandation) si l'association est supprimée.

### Grammaire textuelle concrète

```

<association> ::=
    association [<association name>] <association kind> <end>

<association kind> ::=
    | <association not bound kind>
    | <association end bound kind>
    | <association two ends bound kind>
    | <composition not bound kind>
    | <composition part end bound kind>
    | <composition composite end bound kind>
    | <composition two ends bound kind>
    | <aggregation not bound kind>
    | <aggregation part end bound kind>
    | <aggregation aggregate end bound kind>
    | <aggregation two ends bound kind>

<association not bound kind> ::=
    from <association end> from <association end>

<association end bound kind> ::=
    from <association end> to <association end>

<association two ends bound kind> ::=
    to <association end> to <association end>

<composition not bound kind> ::=
    from <association end> composition <association end>

<composition part end bound kind> ::=
    to <association end> composition <association end>

<composition composite end bound kind> ::=
    from <association end> to composition <association end>

<composition two ends bound kind> ::=
    to <association end> to composition <association end>

<aggregation not bound kind> ::=
    from <association end> aggregation <association end>

<aggregation part end bound kind> ::=
    to <association end> aggregation <association end>

<aggregation aggregate end bound kind> ::=
    from <association end> to aggregation <association end>

<aggregation two ends bound kind> ::=
    to <association end> to aggregation <association end>

<association end> ::=
    [<visibility>] [as <role name>] <linked type identifier> [size <multiplicity>] [ordered]

<linked type identifier> ::=
    | <agent type identifier>
    | <data type identifier>
    | <interface identifier>

<multiplicity> ::=
    <range condition>

```

Il est permis de lier une <association> à des types d'agent, des interfaces ou des types de données.

Si une extrémité <association end> identifie un type d'agent ou une interface, la visibilité **protected** ne peut pas être utilisée dans l'autre extrémité <association end> de l'<association>.

Si deux <association> différentes identifient le même type, les noms <role name> (s'ils sont donnés) doivent être différents dans les extrémités <association end> opposées à ce type commun.

La sorte de base de la condition <range condition> dans <multiplicity> doit être la sorte prédéfinie naturelle.

Il ne doit pas y avoir d'ensemble d'<association> contenant une composition telle que le type est relié à lui-même de par sa composition, que se soit directement ou indirectement.

Si une extrémité <association end> est précédée du mot clé **composition** et identifie un type de données ou une interface, l'identificateur <linked type identifier> dans l'autre extrémité <association end> de l'<association> doit identifier respectivement un type de données ou une interface.

### Grammaire graphique concrète

```
<association area> ::=
    <association symbol>
    [ is associated with <association name> ]
    is connected to { <association end area> <association end area> } set
```

```
<association symbol> ::=
    | <association not bound symbol>
    | <association end bound symbol>
    | <association two ends bound symbol>
    | <composition not bound symbol>
    | <composition part end bound symbol>
    | <composition composite end bound symbol>
    | <composition two ends bound symbol>
    | <aggregation not bound symbol>
    | <aggregation part end bound symbol>
    | <aggregation aggregate end bound symbol>
    | <aggregation two ends bound symbol>
```

```
<association not bound symbol> ::=
    _____
```

```
<association end bound symbol> ::=
    _____>
```

```
<association two ends bound symbol> ::=
    <----->
```

```
<composition not bound symbol> ::=
    _____◆
```

```
<composition part end bound symbol> ::=
    <-----◆
```

```
<composition composite end bound symbol> ::=
    _____>◆
```

```
<composition two ends bound symbol> ::=
    <----->◆
```

```
<aggregation not bound symbol> ::=
    _____◇
```

```
<aggregation part end bound symbol> ::=
    <-----◇
```

```
<aggregation aggregate end bound symbol> ::=
    _____>◇
```

```
<aggregation two ends bound symbol> ::=
    <----->◇
```

```
<association end area> ::=
    <linked type reference area> is associated with
    { [<role name>] [<multiplicity>] [<ordering area>] [<symbolic visibility>] } set
```



<ordering area> ::=

**ordered**

<linked type reference area> ::=

<agent type reference area>  
| <data type reference area>  
| <interface reference area>

Une zone <association area> reliant deux zones <linked type reference area> correspond à une <association> dans le diagramme englobant.

Si une zone <association end area> identifie un type d'agent ou une interface, la visibilité **protected** ne doit pas être utilisé dans l'autre zone <association end area> de la zone <association area>.

Si deux zones <association area> différentes identifient le même type, les noms <role name> (s'ils sont donnés) doivent être différents dans les zones <association end area> opposées à ce type commun.

Il ne doit pas y avoir d'ensemble de zones <association area> contenant une composition telle qu'un type soit relié à lui-même de par sa composition, que se soit directement ou indirectement.

Si l'extrémité de composite (l'extrémité avec un diamant) d'un symbole <composition not bound symbol>, <composition part end bound symbol>, <composition composite end bound symbol>, ou <composition two ends bound symbol> est reliée à une zone <linked type reference area> qui identifie un type de données ou une interface, la zone opposée <association end area> doit être reliée à une zone <linked type reference area> qui identifie respectivement un type de données ou une interface.

### Sémantique

Une association relie les deux types d'entité d'une manière qui n'est pas davantage définie dans la présente Recommandation.

## 9 Agents

Une définition d'agent définit un ensemble (arbitrairement grand) d'agents. Un agent se caractérise par des variables, des procédures, une machine à états (donnée par un type d'état composite explicite ou implicite) et des ensembles d'agents contenus.

Il existe deux sortes d'agents: les *blocs* et les *processus*. Un *système* est le bloc le plus à l'extérieur. La machine à états d'un bloc est interprétée *en parallèle* avec ses agents contenus, alors que la machine à états d'un processus est interprétée *en alternance* avec ses agents contenus.

### Grammaire abstraite

<i>Agent-definition</i>	::	<i>Agent-name</i> <i>Number-of-instances</i> <i>Agent-type-identifier</i>
<i>State-machine-definition</i>	::	<i>State-name</i> <i>Composite-state-type-identifier</i>
<i>Agent-kind</i>	=	<b>SYSTEM</b>   <b>BLOCK</b>   <b>PROCESS</b>
<i>Number-of-instances</i>	::	<i>Initial-number</i> [ <i>Maximum-number</i> ]
<i>Initial-number</i>	=	<i>Nat</i>
<i>Maximum-number</i>	=	<i>Nat</i>
<i>State-transition-graph</i>	::	[ <i>On-exception</i> ] <i>State-start-node</i> <i>State-node-set</i> <i>Free-action-set</i> <i>Exception-handler-node-set</i>
<i>Agent-formal-parameter</i>	=	<i>Parameter</i>

## Grammaire textuelle concrète

```
<agent definition> ::=
    <system definition> | <block definition> | <process definition>

<agent structure> ::=
    [<valid input signal set>]
    {
        {
            <entity in agent>
            | <channel to channel connection>
            | <channel definition>
            | <gate in definition>
            | <agent definition>
            | <agent reference>
            | <textual typebased agent definition> }*
        [<state partitioning>]
        | {
            <entity in agent>
            | <gate in definition> }*
        <agent body> }
    }
```

La division `<state partitioning>` doit avoir le même nom que l'agent contenant. Elle définit la machine à états de l'agent. Si la structure `<agent structure>` peut être comprise à la fois comme division `<state partitioning>` et comme corps `<agent body>`, elle est interprétée comme corps `<agent body>`.

```
<agent instantiation> ::=
    [<number of instances>]
    <agent additional heading>

<agent additional heading> ::=
    [<specialization>] [<agent formal parameters>]

<entity in agent> ::=
    <signal definition>
    | <signal reference>
    | <signal list definition>
    | <variable definition>
    | <remote procedure definition>
    | <remote variable definition>
    | <data definition>
    | <data type reference>
    | <timer definition>
    | <interface reference>
    | <macro definition>
    | <exception definition>
    | <procedure reference>
    | <procedure definition>
    | <composite state type definition>
    | <composite state type reference>
    | <select definition>
    | <agent type definition>
    | <agent type reference>
    | <association>

<agent reference> ::=
    <block reference>
    | <process reference>

<valid input signal set> ::=
    signalset [<signal list>] <end>

<agent body> ::=
    <state machine graph>

<state machine graph> ::=
    [<on exception>] <start>
    { <state> | <exception handler> | <free action> } *
```

<agent formal parameters> ::=  
     ( <parameters of sort> {, <parameters of sort>}\* )  
 <parameters of sort> ::=  
     <variable name> {, <variable name>}\* <sort>  
 <number of instances> ::=  
     ([ <initial number> ] [ , [ <maximum number> ] ] )  
 <initial number> ::=  
     <Natural simple expression>  
 <maximum number> ::=  
     <Natural simple expression>

Les informations suivantes sont valables pour tous les agents en règle générale. Les propriétés spéciales des systèmes, des blocs et des processus sont abordées dans des paragraphes distincts relatifs à ces concepts.

Le nombre initial d'instances et le nombre maximal d'instances contenus dans le nombre *Number-of-instances* sont obtenus à partir du nombre <number of instances>. Si le nombre <initial number> est omis, <initial number> est 1. Si le nombre <maximum number> est omis, <maximum number> n'est pas limité.

Le nombre <number of instances> utilisé dans la dérivation est le suivant:

- a) s'il n'y a pas de <agent reference> pour l'agent, le nombre <number of instances> dans la définition <agent definition> ou dans la définition <textual typebased agent definition> est utilisé. S'il ne contient pas un nombre <number of instances>, on utilise le nombre <number of instances> où le nombre <initial number> et le nombre <maximum number> sont omis;
- b) si à la fois le nombre <number of instances> dans la définition <agent definition> et le nombre <number of instances> dans une référence <agent reference> sont omis, on utilise le nombre <number of instances> où le nombre <initial number> et le nombre <maximum number> sont omis;
- c) si soit le nombre <number of instances> dans la définition <agent definition> ou le nombre <number of instances> dans une référence <agent reference> est omis, le nombre <number of instances> est celui qui est présent;
- d) si à la fois le nombre <number of instances> dans la définition <agent definition> et le nombre <number of instances> dans une référence <agent reference> sont spécifiés, les deux nombres <number of instances> doivent être égaux lexicalement et ce nombre <number of instances> est utilisé.

Une relation analogue s'applique au nombre <number of instances> spécifié dans le diagramme <agent diagram> et dans la zone <agent reference area> définis ci-dessous.

Le nombre <initial number> d'instances doit être inférieur ou égal au nombre <maximum number> et celui-ci doit être supérieur à zéro.

Dans l'instanciation <agent instantiation>, si des paramètres <agent formal parameters> sont présents, le nombre <number of instances> doit être présent, même si les deux nombres <initial number> et <maximum number> sont omis.

L'utilisation et la syntaxe de l'ensemble <valid input signal set> sont définis au paragraphe 9.

### *Grammaire graphique concrète*

<agent diagram> ::=  
     <system diagram> | <block diagram> | <process diagram>  
 <agent diagram content> ::=  
     {     {<agent text area>}\*  
        {<type in agent area>}\*  
        { <interaction area> | <agent body area> } } *set*

<agent body area> ::=

<state machine graph area>

<frame symbol> ::=



La zone <package use area> doit être placée en tête du symbole <frame symbol> du diagramme <system diagram>, <block diagram>, ou du diagramme <process diagram>.

<agent text area> ::=

<text symbol>

*contains* {

[<valid input signal set>]  
{  
| <signal definition>  
| <signal reference>  
| <signal list definition>  
| <variable definition>  
| <remote procedure definition>  
| <remote variable definition>  
| <data definition>  
| <data type reference>  
| <timer definition>  
| <interface reference>  
| <macro definition>  
| <exception definition>  
| <procedure definition>  
| <procedure reference>  
| <select definition>  
| <agent type reference>  
| <agent reference> }\* }

<state machine graph area> ::=

{  
[ <on exception association area> ] <start area>  
{ <state area> | <exception handler area> | <in connector area> }\* } *set*

<interaction area> ::=

{{  
| <agent area>  
| <create line area>  
| <channel definition area>  
| <state machine area> }+ } *set*

<agent area> ::=

| <agent reference area>  
| <agent diagram>  
| <graphical typebased agent definition>  
| <inherited agent definition>

<agent reference area> ::=

{  
| <system reference area>  
| <block reference area>  
| <process reference area> }  
[ *is connected to* { <graphical package use area>+ } *set* ]

<state machine area> ::=

<state partition area>

<create line area> ::=

<create line symbol>  
*is connected to* { <create line endpoint area> <create line endpoint area> }

```

<create line endpoint area> ::=
    <agent area> | <agent type area> | <state machine area>

<agent type area> ::=
    <agent type reference area>
    | <agent type diagram>

<create line symbol> ::=
    <dependency symbol>

```

Il est permis à une zone <agent text area> de contenir une référence <agent reference> seulement si le contenu <agent diagram content> directement englobant contient une zone <interaction area>.

Dans un diagramme <agent diagram>, les diagrammes <gate on diagram> doivent se trouver à l'extérieur du cadre de diagramme.

L'ensemble *Agent-definition-set* dans la *grammaire abstraite* du type d'agent implicite (voir *Modèle*) correspond aux zones <agent area>.

L'ensemble *Channel-definition-set* dans la *grammaire abstraite* du type d'agent implicite correspond aux zones <channel definition area>.

La flèche placée sur le symbole <create line symbol> indique la zone <agent area> ou la zone <agent type area> d'un agent ou d'un type d'agent sur laquelle une action de création est menée. Les symboles <create line symbol> sont facultatifs, mais s'ils sont utilisés, il doit exister dans l'agent (le type d'agent ou la machine à états) à l'extrémité de départ du symbole <create line symbol> une demande de création l'agent (ou type d'agent) à l'extrémité de la tête de flèche du symbole <create line symbol>. Cette action de création peut être héritée et ne nécessite donc pas de spécification directe dans l'agent ou le type d'agent. Cette règle s'applique après transformation dans la zone <option area>.

NOTE 1 – Cette règle peut être appliquée de manière indépendante, avant ou après transformation de l'option <transition option>.

La zone <state machine area> de la zone <interaction area> désigne l'automate à états (composites) de l'agent, qui peut être représenté directement sous la forme d'un graphe d'agent ou par référence à une définition d'état.

S'il existe une zone <agent reference area> pour l'agent, il doit y avoir une zone <gate property area> reliée à la zone <agent reference area> ou il doit y avoir un accès <gate> contenu dans le symbole pour chaque diagramme <gate on diagram> relié au diagramme <agent diagram>. Si le diagramme <gate on diagram> est une définition <graphical gate definition>, la zone <gate property area> doit être respectivement une définition <graphical gate definition> ou un accès <gate>, mentionnant le même nom <gate name>. Si le diagramme <gate on diagram> est une définition <graphical interface gate definition>, la zone <gate property area> doit être une définition <graphical interface gate definition> mentionnant le même identificateur <interface identifier>. Une règle correspondante s'applique s'il existe une référence <agent reference> pour l'agent.

Une zone <graphical package use area> reliée à une zone <agent reference area> doit être cohérente avec la <graphical package use area> du diagramme référencé.

### Sémantique

Une définition *Agent-definition* a un nom qui peut être utilisé dans des qualificatifs en liaison avec **system**, **block** ou **process** selon le genre de l'agent.

Une définition d'agent définit un ensemble d'agents. Plusieurs instances du même agent peuvent exister au même moment et s'exécuter de manière asynchrone et en parallèle ou en alternance entre elles et avec des instances d'autres ensembles d'agents dans le système.

Dans le nombre *Number-of-instances*, la première valeur représente le nombre d'instances de l'ensemble d'agents qui existe à la création du système ou de l'entité contenante (instances initiales), la seconde valeur représente le nombre maximal d'instances simultanées de l'ensemble d'agents.

Selon qu'il s'agisse d'un bloc ou d'un processus, une définition *Agent-definition* dans un ensemble *Agent-definition-set* aura un comportement différent et sera donc définie séparément pour les blocs et les processus.

Une instance d'agent possède une machine à états finis étendus communicante définie par sa définition explicite ou implicite de machine à états. Dès qu'une machine à états est dans un état, elle assure un certain nombre d'actions, appelées transitions, suite à la réception d'un signal donné. La réalisation de la transition aboutit dans la machine à états à l'attente de l'agent dans un autre état, qui n'est pas nécessairement différent de l'état d'origine.

Lorsqu'un agent est interprété, les agents initiaux qu'il contient sont créés. La communication de signaux entre les machines à états finis de ces agents initiaux, la machine à états finis de l'agent et leur environnement ne commence que lorsque les agents initiaux ont été créés. La durée nécessaire à la création d'un agent peut ou non être significative. Les paramètres formels de ces agents initiaux n'ont pas d'éléments de données associés (ils sont initialisés avec une valeur "non définie").

Les instances d'agent existent à partir du moment où l'agent contenant est créé ou peuvent être créées par une des actions de demande de création d'agents qui lance les processus à interpréter; leur interprétation commence lorsque l'action de départ est interprétée; des actions d'arrêt peuvent arrêter leur existence.

Lorsque la machine à états d'un agent interprète un arrêt, si cet agent était un conteneur concurrent, elle continuera à traiter les appels de procédures distantes implicites en servant de médiateur entre l'accès et les variables globales. La machine à états d'un tel agent reste dans cette "condition d'arrêt" jusqu'à ce que tous les agents contenus cessent, après quoi l'agent prend fin. Lorsqu'il est en condition d'arrêt, l'agent n'accepte aucun stimulus, excepté l'ensemble implicite et introduit les appels de procédures distantes pour chaque variable globale, le cas échéant. Après terminaison d'un agent, son pid n'est plus valide.

Les signaux reçus par les instances d'agents sont appelés signaux d'entrée, et les signaux envoyés par les instances d'agents sont appelés signaux de sortie. L'ensemble <valid input signal set> d'un agent définit l'ensemble <valid input signal set> de sa machine à états.

Appeler et servir des appels de procédures distantes, et accéder à des variables distantes correspond également à des échanges de signaux ( voir respectivement 10.5 et 10.6).

Les signaux peuvent être traités par la machine à états d'une instance d'agent uniquement lorsque celle-ci se trouve dans un certain état. L'ensemble complet de signaux d'entrée valides est l'union de:

- a) l'ensemble des signaux se trouvant dans tous les canaux ou accès conduisant à l'ensemble des instances d'agent;
- b) l'ensemble <valid input signal set>;
- c) les signaux d'entrée implicites définis en 10.5, 10.6, 11.4, 11.5 et 11.6;
- d) les signaux de temporisation.

Un accès d'entrée unique est associé à la machine à états finis de chaque instance d'agent. Les signaux qui sont envoyés à un agent conteneur sont envoyés vers l'accès d'entrée de l'agent, sous réserve que le signal apparaisse sur un canal (explicite ou implicite) relié à sa machine à états. Les signaux se produisant uniquement dans l'ensemble <valid input signal set> ne doivent pas être utilisés pour les communications externes. Ils servent pour les communications entre des instances à l'intérieur de l'ensemble d'instances

La machine à états finis d'un agent est en attente dans un état, ou active, en train d'effectuer une transition. A chaque état correspond un ensemble de signaux de sauvegarde (voir également 11.7).

Lorsqu'il est en attente dans un état, le premier signal d'entrée dont l'identificateur ne fait pas partie de l'ensemble des signaux de sauvegarde, est pris dans la file d'attente et traité par l'agent. Une transition peut également être lancée en tant que transition spontanée, indépendamment de la présence de tout signal dans la file d'attente.

L'accès d'entrée peut retenir un nombre quelconque de signaux d'entrée, de sorte que plusieurs signaux entrants peuvent être mis dans la file d'attente pour la machine à états finis de l'instance d'agent. L'ensemble des signaux retenus est ordonné dans la file d'attente, selon l'ordre d'arrivée. Si deux ou plusieurs signaux arrivent "simultanément", ils sont ordonnés arbitrairement.

Lorsqu'un agent est créé, on attribue à sa machine à états finis un accès d'entrée vide, et il y a alors création de variables locales.

Lorsqu'une instance d'agent conteneur est créée, il y a création des agents initiaux des ensembles de l'agent contenu. Si le conteneur est créé au moyen d'une demande <create request>, le **parent** des agents contenus (voir *Modèle* ci-dessous) reçoit le pid du conteneur. Les paramètres formels sont des variables qui sont créées soit lorsque le système est créé (mais aucun paramètre réel ne lui est transmis et par conséquent ces paramètres sont "non définis"), soit lorsque l'instance d'agent est dynamiquement créée.

Si une définition <agent definition> ou <agent type definition>, utilisée dans une définition <textual typebased agent definition>, contient des définitions <channel definition> et <textual typebased agent definition>, chaque accès des définitions <agent type definition> des définitions <textual typebased agent definition> contenues doit être relié à un canal.

La définition d'un agent implique la définition d'une interface dans la même portée de l'agent (voir 12.1.2). La sorte de pid définie de manière implicite par cette interface est identifiée avec le nom *Agent-name* et elle est visible dans la même unité de portée que celle dans laquelle l'agent est défini.

NOTE 2 – Etant donné que chaque agent possède une interface définie implicitement avec le même nom, l'agent doit toujours avoir un nom différent de chaque interface définie explicitement et de chacun des types d'agent (lesquels possèdent également des interfaces implicites) définis dans le même domaine de visibilité, afin d'éviter des collisions de noms.

### *Modèle*

Une définition *Agent-definition* possède un type d'agent anonyme implicite qui définit les propriétés de l'agent.

Un agent possédant un corps <agent body> ou une zone <agent body area> est une abréviation pour un agent n'ayant qu'une seule machine à états mais pas d'agents contenus. Cette machine à états est obtenue en remplaçant le corps <agent body> ou la zone <agent body area> par une définition d'état composite. Cette définition d'état composite a le même nom que l'agent et son graphe *State-transition-graph* est représenté par le corps <agent body> ou la zone <agent body area>.

Un agent qui est une spécialisation est une abréviation pour définir un type d'agent implicite et un agent fondé sur le type de ce type.

Dans toutes les instances d'agent, quatre variables anonymes de la sorte de pid de l'agent (pour les agents qui ne sont pas fondés sur un type d'agent) ou de la sorte de pid du type d'agent (pour les agents fondés sur le type) sont déclarées et sont appelées **self**, **parent**, **offspring** et **sender**. Elles donnent un résultat pour:

- a) l'instance d'agent (**self**);
- b) l'instance de l'agent créateur (**parent**);
- c) l'instance d'agent la plus récente créée par l'instance d'agent (**offspring**);
- d) l'instance d'agent en provenance de laquelle le dernier signal entrant a été utilisé (**sender**) (voir également 11.2.2).

Ces variables anonymes sont accessibles au moyen d'expressions de pid comme expliqué plus en détails en 12.3.4.3.

Pour toutes les instances d'agent qui se trouvent présentes au moment de l'initialisation du système, l'expression **parent** est initialisée à la valeur Null.

Pour toutes les instances d'agent nouvellement créées, les expressions **sender** et **offspring** sont initialisées à la valeur Null.

## 9.1 Système

Un système est l'agent le plus à l'extérieur et a le genre *Agent-kind* **SYSTEM**. Il est défini par une définition <system definition> ou un diagramme <system diagram>. La sémantique des agents s'applique, avec les adjonctions présentées dans le présent sous-paragraphe.

### *Grammaire abstraite*

Un *Agent* avec le genre *Agent-kind* **SYSTEM** ne doit contenir aucun autre *Agent*. Il doit contenir au minimum une définition *Agent-definition* ou une définition explicite ou implicite *State-machine-definition*.

Les définitions de tous les signaux, les types de données et syntypes utilisées dans l'interface avec l'environnement et entre les agents contenus du système (y compris lui-même) sont contenues dans la définition *Agent-definition* du système.

Le nombre *Initial-number* d'instances est 1 et le nombre *Maximum-number* d'instances est 1.

NOTE – Le nombre <number of instances> ne peut pas être spécifié.

### *Grammaire textuelle concrète*

```
<system definition> ::=  
    {<package use clause>}*  
    <system heading> <end> <agent structure>  
    endsystem [<system name>] <end>
```

```
<system heading> ::=  
    system <system name> <agent additional heading>
```

L'en-tête <agent additional heading> dans une définition <system definition> ou dans un diagramme <system diagram> peut ne pas inclure de paramètres <agent formal parameters>.

### *Grammaire graphique concrète*

```
<system diagram> ::=  
    <frame symbol> contains {<system heading> <agent diagram content> }  
    [ is associated with <package use area> ]
```

```
<system reference area> ::=  
    <block symbol> contains  
    { system <system name> }
```

Les diagrammes <gate on diagram> dans un diagramme <system diagram> peuvent ne pas inclure d'identificateurs <channel identifier>.

Une zone <system reference area> doit être utilisée uniquement dans le cadre d'une spécification <sdl specification>.

### *Sémantique*

Une définition *Agent-definition* avec le genre *Agent-kind* **SYSTEM** est la représentation ou la description en SDL d'un système. Un système est le bloc le plus à l'extérieur. Cela signifie que les agents à l'intérieur d'un système sont des blocs et des processus qui sont interprétés en parallèle, avec éventuellement la machine à états du système.



Un système est séparé de son environnement par une frontière de système et contient un ensemble d'agents. La communication entre le système et l'environnement ou entre les agents à l'intérieur du système, peut se faire au moyen de signaux, de procédures distantes et de variables distantes. A l'intérieur d'un système, ces moyens de communication sont véhiculés dans des canaux explicites ou implicites. Les canaux relient les agents contenus entre eux ou à la frontière du système.

Une instance de système est une instantiation d'un type de système définie par une définition *Agent-definition* avec le genre *Agent-kind* **SYSTEM**. L'interprétation d'une instance de système est réalisée par une machine abstraite SDL, qui en conséquence donne la sémantique aux concepts du SDL. Pour interpréter une instance de système, il faut:

- a) initialiser le temps du système;
- b) interpréter les agents contenus et les canaux qui leur sont reliés;
- c) interpréter la machine à états facultative du système.

### Modèle

Pour chaque définition <channel definition> dans un système mentionnant **env**, un accès avec un nom anonyme est ajouté à la définition Agent-definition. La définition de canal est modifiée pour mentionner cet accès dans le trajet <channel path> dirigé vers l'environnement du système.

## 9.2 Bloc

Un bloc est un agent avec le genre *Agent-kind* **BLOCK**. La sémantique des agents s'applique donc, avec les adjonctions présentées dans le présent sous-paragraphe. Un bloc est défini par une définition <block definition> ou un diagramme <block diagram>.

Les instances contenues dans une instance de bloc sont interprétées en parallèle et de manière asynchrone les unes par rapport aux autres, avec la machine à états de l'instance de bloc contenant. Toutes les communications entre les différentes instances contenues dans un bloc sont effectuées de manière asynchrone au moyen d'échanges de signaux, explicitement ou implicitement en utilisant par exemple des appels de procédures distantes.

### Grammaire textuelle concrète

```
<block definition> ::=
    {<package use clause>}*
    <block heading> <end> <agent structure>
    endblock [ [<qualifier>] <block name> ] <end>

<block heading> ::=
    block [<qualifier>] <block name> <agent instantiation>

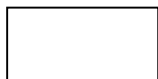
<block reference> ::=
    block <block name> [<number of instances>] referenced <end>
```

### Grammaire graphique concrète

```
<block diagram> ::=
    <frame symbol> contains {<block heading> <agent diagram content> }
    is connected to { {<gate on diagram>}* <external channel identifiers> } set
    [ is associated with <package use area> ]

<block reference area> ::=
    <block symbol> contains
        { { <block name> [<number of instances>] } {<gate>*} set }
    is connected to { <gate property area>* } set

<block symbol> ::=
```



Les accès <gate> sont placés près de la frontière des symboles et sont associés aux points de connexion à des canaux.

L'identificateur <external channel identifiers> identifie les canaux extérieurs reliés aux canaux dans le diagramme <block diagram>. Il est placé à l'extérieur du symbole <frame symbol>, proche du point extrême des canaux au niveau du symbole <frame symbol>.

### *Sémantique*

Une définition de bloc est une définition d'agent qui définit des conteneurs pour une ou plusieurs définitions de processus ou de bloc.

Une instance de bloc est une instanciation d'un type de bloc définie par une *Agent-definition* avec le genre *Agent-kind* **BLOCK**. Pour interpréter une instance de bloc, il faut:

- a) interpréter les agents contenus et les canaux qui leur sont reliés;
- b) interpréter la machine à états du bloc (si elle est présente).

Pour un bloc ayant une machine à états finis, la création du bloc (et de ses agents contenus) comprend la création de la machine à états finis et elle est interprétée en parallèle avec les agents du bloc.

Un bloc ayant une définition de variable mais pas de machine à états, possède une machine à états implicite associée qui est interprétée en parallèle avec les agents dans le bloc.

L'accès entre des agents contenus dans le bloc et une variable accessible du bloc est couvert par des procédures distantes implicitement définies permettant le réglage et l'obtention des éléments de données associés à la variable. Ces procédures sont fournies par la machine à états du bloc.

### *Modèle*

Un bloc *b* avec une machine à états et des variables, est modélisé en conservant le bloc *b* (sans les variables) et en transformant l'entité d'état et les variables dans une machine à états (*sm*) séparée dans le bloc *b*. Pour chaque variable *v* dans le bloc *b*, cette machine à états aura une variable *v* et deux procédures exportées *set\_v* (avec un paramètre IN de la sorte de *v*) et *get\_v* (avec pour sorte de *v* un type de retour). Chaque affectation vers *v* émanant de définitions englobées est transformée en un appel distant de *set\_v*. Chaque occurrence de *v* en expressions dans les définitions englobées est transformée en un appel distant de *get\_v*. Ces occurrences s'appliquent également aux occurrences dans les procédures définies dans le bloc *b*, dans la mesure où elles sont transformées en procédures locales vers les agents appelants.

Un bloc *b* avec uniquement des variables et des procédures est transformé de la même manière que ci-dessus, avec le graphe de la machine à états générée possédant uniquement un état, dans lequel elle entre les procédures *set* et *get* générées.

Les canaux reliés à la machine à états sont transformés de manière à les relier à la machine *sm*.

Cette transformation est effectuée après la transformation des types et des paramètres de contexte.

## **9.3 Processus**

Un processus est un agent avec le genre *Agent-kind* **PROCESS**. La sémantique des agents s'applique donc, avec les adjonctions présentées dans le présent sous-paragraphe. Un processus est défini par une définition <process definition> ou un diagramme <process diagram>.

Un processus est utilisé pour introduire des données partagées dans une spécification, permettant ainsi l'utilisation des variables du processus contenant ou l'utilisation d'objets. Toutes les instances dans un processus peuvent accéder aux variables locales du processus.

Pour permettre des communications sûres malgré le partage des données dans un processus, toutes les instances sont interprétées en utilisant une sémantique alternative. Ceci signifie que pour deux

instances quelconques à l'intérieur d'un processus, deux transitions ne seront pas interprétées en parallèle, cela signifie également que l'interprétation d'une transition dans une instance n'est pas interrompue par une autre instance. Lorsqu'une instance est en attente par exemple d'un retour d'appel de procédure distante, elle est dans un certain état; une autre instance peut donc être interprétée.

### Grammaire abstraite

Une définition *Agent-definition* possédant le genre *Agent-kind* **PROCESS** doit contenir au moins une définition *Agent-definition* ou elle doit avoir une définition *State-machine-definition* explicite ou implicite.

Les définitions contenues *Agent-definitions* d'une définition *Agent-definition* possédant un genre *Agent-kind* **PROCESS** doivent avoir le genre *Agent-kind* **PROCESS**.

### Grammaire textuelle concrète

```
<process definition> ::=
    {<package use clause>}*
    <process heading> <end> <agent structure>
    endprocess [ [<qualifier>] <process name> ] <end>

<process heading> ::=
    process [<qualifier>] <process name> <agent instantiation>

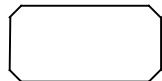
<process reference> ::=
    process <process name> [<number of instances>] referenced <end>
```

### Grammaire graphique concrète

```
<process diagram> ::=
    <frame symbol> contains {<process heading> <agent diagram content> }
    is connected to { {<gate on diagram>}* <external channel identifiers> } set
    [ is associated with <package use area> ]

<process reference area> ::=
    <process symbol> contains
        { {<process name> [<number of instances>] } {<gate>}* } set
    is connected to { {<gate property area>}* } set

<process symbol> ::=
```



Les accès <gate> sont placés à proximité de la frontière des symboles et sont associés au point de connexion aux canaux.

L'identificateur <external channel identifiers> identifie les canaux extérieurs reliés aux canaux dans le diagramme <process diagram>. Il est placé à l'extérieur du symbole <frame symbol>, proche du point extrême des canaux au niveau du symbole <frame symbol>.

### Sémantique

Une instance d'un processus ayant des ensembles d'instances de processus est interprétée au moyen de l'interprétation des instances présentes dans les ensembles d'instances contenus, en alternance les uns par rapport aux autres et avec la machine à états de l'instance du processus contenant, si elle existe. Une interprétation alternante implique que seule une des instances à l'intérieur du contexte alternant peut interpréter une transition à la fois, et également que dès que l'interprétation d'une transition d'une instance de processus impliquée a débuté, elle continue jusqu'à ce qu'un état (explicite ou implicite) soit atteint ou que l'instance du processus se termine. L'état peut être un état implicite introduit par des transformations (par exemple à la suite d'un appel de procédure distante).

Un processus ayant des définitions de variable et des processus contenus mais qui n'a pas de machine à états, possède une machine à états implicite associée qui est interprétée en alternance avec les processus contenus.

NOTE – L'agrégat d'états a également une interprétation en alternance. Cependant, les processus alternants d'un processus ont chacun leur propre accès d'entrée et leurs expressions **self**, **parent**, **offspring** et **sender**. Dans le cas de l'agrégat d'états, il n'existe qu'un seul accès d'entrée et un seul ensemble d'expressions **self**, **parent**, **offspring** et **sender** appartenant à l'agent conteneur.

## 9.4 Procédure

Les procédures sont définies au moyen de définitions de procédure. On fait appel à une procédure au moyen d'un appel de procédure identifiant la définition de procédure. Des paramètres sont associés à un appel de procédure. C'est du mécanisme de transfert des paramètres que dépendent les variables affectées par l'interprétation d'une procédure. Les appels de procédure peuvent être des actions ou des parties d'expression (procédures retournant une valeur uniquement).

### *Grammaire abstraite*

<i>Procedure-definition</i>	::	<i>Procedure-name</i> <i>Procedure-formal-parameter*</i> [ <i>Result</i> ] <i>Procedure-identifiser</i> <i>Data-type-definition-set</i> <i>Syntype-definition-set</i> <i>Variable-definition-set</i> <i>Composite-state-type-definition-set</i> <i>Procedure-definition-set</i> <i>Procedure-graph</i>
<i>Procedure-name</i>	=	<i>Name</i>
<i>Procedure-formal-parameter</i>	=	<i>In-parameter</i>   <i>Inout-parameter</i>   <i>Out-parameter</i>
<i>In-parameter</i>	::	<i>Parameter</i>
<i>Inout-parameter</i>	::	<i>Parameter</i>
<i>Out-parameter</i>	::	<i>Parameter</i>
<i>Parameter</i>	::	<i>Variable-name</i> <i>Sort-reference-identifiser</i>
<i>Result</i>	::	<i>Sort-reference-identifiser</i>
<i>Procedure-graph</i>	::	[ <i>On-exception</i> ] <i>Procedure-start-node</i> <i>State-node-set</i> <i>Free-action-set</i> <i>Exception-handler-node-set</i>
<i>Procedure-start-node</i>	::	[ <i>On-exception</i> ] <i>Transition</i>
<i>Procedure-identifiser</i>	=	<i>Identifiser</i>

Si une définition *Procedure-definition* contient un résultat *Result*, elle correspond à une procédure retournant une valeur.

## Grammaire textuelle concrète

```

<procedure definition> ::=
    | <external procedure definition>
    | {<package use clause>}*
      <procedure heading> <end>
      <entity in procedure>*
      [<procedure body>]
      endprocedure [ [<qualifier>] <procedure name> ] <end>
    | {<package use clause>}*
      <procedure heading>
      [ <end> <entity in procedure>+ ]
      [<virtuality>] <left curly bracket>
      <statement list>
      <right curly bracket>

<procedure preamble> ::=
    <type preamble> [ exported [ as <remote procedure identifier> ] ]

<procedure heading> ::=
    <procedure preamble>
    procedure [<qualifier>] <procedure name>
    [<formal context parameters>] [<virtuality constraint>]
    [<specialization>]
    [<procedure formal parameters>]
    [<procedure result>] [<raises>]

<procedure formal parameters> ::=
    ( <formal variable parameters> { , <formal variable parameters> } * )

<formal variable parameters> ::=
    <parameter kind> <parameters of sort>

<parameter kind> ::=
    [ in/out | in | out ]

<procedure result> ::=
    <result sign> [<variable name>] <sort>

<raises> ::=
    raise <exception identifier> { , <exception identifier> } *

<entity in procedure> ::=
    | <variable definition>
    | <data definition>
    | <data type reference>
    | <procedure reference>
    | <procedure definition>
    | <composite state type definition>
    | <composite state type reference>
    | <exception definition>
    | <select definition>
    | <macro definition>

<procedure signature> ::=
    [ ( <formal parameter> { , <formal parameter> } * ) ] [<result>] [<raises>]

<procedure body> ::=
    [<on exception>] [<start>] { <state> | <exception handler> | <free action> } *

<external procedure definition> ::=
    procedure <procedure name> <procedure signature> external <end>

<procedure reference> ::=
    <type preamble>
    procedure <procedure identifier> <type reference properties>

```

Le symbole facultatif <virtuality> placé avant la liste <left curly bracket> <statement list> dans la définition <procedure definition> s'applique à la transition initiale de la procédure, qui est en l'occurrence la liste de déclarations.

Une procédure exportée ne peut pas avoir de paramètres de contexte formel et sa portée englobante doit être un type d'agent ou une définition d'agent.

Dans une définition <procedure definition>, la définition <variable definition> ne peut contenir les noms **exported** <variable name> (voir 12.3.1).

S'il est présent, l'attribut **exported** est hérité par n'importe quel sous-type d'une procédure. Une procédure exportée virtuelle doit contenir **exported** dans toutes les redéfinitions. Les types virtuels, y compris les procédures virtuelles, sont décrits en 8.3.2. La clause facultative **as** dans une redéfinition doit désigner le même identificateur <remote procedure identifier> que dans le supertype. Si cette clause est omise dans une redéfinition, cela indique qu'il s'agit de l'identificateur <remote procedure identifier> du supertype.

Deux procédures exportées dans un agent ne peuvent pas mentionner le même identificateur <remote procedure identifier>.

Une procédure externe ne peut pas être mentionnée dans une expression <type expression>, dans un paramètre <formal context parameter> ou dans une contrainte <procedure constraint>.

Il n'existe pas de syntaxe graphique correspondante pour la définition <external procedure definition>.

Si une exception peut être déclenchée dans une procédure dans laquelle aucun gestionnaire d'exception n'est actif avec la clause de gestionnaire correspondante (c'est-à-dire que l'exception n'est pas gérée), <raises> doit mentionner cette exception. Une exception est considérée comme n'étant pas traitée s'il n'y a pas de débit de contrôle potentiel dans la procédure produisant l'exception et qu'aucun gestionnaire d'exception activé dans ce débit de contrôle ne traite l'exception.

Si une définition <procedure definition> apparaît à l'intérieur d'un symbole <text symbol>, elle doit contenir l'état <state>.

Une procédure **endprocedure**, contenue dans une définition <procedure definition> interne d'une procédure <entity in procedure> d'une définition <procedure definition> externe, appartient à cette définition <procedure definition> interne.

### Grammaire graphique concrète

```
<procedure diagram> ::=
    <frame symbol> contains {
        <procedure heading>
        {
            <procedure text area>*
            <procedure area>*
            <procedure graph area> } set }
    [ is associated with <package use area> ]
```

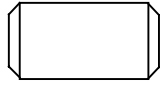
```
<procedure area> ::=
    <procedure diagram>
    | <procedure reference area>
```

```
<procedure text area> ::=
    <text symbol> contains
    {
        <variable definition>
        | <data definition>
        | <data type reference>
        | <procedure reference>
        | <procedure definition>
        | <exception definition>
        | <select definition>
        | <macro definition> }*
```

<procedure reference area> ::=  
     <type reference area>

La zone <type reference area> qui fait partie d'une zone <procedure reference area> doit avoir un en-tête <graphical type reference heading> qui contient un nom <procedure name>.

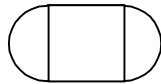
<procedure symbol> ::=



<procedure graph area> ::=  
     [ <on exception association area> ] [ <procedure start area> ]  
     { <state area> | <exception handler area> | <in connector area> }\*

<procedure start area> ::=  
     <procedure start symbol>  
     **contains** { [ <virtuality> ] }  
     [ **is connected to** <on exception association area> ]  
     **is followed by** <transition area>

<procedure start symbol> ::=



La zone <package use area> doit être placée en tête du symbole <frame symbol>.

La zone <on exception association area> d'une zone <procedure graph area> identifie le gestionnaire d'exception associé au graphe complet. L'extrémité de départ ne doit être reliée à aucun symbole.

### Sémantique

Une procédure est un moyen de donner un nom à un assemblage d'objets et de le représenter par une référence unique. Les règles relatives aux procédures imposent une discipline sur la manière dont l'assemblage d'objets est choisi et elles limitent la portée du nom des variables définies dans la procédure.

**exported** dans un <procedure preamble> implique que la procédure peut être appelée comme une procédure distante, conformément au modèle donné en 10.5.

Une variable de procédure est une variable locale à la procédure qui ne peut pas être exportée. Elle est créée lors de l'interprétation du nœud de départ de procédure et cesse d'exister lors de l'interprétation du nœud de retour du graphe de procédure.

L'interprétation d'un appel *Call-node* (représenté par un appel <procedure call>, voir 11.13.3), un nœud *Value-returning-call-node* (représenté par un appel <value returning procedure call>, voir 12.3.5), ou une application *Operation-application* (représentée par une application <operation application>, voir 12.2.7) provoque la création d'une instance de procédure et l'interprétation commence de la manière suivante:

- a) une variable locale est créée pour tout paramètre *In-parameter* ayant le *Name* et la *Sort* du paramètre *In-parameter*. A cette variable on affecte le résultat de l'expression en interprétant une affectation entre la variable et l'expression donnée par le paramètre réel s'il est présent. Autrement, la variable n'obtient pas d'élément de données associé; c'est-à-dire qu'elle devient "non définie";
- b) une variable locale est créée pour tout paramètre *Out-parameter* ayant le *Name* et la *Sort* du paramètre *Out-parameter*. La variable n'obtient pas d'élément de données associé; c'est-à-dire qu'elle devient "non définie";
- c) une variable locale est créée pour chaque définition *Variable-definition* dans la définition *Procedure-definition*;

- d) chaque paramètre *Inout-parameter* décrit une variable qui est donnée par l'expression des paramètres réels dans 11.13.3. Le nom *Variable-name* contenu est utilisé tout au long de l'interprétation du graphe *Procedure-graph* lorsqu'on se réfère à l'élément de données associé à la variable ou lorsque l'on affecte un nouvel élément de données à la variable;
- e) la *Transition* contenue dans le nœud *Procedure-start-node* est interprétée;
- f) avant d'interpréter un nœud *Return-node* contenu dans le graphe *Procedure-graph*, on donne aux paramètres *Out-parameters* les éléments de données de la variable locale correspondante.

Les nœuds du graphe de procédure sont interprétés de la même manière que les nœuds équivalents d'un agent; c'est-à-dire que la procédure a le même ensemble complet de signaux d'entrée valides que l'agent englobant, et les mêmes accès d'entrée que l'instance de l'agent englobant qui l'a appelée, soit directement ou indirectement.

Une procédure externe est une procédure dont le corps `<procedure body>` n'est pas inclus dans la description en SDL. De manière conceptuelle, il est supposé qu'un corps `<procedure body>` est donné à une procédure externe et que celle-ci sera transformée en une définition `<procedure definition>` qui fait partie de la transformation du système générique, voir Annexe F).

### Modèle

Un paramètre formel n'ayant pas de genre explicite `<parameter kind>` a un genre implicite `<parameter kind>` **in**.

Lorsqu'un nom `<variable name>` est présent dans le résultat `<procedure result>`, tous les retours `<return>` ou zones `<return area>` dans le graphe de procédure sans `<expression>` sont remplacés respectivement par le retour `<return>` ou la zone `<return area>` contenant le nom `<variable name>` comme l'`<expression>`.

Un résultat `<procedure result>` avec un nom `<variable name>` est une syntaxe dérivée pour une définition `<variable definition>` avec un nom `<variable name>` et une sorte `<sort>` dans des variables `<variables of sort>`. S'il existe une définition `<variable definition>` impliquant un nom `<variable name>`, aucune autre définition `<variable definition>` n'est ajoutée.

Une zone `<procedure start area>` qui contient `<virtuality>`, une procédure `<start>` qui contient `<virtuality>` ou une liste `<statement list>` dans une définition `<procedure definition>` suivant `<virtuality>` est appelée départ de procédure virtuelle. Ce départ est décrit avec plus de détails en 8.3.3.

La deuxième forme de la définition `<procedure definition>` est une syntaxe dérivée pour la définition `<procedure definition>` suivante:

```

<procedure preamble>
<procedure heading>;
  start <virtuality>;
  task { <statement list>-transform };
  return ;
endprocedure ;

```

Cette transformation se produit après la transformation de l'instruction `<compound statement>`.



## 10 Communication

### 10.1 Canal

#### Grammaire abstraite

<i>Channel-definition</i>	::	<i>Channel-name</i> [ <b>NODELAY</b> ] <i>Channel-path-set</i>
<i>Channel-path</i>	::	<i>Originating-gate</i> <i>Destination-gate</i> <i>Signal-identifiant-set</i>
<i>Originating-gate</i>	=	<i>Gate-identifiant</i>
<i>Destination-gate</i>	=	<i>Gate-identifiant</i>
<i>Gate-identifiant</i>	=	<i>Identifiant</i>
<i>Agent-identifiant</i>	=	<i>Identifiant</i>
<i>Channel-name</i>	=	<i>Name</i>

L'ensemble *Channel-path-set* contient au moins un trajet *Channel-path* et pas plus de deux. Lorsqu'il y a deux trajets, le canal est bidirectionnel et l'accès *Originating-gate* de chaque trajet *Channel-path* doit être le même que l'accès *Destination-gate* de l'autre trajet *Channel-path*.

Si l'accès *Originating-gate* et l'accès *Destination-gate* sont le même agent, le canal doit être unidirectionnel (il ne doit y avoir qu'un seul élément dans l'ensemble *Channel-path-set*).

L'accès *Originating-gate* ou *Destination-gate* doit être défini dans la même unité de portée dans la syntaxe abstraite dans laquelle le canal est défini.

**NODELAY** indique le fait que le canal ne cause pas de retard.

Il est permis à un canal de relier entre elles les deux directions d'un accès bidirectionnel.

#### Grammaire textuelle concrète

```
<channel definition> ::=
    channel [<channel name>] [nodelay]
        <channel path> [<channel path>]
    endchannel [<channel name>] <end>

<channel path> ::=
    from <channel endpoint>
    to <channel endpoint> [ with <signal list> ] <end>

<channel endpoint> ::=
    { <agent identifiant> | <state identifiant> | env | this } [<via gate>]

<via gate> ::=
    via <gate>
```

Le nom <channel name> de fin ne peut être spécifié que si le nom <channel name> de départ est spécifié. Si le nom <channel name> de départ n'est pas spécifié, le canal ne peut pas être désigné par un nom.

L'extrémité <channel endpoint> **this** désigne la machine à états de l'agent englobant directement la définition de canal.

L'accès <gate> doit être spécifié si:

- <channel endpoint> désigne une connexion à une définition <textual typebased agent definition> ou une <textual typebased state partition definition>, auquel cas l'accès <gate> doit être directement défini dans le type d'agent ou le type d'état pour cet agent ou cet état respectivement;
- env** est spécifié et le canal est défini dans un type d'agent, auquel cas l'accès <gate> doit être défini dans ce type d'agent.

Si l'accès <gate> est spécifié, le canal est relié à cet accès. L'accès et le canal doivent avoir au minimum un élément en commun dans leurs listes de signaux dans la même direction. Si aucun accès <gate> n'est spécifié, les règles suivantes s'appliquent:

- a) si l'extrémité de canal est un agent ou une machine à états et que cet agent/état contient une connexion <channel to channel connection> pour le canal, ce dernier est relié à l'accès implicite introduit par la connexion <channel to channel connection>,
- b) si l'extrémité de canal est un état, le canal est relié à l'accès implicite de cette machine à états (voir Annexe F),

autrement, le canal introduit un accès implicite sur l'agent ou l'état mentionné dans l'extrémité <channel endpoint>. Cet accès obtient la liste <signal list> des trajets <channel path> respectifs comme ses contraintes d'accès correspondantes. Le canal est relié à cet accès.

### Grammaire graphique concrète

```
<channel definition area> ::=
    <channel symbol>
    is associated with
        { [<channel name>] { [<gate>] [<signal list area>] [<signal list area>] }set }
    is connected to {
        { <agent area> | <state machine area>
        | <gate property area> | <gate on diagram> }
        { <agent area> | <state machine area>
        | <gate property area> | <gate on diagram> } }set
```

Lorsque le symbole <channel symbol> est relié à une zone <state machine area>, il désigne la machine à états de l'agent englobant directement la définition de canal.

```
<channel symbol> ::=
    <delaying channel symbol 1>
    | <delaying channel symbol 2>
    | <nondelaying channel symbol 1>
    | <nondelaying channel symbol 2>
```

```
<delaying channel symbol 1> ::=
    _____>
```

```
<delaying channel symbol 2> ::=
    <----->
```

```
<nondelaying channel symbol 1> ::=
    _____>
```

```
<nondelaying channel symbol 2> ::=
    <----->
```

Pour chaque flèche placée sur le symbole <channel symbol>, il doit y avoir au plus une zone <signal list area>. Une zone <signal list area> doit être suffisamment proche de la flèche à laquelle elle est associée pour éviter toute ambiguïté.

Les flèches des symboles <nondelaying channel symbol 1> et <nondelaying channel symbol 2> sont placées aux extrémités du canal et indiquent que le canal ne cause pas de retard.

### Sémantique

Une définition *Channel-definition* représente un trajet de transport pour les signaux (y compris les signaux implicites induits par des procédures distantes et des variables distantes, voir 10.5 et 10.6). Un canal peut être considéré comme un ou deux trajets indépendants de canal unidirectionnel entre deux agents ou entre un agent et son environnement. Un canal peut également relier la machine à états (état composite) d'un agent avec l'environnement et avec des agents contenus.

L'ensemble *Signal-identifieur-set* dans chaque trajet *Channel-path* de la définition *Channel-definition* contient les signaux qui peuvent être acheminés sur ce trajet *Channel-path*.

Les signaux acheminés par les canaux sont véhiculés jusqu'au point extrême de destination.

Au point extrême de destination d'un canal, les signaux se présentent dans le même ordre que celui des signaux à son origine. Si deux ou plusieurs signaux arrivent simultanément sur le canal, ils sont ordonnés arbitrairement.

Un canal avec retard peut retarder l'acheminement des signaux sur le canal. Cela signifie qu'une file d'attente du type premier entré – premier sorti (FIFO, *first-in-first-out*) se trouve associée à chaque direction dans un canal. Quand un signal se présente sur le canal, il est placé dans la file d'attente. Après un intervalle de temps non déterminé et non constant, la première instance de signal dans la file d'attente est libérée et appliquée à l'une des extrémités qui se trouve connectée au canal.

Plusieurs canaux peuvent exister entre deux points d'extrémité. Le même type de signal peut être acheminé sur des canaux différents.

Lorsqu'une instance de signal est envoyée à une instance du même ensemble d'instances d'agent, l'interprétation du nœud *Output-node* implique que le signal est directement mis dans l'accès d'entrée de l'agent de destination, ou que le signal est envoyé au moyen d'un canal sans retard qui relie l'ensemble d'instances d'agent à lui-même.

Une procédure distante ou une variable distante placée sur un canal est mentionnée comme sortant d'un importateur et entrant dans un exportateur.

### Modèle

Si le nom `<channel name>` est omis dans une définition `<channel definition>` ou une zone `<channel definition area>`, le canal est nommé de manière implicite et unique.

Si un agent ou un type d'agent contient des accès explicites ou implicites qui ne sont pas reliés par des canaux explicites, des canaux implicites sans listes de signaux sont établis. Ensuite, on introduit des signaux, des procédures distantes et des variables distantes dans les canaux sans listes de signaux explicites. De plus amples détails sont présentés dans l'Annexe F.

Les trajets `<channel path>` définis explicitement possèdent des listes de signaux non vides après ces transformations.

Un canal dont les deux extrémités sont des accès d'une définition `<textual typebased agent definition>` représente des canaux individuels reliant chacun des agents contenus dans cet ensemble à tous les agents de l'ensemble, y compris l'agent d'origine. Tout canal bidirectionnel qui en résulte reliant un agent dans l'ensemble à l'agent lui-même est divisé en deux canaux unidirectionnels.

## 10.2 Connexion

### Grammaire textuelle concrète

```
<channel to channel connection> ::=  
    connect <external channel identifiers>  
    and <channel identifiers> <end>  
  
<external channel identifiers> ::=  
    <channel identifier> { , <channel identifier> } *  
  
<channel identifiers> ::=  
    <channel identifier> { , <channel identifier> } *
```

Aucun canal ne peut être mentionné après le mot clé **and** dans plus d'une connexion `<channel to channel connection>` d'une unité de portée donnée.

Pour toute paire de connexions `<channel to channel connection>` d'une unité de portée donnée, les identificateurs `<external channel identifiers>` doivent mentionner le même ensemble de canaux ou ne doivent pas avoir de canal en commun. Si au moins deux connexions `<channel to channel connection>` d'une unité de portée donnée ont le même ensemble de canaux externes, ceci sert de syntaxe dérivée pour une connexion unique `<channel to channel connection>` qui utilise l'un des

identificateurs <external channel identifiers> comme son identificateur <external channel identifiers> et ses identificateurs <channels identifiers> formés en énumérant tous les identificateurs internes <channel identifier>.

#### *Grammaire graphique concrète*

Sur le plan graphique, la construction connexion est représentée par la connexion graphique d'un symbole <channel symbol> dans une zone <channel definition area> à un identificateur <external channel identifiers> dans un diagramme <gate on diagram>.

Aucun identificateur <channel identifier> ne doit être mentionné plus d'une fois dans les connexions d'un diagramme.

NOTE – A cause de l'élément **connect**, un canal (externe) pouvant être anonyme dans la version graphique d'une spécification, peut nécessiter l'attribution d'un nom dans la version textuelle correspondante. Ceci est rigoureusement similaire au cas des zones <merge area> dans les graphes de processus ou de procédure. Il convient par conséquent qu'un outil permettant de convertir la version graphique d'une spécification en version textuelle puisse générer des noms de canaux implicites.

#### *Sémantique*

Les identificateurs <channel identifier> dans la partie <external channel identifiers> d'une connexion <channel to channel connection> doivent désigner des canaux reliés à l'agent englobant. Chaque canal relié à l'agent englobant doit être mentionné dans la partie <external channel identifiers> d'au moins une connexion <channel to channel connection>.

Chaque canal identifié par un identificateur <channel identifier> dans la partie <channel identifiers> d'une connexion <channel to channel connection> doit être défini dans le même agent dans lequel est définie la connexion <channel to channel connection> et il doit avoir la frontière de cet agent comme l'une de ses extrémités. Chaque canal défini dans l'agent qui l'entoure et qui a son environnement comme étant l'une de ses extrémités, doit être mentionné dans la partie <channel identifiers> d'une seule et unique connexion <channel to channel connection>.

#### *Modèle*

Les connexions sont des éléments abrégés et sont transformées en accès.

Chaque connexion <channel to channel connection> définit un accès implicite sur l'unité de portée donnée. Tous les canaux dans la connexion <channel to channel connection> sont reliés à cet accès dans leurs unités de portée respectives. Les contraintes d'accès de l'accès implicite sont obtenues à partir des canaux reliés à l'accès.

Le nom de l'accès est un nom dérivé unique et non ambigu. Dans l'unité de portée qui l'entoure, la définition <channel definition> qui est définie par l'identificateur <channel identifier> est étendue avec une partie de l'accès <via gate>. La partie de l'accès <via gate> est ajoutée à l'extrémité <channel endpoint> qui référence l'unité de portée courante et elle mentionne l'accès implicite. A l'intérieur de l'unité de portée, les canaux qui sont associés au canal externe au moyen de la connexion <channel to channel connection> sont modifiés par extension de l'extrémité <channel endpoint> qui mentionne **env** avec une partie de l'accès <via gate> pour l'accès implicite.

### **10.3 Signal**

#### *Grammaire abstraite*

<i>Signal-definition</i>	::	<i>Signal-name</i> <i>Sort-reference-identifier*</i>
<i>Signal-identifier</i>	=	<i>Identifier</i>
<i>Signal-name</i>	=	<i>Name</i>

### Grammaire textuelle concrète

```
<signal definition> ::=
    <type preamble>
    signal <signal definition item> { , <signal definition item> } * <end>

<signal definition item> ::=
    <signal name>
    [<formal context parameters>]
    [<virtuality constraint>]
    [<specialization>]
    [<sort list>]

<sort list> ::=
    ( <sort> { , <sort> } * )

<signal reference> ::=
    <type preamble>
    signal <signal identifier> <type reference properties>
```

Le paramètre <formal context parameter> dans les paramètres <formal context parameters> doit être un paramètre <sort context parameter>. Le type <base type> qui fait partie de la <specialization> doit être un identificateur <signal identifier>.

Un signal abstrait ne peut être utilisé que dans des contraintes de spécialisation et de signal.

### Grammaire graphique concrète

```
<signal reference area> ::=
    <type reference area>
```

La zone <type reference area> qui fait partie d'une zone <signal reference area> doit avoir un en-tête <graphical type reference heading> qui contient un nom <signal name>.

### Sémantique

Une instance de signal est un flot d'informations entre des agents; c'est une instanciation d'un type de signal défini par une définition de signal. Une instance de signal peut être envoyée par l'environnement ou par un agent, elle se dirige toujours vers un agent ou vers l'environnement. Une instance de signal est créée lorsqu'un nœud *Output-node* est interprété et cesse d'exister lorsqu'un nœud *Input-Node* est interprété.

La sémantique de <virtuality> est définie en 8.3.2.

## 10.4 Définition de liste de signaux

Un identificateur <signal list identifier> peut être utilisé dans une liste <signal list> comme moyen abrégé pour énumérer les identificateurs de signaux, les procédures distantes, les variables distantes, les signaux de temporisation et les interfaces.

### Grammaire textuelle concrète

```
<signal list definition> ::=
    signallist <signal list name> <equals sign> <signal list> <end>

<signal list> ::=
    <signal list item> { , <signal list item> } *

<signal list item> ::=
    <signal identifier>
    | ( <signal list identifier> )
    | <timer identifier>
    | [ procedure ] <remote procedure identifier>
    | [ interface ] <interface identifier>
    | [ remote ] <remote variable identifier>
```

La liste <signal list>, qui est établie en remplaçant tous les identificateurs <signal list identifieur> dans la liste par la liste des identificateurs <signal identifieur>s ou <timer identifieur> qu'ils désignent, et en remplaçant tous les identificateurs <remote procedure identifieur> et tous les identificateurs <remote variable identifieur> par l'un des signaux implicites que chacun d'eux désigne (voir 10.5 et 10.6), correspond à un ensemble *Signal-identifieur-set* dans la *grammaire abstraite*.

Un élément <signal list item> qui est un identifieur <identifieur> désigne un identifieur <signal identifieur> ou <timer identifieur> si cela est possible conformément aux règles de visibilité, ou bien un identifieur <remote procedure identifieur> si cela est possible conformément aux règles de visibilité, ou bien il désigne un identifieur <remote variable identifieur>. Pour forcer un élément <signal list item> à désigner un identifieur <remote procedure identifieur>, un <interface identifieur> ou un <remote variable identifieur>, les mots clés **procedure**, **interface** ou **remote** peuvent être utilisés respectivement.

La liste <signal list> ne doit pas contenir l'identifieur <signal list identifieur> défini par la définition <signal list definition> soit directement ou indirectement (par le biais d'un autre identifieur <signal list identifieur>).

Il est permis d'utiliser **this** dans les listes <signal list> qui font partie des contraintes <gate constraint> (voir 8.1.6).

### Grammaire graphique concrète

```
<signal list area> ::=
    <signal list symbol> contains <signal list>

<signal list symbol> ::=
    [ ]
```

## 10.5 Procédures distantes

Un agent client peut appeler une procédure définie dans un autre agent, par demande auprès de l'agent serveur au moyen d'un appel de procédure distante dans l'agent serveur.

### Grammaire textuelle concrète

```
<remote procedure definition> ::=
    remote procedure <remote procedure name> [nodelay]
    <procedure signature> <end>

<remote procedure call> ::=
    call <remote procedure call body>

<remote procedure call body> ::=
    <remote procedure identifieur> [<actual parameters>]
    <communication constraints>

<communication constraints> ::=
    {to <destination> | timer <timer identifieur> | <via path>}*
```

L'utilisation de **nodelay** est décrite en 10.1.

L'identifieur <remote procedure identifieur> suivant **as** dans une définition de procédure exportée désigne une définition <remote procedure definition> avec la même signature que celle de la procédure exportée. Dans une définition de procédure exportée sans clause **as**, le nom de la procédure exportée est implicite et la définition <remote procedure definition> dans la portée qui l'entoure la plus proche avec le même nom est implicite.

Une procédure distante mentionnée dans un appel <remote procedure call> doit être dans l'ensemble de sortie complet (voir Annexe F) d'un type d'agent ou d'un ensemble d'agents englobant.

Si une définition <destination> dans un corps <remote procedure call body> est une expression <pid expression> avec une sorte autre que Pid (voir 12.1.6), l'identificateur <remote procedure identifier> doit représenter une procédure distante contenue dans l'interface ayant défini la sorte pid.

Lorsque la <destination> et le trajet <via path> sont omis, il y a une ambiguïté syntaxique entre <remote procedure call body> et <procedure call>. Dans ce cas, l'identificateur <identifier> contenu désigne un identificateur <procedure identifier> si cela est possible conformément aux règles de visibilité et, autrement, un identificateur <remote procedure identifier>.

L'identificateur <timer identifier> d'une contrainte <communication constraints> ne doit pas avoir le même identificateur <identifier> qu'un identificateur <exception identifier>.

Dans un corps <remote procedure call body>, une liste de <communication constraints> est associée au dernier identificateur <remote procedure identifier>. Par exemple, dans

**call** p **to** **call** q **timer** t **via** g

le temporisateur **timer** t ainsi que l'accès **gate** g s'appliqueraient à l'appel **call** de q.

Une contrainte <communication constraints> ne doit pas contenir plus d'une destination <destination> et plus d'un identificateur <timer identifier>.

### *Grammaire graphique concrète*

<remote procedure call area> ::=  
    <procedure call symbol> **contains** <remote procedure call body>  
    [ **is connected to** <on exception association area> ]

### *Sémantique*

Une définition <remote procedure definition> introduit le nom et la signature pour les procédures importées et exportées.

Une procédure exportée est une procédure contenant le mot clé **exported**.

L'association entre une procédure importée et une procédure exportée est établie par le fait que les deux font référence à la même définition <remote procedure definition>.

Un appel de procédure distante par un agent demandeur conduit ce dernier à attendre que l'agent serveur exécute la procédure. Les signaux envoyés à l'agent demandeur pendant qu'il est en attente sont sauvegardés. L'agent serveur interprétera la procédure demandée dans l'état suivant où la sauvegarde de la procédure n'est pas spécifiée, selon l'ordre normal de réception des signaux. Si aucune partie <save part> ni aucune partie <input part> n'est spécifiée pour un état, une transition implicite qui consiste uniquement en un appel de procédure ramenant au même état est ajoutée. Si une partie <input part> est spécifiée pour un état, une transition implicite consistant en l'appel de procédure suivi de la <transition> est ajoutée. Si une partie <save part> est spécifiée pour un état, une sauvegarde implicite du signal pour la procédure demandée est ajoutée.

### *Modèle*

Un appel de procédure distante

**call** Proc(apar) **to** destination **timer** timerlist **via** viapath

est modélisé par un échange de signaux implicitement définis. Si les clauses **to** ou **via** sont omises dans l'appel de procédure distante, elles sont également omises dans les transformations suivantes. Les canaux sont explicites si la procédure distante a été mentionnée dans la liste <signal list> (sortante pour l'importateur et entrante pour l'exportateur) d'au moins un accès ou un canal relié à l'importateur ou à l'exportateur. Lorsqu'une procédure distante est acheminée sur des canaux explicites, le mot clé **nodelay** de la définition <remote procedure definition> est ignoré. L'agent demandeur envoie un signal contenant les paramètres réels de l'appel de procédure, à l'exception des paramètres réels correspondant aux paramètres **out**, aux agents du serveur et attend la réponse. En réponse à ce signal, l'agent serveur interprète la procédure distante correspondante, envoie un signal

de retour à l'agent demandeur avec les résultats de tous les paramètres **in/out** et des paramètres **out**, puis interprète la transition.

Il existe deux définitions implicites <signal definition> pour chaque définition <remote procedure definition> dans une définition <system definition>. Les noms <signal name> dans ces définitions <signal definition> sont désignés respectivement par pCALL et pREPLY, où p est déterminé de façon unique. Les signaux sont définis dans la même unité de portée que la définition <remote procedure definition>. Le premier paramètre de pCALL et de pREPLY appartient à la sorte des entiers (Integer) prédéfinie.

Sur chaque canal mentionnant la procédure distante, celle-ci est remplacée par *pCALL*. Pour chacun de ces canaux, un nouveau canal est ajouté dans la direction opposée, il transporte le signal *pREPLY*. Le nouveau canal a les mêmes propriétés relatives au retard que le canal d'origine.

- a) pour chaque procédure importée, deux variables entières implicites *n* et *newn* sont définies, et *n* est initialisée à 0.

NOTE 1 – Le paramètre *n* est introduit pour reconnaître et éliminer des signaux de réponse d'appels de procédure distante qui ont été laissés par le biais d'une durée de validité de temporisation associée.

L'appel <remote procedure call> est transformé comme suit.

```
task n:= n + 1;
output pCALL(apar,n) to destination via viapath;
wait in state pWAIT, saving all other signals;
input pREPLY(aINOUTpar,newn);
decision newn = n;
(true):
(false): nextstate pWAIT;
enddecision;
return;
```

Lorsque *apar* est la liste des paramètres réels, excepté les paramètres correspondant aux paramètres *out*, et que *aINOUTpar* est la liste modifiée des paramètres *in/out* réels et des paramètres *out*, y compris un paramètre supplémentaire si un appel de procédure distante retournant une valeur est transformé.

NOTE 2 – L'instruction de retour termine la procédure implicite introduite conformément à 11.12.1.

Pour chaque exception contenue dans <raises> d'une procédure distante *p* et toutes les exceptions *e* prédéfinies, un signal *eRAISE* est défini qui peut transporter tous les paramètres d'exception de *e*. Les éléments suivants seront insérés dans l'état *pWAIT*:

```
state pWAIT;
input eRAISE(params,newn);
decision newn = n;
(true): raise e(params);
(false): nextstate pWAIT;
enddecision;
```

Pour un temporisateur *t* inclus dans la contrainte <communication constraints>, une exception supplémentaire avec le même nom et les mêmes paramètres est insérée implicitement dans la même portée dans la définition de temporisateur, et il ne doit pas y avoir d'exception explicitement définie avec le même nom que le temporisateur dans l'unité de portée dans laquelle le temporisateur est défini.

De plus, les éléments suivants seront insérés pour un temporisateur *t* inclus dans la contrainte <communication constraints>

```
state pWait;
input t(aParams);
raise t(aParams);
```

où *aParams* correspond à des variables implicitement définies avec la sorte des paramètres contenue dans la définition de temporisateur.



Dans tous les états de l'agent, excepté pWAIT

```
input pReply, eRAISE;  
nextstate actual state;
```

sont insérés;

- b) dans l'agent serveur, un gestionnaire d'exception implicite pEXC et une variable entière implicite n sont définis pour chaque partie explicite ou implicite <input part> étant une entrée de procédure distante. De plus, il existe une variable pour chacune de ces <input part> définies dans la portée dans laquelle apparaît la procédure distante explicite ou implicite. Si un appel de procédure distante retournant une valeur est transformé, une variable implicite res avec la même sorte que la <sort> dans le résultat <procedure result> est définie.

La partie <input part> suivante est ajoutée à tous les états <state> ayant une transition d'entrée de procédure distante:

```
input pCALL(fpar,n);  
task ivar:= sender;  
call Proc(fpar); onexception pEXC;  
output pREPLY(INOUTpar,n) to ivar;  
transition;
```

ou,

```
input pCALL(fpar,n);  
task ivar:= sender;  
task res := call Proc(fpar); onexception pEXC;  
output pREPLY(INOUTpar,res,n) to ivar;  
transition;
```

Si un appel de procédure distante retournant une valeur a été transformé.

La partie <save part> suivante est ajoutée à tous les états <state> ayant une sauvegarde de procédure distante:

```
save pCall;
```

La partie <input part> suivante est ajoutée à tous les états <state> ayant un rejet <remote procedure reject>:

```
input pCALL;  
output eRAISE(params,n) to sender;  
transition;
```

On ajoute la partie <input part> à tous les autres états <state> exception faite des états implicites obtenus à partir d'une importation:

```
input pCALL(fpar,n);  
task ivar:= sender;  
call Proc(fpar); onexception pEXC;  
output pREPLY(INOUTpar,n) to ivar;  
/* next state the same */
```

Pour chaque exception e contenue dans les <raises> de la procédure distante et pour chaque exception prédéfinie, les éléments suivants sont insérés:

```
exceptionhandler pEXC;  
handle e(params);  
output eRAISE(params,n) to ivar;  
raise e(params);
```

Si un gestionnaire d'exception est associé à une importation de procédure distante, il devient associé à l'importation de signaux qui en résulte (n'apparaît pas dans le modèle ci-dessus).

NOTE 3 – Un blocage fatal peut se produire en utilisant la construction de procédure distante, particulièrement si aucune <destination> n'est donnée, ou si <destination> ne désigne pas une expression <pid expression> d'un agent dont la spécification garantit l'existence au moment de la réception du signal pCALL. Les temporisateurs associés permettent d'empêcher les blocages fatals.

## 10.6 Variables distantes

Dans le SDL, une variable appartient toujours à une instance d'agent dont elle est une variable locale. En général, la variable est visible uniquement à l'instance d'agent qui la possède et aux agents contenus. Si une instance d'agent d'un autre agent doit accéder aux éléments de données associés à une variable, on a besoin pour cela d'un échange de signaux avec l'instance d'agent à laquelle cette variable appartient.

Cette opération peut être réalisée en utilisant l'abréviation variable importée et exportée. Cette abréviation peut également être utilisée pour exporter des éléments de données vers d'autres instances d'agent à l'intérieur du même agent.

### Grammaire textuelle concrète

```
<remote variable definition> ::=
    remote <remote variable name> {,<remote variable name>}* <sort> [ nodelay ]
        {,<remote variable name> {,<remote variable name>}* <sort> [ nodelay ]}*
        <end>

<import expression> ::=
    import ( <remote variable identifier> <communication constraints> )

<export> ::=
    export ( <variable identifier> { , <variable identifier> }* )
```

L'utilisation de **nodelay** est décrite dans l'Annexe F.

L'identificateur <remote variable identifier> qui suit **as** dans une définition de variable exportée doit désigner une définition <remote variable definition> de la même sorte que la définition de variable exportée. En cas d'absence de la clause **as**, la définition de variable distante dans l'unité de portée englobante la plus proche avec le même nom et la même sorte que la définition de variable exportée est désignée.

Une variable distante mentionnée dans une expression <import expression> doit être dans l'ensemble de sortie complet (voir Annexe F) d'un type d'agent ou d'un ensemble d'agents englobant.

L'identificateur <variable identifier> dans <export> doit désigner une variable définie avec **exported**.

Si la <destination> dans une expression <import expression> est une expression <pid expression> avec une sorte autre que Pid (voir 12.1.6), l'identificateur <remote variable identifier> doit représenter une variable distante contenue dans l'interface ayant défini la sorte pid.

Il n'y a pas de syntaxe graphique correspondante pour <export>.

### Sémantique

Une définition <remote variable definition> introduit le nom et la sorte pour les procédures importées et exportées.

Une définition de variable exportée est une définition de variable avec le mot clé **exported**.

L'association entre une variable importée et une variable exportée est établie par le fait que les deux font référence à la même définition <remote variable definition>.

Les variables importées sont spécifiées comme faisant partie de l'ensemble de sortie de l'entité active englobante. Les variables exportées sont spécifiées comme faisant partie de l'ensemble de sortie de l'entité active englobante.

L'instance d'agent à laquelle appartient une variable dont les éléments de données sont exportés vers d'autres instances d'agent est appelée exportateur de la variable. Les autres instances d'agent qui utilisent ces éléments de données sont les importateurs de la variable. La variable est appelée variable exportée.

Une instance d'agent peut être à la fois l'importateur et l'exportateur de la même variable distante.

a) *opération d'export*

Les variables exportées ont le mot clé **exported** dans leurs définitions <variable definition> et ont une copie implicite qu'elles utilisent dans les opérations d'import.

Une opération d'export est l'interprétation d'un <export> par lequel un exportateur divulgue le résultat courant d'une variable exportée. Une opération d'export provoque le stockage du résultat courant de la variable exportée dans sa copie implicite;

b) *opération d'import*

Une opération d'import est l'interprétation d'une expression <import expression> grâce à laquelle un importateur accède au résultat d'une variable exportée. Le résultat est stocké dans une variable implicite désignée par l'identificateur <remote variable identifier> dans l'expression <import expression>. L'exportateur contenant la variable exportée est spécifié par la <destination> dans l'expression <import expression>. Si aucune <destination> n'est spécifiée, l'importation provient alors d'une instance d'agent arbitraire exportant la même variable distante. L'association entre la variable exportée dans l'exportateur et la variable implicite dans l'importateur est spécifiée par leur référence commune à la même variable distante dans la définition de variable exportée et l'expression <import expression>

*Modèle*

Une opération d'import est modélisée par l'échange de signaux définis implicitement. Lorsqu'une procédure distante est acheminée sur des canaux explicites, le mot clé **nodelay** de la définition <remote variable definition> est ignoré. L'importateur envoie un signal vers l'exportateur et attend la réponse. En réponse à ce signal, l'exportateur renvoie un signal vers l'importateur avec le même résultat qui se trouve contenu dans la copie implicite de la variable exportée.

Si une initialisation par défaut est associée à la variable d'export ou si la variable d'export est initialisée lorsqu'elle est définie, la copie implicite est aussi initialisée, avec le même résultat que la variable d'export.

Il y a deux définitions <signal definition> implicites pour chaque définition <remote variable definition> dans une définition de système. Les noms <signal name> dans ces définitions <signal definition> sont désignés respectivement par xQUERY et xREPLY, où x désigne le nom <name> de la définition <remote variable definition>. Les signaux sont définis dans la même unité de portée que la définition <remote variable definition>. Le signal xQUERY a un argument de la sorte des entiers prédéfinie et le signal xREPLY a des arguments de la sorte de la variable et de la sorte des entiers. La copie implicite de la variable exportée est appelée imcx.

Sur chaque canal mentionnant la variable distante, celle-ci est remplacée par xQUERY. Pour chacun de ces canaux, un nouveau canal est ajouté dans la direction opposée, il transporte le signal xREPLY. Dans le cas d'un canal, le nouveau canal a les mêmes propriétés relatives au retard que le canal d'origine.

Pour chaque exception prédéfinie (désignée predefExc), un signal anonyme supplémentaire (désigné predefExcRAISE) est défini.

a) *importateur*

Pour chaque variable importée, deux variables entières implicites n et newn sont définies, et n est initialisée à 0. De plus, une variable implicite x de la sorte de la variable distante est définie.

L'expression <import expression>  
**import** (x to destination via via-path)

est transformée en l'expression suivante, où la clause **to** est omise si la destination n'est pas présente, et où la clause **via** est omise si elle n'est pas présente dans l'expression d'origine:

```
task n:= n + 1;  
output xQUERY(n) to destination via via-path;  
wait in state xWAIT, saving all other signals;  
input xREPLY(x,newn);  
decision newn = n;  
(true):  
(false): nextstate xWAIT;  
enddecision;  
return;  
state xWAIT  
input predefExcRAISE;  
raise predefExc;
```

Dans tous les autres états, xREPLY est sauvegardé.

NOTE 1 – L'instruction de retour termine la procédure implicite introduite conformément à 11.12.1.

Pour chaque temporisateur t inclus dans la contrainte <communication constraints>, une exception supplémentaire ayant le même nom et les mêmes paramètres est implicitement insérée dans la même portée que la définition de temporisateur. Dans ce cas, il ne doit pas exister d'exception ayant le même nom dans l'unité de portée de la définition de temporisateur.

De plus, les éléments suivants seront insérés pour chaque temporisateur t inclus dans la contrainte <communication constraints>:

```
state xWait;  
input t(aParams);  
raise t(aParams);
```

où aParams correspond à des variables implicitement définies avec la sorte des paramètres contenue dans la définition de temporisateur.

Le résultat de la transformation est encapsulé dans une procédure implicite, comme décrit en 11.12.1. Chaque exception <on exception> associée à l'action d'import doit être associée à un appel de la procédure implicite;

b) *exporter*

A tous les états <state> de l'exportateur, exception faite des états implicites, obtenus à partir d'une importation, la partie <input part> suivante est ajoutée:

```
input xQUERY(n);  
task ivar := sender;  
output xREPLY(imcx,n) to ivar; onexception xEXC;  
nextstate the state containing this input;  
exceptionhandler xEXC;  
handle predefExc;  
output predefExcRAISE to ivar;  
raise predefExc;
```

Pour chacun de ces états, ivar sera définie comme une variable de sorte Pid et n comme une variable de type entier.

L' <export>

```
export x
```

est transformé en ce qui suit:

```
task imcx := x;
```

NOTE 2 – Un blocage fatal peut se produire en utilisant la construction d'import, particulièrement si aucune <destination> n'est donnée, ou si <destination> ne désigne pas une expression <pid expression> d'un agent dont la spécification garantit l'existence au moment de la réception du signal xQUERY. La spécification d'un temporisateur réglé dans l'expression <import expression> permet d'empêcher un tel blocage.

## 11 Comportement

### 11.1 Départ

#### Grammaire abstraite

*State-start-node* :: [*On-exception*]  
[*State-entry-point-name*]  
*Transition*

#### Grammaire textuelle concrète

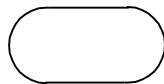
<start> ::= **start** [<virtuality>] [<state entry point name>] <end> [<on exception>] <transition>

Si le nom <state entry point name> est donné dans un départ <start>, le départ <start> doit être le départ <start> d'un état <composite state>.

#### Grammaire graphique concrète

<start area> ::= **<start symbol>** *contains* { [<virtuality>] [<state entry point name>] }  
[ *is connected to* <on exception association area> ]  
*is followed by* <transition area>

<start symbol> ::=



Si le nom <state entry point name> est donné dans un départ <start>, la zone <start area> doit être la zone <start area> d'un état <composite state>.

#### Sémantique

La transition du nœud *Transition* of the *State-start-node* est interprétée.

#### Modèle

Un départ <start> ou une zone <start area> qui contient <virtuality> est appelé départ virtuel. Le départ virtuel est décrit en détail en 8.3.3.

Un départ <start> ou une zone <start area> contenu dans un état <composite state> ou dans une zone <composite state area> est défini en 11.11.

### 11.2 Etat

#### Grammaire abstraite

*State-node* :: *State-name*  
[*On-exception*]  
*Save-signalset*  
*Input-node-set*  
*Spontaneous-transition-set*  
*Continuous-signal-set*  
[*Composite-state-type-identifier*]  
*State-name* = *Name*

Les nœuds *State-node* dans un graphe *State-transition-graph* ou *Procedure-graph* doivent avoir des noms *State-name* différents.

Pour chaque nœud *State-node*, tous les identificateurs *Signal-identifier* (dans l'ensemble complet des signaux d'entrée valides) apparaissent soit dans un ensemble *Save-signalset*, ou dans un nœud *Input-node*.

Les identificateurs *Signal-identifier* dans l'ensemble *Input-node-set* doivent être distincts.

Un nœud *State-node* ayant un identificateur *Composite-state-type-identifïer* représente une application <composite state application>.

#### Grammaire textuelle concrète

```
<state> ::=
    <basic state> | <composite state application>
```

Une application <composite state application> représente un état avec un identificateur *Composite-state-type-identifïer*.

#### Grammaire graphique concrète

```
<state area> ::=
    <basic state area> | <composite state application area>
```

Une zone <composite state area> représente un état avec un identificateur *Composite-state-type-identifïer*.

### 11.2.1 Etat de base

#### Grammaire textuelle concrète

```
<basic state> ::=
    state <state list> <end> [ <on exception> ]
        {
        |   <input part>
        |   <priority input>
        |   <save part>
        |   <spontaneous transition>
        |   <continuous signal> }*
    [ endstate [ <state name> ] <end> ]

<state list> ::=
    <state name> { , <state name> }*
    |
    <asterisk state list>

<asterisk state list> ::=
    <asterisk> [ ( <state name> { , <state name> }* ) ]
```

Un état donné ne peut pas avoir plus d'un gestionnaire d'exception associé.

Lorsque la liste <state list> contient un nom <state name>, alors le <state name> représente un nœud *State-node*. Pour chaque *State-node*, l'ensemble *Save-signalset* est représenté par la partie <save part> et toute sauvegarde de signal d'entrée implicite. Pour chaque *State-node*, l'ensemble *Input-node-set* est représenté par la partie <input part> et tout signal d'entrée implicite. Pour chaque *State-node*, une transition *Spontaneous-transition* est représentée par une transition <spontaneous transition>.

Un nom <state name> peut apparaître dans plusieurs états <state> d'un corps.

Les noms <state name> dans une liste <asterisk state list> doivent être distincts et contenus dans d'autres listes <state list> dans le corps englobant ou dans le corps d'un supertype.

Le nom <state name> facultatif terminant un état <state> peut être spécifié seulement si la liste <state list> dans l'état <state> consiste en un seul <state name>, auquel cas il doit s'agir de ce même <state name>.

## Grammaire graphique concrète

<basic state area> ::=

```
<state symbol> contains <state list>
[ is connected to <on exception association area> ]
is associated with
  {
  | <input association area>
  | <priority input association area>
  | <continuous signal association area>
  | <spontaneous transition association area>
  | <save association area> }*
```

<state symbol> ::=



<input association area> ::=

<solid association symbol> **is connected to** <input area>

<save association area> ::=

<solid association symbol> **is connected to** <save area>

<spontaneous transition association area> ::=

<solid association symbol> **is connected to** <spontaneous transition area>

Une zone <state area> représente un ou plusieurs nœuds *State-node*.

Les symboles <solid association symbol> provenant d'un symbole <state symbol> peuvent avoir un trajet d'origine commun.

### Sémantique

Un état représente une condition particulière dans laquelle la machine à états d'un agent peut traiter une instance de signal. Si une instance de signal est traitée, la transition associée est interprétée. Une transition peut également être interprétée comme le résultat d'un signal continu ou d'une transition spontanée.

Pour chaque état, les signaux *Save-signal*, les nœuds *Input-node*, les signaux *Spontaneous-signal* et les signaux *Continuous-signal* sont interprétés dans l'ordre suivant:

- a) si un accès d'entrée contient un signal correspondant à une entrée prioritaire de l'état courant, le premier signal est traité (voir 11.4). Autrement,
- b) dans l'ordre des signaux sur l'accès d'entrée,
  - 1) les expressions *Provided-expression* du nœud *Input-node* correspondant au signal courant sont interprétées dans un ordre aléatoire, si elles existent,
  - 2) si le signal courant est validé, ce signal est traité (voir 11.6). Autrement,
  - 3) le signal suivant sur l'accès d'entrée est sélectionné;
- c) si aucun signal validé n'a été trouvé, dans l'ordre de priorité des signaux *Continuous-signal*, s'ils existent, avec les signaux *Continuous-signal* de priorité égale traité dans un ordre aléatoire et aucune priorité traitée en tant que priorité inférieure:
  - 1) l'expression *Continuous-expression* contenue dans le signal *Continuous-signal* courant est interprétée,
  - 2) si le signal continu courant est validé, ce signal est traité (voir 11.5). Autrement,
  - 3) le signal continu suivant est sélectionné;
- d) si aucun signal validé n'a été trouvé, la machine à états attend dans l'état jusqu'à ce qu'une autre instance de signal soit reçue. Si l'état a des conditions de validation ou des signaux continuent, ces étapes sont répétées même si aucun signal n'est reçu.

A tout moment dans un état qui contient des transitions *Spontaneous-transition*, la machine à états peut interpréter les expressions *Provided-expression* d'une transition *Spontaneous-transition* et, par la suite, si la transition *Spontaneous-transition* a été validée, la *Transition* de l'une des transitions *Spontaneous-transition* (voir 11.9), ou la *Transition* de l'une des transitions *Spontaneous-transition* s'il n'y a pas d'expression *Provided-expression*.

### Modèle

Lorsque la liste <state list> d'un état <state> contient plus d'un nom <state name>, une copie de cet état <state> est créée pour chacun de ces noms <state name>. Ensuite l'état <state> est remplacé par ces copies.

Lorsque plusieurs états <state> contiennent le même nom <state name>, ces états <state> sont concaténés dans un état <state> ayant ce nom <state name>.

Un état <state> avec une liste <asterisk state list> est transformé en une liste d'états <state>, un pour chaque nom <state name> du corps en question, excepté pour les noms <state name> contenus dans la liste <asterisk state list>.

## 11.2.2 Application d'état composite

Une application <composite state application> indique que la machine à états a un état composite. Les propriétés de l'état composite sont définies soit comme une partie de l'application <composite state application>, par un état composite référencé, ou par un type d'état composite.

### Grammaire textuelle concrète

```
<composite state application> ::=
    state <composite state list> <end> [<on exception>]
        {
            | <input part>
            | <priority input>
            | <save part>
            | <spontaneous transition>
            | <continuous signal>
            | <connect part> }*
    [ endstate [ <state name> ] <end> ]
```

Une référence <composite state reference> au même état composite ne doit apparaître que dans l'une des applications <composite state application> de la machine à états qui l'entoure.

Un nœud *State-node* ayant un identificateur *Composite-state-type-identififier* représente une application <composite state application>.

### Grammaire graphique concrète

```
<composite state application area> ::=
    <state symbol> contains <composite state list>
    [ is connected to <on exception association area> ]
    is associated with
        {
            | <input association area>
            | <priority input association area>
            | <continuous signal association area>
            | <spontaneous transition association area>
            | <save association area>
            | <connect association area> }*

<connect association area> ::=
    <solid association symbol> is connected to <connect area>

<composite state list> ::=
    { <state list token> { , <state list token> }* }
    | { <asterisk state list> }
```



```

<state list token> ::=
    <state name> [<actual parameters>]
    | <typebased composite state>

```

Une marque <state list token> ne doit contenir que des paramètres <actual parameters> si la zone <state area> coïncide avec une zone <nextstate area>. Dans ce cas, la zone <state area> ne doit contenir qu'un nom <state name> et, de manière facultative, des paramètres <actual parameters>.

### Sémantique

La sémantique d'une application <composite state application> est définie en 11.11.

## 11.3 Entrée

### Grammaire abstraite

```

Input-node          ::  [PRIORITY]
                       Signal-identifiant
                       [Variable-identifiant]*
                       [Provided-expression]
                       [On-exception]
                       Transition
Variable-identifiant =  Identifiant

```

La longueur de la liste des identificateurs *Variable-identifiant* facultatifs doit être égale au nombre d'identificateurs *Sort-reference-identifiants* de la définition *Signal-définition* désigné par l'identificateur *Signal-identifiant*.

Les sortes de variables doivent correspondre par leur position aux sortes des éléments de données qui peuvent être acheminés par le signal.

### Grammaire textuelle concrète

```

<input part> ::=
    input [<virtuality>] <input list> <end>
    [<on exception>] [<enabling condition>] <transition>
<input list> ::=
    <stimulus> { , <stimulus> } *
    | <asterisk input list>
<stimulus> ::=
    <signal list item>
    [ ( [ <variable> ] { , [ <variable> ] } * ) | <remote procedure reject> ]
<remote procedure reject> ::=
    raise <exception raise>
<asterisk input list> ::=
    <asterisk>

```

Un état <state> peut contenir au plus une liste <asterisk input list>. Un état <state> ne peut contenir à la fois une liste <asterisk input list> et une liste <asterisk save list>.

Un rejet <remote procedure reject> peut être spécifié uniquement si l'élément <signal list item> désigne un identificateur <remote procedure identifiant>. L'identificateur <exception identifiant> dans le rejet <remote procedure reject> doit être mentionné dans la définition <remote procedure definition>.

Un élément <signal list item> ne doit pas désigner un identificateur <remote variable identifiant> et, s'il désigne un identificateur <remote procedure identifiant> ou un identificateur <signal list identifiant>, les paramètres <stimulus> (y compris les parenthèses) doivent être omis.

Lorsque la liste <input list> contient un <stimulus>, la partie <input part> représente un nœud *Input-node*. Dans la *grammaire abstraite*, les signaux de temporisation (<timer identifiant>) sont

également représentés par l'identificateur *Signal-identifieur*. En raison de leurs propriétés voisines à beaucoup d'égards, la distinction entre les signaux de temporisation et les signaux courants n'est faite qu'en cas de nécessité. Les propriétés exactes des signaux de temporisation sont décrites en 11.15.

Les virgules peuvent être omises après la dernière variable <variable> dans le <stimulus>.

Dans la *grammaire abstraite*, les identificateurs <remote procedure identifier> sont également représentés par des identificateurs *Signal-identifieur*.

### Grammaire graphique concrète

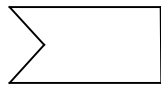
<input area> ::=

<input symbol> *contains* { [<virtuality>] <input list> }  
 [ *is connected to* <on exception association area> ]  
 [ *is associated with* <solid association symbol> *is connected to* <enabling condition area> ]  
*is followed by* <transition area>

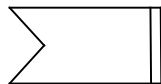
<input symbol> ::=

<plain input symbol>  
 |  
 <internal input symbol>

<plain input symbol> ::=



<internal input symbol> ::=



Une zone <input area> dont la liste <input list> contient un <stimulus> correspond à un nœud *Input-node*. Chacun des identificateurs <signal identifier> ou <timer identifier> contenus dans un symbole <input symbol> donne le nom de l'un des nœuds *Input-node* que ce symbole <input symbol> représente.

NOTE – Il n'y a pas de différence de signification entre un symbole <plain input symbol> et un symbole <internal input symbol>.

### Sémantique

Une entrée permet le traitement de l'instance de signal d'entrée spécifiée. Le traitement du signal d'entrée rend l'information véhiculée par le signal disponible pour l'agent. Aux variables associées à l'entrée, on affecte les éléments de données acheminés par le signal utilisé.

Les éléments de données seront assignés aux variables de gauche à droite. Si aucune variable n'est associée à l'entrée pour une sorte spécifiée dans le signal, l'élément de données correspondant est ignoré. S'il n'y a pas d'élément de données associé à une sorte spécifiée dans le signal, la variable correspondante devient "non définie".

A l'expression sender de l'agent récepteur (voir paragraphe 9 *Modèle*) est donnée la valeur pid de l'agent d'origine, acheminée par l'instance de signal.

Les instances de signal circulant de l'environnement vers une instance d'agent dans le système achemineront toujours une valeur pid différente de toutes les valeurs dans le système.

### Modèle

Un <stimulus>, dont l'élément <signal list item> est un identificateur <signal list identifier>, est une syntaxe dérivée pour une liste de <stimulus> sans paramètres et est inséré dans la liste englobante <input list> ou <priority input list>). Dans cette liste, il existe une correspondance biunivoque entre les <stimulus> et les membres de la liste de signaux.

Lorsque la liste des <stimulus> d'une partie <input part> contient plus d'un <stimulus>, une copie de la partie <input part> est créée pour chacun de ces <stimulus>. Ensuite, la partie <input part> est remplacée par ces copies.

Lorsqu'une ou plusieurs variables <variable> d'un certain <stimulus> sont des variables <indexed variable> ou <field variable>, toutes les variables sont remplacées par des identificateurs <variable identifier> uniques, nouveaux et déclarés implicitement. Une tâche <task> est insérée immédiatement après la partie <input part>; cette tâche contient dans son corps <textual task body> une affectation <assignment> pour chacune des variables <variable>, affectant à la <variable> le résultat de la nouvelle variable correspondante. Les résultats sont affectés dans l'ordre de gauche à droite de la liste des variables <variable>. Cette tâche <task> devient la première instruction <action statement> dans la <transition>.

Une liste <asterisk input list> est transformée en une liste de parties <input part>, une pour chaque membre de l'ensemble complet de signaux d'entrée valides de la définition englobante <agent definition>, excepté pour les identificateurs <signal identifier> de signaux d'entrée implicites introduits par les concepts en 10.5, 10.6, 11.4, 11.5 et 11.6, et pour les identificateurs <signal identifier> contenus dans les autres listes <input list> et <save list> de l'état <state>.

Une partie <input part> ou une zone <input area> qui contient <virtuality> est appelée transition d'entrée virtuelle. Les transitions d'entrée virtuelles sont décrites en détail en 8.3.3.

## 11.4 Entrée prioritaire

Dans certains cas, il convient d'exprimer le fait que la réception d'un signal est prioritaire par rapport à d'autres signaux. Cela peut être exprimé au moyen de l'entrée prioritaire.

### Grammaire textuelle concrète

```
<priority input> ::=
    priority input [<virtuality>]
    <priority input list> <end> [<on exception>]<transition>

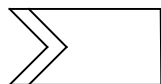
<priority input list> ::=
    <stimulus> {, <stimulus>}*
```

### Grammaire graphique concrète

```
<priority input association area> ::=
    <solid association symbol> is connected to <priority input area>

<priority input area> ::=
    <priority input symbol> contains { [ <virtuality> ] <priority input list> }
    [ is connected to <on exception association area> ]
    is followed by <transition area>

<priority input symbol> ::=
```



Une entrée <priority input> ou une zone <priority input association area> représente un nœud *Input-node* avec **PRIORITY**.

### Sémantique

Si un nœud *Input-node* d'un état a **PRIORITY**, le signal est un signal prioritaire et sera traité avant tout autre signal, sous réserve qu'il ait une transition validée.

### Modèle

Une entrée <priority input> ou une zone <priority input area> qui contient <virtuality> est appelée entrée prioritaire virtuelle. Elle est décrite plus en détail en 8.3.3.

## 11.5 Signal continu

Lorsqu'on décrit des systèmes, on peut être confronté à une situation dans laquelle on souhaiterait interpréter une transition lorsqu'une certaine condition est remplie. Un signal continu interprète une expression booléenne et la transition associée est interprétée lorsque l'expression renvoie la valeur booléenne prédéfinie Vrai (true).

### Grammaire abstraite

*Continuous-signal* ::= *Continuous-expression* [*Priority-name*] *Transition*  
*Continuous-expression* = *Boolean-expression*  
*Priority-name* = *Nat*

### Grammaire textuelle concrète

<continuous signal> ::=  
    **provided** [<virtuality>]  
        <continuous expression> <end>  
    [ **priority** <priority name> <end> ] [<on exception>] <transition>  
  
<continuous expression> ::=  
    <Boolean expression>  
  
<priority name> ::=  
    <Natural literal name>

### Grammaire graphique concrète

<continuous signal association area> ::=  
    <solid association symbol> **is connected to** <continuous signal area>  
  
<continuous signal area> ::=  
    <enabling condition symbol>  
    **contains** {  
        [<virtuality>] <continuous expression>  
        [ [<end>] **priority** <priority name> ] }  
    [ **is connected to** <on exception association area> ]  
    **is followed by** <transition area>

### Sémantique

L'expression *Continuous-expression* est interprétée avant d'entrer dans l'état auquel est associé son signal *Continuous-signal*, et tant que l'on attend dans l'état, chaque fois que l'on ne trouve dans l'accès d'entrée aucun <stimulus> d'une liste <input list> adjointe. Si l'expression *Continuous-expression* renvoie la valeur booléenne prédéfinie Vrai (true), le signal continu est validé.

Le signal continu ayant la valeur la plus faible pour le nom *Priority-name* présente la priorité la plus élevée.

### Modèle

Un signal <continuous signal> ou une zone <continuous signal area> qui contient <virtuality> est appelé(e) signal continu virtuel. Les transitions continues virtuelles sont décrites avec plus de détails en 8.3.3.

## 11.6 Condition de validation

Une condition de validation permet d'imposer une condition supplémentaire sur le traitement d'un signal, au-delà de sa réception, ainsi que sur une transition spontanée.

### Grammaire abstraite

*Provided-expression* = *Boolean-expression*

### Grammaire textuelle concrète

<enabling condition> ::=  
                                  **provided** <provided expression> <end>

<provided expression> ::=  
                                  <Boolean expression>

### Grammaire graphique concrète

<enabling condition area> ::=  
                                  <enabling condition symbol> **contains** <provided expression>

<enabling condition symbol> ::=  
                                  < >

Lorsque l'expression <provided expression> contient une expression <imperative expression>, l'expression *Provided-expression* est un nœud *Value-returning-call-node* contenant uniquement l'identificateur *Procedure-identifier* de la procédure implicitement définie par le *Modèle* ci-dessous. Autrement, l'expression *Provided-expression* est représentée par <provided expression>.

### Sémantique

L'expression *Provided-expression* d'un nœud *Input-node* est interprétée lors de l'entrée dans l'état auquel ce nœud *Input-node* est associé et chaque fois que l'on se retrouve dans cet état en raison de l'arrivée d'un <stimulus>. En cas de conditions de validation multiple, celles-ci sont interprétées séquentiellement dans un ordre arbitraire lors de l'entrée dans l'état.

Un signal dans l'accès d'entrée est validé si toutes les expressions *Provided-expression* d'un nœud *Input-node* renvoient la valeur booléenne prédéfinie Vrai (true), ou si le nœud *Input-node* n'a pas d'expression *Provided-expression*. L'expression *Provided-expression* d'une transition *Spontaneous-transition* peut être interprétée à tout moment lorsque l'agent est dans l'état.

### Modèle

Lorsque l'expression <provided expression> contient une expression <imperative expression>, la procédure suivante avec un nom anonyme désigné *isEnabled* est implicitement définie.

```
procedure isEnabled -> Boolean;  
  start ;  
  return <provided expression>;  
endprocedure ;
```

NOTE – L'expression <Boolean expression> peut être encore transformée conformément au modèle de l'expression <import expression>.

## 11.7 Sauvegarde

Une sauvegarde spécifie un ensemble d'identificateurs de signaux et d'identificateurs de procédures distantes dont les instances ne peuvent pas être traitées par l'état auquel la sauvegarde est associée, et qui nécessitent une sauvegarde pour un traitement ultérieur.

### Grammaire abstraite

*Save-signalset* :: *Signal-identifier-set*

### Grammaire textuelle concrète

<save part> ::=  
                                  **save** [<virtuality>] <save list> <end>

<save list> ::=  
                                  <signal list>  
                                  | <asterisk save list>

<asterisk save list> ::=

<asterisk>

Une liste <save list> représente l'ensemble *Signal-identifieur-set*.

Un état <state> peut contenir au plus une liste <asterisk save list>. Un état <state> ne peut contenir à la fois une liste <asterisk input list> et une liste <asterisk save list>.

### *Grammaire graphique concrète*

<save area> ::=

<save symbol> *contains* { [<virtuality>] <save list> }

<save symbol> ::=



### *Sémantique*

Un signal dans un ensemble Save-signalset n'est pas validé.

Les signaux sauvegardés sont bloqués à l'accès d'entrée dans l'ordre de leur arrivée.

La sauvegarde n'a d'effet que sur les états auxquels la sauvegarde est associée. Dans l'état suivant, les instances de signaux qui ont été "sauvegardées" sont traitées comme des instances de signaux normales.

### *Modèle*

Une liste <asterisk save list> est transformée en une liste de <stimulus> contenant l'ensemble complet de signaux d'entrée valides de la définition englobante <agent definition>, excepté pour les identificateurs <signal identifieur> de signaux d'entrée implicites introduits par les concepts en 10.5, 10.6, 11.4, 11.5 et 11.6 et pour les identificateurs <signal identifieur> contenus dans les autres listes <input list> et <save list> de l'état <state>.

Une partie <save part> ou une zone <save area> qui contient <virtuality> est appelée sauvegarde virtuelle. Cette dernière est décrite en 8.3.3.

## **11.8 Transition implicite**

### *Grammaire textuelle concrète*

Un identificateur <signal identifieur> contenu dans l'ensemble complet de signaux d'entrée valides d'une définition <agent definition> peut être omis dans l'ensemble des identificateurs <signal identifieur> contenus dans les listes <input list>, <priority input list> et dans la liste <save list> d'un état <state>.

### *Modèle*

Pour chaque état <state>, il existe une partie implicite <input part> contenant une <transition> qui contient uniquement un état suivant <nextstate> revenant au même état <state>.

## **11.9 Transition spontanée**

Une transition spontanée spécifie une transition d'état sans aucune réception de signal.

### *Grammaire abstraite*

*Spontaneous-transition*

:: [On-exception]  
[Provided-expression]  
*Transition*









<composite state heading> ::=  
     **state** [<qualifier>] <composite state name>  
     [<agent formal parameters>]

<entity in composite state> ::=  
     <variable definition>  
     | <data definition>  
     | <select definition>  
     | <data type reference>  
     | <macro definition>  
     | <procedure definition>  
     | <exception definition>  
     | <composite state type definition>  
     | <composite state type reference>

<composite state body> ::=  
     [<on exception>] <start>\* { <state> | <exception handler> | <free action> }\*

<composite state reference> ::=  
     **state substructure** <composite state name> **referenced** <end>

*Composite-state-graph* représente un corps <composite state body>.

Si un corps <composite state body> contient au moins un départ <start> mais pas d'état <state>, le corps <composite state body> doit être interprété comme une partie encapsulée d'une transition.

Un seul de ces départs <start> doit être non étiqueté. Chaque point d'entrée et de sortie étiqueté supplémentaire doit être défini par un point <state connection points> correspondant.

Si un corps <composite state body> contient au moins un état <state> différent de l'état astérisque, un départ <start> doit être présent.

Une définition <variable definition> dans un état <composite state> ne peut pas contenir de noms exportés **exported** <variable name> si l'état <composite state> est englobé par une définition <procedure definition>.

### *Grammaire graphique concrète*

<composite state graph area> ::=  
     <frame symbol> **contains** {  
         <composite state heading>  
         { <composite state text area>\*  
           <type in composite state area>\*  
           <composite state body area> } **set** }  
     **is associated with** {<graphical state connection point>\* } **set**  
     **is connected to** { {<gate on diagram>}\* } **set**  
     [ **is associated with** <package use area> ]

<composite state text area> ::=  
     <text symbol> **contains**  
     { <valid input signal set>  
     | <variable definition>  
     | <data definition>  
     | <data type reference>  
     | <procedure definition>  
     | <procedure reference>  
     | <select definition>  
     | <macro definition>}\*

<type in composite state area> ::=  
     <procedure area>  
     | <data type reference area>  
     | <composite state type diagram>  
     | <composite state type reference area>

```

<composite state body area> ::=
    [<on exception association area>]
    <start area>* { <state area> | <exception handler area> | <in connector area> }*

```

Il doit y avoir au plus un ensemble <valid input signal set> dans les zones <composite state text area> d'une zone <composite state graph area>.

La zone <package usage area> doit être placée en tête du symbole <frame symbol>.

### Sémantique

Le nœud non étiqueté *State-start-node* du graphe *Composite-state-graph* est interprété comme le point d'entrée par défaut de l'état composite. Il est interprété lorsque le nœud *Nextstate-node* n'a pas de point *State-entry-point*. Les nœuds *Named-start-nodes* sont interprétés comme des points d'entrée supplémentaires de l'état composite. Le point *State-entry-point* d'un nœud *Nextstate-node* définit la transition de départ nommée qui est interprétée.

Un nœud *Action-return-node* dans un état composite (sans nom <state exit point name>) doit être interprété comme le point de sortie par défaut de l'état composite. L'interprétation d'un nœud *Action-return-node* déclenche le nœud *Connect-node* sans nom *Name* dans l'entité englobante. Des nœuds *Named-return-nodes* supplémentaires, c'est-à-dire le retour <return> avec un nom <state exit point name>, doivent être interprétés comme des points de sortie supplémentaires de l'état composite. L'interprétation d'un nœud *Named-return-node* déclenche une transition de sortie dans l'entité englobante contenue dans un nœud *Connect-node* avec le même nom *Name*.

Les nœuds du graphe d'état sont interprétés de la même manière que les nœuds équivalents d'un graphe d'agent, de service ou de procédure, c'est-à-dire que le graphe d'état a le même ensemble complet de signaux d'entrée valides que l'agent englobant et le même accès d'entrée que l'instance de l'agent englobant.

Il est possible de spécifier un état <composite state> qui consiste uniquement en transitions reçues dans l'état astérisque, sans départ <start> et sans sous-état. Un tel état <composite state> est interprété comme un état de base contenant des transitions encapsulées. Les transitions encapsulées peuvent être terminées par l'état suivant <dash nextstate> ou par le retour <return>.

### Modèle

Si l'état <composite state> ne contient aucun <state> ayant un nom <state name> mais contient uniquement un état <state> avec <asterisk>, transformer l'état astérisque en un état <state> ayant un nom anonyme <state name> et un départ <start> menant à cet état <state>.

## 11.11.2 Agrégat d'état

Un agrégat d'état est une division d'un état composite. Il consiste en plusieurs états composites, dont l'interprétation est entrelacée au niveau des transitions. A tout moment, chaque division de l'agrégat d'état se trouve dans un des états de cette division ou (pour une seule des divisions) dans une transition ou elle s'est achevée et attend que les autres divisions s'achèvent. Chaque transition va jusqu'à son achèvement. Les agrégats d'état peuvent être utilisés pour diviser le graphe d'un état.

### Grammaire abstraite

```

State-aggregation-node      ::      State-partition-set
                               [Entry-procedure-definition]
                               [Exit-procedure-definition]
State-partition              ::      Name
                               Composite-state-type-identifier
                               Connection-definition-set
Connection-definition        ::      Entry-connection-definition | Exit-connection-definition
Entry-connection-definition  ::      Outer-entry-point Inner-entry-point
Outer-entry-point            ::      State-entry-point-name | DEFAULT
Inner-entry-point            ::      State-entry-point-name | DEFAULT
Exit-connection-definition   ::      Outer-exit-point Inner-exit-point

```

*Outer-exit-point* :: *State-exit-point-name* | **DEFAULT**  
*Inner-exit-point* :: *State-exit-point-name* | **DEFAULT**

*State-entry-point-name* dans un point d'entrée *Outer-entry-point* doit désigner une définition *State-entry-point-definition* d'une définition *Composite-state-type-definition* où se produit le nœud *State-aggregation-node*. Le nom *State-entry-point-name* d'un point d'entrée *Inner-entry-point* doit désigner une définition *State-entry-point-definition* de l'état composite dans la division *State-partition*. De même, les points de sortie *State-exit-points* doivent désigner respectivement des points de sortie dans l'état composite interne et externe. **DEFAULT** indique les points d'entrée et de sortie non étiquetés.

Tous les points d'entrée et de sortie de l'état conteneur et des divisions d'état doivent apparaître dans une seule et unique définition *Connection-definition*.

Les ensembles de signaux d'entrée dans des divisions *State-partition-set* au sein d'un état composite doivent être disjoints. L'ensemble de signaux d'entrée d'une division *State-partition* est défini comme l'union de tous les signaux apparaissant dans un nœud *Input-node* ou l'ensemble *Save-signalset* à l'intérieur du type d'état composite, y compris les états imbriqués, et les procédures mentionnées dans les nœuds *Call-nodes*.

### Grammaire textuelle concrète

```
<state aggregation> ::=
    {<package use clause>}*
    <state aggregation heading> <end> substructure
    <state connection points>*
    <entity in composite state>*
    <state aggregation body>
    endsubstructure [ [<qualifier>] <composite state name> ] <end>
```

```
<state aggregation heading> ::=
    state aggregation [<qualifier>] <composite state name>
    [<agent formal parameters>]
```

```
<state aggregation body> ::=
    {
    | <state partitioning>
    | <state partition connection> }+
```

```
<state partitioning> ::=
    <textual typebased state partition definition>
    | <composite state reference>
    | <composite state>
```

```
<state partition connection> ::=
    connect <outer entry point> and <inner entry point> <end>
```

```
<outer entry point> ::=
    <state partition identifier> via <point>
```

```
<inner entry point> ::=
    <state partition identifier> via <point>
```

```
<point> ::=
    <state entry point> | <state exit point> | default
```

### Grammaire graphique concrète

```
<state aggregation area> ::=
    <frame symbol> contains {
    <state aggregation heading>
    { <composite state text area>*
    <type in composite state area>*
    <state aggregation body area> } set }
    is associated with {<graphical state connection point>* } set
    is connected to { {<gate on diagram>}* } set
    [ is associated with <package use area> ]
```

<state aggregation body area> ::=  
 { { <state partition area> | <connection definition area> }+ } **set**

<state partition area> ::=  
 | <state partition reference area>  
 | <composite state area>  
 | <graphical typebased state partition definition>  
 | <inherited state partition definition>

<state partition reference area> ::=  
 <state symbol> **contains** <state name>

<graphical typebased state partition definition> ::=  
 <state symbol> **contains** <typebased state partition heading>

<typebased state partition heading> ::=  
 <state name> <colon> <composite state type expression>

<inherited state partition definition> ::=  
 <dashed state symbol> **contains** <state partition identifier>

<dashed state symbol> ::=



<connection definition area> ::=  
 <solid association symbol> **is connected to** <graphical point>

<graphical point> ::=  
 | <graphical state connection point>  
 | { <state entry points> | <state exit points> } **is associated with** <state partition area>  
 | <state connection point symbol> **is connected to** <frame symbol>

Le même point de sortie d'état d'une division d'état ne doit pas être relié à plus d'un point de sortie d'état de l'état englobant.

Un point d'entrée d'état de l'état englobant ne doit pas être relié à deux points d'entrée d'état différents de la même division d'état.

### Sémantique

Si une définition *Composite-state-type-definition* contient un nœud *State aggregation-node*, les états composites de chaque division *State-partition* sont interprétés de manière entrelacée au niveau des transitions. Chaque transition va jusqu'à son achèvement avant qu'une autre transition ne soit interprétée. La création d'un état composite ayant une division d'état implique la création des divisions *State-partition-set* contenues et leurs connexions. Si la définition *Composite-state-type-definition* d'une division *State-partition* a des paramètres *Composite-state-formal-parameters*, ces paramètres formels sont non définis lorsqu'on rentre dans l'état.

Les nœuds *State-start-node* non étiquetés des divisions sont interprétés dans n'importe quel ordre comme le point d'entrée par défaut de l'état composite. Ils sont interprétés lorsque le nœud *Nextstate-node* n'a pas de point d'entrée *State-entry-point*. Les nœuds *Named-start-nodes* sont interprétés comme des points d'entrée supplémentaires de l'état composite. Si l'entrée dans l'état composite se fait par le point d'entrée *Outer-entry-point* des définitions *Entry-connection-definitions*, la transition de départ de la division ayant le point d'entrée *Inner-entry-point* correspondant est interprétée. On entre dans les divisions d'état dans un ordre indéterminé, après l'achèvement de la procédure d'entrée d'agrégat d'état.

Lorsque toutes les divisions ont chacune interprété (dans un ordre indifférent) un nœud *Action-Return-node* ou *Named-return-node*, les divisions sortent de l'état composite. Les définitions *Exit-connection-definitions* associent les points de sortie des divisions aux points de sorties de l'état composite. Si des divisions différentes sortent de l'état composite par des points de sorties différents, le point de sortie de l'état composite est choisi de façon non déterministe. La procédure de sortie de

l'agrégat d'état est interprétée après l'achèvement de toutes les divisions d'état. Les signaux dans l'ensemble d'entrée d'une division qui a achevé son nœud de retour sont sauvegardés jusqu'à ce que toutes les autres divisions soient achevées.

Les nœuds des graphes de divisions d'état sont interprétés de la même manière que les nœuds équivalents d'un agent ou d'un graphe de procédure, avec la seule différence qu'ils ont des ensembles disjoints de signaux d'entrée. Les divisions d'état partagent le même port d'entrée que l'agent englobant.

Une transition d'entrée associée à une application d'état composite qui contient un nœud *State-aggregation-node* s'applique à tous les nœuds de toutes les divisions d'état et elle implique une terminaison par défaut de toutes celles-ci. Si une telle transition se termine par un nœud *Nextstate-node* avec **HISTORY**, toutes les divisions pénètrent de nouveau dans leurs sous-états respectifs.

### Modèle

Si un point d'entrée de l'agrégat d'état n'est relié à aucun point d'entrée d'une division d'état, une connexion implicite à l'entrée non étiquetée est ajoutée. De même, si un point de sortie d'une division n'est relié à aucun point de sortie de l'agrégat d'état, une connexion à la sortie non étiquetée est ajoutée.

S'il y a des signaux dans l'ensemble d'entrée valide complet d'un agent qui ne sont traités par aucune division d'état d'un état composite donné, une division d'état implicite supplémentaire est ajoutée à cet état composite. Cette division implicite a uniquement une transition de départ non étiquetée et un seul état contenant toutes les transitions implicites (y compris celles pour les procédures exportées et les variables exportées). Si le signal de l'autre division sort, un signal implicite est envoyé à l'agent, qui est traité par la division implicite. Après le traitement de tous les signaux implicites par la division implicite, il sort par le nœud *State-return-node*.

### 11.11.3 Point de connexion d'état

Les points de connexion d'état sont définis dans des états composites, à la fois des états composites directement spécifiés et des types d'état, et ces points représentent des points de connexion pour l'entrée et la sortie d'un état composite.

#### Grammaire textuelle concrète

```
<state connection points> ::=
    { in <state entry points> | out <state exit points> } <end>

<state entry points> ::=
    <state entry point>
    | ( <state entry point> { , <state entry point> } * )

<state exit points> ::=
    <state exit point>
    | ( <state exit point> { , <state exit point> } * )

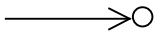
<state entry point> ::=
    <state entry point name>

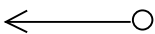
<state exit point> ::=
    <state exit point name>
```

#### Grammaire graphique concrète

```
<graphical state connection point> ::=
    <state connection point symbol>
    is associated with { <state entry points> | <state exit points> }
    is connected to <frame symbol>

<state connection point symbol> ::=
    <state connection point symbol 1> | <state connection point symbol 2>
```

<state connection point symbol 1> ::=  


<state connection point symbol 2> ::=  


Pour le symbole <state connection point symbol 1>, le point <graphical state connection point> doit alors contenir des points <state entry points>; autrement, le point <graphical state connection point> doit contenir des points <state exit points>.

Dans le symbole <state connection point symbol 1> et dans le <state connection point symbol 2>, le centre du cercle doit être situé sur le bord du symbole <frame symbol> auquel il est relié.

### Sémantique

Un point <state entry point> définit un point d'entrée sur un état <composite state>. Un point <state exit point> définit un point de sortie sur un état <composite state>.

Chaque état <composite state> possède deux points <state connection points> définis implicitement. Ceux-ci sont les points d'entrée et de sortie qui correspondent respectivement à un départ non étiqueté <start> et à un retour <return>.

Un état <composite state> ne peut correspondre qu'à ses propres points <state entry point> dans des départs <start> étiquetés et à ses propres points <state exit point> dans des retours <return> étiquetés.

## 11.11.4 Connexion

### Grammaire abstraite

*Connect-node* ::= [*State-exit-point-name*]  
 [*On-exception*]  
*Transition*  
*State-exit-point-name* = *Name*

### Grammaire textuelle concrète

<connect part> ::=  
**connect** [<virtuality>] [<connect list>] <end>  
 [<on exception>] <exit transition>  
 <connect list> ::=  
 <state exit point name> { , <state exit point name> } \*  
 | <asterisk connect list>  
 <exit transition> ::=  
 <transition>  
 <asterisk connect list> ::=  
 <asterisk>

Une partie <connect part> avec au plus un point <state exit point> représente un nœud *Connect-node*. Si aucune liste <connect list> n'est donnée, le nom *State-exit-point-name* est omis.

La liste <connect list> doit correspondre uniquement à des points <state exit point> visibles.

### Grammaire graphique concrète

<connect area> ::=  
 [<virtuality>] [<connect list>]  
 [ **is connected to** <on exception association area> ]  
**is followed by** <exit transition area>  
 <exit transition area> ::=  
 <transition area>

Une partie <connect part> avec au plus un point <state exit point> représente un nœud *Connect-node*. Si aucune liste <connect list> n'est donnée, le nom *State-exit-point-name* est omis.

La liste <connect list> doit correspondre uniquement à des points <state exit point> visibles.

### Sémantique

<connect part> représente un point de sortie sur un état <composite state>. L'interprétation est reprise à ce point si, dans l'état <composite state>, il y a interprétation d'un retour <return> accédant à un point <state exit point> trouvé dans la liste <connect list>.

Une partie <connect part> avec une liste <connect list> vide correspond à un retour <return> non étiqueté, c'est-à-dire un retour <return> sans <expression>, dans un état <composite state>.

### Modèle

Lorsque la liste <connect list> d'une certaine partie <connect part> contient d'un point <state exit point>, une copie de la partie <connect part> est créée pour chacun de ces points <state exit point>. La partie <connect part> est ensuite remplacée par ces copies.

Une liste <connect list> qui contient une liste <asterisk connect list> est transformée en une liste de points <state exit point>, une liste pour chaque point <state exit point> de l'état <composite state> en question. La liste des points <state exit point> est transformée comme décrit ci-dessus.

## 11.12 Transition

### 11.12.1 Corps de transition

#### Grammaire abstraite

```
Transition          ::      Graph-node*
                        ( Terminator | Decision-node )
Graph-node          ::      ( Task-node
                        | Output-node
                        | Create-request-node
                        | Call-node
                        | Compound-node
                        | Set-node
                        | Reset-node ) [On-exception]
Terminator          ::      ( Nextstate-node
                        | Stop-node
                        | Return-node
                        | Join-node
                        | Continue-node
                        | Break-node
                        | Raise-node ) [On-exception]
```

#### Grammaire textuelle concrète

```
<transition> ::=
                {<transition string> [<terminator statement>] }
                |
                <terminator statement>
<transition string> ::=
                {<action statement>}+
<action statement> ::=
                [<label>]
                { <action 1> <end> [ <on exception> ] | <action 2> <end> }
```



```

<action 1> ::=
    | <task>
    | <output>
    | <create request>
    | <decision>
    | <set>
    | <reset>
    | <export>
    | <procedure call>
    | <remote procedure call>

<action 2> ::=
    <transition option>

<terminator statement> ::=
    [<label>]
    { <terminator 1> <end> [ <on exception> ] | <terminator 2> <end> }

<terminator 1> ::=
    | <return>
    | <raise>

<terminator 2> ::=
    | <nextstate>
    | <join>
    | <stop>

```

Lorsque l'instruction <terminator statement> d'une transition <transition> est omise, la dernière action dans la <transition> doit contenir une décision <decision> terminale (voir 11.13.5) ou une option <transition option> terminale, sauf pour toutes les <transition> contenues dans une décision <decision> ou une option <transition option>.

### Grammaire graphique concrète

```

<transition area> ::=
    [ <transition string area> is followed by ]
    {
    | <state area>
    | <nextstate area>
    | <decision area>
    | <stop symbol>
    | <merge area>
    | <out connector area>
    | <return area>
    | <transition option area>
    | <raise area> }

<transition string area> ::=
    {
    | <task area>
    | <output area>
    | <create request area>
    | <procedure call area>
    | <remote procedure call area> }
    [ is followed by <transition string area> ]

```

Une transition consiste en une séquence d'actions qui doivent être effectuées par l'agent.

La zone <transition area> représente *Transition* et la zone <transition string area> représente la liste de nœuds *Graph-node*.

### Sémantique

Une transition réalise une séquence d'actions. Pendant la transition, les données de l'agent peuvent être manipulées et des signaux peuvent être envoyés. La transition s'arrêtera lorsque la machine à états de l'agent entrera dans un état, ou lors d'un arrêt ou d'un retour ou d'un transfert de contrôle à une autre transition.

Une transition dans un processus ou dans un bloc peut être interprétée en même temps qu'une transition dans un autre processus du même bloc (sous réserve qu'elles soient toutes les deux englobées dans un processus) ou d'un autre bloc. Les transitions de processus contenues dans un processus sont interprétées par entrelacement, c'est-à-dire qu'un seul processus interprète une seule transition à la fois jusqu'à ce qu'il atteigne un état suivant (*nextstate*) (fonctionnement jusqu'à réalisation). Un modèle valide pour l'interprétation d'un système de SDL est l'entrelacement complet de différents processus au niveau de toutes les actions qui ne peuvent pas être transformées (par les règles dans l'Annexe F) en d'autres actions et qui ne sont pas exclues parce qu'elles sont dans une transition alternant avec une transition en cours d'interprétation (voir 9.3).

Une durée non définie peut s'écouler pendant qu'une action est interprétée. Il est normal que la durée nécessaire varie chaque fois qu'une action est interprétée. Il est également normal que la durée nécessaire soit différente à chaque interprétation, ou qu'elle soit égale à zéro (ce qui est le résultat de **now**, voir 12.3.4.1, non modifié).

### Modèle

Une action de transition peut être transformée en une liste d'actions (contenant éventuellement des états implicites), conformément aux règles de transformation pour l'expression `<import expression>` et l'appel `<remote procedure call>`. Pour conserver un gestionnaire d'exception associé à l'action, au terminateur ou à la décision d'origine, cette liste d'actions est encapsulée dans une nouvelle procédure implicitement définie avec un nom anonyme, désigné ici *Actions*, comme suit, où la liste *list-of-actions* correspond à la liste d'actions résultante:

```

procedure Actions;
  start;
  list-of-actions;
  return;
endprocedure;

```

L'ancienne action est remplacée par un appel à *Actions*. Si un gestionnaire d'exception était associé à l'action d'origine, il est associé à l'appel à *Actions*.

Si la construction transformée est apparue dans un terminateur ou dans une décision, le terminateur ou la décision d'origine est remplacé(e) par un appel à *Actions*, suivi d'un nouveau terminateur ou d'une nouvelle décision. Si un gestionnaire d'exception était associé au terminateur ou à la décision d'origine, il est associé à l'appel à *Actions* et au nouveau terminateur ou à la nouvelle décision.

Aucun gestionnaire d'exception n'est associé au corps de *Actions* ni avec aucune partie de ce corps.

## 11.12.2 Terminateur de transition

### 11.12.2.1 Etat suivant

#### Grammaire abstraite

```

Nextstate-node           ::   State-name
                               [Nextstate-parameters]
Nextstate-parameters   ::   [Expression]*
                               [State-entry-point-name]
                               [HISTORY]

```

Les paramètres *Nextstate-parameters* ne peuvent être présents que si le nom *State-name* désigne un état composite.

Le nom *State-name* spécifié dans un état suivant doit avoir le même nom que l'état à l'intérieur du même graphe *State-transition-graph* ou *Procedure-graph*.

## Grammaire textuelle concrète

```
<nextstate> ::=
    nextstate <nextstate body>

<nextstate body> ::=
    <state name> [<actual parameters>] [ via <state entry point name> ]
    | <dash nextstate>

<dash nextstate> ::=
    <hyphen>
    | <history dash nextstate>

<history dash nextstate> ::=
    <history dash sign>
```

Un nœud *Nextstate-node* avec **HISTORY** représente un état suivant <history dash nextstate>.

Si une transition est terminée par un état suivant <history dash nextstate> l'état <state> doit être un état <composite state>.

La <transition> contenue dans un départ <start> ne doit pas mener, directement ou indirectement, à un état suivant <dash nextstate>. La <transition> contenue dans un départ <start> ou un traitement <handle> ne doit pas mener, directement ou indirectement, à un état suivant <history dash nextstate>.

Une exception <on exception> à l'intérieur d'un départ <start> ou associée à un corps complet, ne doit pas mener, directement ou indirectement (par le biais d'exceptions <on exception> à l'intérieur de gestionnaires <exception handler>), à un gestionnaire <exception handler> contenant des états suivants <dash nextstate>.

## Grammaire graphique concrète

```
<nextstate area> ::=
    <state symbol> contains <nextstate body>
```

Si le nom <state entry point name> est donné, l'état suivant <nextstate> doit correspondre à un état composite avec le point d'entrée d'état.

Si les paramètres <actual parameters> sont donnés, l'état suivant <nextstate> doit correspondre à un état composite avec des paramètres <agent formal parameters>.

## Sémantique

Un état suivant représente un terminateur ou une transition. Il spécifie l'état qu'aura l'agent, la procédure ou l'état composite lors de l'achèvement d'une transition.

Un état suivant pointillé pour un état composite indique que l'état suivant est l'état composite.

Si un nom *State-entry-point-name* est donné, l'état suivant est un état composite, et l'interprétation continue avec le nœud *State-start-node* portant le même nom dans le graphe *Composite-state-graph*.

Lorsqu'un nœud *Nextstate-node* avec **HISTORY** est interprété, l'état suivant est celui dans lequel la transition courante a été activée. Si l'interprétation entre à nouveau dans un état composite, sa procédure d'entrée est invoquée.

## Modèle

Dans chaque état <nextstate> d'un état <state>, l'état <dash nextstate> est remplacé par le nom <state name> de l'état <state>. Ce modèle est appliqué après la transformation des états <state> et après toutes les autres transformations sauf celles qui concernent les virgules finales, les synonymes, les entrées prioritaires, les signaux continus, les conditions d'activation, les tâches implicites pour actions impératives et les variables ou procédures distantes.

Le reste de cette section *Modèle* décrit la façon de déterminer la signification de l'état <dash nextstate> dans les gestionnaires d'exception.

Un gestionnaire d'exception est appelé atteignable à partir d'un état ou d'un gestionnaire d'exception s'il est associé à l'état ou au gestionnaire d'exception, aux stimuli associés à l'état ou au gestionnaire d'exception, ou s'il est associé aux actions de transition suivant les stimuli. Tous les gestionnaires d'exception accessibles depuis un gestionnaire d'exception qui est lui-même atteignable à partir de l'état sont également appelés accessibles à partir de l'état.

NOTE – L'accessibilité est transitive.

Pour chaque état <state>, la règle suivante s'applique: tous les gestionnaires d'exception atteignables sont rendus distincts pour l'état en copiant chaque gestionnaire d'exception dans un gestionnaire <exception handler> portant un nouveau nom. Les exceptions <on exception> sont modifiées en utilisant ce nouveau nom. Ensuite, les gestionnaires d'exception qui ne sont pas atteignables à partir d'un quelconque état sont éliminés.

Après ce remplacement, un gestionnaire <exception handler> donné contenant des états suivants <dash nextstate> peut être accessible, directement ou indirectement, à partir d'un seul et unique état <state>. Les états suivants <dash nextstate> à l'intérieur de chacun de ces gestionnaires <exception handler> sont remplacés par le nom <state name> de cet état <state>.

### 11.12.2.2 Branchement

Un branchement modifie le flot dans un corps en exprimant que l'instruction <action statement> suivante qui doit être interprétée est celle qui contient le même nom <connector name>.

#### *Grammaire abstraite*

Join-node :: Connector-name

#### *Grammaire textuelle concrète*

<join> ::= **join** <connector name>

Il doit y avoir un seul et unique nom <connector name> correspondant à un branchement <join> à l'intérieur du même corps. La règle concernant le corps <agent type body> est énoncée en 8.3.1.

#### *Grammaire graphique concrète*

<merge area> ::= <merge symbol> **is connected to** <flow line symbol>

<merge symbol> ::= <flow line symbol>

<flow line symbol> ::=

\_\_\_\_\_

<out connector area> ::= <out connector symbol> **contains** <connector name>

<out connector symbol> ::= <in connector symbol>

Pour chaque zone <out connector area> d'une zone <state machine graph area>, on doit avoir une seule et unique zone <in connector area> dans cette zone <state machine graph area>.

Une zone <out connector area> correspond à un branchement <join> dans la *grammaire textuelle concrète*. Si une zone <merge area> est incluse dans une zone <transition area>, elle est équivalente à la spécification d'une zone <out connector area> dans la zone <transition area> qui contient un unique nom <connector name> et à l'adjonction d'une zone <in connector area> avec le même nom <connector name> au symbole <flow line symbol> dans la zone <merge area>.

## Sémantique

Lorsqu'un nœud *Join-node* est interprété, l'interprétation se poursuit avec l'action *Free-action* nommé par le nom *Connector-name*.

### 11.12.2.3 Arrêt

#### Grammaire abstraite


*Stop-node* ::= ()

Un nœud *Stop-node* ne doit pas être situé à l'intérieur d'un graphe *Procedure-graph*.

#### Grammaire textuelle concrète

<stop> ::= **stop**

#### Grammaire graphique concrète

<stop symbol> ::= 

## Sémantique

L'arrêt provoque l'arrêt de l'agent qui l'interprète.

Cela signifie que les signaux bloqués à l'accès d'entrée sont supprimés et que la machine à états de l'agent se met en arrêt. Lorsque tous les agents contenus cessent d'exister, l'agent lui-même cesse d'exister.

### 11.12.2.4 Retour

#### Grammaire abstraite

*Return-node* = *Action-return-node*  
| *Value-return-node*  
| *Named-return-node*  
*Action-return-node* ::= ()  
*Value-return-node* ::= *Expression*  
*Named-return-node* ::= *State-exit-point-name*

Un nœud *Return-node* doit être situé à l'intérieur d'un graphe *Procedure-graph* ou d'un *Composite-state-graph*. Un nœud *Action-return-node* doit être situé uniquement à l'intérieur du graphe *Procedure-Graph* d'une définition *Procedure-definition* sans résultat *Result* ou d'un *Composite-state-graph*. Un nœud *Value-return-node* doit être situé uniquement à l'intérieur du graphe *Procedure-Graph* d'une définition *Procedure-definition* contenant le résultat *Result*. Un nœud *Named-return-node* doit être situé uniquement à l'intérieur d'un graphe *Composite-state-graph*.

#### Grammaire textuelle concrète

<return> ::= **return** [<expression> | **{via** <state exit point>}]

#### Concrete graphical grammar

<return area> ::= **<return symbol>**  
[ **is connected to** <on exception association area> ]  
[ **is associated with** {<expression> | <state exit point>} ]

<return symbol> ::=



L'<expression> dans un retour <return> ou une zone <return area> est permise si et seulement si la portée qui l'englobe est un opérateur, une méthode ou une procédure ayant un résultat <procedure result>.

Le point <state exit point> est permis si et seulement si la portée qui l'englobe est un état composite contenant le point <state exit point> spécifié.

### Sémantique

Un nœud *Return-node* dans une procédure est interprété de la manière suivante:

- toutes les variables créées par l'interprétation du nœud *Procedure-start-node* cesseront d'exister;
- l'interprétation du graphe *Procedure-graph* est terminée et l'instance de procédure cesse d'exister;
- si un nœud *Value-return-node* est interprété, le résultat de l'*Expression* est renvoyé dans le contexte appelant;
- désormais, l'interprétation du contexte appelant continue au nœud suivant l'appel.

Un nœud *Return-node* dans un état composite entraîne l'activation d'un nœud *Connect-node*. Pour un nœud *Named-return-node*, l'interprétation continue au nœud *Connect-node* portant le même nom. Pour un nœud *Action-return-node*, l'interprétation continue au nœud *Connect-node* sans nom.

### 11.12.2.5 Déclenchement d'exception (Raise)

#### Grammaire abstraite

*Raise-node* ::= *Exception-identifiant*  
[*Expression*]\*

La longueur de la liste des expressions facultatives *Expression* doit être la même que le nombre d'identificateurs *Sort-reference-identifiers* dans la définition *Exception-definition* désignée par l'identificateur *Exception-identifiant*.

Chaque *Expression* doit avoir une sorte compatible (par position) avec l'identificateur *Sort-reference-identifiant* correspondant de la définition *Exception-definition*.

#### Grammaire textuelle concrète

<raise> ::=

**raise** <raise body>

<raise body> ::=

<exception raise>

<exception raise> ::=

<exception identifiant> [<actual parameters>]

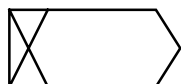
Un déclenchement d'exception <raise> représente un nœud *Raise-node*.

#### Grammaire graphique concrète

<raise area> ::=

<raise symbol> **contains** <raise body>

<raise symbol> ::=



## Sémantique

L'interprétation d'un nœud *Raise-node* crée une instance d'exception (voir 11.16 pour l'interprétation d'une instance d'exception). Les éléments de données qui sont acheminés par l'instance d'exception sont les résultats des paramètres réels du déclenchement d'exception <raise>. Si une *Expression* d'une liste d'*Expressions* facultatives est omise (c'est-à-dire si l'<expression> correspondante dans les paramètres <actual parameters> est omise), aucun élément de données n'est acheminé avec la place correspondante de l'instance d'exception, c'est-à-dire que la place correspondante est "non définie".

Si un syntype est spécifié dans la définition d'exception et qu'une expression est spécifiée dans l'instruction <raise>, la vérification d'intervalle définie au 12.1.9.5 est appliquée à l'expression.

## Modèle

Un déclenchement d'exception peut être transformé en une liste d'actions (contenant éventuellement des états implicites) ajoutée à un nouveau déclenchement d'exception conformément au modèle (des appels de procédure distante par exemple). Le modèle relatif aux terminateurs de transition défini en 11.12.1 s'applique ensuite.

## 11.13 Action

### 11.13.1 Tâche


#### Grammaire abstraite

*Task-node* = *Assignment*  
| *Assignment-attempt*  
| *Informal-text*

#### Grammaire textuelle concrète

<task> ::= **task** <textual task body>  
  
<textual task body> ::=   
| <assignment>  
| <informal text>  
| <compound statement>

#### Grammaire graphique concrète

<task area> ::=   
| <task symbol> **contains** <graphical task body>  
| [ **is connected to** <on exception association area> ]  
  
<graphical task body> ::=   
| <statement list>  
| <informal text>  
  
<task symbol> ::=   


L'extrémité de poursuite <end> dans la liste <statement list> d'un corps <graphical task body> peut être omise.

## Sémantique

L'interprétation d'un nœud *Task-node* est l'interprétation de l'instruction *Assignment*, *Assignment-attempt* ou l'interprétation du texte *Informal-text*.

Une zone de tâche crée sa propre portée.

## Modèle

Si la liste <statement list> dans l'instruction <compound statement> du corps <textual task body> est vide, la tâche <task> est supprimée. Si la liste <statement list> dans un corps <graphical task body> est vide, la zone <task area> est supprimée. Tout élément syntaxique menant à la tâche <task> ou à la zone <task area> doit alors mener directement à l'élément suivant la <task> ou la zone <task area>, respectivement.

Une tâche <task> contenant une instruction <compound statement> est transformée comme décrit au 11.14.1. Le résultat de cette transformation est inséré à la place de la tâche <task>.

Une zone <task area> contenant une liste <statement list> est transformée en une instruction <compound statement> de la même manière que l'instruction <compound statement> en une tâche <task> (voir ci-dessus), si ce n'est que le résultat de la transformation de l'instruction <compound statement> au 11.14.1 est ensuite transformé en son équivalent graphique qui remplace la zone <task area>.

NOTE – La transformation d'une tâche <task> ou d'une zone <task area> contenant une liste <statement list> ne correspond pas nécessairement à un nœud *Task-node* dans la grammaire abstraite.

### 11.13.2 Création

#### Grammaire abstraite

*Create-request-node* :: [Variable-identifiant]  
Agent-identifiant  
[Expression]\*

La longueur de la liste d'*Expressions* facultatives doit être la même que le nombre de paramètres *Agent-formal-parameter* dans la définition *Agent-definition* de l'identificateur *Agent-identifiant*.

Chaque *Expression* correspondant par sa position à un paramètre *Agent-formal-parameter* doit avoir une sorte compatible à la sorte du paramètre *Agent-formal-parameter* dans la définition *Agent-definition* désignée par l'identificateur *Agent-identifiant*.

#### Grammaire textuelle concrète

<create request> ::=  
                                  **create** <create body>  
  
<create body> ::=  
                                  { <agent identifiant> | <agent type identifiant> | **this** } [<actual parameters>]  
  
<actual parameters> ::=  
                                  ( <actual parameter list> )  
  
<actual parameter list> ::=  
                                  [<expression>] { , [<expression>] }\*

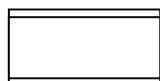
Les virgules après la dernière <expression> dans la liste <actual parameter list> peuvent être omises.

**this** ne peut être spécifié que dans une définition <agent type definition> et dans des portées englobées par une définition <agent type definition>.

#### Grammaire graphique concrète

<create request area> ::=  
                                  <create request symbol> **contains** <create body>  
                                  [ **is connected to** <on exception association area> ]

<create request symbol> ::=



Une zone <create request area> représente un nœud *Create-request-node*.



## Sémantique

L'action de création entraîne la création d'une instance d'agent à l'intérieur de l'agent qui effectue la création ou dans l'agent qui contient celui qui effectue la création. L'agent parent des agents créés (voir paragraphe 9, *Modèle*) a la même valeur pid que celle retournée par **self** de l'agent créateur. L'expression **self** des agents créés (voir paragraphe 9, *Modèle*) et l'expression **offspring** de l'agent créateur (voir paragraphe 9, *Modèle*) ont toutes deux la même nouvelle et unique valeur pid.

Lorsqu'une instance d'agent est créée, on lui attribue un accès d'entrée vide, les variables sont créées et les expressions de paramètres réels sont interprétées dans l'ordre donné, et affectées (voir 12.3.3) aux paramètres formels correspondants. Si l'agent créé contient des ensembles d'agents, les instances initiales de ces ensembles sont créées. L'agent débute alors l'interprétation du nœud de départ dans le graphe d'agent, les nœuds de départ des agents contenus initiaux sont interprétés dans un ordre donné avant d'interpréter les transitions causées par des signaux.

L'agent créé est ensuite interprété de manière asynchrone et en parallèle ou en alternance avec les autres agents, selon le genre de l'agent contenant (système, bloc, processus).

Si l'on tente de créer un nombre d'instances d'agent supérieur à celui qui est spécifié par le nombre maximal d'instances dans la définition d'agent, aucune nouvelle instance n'est créée, l'expression **offspring** de l'agent de création (voir paragraphe 9, *Modèle*) a le résultat Null et l'interprétation se poursuit.

Si une *<expression>* dans *<actual parameters>* est omise, le paramètre formel correspondant n'a pas d'élément de données associé; c'est-à-dire qu'il est "non défini".

Si l'identificateur *<agent type identifier>* est utilisé dans une demande *<create request>*, le type d'agent correspondant peut ne pas être défini comme *<abstract>* ou ne pas contenir de paramètres de contexte formels.

Si un ensemble d'instances et un type d'agent ayant le même nom sont définis dans une unité de portée et qu'une instruction de création dans cette unité de portée utilise le même nom, une instance est alors créée dans l'ensemble d'instances et elle n'est pas fondée sur le type d'agent. Il est à noter qu'il est possible de créer une instance du type d'agent en définissant un ensemble d'instances fondé sur le type d'agent et en créant ensuite une instance dans cet ensemble.

## Modèle

L'énoncé **this** est une syntaxe dérivée pour l'identificateur implicite *<process identifier>* qui identifie l'ensemble d'instances de l'agent dans lequel la création est interprétée.

Si l'identificateur *<agent type identifier>* est utilisé dans une demande *<create request>* les modèles suivants s'appliquent:

- a) s'il existe un ensemble d'instances du type d'agent indiqué dans l'agent contenant l'instance qui exécute la création, l'identificateur *<agent type identifier>* est une syntaxe dérivée désignant cet ensemble d'instances;
- b) s'il y a plus d'un ensemble d'instances, il est déterminé dans quel ensemble sera créée l'instance au moment de l'interprétation. La demande *<create request>* est dans ce cas remplacée par une décision non déterministe au moyen de **any** suivi d'une branche pour chaque ensemble d'instances. Dans chacune de ces branches est insérée une demande de création pour l'instance correspondante;
- c) s'il n'existe aucun ensemble d'instances du type d'agent indiqué dans l'agent contenant:
  - i) un ensemble d'instances implicite du type donné est créé avec un nom unique dans l'agent contenant;
  - ii) l'identificateur *<agent identifier>* dans la demande *<create request>* est une syntaxe dérivée pour cet ensemble d'instances implicite.

### 11.13.3 Appel de procédure

#### Grammaire abstraite

*Call-node* :: *Procedure-identifïer*  
[*Expression*]\*  
*Value-returning-call-node* :: *Procedure-identifïer*  
[*Expression*]\*

La longueur de la liste d'*Expressions* facultatives doit être la même que le nombre de paramètres *Procedure-formal-parameter* dans la définition *Procedure-definition* désignée par l'identificateur *Procedure-identifïer*.

Chaque *Expression* correspondant par sa position à un paramètre *In-parameter* doit avoir une sorte compatible avec la sorte du paramètre *Procedure-formal-parameter*.

Chaque *Expression* correspondant par sa position à un paramètre *Inout-parameter* ou *Out-parameter* doit être un identificateur *Variable-identifïer* dont la sorte est compatible avec la sorte identifiée par l'identificateur *Sort-reference-identifïer* du paramètre *Procedure-formal-parameter*.

#### Grammaire textuelle concrète

<procedure call> ::= **call** <procedure call body>  
<procedure call body> ::= [ **this** ] { <procedure identifïer> | <procedure type expression> } [<actual parameters>]


Une <expression> dans <actual parameters> correspondant à un paramètre formel **in/out** ou **out** ne peut être omise et doit être un accès <variable access> ou un primaire <extended primary>.

Une fois le *Modèle* de **this** appliqué, l'identificateur <procedure identifïer> doit désigner une procédure contenant une transition de départ.

Si **this** est utilisé, l'identificateur <procedure identifïer> doit désigner une procédure englobante.

L'appel <procedure call> représente un nœud *Call-node*. Un appel <value returning procedure call> (voir 12.3.5) représente un nœud *Value-returning-call-node*.

#### Grammaire graphique concrète

<procedure call area> ::= <procedure call symbol> **contains** <procedure call body>  
[ **is connected to** <on exception association area> ]  
<procedure call symbol> ::= 

La zone <procedure call area> représente un nœud *Call-node*.

#### Sémantique

L'interprétation d'une procédure *Call-node* ou *Value-returning-call-node* interprète les expressions des paramètres réels dans l'ordre donné et ensuite transfère l'interprétation vers la définition de procédure référencée par l'identificateur *Procedure-identifïer* et le graphe de procédure est interprété (l'explication est donnée en 9.4).

L'interprétation de la transition contenant un nœud *Call-node* continue lorsque l'interprétation de la procédure appelée est terminée.

L'interprétation de la transition contenant un nœud *Value-returning-call-node* continue lorsque l'interprétation de la procédure appelée est terminée. Le résultat de la procédure appelée est retourné par le nœud *Value-returning-call-node*.

Si une <expression> dans <actual parameters> est omise, le paramètre formel correspondant n'a pas d'élément de données associé; c'est-à-dire qu'il est "non défini".

### Modèle

Si l'identificateur <procedure identifier> n'est pas défini à l'intérieur de l'agent englobant, l'appel de procédure est transformé en un appel d'un sous-type local de la procédure créé implicitement.

**this** implique que lorsque la procédure est spécialisée, l'identificateur <procedure identifier> est remplacé par l'identificateur de la procédure spécialisée.

#### 11.13.4 Sortie

##### Grammaire abstraite

```

Output-node                ::   Signal-identifier
                               [Expression]*
                               [Signal-destination]
                               Direct-via
Signal-destination         =   Expression | Agent-identifier
Direct-via                 =   ( Channel-identifier | Gate-identifier )-set
Channel-identifier         =   Identifier

```

La longueur de la liste d'*Expressions* facultatives doit être la même que le nombre d'identificateurs *Sort-reference-identifier* dans la définition *Signal-definition* désignée par l'identificateur *Signal-identifier*.

Chaque *Expression* doit avoir une sorte compatible avec la sorte de l'identificateur *Sort-identifier-reference* correspondant (par position) qui figure dans la définition *Signal-definition*.

Pour chaque identificateur *Channel-identifier* dans le trajet *Direct-via*, il doit exister zéro ou plusieurs canaux de sorte que le canal soit atteignable via ce trajet avec l'identificateur *Signal-identifier* de l'agent et le trajet *Channel-path* dans la direction à partir de l'agent doit inclure l'identificateur *Signal-identifier* dans son ensemble d'identificateurs *Signal-identifier*.

Pour chaque identificateur *Gate-identifier* dans *Direct-via*, il doit y avoir zéro ou davantage de canaux de manière à ce que l'accès par ce trajet soit accessible avec l'identificateur *Signal-identifier* de l'agent et l'ensemble *Out-signal-identifier-set* de l'accès doit être inclure l'identificateur *Signal-identifier*.

##### Grammaire textuelle concrète

```

<output> ::=
    output <output body>
<output body> ::=
    <signal identifier> [<actual parameters>] { , <signal identifier> [<actual parameters>] } *
    <communication constraints>
<destination> ::=
    <pid expression> | <agent identifier> | this
<via path> ::=
    via { <channel identifier> | <gate identifier> }

```

L'expression <pid expression> ou l'identificateur <agent identifier> dans la <destination> représente la destination *Signal-destination*. Il existe une ambiguïté syntaxique entre <pid expression> et <agent identifier> dans la <destination>. Si la <destination> peut être interprétée comme une expression <pid expression> sans violer aucune condition statique, elle est interprétée comme une expression <pid expression> sinon comme un identificateur <agent identifier>. L'identificateur <agent identifier> doit désigner un agent atteignable à partir de l'agent d'origine.

La contrainte <communication constraints> (voir 10.5) dans une sortie <output> ne doit pas contenir de clause **timer** <timer identifier>. Elle contient au plus une clause **to** <destination> et zéro ou davantage de trajets <via path>.

Chaque trajet <via path> de contrainte <communication constraints> représente un identificateur *Channel-identifier* ou *Gate-identifier* dans le trajet *Direct-via*.

**this** ne peut être spécifié que dans une définition <agent type definition> et dans des portées englobées par une définition <agent type definition>.

Si la <destination> est une expression <pid expression> avec une sorte autre que Pid (voir 12.1.6), l'identificateur <signal identifier> doit représenter un signal défini ou utilisé par l'interface qui a défini la sorte pid.

L'identificateur <gate identifier> dans <via path> peut être utilisé pour identifier un accès qui est défini à l'aide d'une définition <textual interface gate definition> ou <graphical interface gate definition>.

### Grammaire graphique concrète

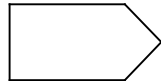
<output area> ::=

<output symbol> **contains** <output body>  
[ **is connected to** <on exception association area> ]

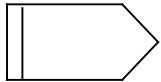
<output symbol> ::=

| <plain output symbol>  
| <internal output symbol>

<plain output symbol> ::=



<internal output symbol> ::=



NOTE 1 – Il n'y a pas de différence de signification entre un symbole <plain output symbol> et un symbole <internal output symbol>

### Sémantique

L'énoncé d'un identificateur *Agent-identifier* dans une destination *Signal-destination* désigne la destination *Signal-destination* comme toute instance existante de l'ensemble des instances d'agent désigné par l'identificateur *Agent-identifier*. S'il n'existe pas d'instances, le signal est ignoré.

Si aucun identificateur *Channel-identifier* ou *Gate-identifier* n'est spécifié dans le trajet *Direct-via* et qu'aucune destination *Signal-destination* n'est spécifiée, tout agent pour lequel il existe un trajet de communication peut recevoir le signal.

S'il existe une instance de processus contenant à la fois l'émetteur et le récepteur, les éléments de données acheminés par l'instance de signal sont les résultats des paramètres réels de la sortie. Sinon, les éléments de données acheminés par l'instance de signal sont les répliques nouvellement créées des résultats des paramètres réels de la sortie et n'ont pas de références communes avec les résultats des paramètres réels de la sortie. Lorsqu'il y a des cycles de références dans le résultat des paramètres réels, les éléments de données acheminés contiennent également ces cycles. Chaque élément de données acheminé est égal au paramètre réel correspondant de la sortie.

Si une expression <expression> dans <actual parameters> est omise, aucun élément de données n'est acheminé avec la place correspondante de l'instance du signal, c'est-à-dire que la place correspondante est "non définie".

Le pid de l'agent d'origine est également acheminé par l'instance du signal.

Si un syntype est spécifié dans la définition du signal et qu'une expression est spécifiée dans la sortie, la vérification d'intervalle définie au 12.1.9.5 s'applique à l'expression.

Si la <destination> est une expression <pid expression> et que la sorte statique de l'expression du pid est Pid, la vérification de la compatibilité pour la sorte dynamique de l'expression du pid (voir 12.1.6) est effectuée pour le signal désigné par l'identificateur *Signal-identif*.

L'instance du signal est ensuite remise à un trajet de communication capable de l'acheminer. L'ensemble des trajets de communication capables d'acheminer l'instance du signal peut être restreint par la clause <via path> de façon à inclure au moins un des ensembles de trajets mentionnés dans le trajet *Direct-via*.

Si la destination *Signal-destination* est une *Expression*, l'instance du signal est délivrée à une instance d'agent désignée par *Expression*. Si cette instance n'existe pas, ou si elle est non atteignable à partir de l'agent d'origine, l'instance du signal est ignorée.

Si la destination *Signal-destination* est un identificateur *Agent-identif*, l'instance du signal est délivrée à une instance arbitraire de l'ensemble d'instances d'agent désigné par l'identificateur *Agent-identif*. S'il n'existe pas de telle instance, l'instance du signal est ignorée.

NOTE 2 – Si la destination *Signal-destination* est Null dans un nœud *Output-node*, l'instance du signal sera ignorée lorsque le nœud *Output-node* est interprété.

Si aucune destination *Signal-destination* n'est spécifiée, le récepteur est sélectionné en deux étapes. Tout d'abord, le signal est envoyé à un ensemble d'instances d'agent atteignable par le trajet de communication capable d'acheminer l'instance du signal. Cet ensemble d'instances est choisi arbitrairement. Ensuite, lorsque l'instance du signal arrive à l'extrémité du trajet de communication, elle est délivrée à une instance arbitraire de l'ensemble d'instances d'agent. L'instance est choisie arbitrairement. S'il n'est pas possible de sélectionner une instance, l'instance du signal est ignorée.

Il faut noter que spécifier le même identificateur *Channel-identif* ou *Gate-identif* dans le trajet *Direct-via* de deux nœuds *Output-node*, cela ne signifie pas automatiquement que les signaux sont mis en file d'attente à l'accès d'entrée dans le même ordre que celui où les nœuds *Output-node* sont interprétés. Toutefois, l'ordre est préservé si les deux signaux sont acheminés par des canaux à retard identiques, ou acheminés uniquement par des canaux sans retard.

### Modèle

Si plusieurs paires d'identificateur <signal identifieur> et de paramètres <actual parameters> se trouvent spécifiées dans un corps <output body>, on a une syntaxe dérivée pour spécifier la séquence des sorties <output> ou des zones <output area> dans le même ordre spécifié dans le corps <output body> d'origine, chacune contenant une seule paire d'identificateur <signal identifieur> et de paramètres <actual parameters>. la clause **to** <destination> et les trajets <via path> sont répétés dans chacune des sorties <output> ou zones <output area>.

L'énoncé **this** dans la <destination> constitue une syntaxe dérivée pour l'identificateur implicite <agent identifieur> qui identifie l'ensemble d'instances pour l'agent dans lequel la sortie est interprétée.

### 11.13.5 Décision

#### Grammaire abstraite

<i>Decision-node</i>	::	<i>Decision-question</i> [ <i>On-exception</i> ] <i>Decision-answer-set</i> [ <i>Else-answer</i> ]
<i>Decision-question</i>	=	<i>Expression</i>   <i>Informal-text</i>

*Decision-answer* :: ( *Range-condition* | *Informal-text* )  
*Transition*

*Else-answer* :: *Transition*

Les conditions *Range-condition* des réponses *Decision-answer* doivent s'exclure mutuellement et les expressions *Constant-expression* des conditions *Range-condition* doivent avoir des sortes compatibles. Si la question *Decision-question* est une *Expression*, la condition *Range-condition* des réponses *Decision-answer* doit avoir une sorte compatible avec la sorte de la question *Decision-question*.

### Grammaire textuelle concrète

<decision> ::=

**decision** <question> <end> [<on exception>]  
                   <decision body>  
**enddecision**

<decision body> ::=

<answer part>+ [<else part>]

<answer part> ::=

( [<answer> ] ) <colon> [<transition>]

<answer> ::=

<range condition> | <informal text>

<else part> ::=

**else** <colon> [<transition>]

<question> ::=

<expression> | <informal text> | **any**

Une partie <answer part> ou une partie <else part> dans une décision ou une option de transition est une partie <answer part> terminale ou une partie <else part> terminale respectivement si elle contient une <transition> dans laquelle une instruction <terminator statement> est spécifiée, ou contient une chaîne <transition string> dont la dernière instruction <action statement> contient une décision terminale ou une option terminale. Une décision <decision> ou une option <transition option> est une décision terminale et une option terminale respectivement, si chaque partie <answer part> et partie <else part> dans le corps <decision body> est une partie terminale <answer part> ou une partie terminale <else part> respectivement.

La réponse <answer> de la partie <answer part> doit être omise si et seulement si la <question> est constituée par le mot clé **any**. Dans ce cas, il n'existe pas de partie <else part>.

Il y a une ambiguïté syntaxique entre le texte <informal text> et la chaîne <character string> dans <question> et réponse <answer>. Si la <question> et toutes les réponses <answer> sont des chaînes <character string>, tous ces éléments sont interprétés comme texte <informal text>. Si la <question> ou une réponse <answer> quelconque est une chaîne <character string> qui ne correspond pas au contexte de la décision, la chaîne <character string> désigne un texte <informal text>.

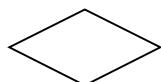
Le contexte de la décision (c'est-à-dire la sorte) est déterminé sans considération pour les réponses <answer> qui sont des chaînes <character string>.

### Grammaire graphique concrète

<decision area> ::=

<decision symbol> **contains** <question>  
                   [ **is connected to** <on exception association area> ]  
                   **is followed by** <graphical decision body>

<decision symbol> ::=



```

<graphical decision body> ::=
    { <graphical answer part>+ [<graphical else part>] } set

<graphical answer part> ::=
    <flow line symbol> is associated with <graphical answer>
    is followed by <transition area>

<graphical answer> ::=
    [<answer>] | ( [<answer>] )

<graphical else part> ::=
    <flow line symbol> is associated with else
    is followed by <transition area>

```

Les termes <graphical answer> et **else** peuvent être placés le long du symbole <flow line symbol> associé, ou sur le symbole <flow line symbol>.

Les symboles <flow line symbol> provenant d'un symbole <decision symbol> peuvent avoir un trajet d'origine commun.

Une zone <decision area> représente un nœud *Decision-node*.

La réponse <answer> de <graphical answer> doit être omise si et seulement si la <question> est constituée par le mot clé **any**. Dans ce cas, il n'existe pas de partie <graphical else part>.

### Sémantique

Une décision transfère l'interprétation vers le chemin sortant dont la condition d'intervalle contient le résultat donné par l'interprétation de la question. Un ensemble de réponses possibles à la question est défini, chacune d'entre elles spécifiant un ensemble d'actions à interpréter pour ce choix de chemin.

Une des réponses peut être le complément des autres. C'est le cas lorsqu'on spécifie la réponse *Else-answer*, qui indique l'ensemble des activités à réaliser quand le résultat de l'expression sur laquelle la question est posée, n'est pas couvert par les résultats spécifiés dans les autres réponses.

Lorsque la réponse *Else-answer* n'est pas spécifiée et que le résultat tiré de l'évaluation de l'expression de question ne correspond pas à l'une des réponses, l'exception prédéfinie `NoMatchingAnswer` est déclenchée.

### Modèle

Si une décision <decision> n'est pas une décision terminale, il s'agit d'une syntaxe dérivée pour une décision <decision> lorsque toutes les parties terminales <answer part> et la partie <else part> ( si elle n'est pas terminale) ont inséré à l'extrémité de leur <transition> un branchement <join> vers la première instruction <action statement> qui suit la décision ou (si la décision est la dernière instruction <action statement> dans une chaîne <transition string>) vers l'instruction <terminator statement> suivante.

L'utilisation du seul mot clé **any** dans une décision <decision> est une abréviation de l'utilisation de l'expression <any expression> dans la décision. Si l'on suppose que le corps <decision body> est suivi par N parties <answer part>, l'utilisation du mot clé **any** dans la décision <decision> est une abréviation de l'écriture **any**(data\_type\_N), où data\_type\_N est un syntype anonyme défini de la manière suivante:

```

syntype data_type_N =
    package Predefined Integer constants 1:N
endsyntype;

```

Les réponses <answer> omises sont des abréviations d'écriture des littéraux 1 à N comme constantes <constant> des conditions <range condition> dans les N réponses <answer>.

## 11.14 Liste d'instructions

Une liste d'instructions peut être utilisée dans une tâche <task>, une zone <task area>, une définition <procedure definition> ou une définition <operation definition> pour définir des variables locales à la liste d'instructions et un certain nombre d'actions à interpréter. L'objectif d'une liste d'instructions est d'obtenir des descriptions textuelles concises d'algorithmes à associer avec le SDL/GR ou la forme moins concise SDL/PR. La sémantique d'une liste d'instructions est déterminée par la transformation des instructions selon les modèles décrits ci-dessous, de sorte que les instructions soient effectivement interprétées de gauche à droite.

Une instruction de définition de variable peut introduire des variables au début d'une liste <statement list>. Contrairement à la définition <variable definition> en 12.3.1, il n'est pas nécessaire que l'initialisation des variables dans ce contexte soit une expression <constant expression>.

### *Grammaire textuelle concrète*

```
<statement list> ::=
    <variable definitions> <statements>

<variable definitions> ::=
    { <variable definition statement> } *

<statements> ::=
    <statement> *

<statement> ::=
    <empty statement>
    | <compound statement>
    | <assignment statement>
    | <algorithm action statement>
    | <call statement>
    | <expression statement>
    | <if statement>
    | <decision statement>
    | <loop statement>
    | <terminating statement>
    | <labelled statement>
    | <exception statement>

<terminating statement> ::=
    <return statement>
    | <break statement>
    | <loop break statement>
    | <loop continue statement>
    | <raise statement>
```

Une instruction <loop break statement> et une instruction <loop continue statement> ne peuvent se produire à l'intérieur d'une instruction <loop statement>.

Une instruction <terminating statement> ne peut se produire qu'en tant que dernière instruction <statement> dans <statements>. Si la dernière instruction <statement> dans la liste <statement list> est une instruction <terminating statement>, la liste <statement list> est terminale.

```
<variable definition statement> ::=
    decl <local variables of sort> { , <local variables of sort> } * <end>

<local variables of sort> ::=
    <variable name> { , <variable name> } * <sort> [ <is assigned sign> <expression> ]
```



### 11.14.1 Instruction composée

Des instructions multiples peuvent être groupées en une seule instruction.

<i>Compound-node</i>	::	<i>Connector-name</i> <i>Variable-definition-set</i> <i>Init-graph-node*</i> <i>Transition</i> <i>Step-graph-node*</i>
<i>Init-graph-node</i>	=	<i>Graph-node</i>
<i>Step-graph-node</i>	=	<i>Graph-node</i>
<i>Continue-node</i>	::	<i>Connector-name</i>
<i>Break-node</i>	::	<i>Connector-name</i>

#### Grammaire textuelle concrète

<compound statement> ::=  
                                <left curly bracket> <statement list> <right curly bracket>

L'instruction <compound statement> représente un nœud *Compound-node*. Le nom *Connector-name* est représenté par le nom anonyme nouvellement créé. L'ensemble *Variable-definition-set* est représenté par la liste de toutes les définitions <variable definition> dans la liste <statement list>. La *Transition* est représentée par la transformation de <statements> dans la liste <statement list> ou par la transformation de <statements> dans la liste <statement list> suivie d'un nœud *Break-node* ayant un nom *Connector-name*, si la liste <statement list> n'est pas terminale.

#### Sémantique

Une instruction <compound statement> crée sa propre portée.

L'interprétation d'un nœud *Compound-node* procède de la manière suivante:

- a) une variable locale est créée pour chaque définition *Variable-definition* dans l'ensemble *Variable-definition-set*;
- b) la liste de *Init-graph-nodes* est interprétée;
- c) la *Transition* est interprétée;
- d) lorsqu'un nœud *Continue-node* ayant un nom *Connector-name* correspondant au nom *Connector-name* est interprété, la liste des nœuds *Step-graph-nodes* est interprétée et une interprétation plus poussée continue à l'étape c);
- e) lorsque l'interprétation du nœud *Compound-node* s'achève, toutes les variables créées par cette interprétation cessent d'exister. L'interprétation d'un nœud *Compound-node* s'achève:
  - i) lorsqu'un nœud *Break-node* est interprété;
  - ii) lorsqu'un nœud *Continue-node* ayant un nom *Connector-name* différent du nom *Connector-name* dans le nœud *Compound-node* est interprété;
  - iii) lorsqu'un nœud *Return-node* est interprété;
  - iv) lorsqu'une instance d'exception est créée qui n'est pas gérée dans la *Transition* du nœud *Compound-node*;
- f) ensuite, l'interprétation se poursuit de la manière suivante:
  - i) si l'interprétation du nœud *Compound-node* s'est achevée du fait de l'interprétation d'un nœud *Break-node* ayant un nom *Connector-name* correspondant au nom *Connector-name*, l'interprétation se poursuit au nœud suivant le nœud *Compound-node*; autrement
  - ii) si l'interprétation du nœud *Compound-node* s'est achevée du fait de l'interprétation d'un nœud *Break-node*, *Continue-node* ou *Return-node*, l'interprétation se poursuit respectivement avec l'interprétation du nœud *Break-node*, *Continue-node* ou *Return-node* au point d'invocation du nœud *Compound-node*; autrement

- iii) si l'interprétation du nœud *Compound-node* s'est achevée du fait de la création d'une instance d'exception, l'interprétation se poursuit comme décrit en 11.16.

### Modèle

Si la liste <statement list> contient une définition <variable definitions>, ce qui suit est exécuté pour chaque instruction <variable definition statement>. Un nouveau nom <variable name> est créé pour chaque nom <variable name> dans l'instruction <variable definition statement>. Chaque occurrence de <variable name> dans les instructions <variable definition statement> suivantes et à l'intérieur de <statements> est remplacée par le nom <variable name> nouvellement créé correspondant.

Pour chaque instruction <variable definition statement>, une définition <variable definition> est formée à partir de l'instruction <variable definition statement> en omettant l'expression <expression> d'initialisation (si elle est présente) et insérée en tant qu'instruction <variable definition statement> à la place de l'instruction <variable definition statement> d'origine. Si une expression <expression> d'initialisation est présente, une instruction <assignment statement> est construite pour chaque nom <variable name> mentionné dans les variables <local variables of sort> dans l'ordre de leur occurrence, où le nom <variable name> reçoit le résultat de l'<expression>. Ces instructions <assignment statement> sont insérées en tête des instructions <statement> dans l'ordre de leur occurrence.

La liste <statement list> est équivalente à la concaténation de la transformation de chaque instruction <variable definition statement> et à la transformation de chaque instruction <statement> dans <statements> (voir 11.14.1 à 11.14.7).

NOTE – La liste <statement list> non vide transformée devient une liste d'instructions <action statement> et d'instructions <terminator statement> séparées par des points virgules et se terminant par un point virgule et peut par conséquent être traitée comme une <transition>.

Si la liste <statement list> est vide, le résultat de sa transformation est le texte vide.

### 11.14.2 Actions et terminateurs de transition en tant qu'instructions

A l'intérieur d'une liste d'instructions, une instruction d'affectation n'est pas précédée du mot clé **task** et un appel de procédure ne nécessite pas le mot clé **call**. La plupart des autres instructions <action statement> qui peuvent être utilisées dans une transition (voir 11.12.1) et le terminateur de retour <return> d'une transition (voir 11.12.1) peuvent être utilisés en tant qu'instructions <statement> dans une liste <statement list>.

#### Grammaire textuelle concrète

```

<assignment statement> ::=
    <assignment> <end>

<algorithm action statement> ::=
    <output> <end>
    | <create request> <end>
    | <set> <end>
    | <reset> <end>
    | <export> <end>

<return statement> ::=
    <return> <end>

<raise statement> ::=
    <raise> <end>

<call statement> ::=
    [call] <procedure call body> <end>

```

Une instruction <return statement> est permise uniquement à l'intérieur d'une définition <procedure definition> ou à l'intérieur d'une définition <operation definition>.

Le mot clé **call** ne peut pas être omis si l'instruction <call statement> présente une ambiguïté syntaxique avec une application ou une variable d'opération portant le même nom.

NOTE – Cette ambiguïté n'est pas résolue par le contexte.

*Modèle*

L'instruction <assignment statement> est transformée en tâche <task>

```
task <assignment> ;
```

Une instruction <call statement> est une syntaxe dérivée pour l'appel <procedure call> et est transformée en un appel <procedure call> ayant le même corps <procedure call body>:

```
call <procedure call body> ;
```

La transformation d'une instruction <algorithm action statement>, d'une instruction <return statement> et d'une instruction <raise statement> est obtenue en supprimant l'extrémité de poursuite <end>.

### 11.14.3 Expressions en tant qu'instructions

Les expressions qui sont des applications d'opération peuvent être utilisées comme des instructions, auquel cas l'application <operation application> est interprétée et le résultat est ignoré.

*Grammaire textuelle concrète*

```
<expression statement> ::=  
    <operation application> <end>
```

*Modèle*

Un nouveau nom <variable name> est créé. Une définition <variable definition> est ajoutée à la fin de la liste *Decl-list* qui déclare le nom nouvellement créé <variable name> comme étant de la même sorte que le résultat de l'application <operation application>. Enfin, l'instruction d'expression est transformée en:

```
task <variable name> := <operation application> ;
```

### 11.14.4 Instruction Si

L'expression <Boolean expression> est interprétée et, si elle renvoie la valeur booléenne prédéfinie Vrai (true), l'instruction <consequence statement> est interprétée, sinon, l'instruction <alternative statement> est interprétée si elle est présente.

*Grammaire textuelle concrète*

```
<if statement> ::=  
    if ( <Boolean expression> ) <consequence statement>  
        [ else <alternative statement> ]
```

```
<consequence statement> ::=  
    <statement>
```

```
<alternative statement> ::=  
    <statement>
```

Une instruction <alternative statement> est associée à l'instruction <consequence statement> qui la précède la plus proche.

## Modèle

L'instruction <if statement> équivaut à l'instruction <action statement> suivante, qui indique une décision <decision>:

```
decision <Boolean expression> ;  
  ( true ) : task { <consequence statement>-transform } ;  
  ( false ) : task { <alternative statement>-transform } ;  
enddecision ;
```

La transformation de l'instruction <alternative statement> est insérée uniquement si l'instruction <alternative statement> était présente.

### 11.14.5 Instruction de décision

L'instruction de décision est une forme concise de décision. L'<expression> est évaluée et la partie <algorithm answer part> dont la condition <range condition> contient le résultat de l'expression est interprétée. Des conditions d'intervalle chevauchant ne sont pas permises. Contrairement à une décision <decision> (voir 11.13.5), il n'est pas nécessaire que l'expression corresponde à l'une des conditions d'intervalle. S'il n'y a pas de correspondance et qu'une instruction <alternative statement> existe, cette instruction <alternative statement> est interprétée. S'il n'y a pas de correspondance et qu'il n'existe pas d'instruction <alternative statement>, l'interprétation se poursuit après l'instruction <decision statement>.

#### Grammaire textuelle concrète

```
<decision statement> ::=  
    decision ( <expression> ) <left curly bracket>  
        <decision statement body>  
    <right curly bracket>  
  
<decision statement body> ::=  
    <algorithm answer part>+ [ <algorithm else part> ]  
  
<algorithm answer part> ::=  
    ( <range condition> ) <colon> <statement>  
  
<algorithm else part> ::=  
    else <colon> <alternative statement>
```

## Modèle

Une partie <algorithm answer part> est transformée en la partie <answer part> suivante.

```
( <range condition> ) : task { <statement>-transform } ;
```

Un corps <decision body> est ensuite formé en prenant la transformation de chaque partie <algorithm answer part> dans l'ordre et en ajoutant la partie <else part> suivante.

```
else : task { <alternative statement>-transform } ;
```

où la transformation de l'instruction <alternative statement> est insérée uniquement si l'instruction <alternative statement> est présente. Le corps <decision body> qui en résulte est désigné par *Body*. L'instruction <decision statement> équivaut à l'instruction <action statement> suivante:

```
decision ( <expression> ) ;  
  Body  
enddecision ;
```

### 11.14.6 Instruction de boucle

L'instruction <loop statement> fournit une facilité généralisée pour les itérations liées et non liées d'une instruction <loop body statement>, ayant un nombre arbitraire de variables de boucle. Ces variables peuvent être définies à l'intérieur de l'instruction <loop statement> et sont divisées en étapes spécifiées par l'étape <loop step>. Elles peuvent être utilisées à la fois pour générer des

résultats successifs et pour accumuler des résultats. Lorsque l'instruction <loop statement> s'achève, une instruction <finalization statement> peut être interprétée dans le contexte des variables de boucle.

L'instruction <loop body statement> est interprétée de manière répétée. L'interprétation de la boucle est contrôlée par la présence d'une clause <loop clause> quelconque. Une clause <loop clause> débute avec une indication <loop variable indication> qui fournit un moyen approprié de déclarer et d'initialiser des variables de boucle locales. La portée et la durée d'une variable quelconque dans une définition <loop variable indication> sont en fait celles de l'instruction <loop statement>. Si l'initialisation est présente en tant qu'<expression> dans une définition <loop variable definition>, l'expression est évaluée une seule fois avant la première interprétation du corps de boucle. Alternativement, toute variable visible peut être définie comme une variable de boucle et peut se voir affecter un élément de données. Avant chaque itération, tous les éléments de l'expression <Boolean expression> sont évalués. L'interprétation de l'instruction <loop statement> est terminée si un quelconque élément de l'expression <Boolean expression> est Faux (false). En conséquence, s'il n'y a pas d'expression <Boolean expression> présente, l'interprétation de l'instruction <loop statement> se poursuivra jusqu'à ce que l'instruction <loop statement> soit sortie non localement. Si une indication <loop variable indication> est présente dans cette clause <loop clause>, l'étape <loop step> dans chaque clause de boucle calcule et affecte le résultat de la variable de boucle respective à la fin de chaque itération. Si aucune indication <loop variable indication> n'était présente dans une clause <loop clause>, ou si aucune étape <loop step> n'était présente, aucune instruction d'affectation de la variable de boucle n'est réalisée. L'indication <loop variable indication>, l'expression <Boolean expression> et l'étape <loop step> sont facultatives. Une variable de boucle est visible mais ne doit pas être affectée dans l'instruction <loop body statement>.

L'interprétation du corps de boucle se termine également lorsqu'une instruction d'interruption **break** est atteinte. Atteindre une instruction de poursuite **continue** entraîne l'interprétation de la boucle à passer immédiatement à l'itération suivante. (Voir également <break statement> en 11.14.7).

Si une instruction <loop statement> se termine "normalement" [c'est-à-dire par une expression <Boolean expression> évaluant à la valeur booléenne prédéfinie Faux (false)], l'instruction <finalization statement> est interprétée. Une variable de boucle est visible et conserve son résultat lorsque l'instruction <finalization statement> est interprétée. Une instruction d'interruption ou de poursuite à l'intérieur de l'instruction <finalization statement> met fin à l'instruction extérieure suivante <loop statement>.

### *Grammaire textuelle concrète*

```

<loop statement> ::=
    for ( [ <loop clause> { ; <loop clause> }* ]
          <loop body statement> [ then <finalization statement> ]

<loop body statement> ::=
    <statement>

<finalization statement> ::=
    <statement>

<loop clause> ::=
    [ <loop variable indication>
      , [ <Boolean expression> ]
      <loop step> ]

<loop step> ::=
    [ , [ { <expression> | [ call ] <procedure call body> } ] ]

<loop variable indication> ::=
    <loop variable definition>
    | <variable identifier> [ <is assigned sign> <expression> ]

<loop variable definition> ::=
    decl <variable name> <sort> <is assigned sign> <expression>

```

<loop break statement> ::=  
    **break** <end>

<loop continue statement> ::=  
    **continue** <end>

Le mot clé **call** ne peut pas être omis dans une étape <loop step> si cela conduit à une ambiguïté avec une application d'opération ou une variable d'opération ayant le même nom.

L'identificateur <procedure identifieur> dans le corps <procedure call body> d'une étape <loop step> ne doit pas se rapporter à un appel de procédure renvoyant une valeur.

Une instruction <finalization statement> est associée avec l'instruction <loop body statement> la plus proche.

Une instruction <loop statement> représente un nœud *Compound-node*. Le nom *Connector-name* est représenté par un nom anonyme nouvellement créé, appelé *Label*.

L'ensemble *Variable-definition-set* est représenté par la liste d'instructions <variable definition statement> construite à partir du nom <variable name> et de la sorte <sort> mentionnés dans chaque définition <loop variable definition>.

La liste des nœuds *Init-graph-nodes* est représentée par la transformation de la liste <statement list> construite à partir d'instructions <assignment statement> formées à partir de chaque indication <loop variable indication> dans l'ordre de leur occurrence.

Une instruction <assignment statement> est construite à partir de chaque clause <loop clause> entre le nom <variable name> ou l'identificateur <variable identifieur> et l'<expression> dans l'étape <loop step>, si à la fois l'indication <loop variable indication> et l'<expression> étaient présentes. Une instruction <statements> est construite en prenant ces éléments <assignment statement> les uns après les autres, ou l'<expression>, l'appel <procedure call> ou le corps <procedure call body> dans l'étape <loop step>, si aucune instruction <assignment statement> n'avait été construite. <statements> représente la liste de nœuds *Step-graph-nodes*.

La *Transition* est représentée par la <decision> qui suit, dans laquelle *Limit* se rapporte à l'<expression> construite en combinant tous les éléments <Boolean expression> à l'aide de l'opérateur prédéfini "and" de type booléen:

```
decision Limit ;  
  ( True ) : task { <loop body statement>-transform }; Continue ;  
  ( False ) : task { <finalization statement>-transform };  
enddecision ;
```

La transformation de l'instruction <finalization statement> est insérée uniquement si une instruction <finalization statement> était présente à l'origine.

Une instruction <loop continue statement> représente un nœud *Continue-node*. Le nom *Connector-name* est représenté par *Label* de l'instruction de boucle englobante la plus à l'extérieur.

### Modèle

Chaque occurrence d'une instruction <loop break statement> à l'intérieur d'une clause <loop clause>, de l'instruction <loop body statement> ou d'une instruction <finalization statement> d'une autre instruction <loop statement> contenue dans cette instruction <loop statement>, toutes ne se produisant pas dans une autre instruction interne <loop statement>, est remplacée par:

```
break Label ;
```

Si l'expression <Boolean expression> est absente d'une clause <loop clause>, la valeur booléenne prédéfinie Vrai (true) est insérée en tant que l'expression <Boolean expression>.

Ensuite l'instruction <loop statement> est remplacée par l'instruction ainsi modifiée <loop statement> suivie d'une instruction <labelled statement> ayant le nom <connector name> *Break*.

### 11.14.7 Instructions d'interruption et instructions étiquetées

Une instruction `<break statement>` est une forme plus restrictive d'un branchement `<join>`.

Un nom peut être attribué à une instruction `<break statement>` et entraîne le transfert immédiat de l'interprétation dans l'instruction suivant celle portant l'étiquette `<label>` correspondante.

#### *Grammaire textuelle concrète*

```
<break statement> ::=  
    break <connector name> <end>  
  
<labelled statement> ::=  
    <label> <statement>
```

Une instruction `<break statement>` doit être contenue dans une instruction qui a été étiquetée avec un nom `<connector name>` donné.

Une instruction `<break statement>` représente un nœud *Break-node* ayant le nom *Connector-name* représenté par le nom `<connector name>`.

Une instruction `<labelled statement>` représente un nœud *Compound-node* ayant le nom *Connector-name* représenté par le nom `<connector name>` dans l'étiquette `<label>`.

### 11.14.8 Instruction vide

Une instruction peut être vide, cela est indiqué par l'utilisation d'un unique point virgule. L'instruction `<empty statement>` n'a pas d'effet.

#### *Grammaire textuelle concrète*

```
<empty statement> ::=  
    <end>
```

#### *Modèle*

La transformation de l'instruction `<empty statement>` est le texte vide.

### 11.14.9 Instruction d'exception

Une instruction peut être encapsulée à l'intérieur d'un gestionnaire d'exception.

#### *Grammaire textuelle concrète*

```
<exception statement> ::=  
    try <try statement> <handle statement>+  
  
<try statement> ::=  
    <statement>  
  
<handle statement> ::=  
    handle ( <exception stimulus list> ) <statement>
```

Une instruction `<handle statement>` est associée à l'instruction `<try statement>` précédente la plus proche. L'instruction `<try statement>` ne doit pas être une instruction `<break statement>`.

Le gestionnaire d'exception construit dans le *Modèle* représente l'exception *On-exception* du nœud *Local-scope-node* représenté par l'instruction `<compound statement>` qui est obtenue à partir de l'instruction `<try statement>` (voir au 11.14.1).

#### *Sémantique*

Une instruction `<exception statement>` crée sa propre portée avec un gestionnaire d'exception.

## Modèle

Si l'instruction <try statement> n'était pas une instruction <compound statement>, l'instruction <try statement> est d'abord transformée en une instruction <compound statement>:

```
{ <try statement> }
```

Ensuite, l'instruction (transformée) <try statement> et toutes les instructions <handle statement> sont transformées. Pour chaque instruction <handle statement>, le traitement <handle> ci-après est construit:

```
handle <exception stimulus list>;
    task { <handle statement>-transform };
```

Les traitements <handle> construits sont recueillis dans une liste appelée *HandleParts*. Un gestionnaire d'exception est construit et porte un nom anonyme. Le nom du gestionnaire d'exception est désigné par *handler*.

```
exceptionhandler handler;
    HandleParts
endexceptionhandler;
```

La transformation de l'instruction <exception statement> est la transformation de l'instruction <try statement>.

## 11.15 Temporisateur

### Grammaire abstraite

```
Timer-definition      ::   Timer-name
                          Sort-reference-identif*
Timer-name            =   Name
Set-node              ::   Time-expression
                          Timer-identif
                          Expression*
Reset-node            ::   Timer-identif
                          Expression*
Timer-identif         =   Identif
Time-expression       =   Expression
```

Les sortes de la liste d'*Expressions* dans le nœud *Set-node* et le nœud *Reset-node* doivent correspondre par leur position à la liste d'identificateurs *Sort-reference-identif* suivant immédiatement le nom *Timer-name* identifié par l'identificateur *Timer-identif*.

### Grammaire textuelle concrète

```
<timer definition> ::=
    timer
    <timer definition item> { , <timer definition item>* <end>
<timer definition item> ::=
    <timer name> [ <sort list> ] [ <timer default initialization> ]
<timer default initialization> ::=
    <is assigned sign> <Duration constant expression>
<reset> ::=
    reset ( <reset clause> { , <reset clause>* } )
<reset clause> ::=
    <timer identif> [ ( <expression list> ) ]
<set> ::=
    set <set clause> { , <set clause>* }
<set clause> ::=
    ( [ <Time expression> , ] <timer identif> [ ( <expression list> ) ] )
```



Une clause <set clause> peut omettre l'expression <Time expression>, si l'identificateur <timer identifier> désigne un temporisateur qui a une initialisation <timer default initialization> dans sa définition.

Une clause <reset clause> représente un nœud *Reset-node*; une clause <set clause> représente un nœud *Set-node*.

Il n'y a pas de syntaxe graphique correspondante pour <reset> et <set>.

### Sémantique

Une instance de temporisateur est un objet qui peut être actif ou inactif. Deux occurrences d'un identificateur de temporisateur suivies par une liste d'expressions se réfèrent à la même instance de temporisateur uniquement si l'expression d'égalité (voir 12.2.7) appliquée à toutes les expressions correspondantes dans les listes donne la valeur booléenne prédéfinie Vrai (true) (c'est-à-dire si les deux listes d'expressions ont le même résultat).

Lorsqu'un temporisateur inactif est initialisé, une valeur de temps est associée au temporisateur. A condition qu'il n'y ait pas de réinitialisation ou d'autres initialisations de ce temporisateur avant que le temps du système atteigne cette valeur de temps, un signal portant le même nom que le temporisateur est appliqué à l'accès d'entrée de l'agent. On agit de la même manière lorsque le temporisateur est initialisé à une valeur de temps inférieure ou égale à **now**. Après traitement d'un signal de temporisateur, l'expression **sender** prend le même résultat que l'expression **self**. Si une liste d'expressions est donnée lors de l'initialisation du temporisateur, les résultats de ces expressions sont contenus dans le même ordre dans le signal du temporisateur. Un temporisateur est actif à partir du moment de son initialisation jusqu'au moment du traitement du signal du temporisateur.

Si une sorte spécifiée dans une définition de temporisateur est un syntype, la vérification d'intervalle définie au 12.1.9.5 et appliquée à l'expression correspondante dans une initialisation ou réinitialisation, doit être la valeur booléenne prédéfinie vrai (True); dans le cas contraire, l'exception prédéfinie *OutOfRange* est déclenchée.

Lorsqu'un temporisateur inactif est réinitialisé, il reste inactif.

Lorsqu'un temporisateur actif est réinitialisé, l'association avec la valeur de temps est perdue, lorsqu'il y a un signal de temporisateur correspondant retenu dans l'accès d'entrée, il est alors supprimé, et le temporisateur devient inactif.

Lorsqu'un temporisateur actif est initialisé, cela équivaut à réinitialiser le temporisateur, opération immédiatement suivie par l'initialisation du temporisateur. Entre cette réinitialisation et cette initialisation, le temporisateur reste actif.

Avant la première initialisation d'une instance de temporisateur, celle-ci est inactive.

Les expressions *Expressions* dans un nœud *Set-node* ou *Reset-node* sont évaluées dans un ordre donné.

### Modèle

Une clause <set clause> qui ne contient pas d'expression <Time expression> est une syntaxe dérivée pour une clause <set clause> dans laquelle l'expression <Time expression> est:

**now** + <Duration constant expression>

où <Duration constant expression> est obtenue à partir de l'initialisation <timer default initialization> dans la définition du temporisateur.

Une réinitialisation <reset> ou une initialisation <set> peut contenir plusieurs clauses <reset clause> ou <set clause> respectivement. Cela constitue une syntaxe dérivée pour spécifier une séquence de <reset> ou <set>, une pour chaque clause <reset clause> ou <set clause> de manière à

conserver l'ordre original dans lequel elles ont été spécifiées dans <reset> ou <set>. Cette abréviation est développée avant toute extension des abréviations qui se trouvent dans les expressions contenues.

## 11.16 Exception

Une instance d'exception transfère le contrôle à un gestionnaire d'exception.

### Grammaire abstraite

*Exception-definition* :: *Exception-name*  
*Sort-reference-identifïer*\*

*Exception-name* = *Name*

*Exception-identifïer* = *Identifïer*

### Grammaire textuelle concrète

<exception definition> ::= **exception** <exception definition item> { , <exception definition item> }\* <end>

<exception definition item> ::= <exception name> [ <sort list> ]

### Sémantique

Une instance d'exception indique qu'une situation exceptionnelle (généralement une situation d'erreur) s'est produite pendant l'interprétation d'un système. Une instance d'exception est créée implicitement par le système sous-jacent ou explicitement par un nœud *Raise-node* et elle cesse d'exister si elle est interceptée par un nœud *Handle-node* ou *Else-handle-node*.

La création d'une instance d'exception interrompt le flux de commande normal à l'intérieur de l'agent, de l'opération ou de la procédure. Si une instance d'exception est créée à l'intérieur d'une procédure appelée, d'une opération appelée ou d'une instruction composée appelée et qu'elle n'est pas interceptée à cet endroit, cette procédure, opération ou instruction composée se termine respectivement et l'instance d'exception se propage (dynamiquement) à l'extérieur vers l'appelant et est traitée comme si elle était créée à la place de l'appel de procédure, à la place de l'application d'opération ou à la place de l'invocation de l'instruction composée. Cette règle s'applique également aux appels de procédures distantes; et, dans ce cas, l'instance d'exception se propage vers l'instance de processus appelante en plus de se propager à l'intérieur de l'instance d'agent appelé.

Plusieurs types d'exception sont prédéfinis à l'intérieur du paquetage *Predefined*. Ces types d'exception sont ceux qui peuvent être créés implicitement par le système sous-jacent. Le spécificateur est également autorisé à créer des instances de ces types d'exception de manière explicite.

Si une instance d'exception est créée à l'intérieur d'une instance d'agent et qu'elle n'est pas interceptée dans cette instance, le comportement ultérieur du système est non défini.

### 11.16.1 Gestionnaire d'exception

#### Grammaire abstraite

*Exception-handler-node* :: *Exception-handler-name*  
*[On-exception]*  
*Handle-node-set*  
*[Else-handle-node]*

*Exception-handler-name* = *Name*

Les nœuds *Exception-handler-nodes* à l'intérieur d'un graphe *State-transition-graph* ou *Procedure-graph* donné doivent tous avoir des noms *Exception-handler-names* différents.

NOTE – Un nom *Exception-handler-name* peut avoir le même nom qu'un *State-name*. Ils sont cependant différents.

Les identificateurs *Exception-identifïer* dans l'ensemble *Handler-node-set* doivent être distincts.

## Grammaire textuelle concrète

```
<exception handler> ::=
    exceptionhandler <exception handler list> <end>
    [<on exception>]
    <handle>*
    [ endexceptionhandler [ <exception handler name> ] <end> ]

<exception handler list> ::=
    <exception handler name> { , <exception handler name> }*
    | <asterisk exception handler list>

<asterisk exception handler list> ::=
    <asterisk> [ ( <exception handler name> { , <exception handler name> }* ) ]
```

Lorsque la liste <exception handler list> contient un nom <exception handler name>, le gestionnaire <exception handler name> représente un nœud *Exception-handler-node*. Pour chaque nœud *Exception-handler-node*, l'ensemble *Handle-node-set* est représenté par les traitements <handle> contenant des identificateurs <exception identifier> dans leurs listes <exception stimulus list>. Pour chaque nœud *Exception-handler-node*, le nœud *Else-handle-node* est représenté par <handle > explicite ou implicite dans laquelle la liste <exception stimulus list> est une liste <asterisk exception stimulus list>.

Les noms <exception handler name> dans une liste <asterisk exception handler list> doivent être distincts et doivent être contenus dans d'autres listes <exception handler list> dans le corps englobant ou dans le corps d'un supertype.

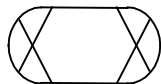
Un gestionnaire <exception handler> contient au plus une liste <asterisk exception stimulus list> (voir 11.16.3).

Un gestionnaire <exception handler> a au plus un gestionnaire <exception handler> associé.

## Grammaire graphique concrète

```
<exception handler area> ::=
    <exception handler symbol> contains <exception handler list>
    [ is connected to <on exception association area> ]
    is associated with <exception handler body area>
```

```
<exception handler symbol> ::=
```



```
<exception handler body area> ::=
    <handle association area>*
```

```
<handle association area> ::=
    <solid association symbol> is connected to <handle area>
```

Une zone <exception handler area> représente un ou plusieurs nœuds *Exception-handler-node*. Les symboles <solid association symbol> provenant d'un symbole <exception handler symbol> peuvent avoir un trajet d'origine commun. Une zone <exception handler area> doit contenir un nom <state name> (pas une liste <asterisk state list>) si elle correspond avec une zone <on exception area>.

## Sémantique

Un gestionnaire d'exception représente une condition particulière dans laquelle un agent, une opération ou une procédure peut gérer une instance d'exception qu'il ou elle a créée. Le traitement d'une instance d'exception entraîne une transition. L'état du processus ou de la procédure reste inchangé.

Si le nœud *Exception-handler-node* n'a pas de nœud *Handle-node* ayant le même identificateur *Exception-identifier* que l'instance d'exception, celle-ci est interceptée par le nœud *Else-handle-node*.

S'il n'y a pas de nœud *Else-handle-node*, l'instance d'exception n'est pas traitée dans ce gestionnaire d'exception.

### Modèle

Lorsque la liste `<exception handler list>` d'un gestionnaire `<exception handler>` contient plus d'un nom `<exception handler name>`, une copie du gestionnaire `<exception handler>` est créée pour chacun de ces noms `<exception handler name>`. Le gestionnaire `<exception handler>` est ensuite remplacé par ces copies.

Un gestionnaire `<exception handler>` ayant une liste `<asterisk exception handler list>` est transformé en une liste de gestionnaires `<exception handler>`, un pour chaque nom `<exception handler name>` du corps en question, à l'exception des noms `<exception handler name>` contenus dans la liste `<asterisk exception handler list>`.

## 11.16.2 On-Exception

### Grammaire abstraite

`On-exception` :: `Exception-handler-name`

Le nom `Exception-handler-name` spécifié dans `On-exception` doit être le nom d'un gestionnaire `<exception handler>` à l'intérieur du même graphe `State-transition-graph` ou `Procedure-graph`.


### Grammaire textuelle concrète

`<on exception>` ::= **onexception** `<exception handler name>` `<end>`

Un nom `<exception handler name>` peut apparaître dans plus d'un gestionnaire `<exception handler>` d'un corps.

### Grammaire graphique concrète

`<on exception association area>` ::= `<solid on exception association symbol>` **is connected to** { `<on exception area>` | `<exception handler area>` }

`<solid on exception association symbol>` ::= 

`<on exception area>` ::= `<exception handler symbol>` **contains** `<exception handler name>`

Un symbole `<solid on exception association symbol>` peut être constitué de plusieurs segments linéaires horizontaux et verticaux. La flèche doit être attachée à la zone `<on exception area>` ou à la zone `<exception handler area>`.

### Sémantique

Une expression `On-exception` indique le gestionnaire d'exception dans lequel il convient qu'un agent, une opération ou une procédure entre si l'agent, l'opération ou la procédure crée une instance d'exception. Un gestionnaire d'exception est associé à une autre entité par le biais d'une expression `<on exception>` ou d'une zone `<on exception association area>`. Un gestionnaire d'exception est dit actif lorsqu'il est capable de réagir à la création d'une instance d'exception.

Plusieurs gestionnaires d'exception peuvent être actifs en même temps. Pour chaque instance d'agent, de procédure ou d'opération, il existe plusieurs portées d'exception qui peuvent contenir un gestionnaire d'exception actif. Les portées d'exception sont, dans l'ordre croissant de localisation:

- a) le graphe entier de l'instance;
- b) les états composites (si un état composite est interprété);
- c) le graphe des états composites (s'il existe);

- d) l'état courant;
- e) la transition pour le stimulus dans l'état courant, ou la transition de départ;
- f) l'état d'exception courant;
- g) la transition pour la clause de gestion courante;
- h) l'action courante.

En raison de l'emboîtement des états composites, plus d'un gestionnaire d'exception peut être actif à tout moment pour un état composite ou un graphe d'état composite.

Lorsqu'une instance d'exception est créée, les gestionnaires d'exception actifs sont visités dans l'ordre croissant de localisation. Lorsqu'un état d'exception est visité, le gestionnaire d'exception appartient à la portée d'exception courante désactivée. Si aucun gestionnaire d'exception n'est actif pour une portée d'exception donnée, ou si l'état d'exception gère l'exception, la portée d'exception suivante est visitée.

Aucun gestionnaire d'exception n'est actif pendant l'interprétation d'une expression <constant expression>.

Un gestionnaire d'exception peut être associé à un graphe complet d'agent/de procédure/d'opération, à une transition de départ, à un état, à un gestionnaire d'exception, à un déclencheur d'état (par exemple entrée ou gestion) avec sa transition associée, à la plupart des actions de transition ou à certains terminateurs de transition. Le texte suivant décrit chaque cas dans lequel le gestionnaire d'exception est activé et désactivé.

- a) *graphe complet d'agent/de procédure/d'opération*  
le gestionnaire d'exception est activé au début de l'interprétation du graphe de l'instance d'agent, d'opération ou de procédure; le gestionnaire d'exception est désactivé lorsque l'instance d'agent, d'opération ou de procédure est dans la condition d'arrêt ou lorsqu'elle cesse d'exister;
- b) *transition de départ*  
le gestionnaire d'exception est activé lorsque l'interprétation de la transition de départ débute dans l'agent, l'opération ou la procédure; le gestionnaire d'exception est désactivé lorsque l'agent ou la procédure interprète un nœud d'état suivant nextstate ou est dans la condition d'arrêt ou cesse d'exister;
- c) *état composite*  
le gestionnaire d'exception est activé lorsqu'on entre dans l'état composite; il est actif pour l'état composite, y compris tous nœuds *Connect-nodes* ou transitions associés à l'état. Il est désactivé lorsque l'interprétation entre dans une autre état;
- d) *graphe d'état composite*  
le gestionnaire d'exception est activé avant d'invoquer la procédure d'entrée d'un état composite; il est désactivé à l'issue de la procédure de sortie de l'état composite;
- e) *état*  
le gestionnaire d'exception est activé lorsque l'agent ou la procédure entre dans l'état donné. Le gestionnaire d'exception est désactivé lorsque l'agent ou la procédure interprète un nœud nextstate ou entre dans une condition d'arrêt ou cesse d'exister;
- f) *gestionnaire d'exception*  
le gestionnaire d'exception est activé lorsque l'agent ou la procédure entre dans le gestionnaire d'exception donné; il est désactivé lorsque l'agent ou la procédure entre dans un nœud nextstate ou entre dans une condition d'arrêt ou cesse d'exister;

g) *entrée*

le gestionnaire d'exception pour le stimulus est activé lorsque l'interprétation du nœud *Input-node* donné est déclenchée dans l'agent ou la procédure. Le gestionnaire d'exception est désactivé lorsque l'agent ou la procédure interprète un nœud *Nextstate-node* ou entre dans une condition d'arrêt ou cesse d'exister;

h) *traitement*

le gestionnaire d'exception pour le nœud courant *Handle-node* est activé lorsque l'interprétation de la *Transition* du nœud *Handle-node* est déclenchée dans l'agent ou la procédure. Il est désactivé lorsqu'un nœud *Nextstate-node* est entré dans l'agent, l'opération ou la procédure;

i) *décision*

le gestionnaire d'exception est activé lorsque l'interprétation de la décision donnée débute dans l'agent, l'opération ou la procédure. Il est désactivé lorsque l'agent ou la procédure entre dans la transition d'une branche de décision (c'est-à-dire que le gestionnaire d'exception couvre l'expression de la décision et si l'expression correspond à l'un quelconque des intervalles des branches de décision);

j) *action de transition (à l'exception de la décision)*

le gestionnaire d'exception est activé lorsque l'interprétation de l'action donnée est déclenchée dans l'agent, l'opération ou la procédure. Il est désactivé lorsque l'interprétation de l'agent ou de la procédure est terminée;

k) *terminateur de transition (avec expressions)*

le gestionnaire d'exception est activé lorsque l'agent, l'opération ou la procédure entre dans le terminateur donné. Il est désactivé lorsque l'interprétation du terminateur est terminée.

Tout gestionnaire d'exception est désactivé lorsqu'il traite une exception et crée une instance d'exception. Le gestionnaire d'exception pour les actions et les terminateurs couvre également les actions qui résultent du modèle de la <transition>, par exemple l'expression <import expression>.

NOTE – Les règles ci-dessus indiquent que dans certains cas, plusieurs gestionnaires d'exception peuvent être désactivés en même temps. Par exemple, si un gestionnaire d'exception pour un état et un gestionnaire d'exception pour une transition d'entrée associée sont actifs en même temps, les deux gestionnaires d'exception sont désactivés lorsque la transition d'entrée entre dans un nœud d'état suivant. Les états ou les stimuli implicites sont couverts par les gestionnaires d'exception du contexte syntaxique; c'est-à-dire que les expressions <on exception> ou les zones <on exception association area> sont copiées dans le modèle.

### *Modèle*

Lorsque plusieurs gestionnaires <exception handler> contiennent le même nom <exception handler name>, ces gestionnaires <exception handler> sont concaténés en un gestionnaire <exception handler> qui porte ce nom <exception handler name>.

Dans une spécialisation, l'association avec le gestionnaire d'exception est considérée comme faisant partie du graphe ou de la transition. Si une transition virtuelle est redéfinie, la nouvelle transition remplace une expression <on exception> de la transition d'origine. Si un graphe ou un état est hérité dans une spécialisation, tout gestionnaire d'exception associé est également hérité.

### **11.16.3 Traitement**

#### *Grammaire abstraite*

<i>Handle-node</i>	::	<i>Exception-identifier</i> <i>[Variable-identifier]*</i> <i>[On-exception]</i> <i>Transition</i>
<i>Else-handle-node</i>	::	<i>[On-exception]</i> <i>Transition</i>

La longueur de la liste d'identificateurs *Variable-identifieur* facultatifs dans le nœud *Handle-node* doit être la même que le nombre d'identificateurs *Sort-reference-identifieur* dans la définition *Exception-definition* désignée par l'identificateur *Exception-identifieur*.

Les sortes des variables doivent correspondre par leur position aux sortes des éléments de données qui peuvent être supportés par l'exception.

#### Grammaire textuelle concrète

```

<handle> ::=
    handle [<virtuality>] <exception stimulus list> <end>
    [<on exception>] <transition>

<exception stimulus list> ::=
    <exception stimulus> { , <exception stimulus> }*
    |
    <asterisk exception stimulus list>

<exception stimulus> ::=
    <exception identifieur> [ ( [ <variable> ] { , [ <variable> ] }* ) ]

<asterisk exception stimulus list> ::=
    <asterisk>
  
```

Lorsque la liste <exception stimulus list> contient un stimulus <exception stimulus>, le traitement <handle > représente un nœud *Handle-node*. Un traitement <handle > comportant une liste <asterisk exception stimulus list> représente un nœud *Else-handle-node*.

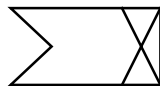
Les virgules peuvent être omises après la dernière <variable> dans le stimulus <exception stimulus>.

#### Grammaire graphique concrète

```

<handle area> ::=
    <handle symbole> contains { [<virtuality>] <exception stimulus list> }
    [ is connected to <on exception association area> ]
    is followed by <transition area>

<handle symbole> ::=
  
```



Le trajet vers la zone <transition area> dans la zone <handle area> doit provenir du symbole <handle symbole>.

Une zone <handle area> dont la liste <exception stimulus list> contient un stimulus <exception stimulus>, correspond à un nœud *Handle-node*. Chacun des identificateurs <exception identifieur> contenus dans un symbole <handle symbole> donne le nom de l'un des nœuds *Handle-node* que ce symbole <handle symbole> représente. Une zone <handle area> comportant une liste <asterisk exception stimulus list> représente un nœud *Else-handle-node*.

#### Sémantique

Un nœud *Handle-node* traite une instance du type d'exception spécifié. Le traitement de l'instance d'exception entraîne l'acheminement des informations par l'instance d'exception disponible vers l'agent ou la procédure. Les éléments de données acheminés par l'instance d'exception traitée sont attribués aux variables mentionnées dans le nœud *Handle-node*.

Les éléments de données seront assignés aux variables de gauche à droite. Si aucune variable n'est mentionnée pour une position de paramètre dans l'exception, les éléments de données à cette position sont ignorés. Si aucun élément de données n'est associé à une position de paramètre donnée, la variable correspondante devient "non définie".

Le même résultat que celui de l'expression **self** est donné à l'expression **sender**.

NOTE – L'expression **state** ne change pas par rapport au nom du gestionnaire d'exception.

## Modèle

Lorsque la liste <exception stimulus list> d'un certain traitement <handle > contient plus d'un stimulus <exception stimulus>, une copie du traitement <handle > est créée pour chaque stimulus <exception stimulus>. Le traitement <handle > est ensuite remplacé par ces copies.

Lorsqu'une ou plusieurs <variable> d'un certain stimulus <exception stimulus> sont des variables <indexed variable> ou <field variable>, toutes les <variable> sont remplacées par des identificateurs <variable identifier> uniques, nouveaux et déclarés implicitement. Immédiatement avant la <transition> du traitement <handle>, une tâche <task> est insérée, qui contient dans son corps <textual task body> une affectation <assignment> pour chacune des <variable>, permettant l'affectation du résultat de la nouvelle variable à la <variable>. Les résultats sont attribués dans l'ordre de gauche à droite de la liste de <variable>. Cette tâche <task> devient la première instruction <action statement> dans la <transition>.

Un traitement <handle > ou une zone <handle area> qui contient <virtuality> est appelée transition de traitement virtuelle. Les transitions de traitement virtuelles sont décrites en détail en 8.3.3.

## 12 Données

Le concept de données dans le SDL est défini dans le présent paragraphe. Ceci inclut la terminologie relative aux données, les concepts à définir et les données prédéfinies.

Les données dans le SDL sont principalement abordées sous l'aspect des types de données. Un type de données définit un ensemble d'éléments ou d'éléments de données, désignés par le terme *sorte*, et un ensemble d'opérations qui peuvent être appliquées à ces éléments de données. Les sortes et les opérations définissent les propriétés du type de données. Ces propriétés sont définies par des définitions de type de données.

Un type de données consiste en un ensemble, qui est la *sorte* du type de données, ainsi qu'en une ou plusieurs *opérations*. Le type de données booléen prédéfini est donné à titre d'exemple. La sorte booléenne du type de données booléen se compose des éléments Vrai (true) et Faux (false). On trouve parmi les opérations du type de données booléen les éléments suivants: "=" (égal), "/=" (différent de), "not", "and", "or", "xor", and "=>" (implique). On considère à titre d'exemple supplémentaire le type de données prédéfini naturel. Il comporte la sorte naturelle, comportant les éléments 0, 1, 2, etc. et les opérations "=", "/=", "+", "-", "\*", "/", "mod", "rem", "<", ">", "<=", ">=", et puissance.

Le SDL fournit plusieurs types de données prédéfinis qui sont connus tant du point de vue de leur comportement que de leur syntaxe. Les types de données prédéfinis sont décrits dans l'Annexe D.

Les variables sont des objets qui peuvent être associés à un élément d'une sorte par affectation. Lorsqu'on accède à la variable, l'élément de données associé est retourné.

Les éléments de la sorte d'un type de données sont des *valeurs*, des *objets* qui sont des références pour les valeurs ou des *pids*, qui sont des références pour les agents. La sorte d'un type de données peut être définie de la manière suivante:

- a) énumération explicite des éléments de la sorte;
- b) formation du produit cartésien des sortes  $S_1, S_2, \dots, S_n$ ; la sorte est égale à l'ensemble comportant tous les tuples pouvant être formés en prenant le premier élément de la sorte  $S_1$ , le deuxième élément de la sorte  $S_2, \dots$ , et enfin, en prenant le dernier élément de la sorte  $S_n$ ;
- c) les sortes de pids sont définies par la définition d'une interface (voir 12.1.2);
- d) plusieurs sortes sont prédéfinies et forment la base des types de données prédéfinis décrits dans l'Annexe D. Les sortes prédéfinies Any et Pid sont décrites en 12.1.5 et 12.1.6.



Si les éléments d'une sorte sont des objets, la sorte est une sorte d'objet. Si les éléments d'une sorte sont des pids, la sorte est une sorte de pid. Les éléments d'une sorte de valeur sont des valeurs.

Les opérations sont définies à partir et vers les éléments des sortes. Par exemple, l'application de l'opération somme ("+") à partir et vers les éléments de la sorte des entiers est valable, tandis que la somme des éléments de la sorte booléenne ne l'est pas.

Chaque élément de données appartient à une seule et unique sorte. C'est-à-dire que les sortes n'ont jamais d'éléments de données en commun.

Pour la plupart des sortes, il y a des formes littérales qui décrivent les éléments de la sorte: par exemple, pour les entiers, "2" est utilisé de préférence à "1 + 1". Il peut y avoir plusieurs littéraux qui décrivent le même élément de données: par exemple, 12 et 012 peuvent être utilisés pour décrire le même élément de données entier. La même description littérale peut être utilisée pour plus d'une sorte; par exemple, 'A' est à la fois un caractère et une chaîne de caractères de longueur 1. Certaines sortes peuvent ne pas avoir de formes littérales pour désigner les éléments de la sorte; par exemple, les sortes peuvent également être formées comme le produit cartésien d'autres sortes. Dans ce cas, les éléments de ces sortes sont désignés par des opérations qui construisent l'élément de données à partir d'éléments de la ou des sortes de composante.

Une expression désigne un élément de données. Si une expression ne contient pas de variable ou d'expression obligatoire, par exemple, si elle est un littéral d'une sorte donnée, chaque occurrence de l'expression décrira toujours le même élément de données. Ces expressions "passives" correspondent à une utilisation fonctionnelle du langage.

Une expression qui contient des variables ou des expressions obligatoires peut être interprétée comme présentant différents résultats durant l'interprétation d'un système SDL selon l'élément de données associé aux variables. L'utilisation active de données comprend l'affectation, l'utilisation et l'initialisation des variables. La différence entre les expressions actives et passives est que le résultat d'une expression passive est indépendant de son instant d'interprétation, tandis qu'une expression active peut avoir des résultats différents selon les valeurs, les objets ou les pids actuels associés aux variables ou à l'état actuel du système.

## 12.1 Définitions de données

Les définitions de données sont utilisées pour définir des types de données. Les mécanismes de base pour définir des données résident dans les définitions de type de données (voir 12.1.1) et les interfaces (voir 12.1.2). La spécialisation (voir 12.1.3) permet de fonder la définition d'un type de données sur un autre type de données, appelé supertype. La définition de la sorte du type de données, ainsi que les opérations prévues pour la sorte, sont élaborées par les constructeurs de type de données (voir 12.1.7). Des opérations supplémentaires peuvent être définies, comme décrit en 12.1.4. Le sous-paragraphe 12.1.8 indique comment définir le comportement des opérations d'un type de données.

Dans la mesure où les données prédéfinies sont définies dans un paquetage Predefined et utilisé de manière implicite (voir 7.2 et D.3), les sortes prédéfinies (par exemple, booléenne et naturelle) et leurs opérations peuvent être librement utilisées dans tout le système. La sémantique d'égalité (12.2.5), des expressions conditionnelles (12.2.6), et des syntypes (12.1.9.4) se fonde sur la définition du type de données booléen (voir D.3.1). La sémantique de la classe de nom (voir 12.1.9.1) se fonde également sur la définition des types de données Character (caractère) et Charstring (chaîne de caractères) (voir D.3.2 et D.3.4).

## Grammaire abstraite

<i>Data-type-definition</i>	=	<i>Value-data-type-definition</i>   <i>Object-data-type-definition</i>   <i>Interface-definition</i>
<i>Value-data-type-definition</i>	::	<i>Sort</i> <i>Data-type-identifiser</i> <i>Literal-signature-set</i> <i>Static-operation-signature-set</i> <i>Dynamic-operation-signature-set</i>
<i>Object-data-type-definition</i>	::	<i>Sort</i> <i>Data-type-identifiser</i> <i>Literal-signature-set</i> <i>Static-operation-signature-set</i> <i>Dynamic-operation-signature-set</i>
<i>Interface-definition</i>	::	<i>Sort</i> <i>Data-type-identifiser*</i>
<i>Data-type-identifiser</i>	=	<i>Identifiser</i>
<i>Sort-reference-identifiser</i>	=	<i>Sort-identifiser</i>   <i>Syntype-identifiser</i>   <i>Expanded-sort-identifiser</i>   <i>Reference-sort-identifiser</i>
<i>Sort-identifiser</i>	=	<i>Identifiser</i>
<i>Expanded-sort-identifiser</i>	=	<i>Sort-identifiser</i>
<i>Reference-sort-identifiser</i>	=	<i>Sort-identifiser</i>
<i>Sort</i>	=	<i>Name</i>

Une définition *Data-type-definition* introduit une sorte qui est visible dans l'unité de portée englobante dans la syntaxe abstraite. Elle peut également introduire un ensemble de littéraux et d'opérations.

## Grammaire textuelle concrète

<code>&lt;data definition&gt; ::=</code>	<code>&lt;data type definition&gt;</code>   <code>&lt;interface definition&gt;</code>   <code>&lt;syntype definition&gt;</code>   <code>&lt;synonym definition&gt;</code>
--	--

Une définition de données représente une définition *Data-type-definition* si elle est une définition `<data type definition>`, `<interface definition>`, ou une définition `<syntype definition>`.

<code>&lt;sort&gt; ::=</code>	<code>&lt;basic sort&gt;</code>   <code>&lt;anchored sort&gt;</code>   <code>&lt;expanded sort&gt;</code>   <code>&lt;reference sort&gt;</code>   <code>&lt;pid sort&gt;</code>
<code>&lt;basic sort&gt; ::=</code>	<code>&lt;sort identifier&gt;</code>   <code>&lt;syntype&gt;</code>
<code>&lt;anchored sort&gt; ::=</code>	<b>this</b> [ <code>&lt;basic sort&gt;</code> ]
<code>&lt;expanded sort&gt; ::=</code>	<b>value</b> <code>&lt;basic sort&gt;</code>
<code>&lt;reference sort&gt; ::=</code>	<b>object</b> <code>&lt;basic sort&gt;</code>
<code>&lt;pid sort&gt; ::=</code>	<code>&lt;sort identifier&gt;</code>

Une sorte `<anchored sort>` ayant une sorte `<basic sort>` est uniquement permise à l'intérieur de la définition de sorte `<basic sort>`.

Une sorte <anchored sort> n'est une forme syntaxique concrète légale que si elle apparaît à l'intérieur d'une définition de type de données <data type definition>. La sorte de base <basic sort> contenue dans la sorte ancrée <anchored sort> doit nommer la sorte <sort> introduite par la définition de type de données <data type definition>.

### Sémantique

Une définition de données est utilisée pour la définition d'un type de données ou d'une interface ou pour la définition d'un synonyme d'une expression conformément à la définition plus complète donnée en 12.1, 12.1.9.4 ou 12.1.9.6.

Chaque définition <data type definition> introduit une sorte ayant le même nom que le nom <data type name> (voir 12.1.1). Chaque définition <interface definition> introduit une sorte ayant le même nom que le nom <interface name> (voir 12.1.2).

NOTE 1 – Pour alléger le texte, il a été convenu d'utiliser la phrase "la sorte S" à la place de "la sorte définie par le type de données S" ou "la sorte définie par l'interface S" lorsque aucune confusion n'est possible.

Un identificateur de sorte <sort identifier> nomme une sorte <sort> introduite par une définition de type de données.

Une sorte est un ensemble d'éléments: valeurs, objets (c'est-à-dire références à des valeurs) ou pids (c'est-à-dire références à des agents). Deux sortes différentes n'ont pas d'élément en commun. Une définition <value data type definition> introduit une sorte qui est un ensemble de valeurs. Une définition <object data type definition> introduit une sorte qui est un ensemble d'objets. Une définition <interface definition> introduit une sorte de pid.

Si une sorte <sort> est une sorte <expanded sort>, les variables, les synonymes, les champs, les paramètres, le retour, les signaux, les temporisateurs et les exceptions définis avec cette sorte <sort> seront associés à des valeurs de la sorte plutôt qu'aux références de ces valeurs, même si la sorte a été définie comme un ensemble d'objets. Un identificateur *Expanded-sort-identifier* est représenté par une sorte <expanded sort>.

Si une sorte <sort> est une sorte <reference sort>, les variables, les synonymes, les champs, les paramètres, le retour, les signaux, les temporisateurs et les exceptions définis avec cette sorte <sort> seront associés aux références des valeurs de la sorte plutôt qu'aux valeurs de la sorte, même si la sorte a été définie comme un ensemble de valeurs. Un identificateur *Referenced-sort-identifier* est représenté par une sorte <reference sort>.

La signification d'une sorte <anchored sort> est donnée en 12.1.3.

L'identificateur <sort identifier> dans une sorte <pid sort> doit référencer une sorte de pid.

### Modèle

Une sorte <expanded sort> avec une sorte <basic sort> qui représente une sorte de valeur est remplacée par la sorte <basic sort>.

Une sorte <reference sort> avec une sorte <basic sort> qui représente une sorte d'objet est remplacée par la sorte <basic sort>.

NOTE 2 – Par conséquent, le mot clé **value** n'a pas d'effet si la sorte a été définie comme un ensemble de valeurs et le mot clé **object** n'a pas d'effet si la sorte a été définie comme un ensemble d'objets.

Une sorte <anchored sort> qui n'a pas de sorte <basic sort> est une abréviation pour spécifier une sorte <basic sort> qui référence le nom de la définition de type de donnée ou la définition de sous-type de donnée dans le contexte duquel la sorte <anchored sort> se produit.

## 12.1.1 Définition des types de données

### Grammaire textuelle concrète

```
<data type definition> ::=
    {<package use clause>}*
    <type preamble> <data type heading> [<data type specialization>]
    {
        <end> [ <data type definition body> <data type closing> <end>]
        | <left curly bracket> <data type definition body> <right curly bracket> }

<data type definition body> ::=
    {<entity in data type>}* [<data type constructor>] <operations>
    [<default initialization>[ <end> ] ]

<data type closing> ::=
    { endvalue | endobject } type [<data type name>]

<data type heading> ::=
    { value | object } type <data type name>
    [ <formal context parameters> ] [<virtuality constraint>]

<entity in data type> ::=
    <data type definition>
    | <syntype definition>
    | <synonym definition>
    | <exception definition>

<operations> ::=
    <operation signatures>
    <operation definitions>

<data type reference> ::=
    <type preamble>
    { value | object } type <data type identifier> <type reference properties>
```

Une définition <value data type definition> contient le mot clé "value" dans l'en-tête <data type heading>. Une définition <object data type definition> contient le mot clé "object" dans l'en-tête <data type heading>.

Un paramètre <formal context parameter> de <formal context parameters> doit être un paramètre <sort context parameter> ou un paramètre <synonym context parameter>.

Le mot clé **value** ou **object** et le nom <data type name> dans la fermeture <data type closing> doivent correspondre aux mots clés **endvalue** ou **endobject**, respectivement, et au nom dans l'en-tête correspondant <data type heading>, s'ils sont présents.

### Grammaire graphique concrète

```
<data type reference area> ::=
    <type reference area>
```

La zone <type reference area> qui fait partie d'une zone <data type reference area> doit avoir un en-tête <graphical type reference heading> qui contient un nom <data type name>.

### Sémantique

Une définition <data type definition> se compose d'un constructeur <data type constructor> qui décrit les éléments de la sorte (voir 12.1.6) et les opérations induites par le mode de construction de la sorte, ainsi que des <operations> qui définissent un ensemble d'opérations pouvant être appliquées aux éléments d'une sorte (voir 12.1.4). Un type de données peut également être fondé sur un supertype par le biais d'une spécialisation (voir 12.1.3).

## 12.1.2 Définition d'interface

Les interfaces sont définies dans des paquetages, des agents ou des types d'agent. Une interface définit une sorte de pid qui a des éléments qui sont des références à des agents.

Une interface peut définir des signaux, des procédures distantes, des variables distantes et des exceptions. Le contexte de définition des entités définies dans l'interface est l'unité de portée de l'interface et les entités définies sont visibles lorsque l'interface est visible. Une interface peut également définir des signaux, des procédures distantes ou des variables distantes définies à l'extérieur de l'interface par la liste <interface use list>.

Une interface est utilisée dans une liste de signaux pour indiquer que les signaux, les appels de procédure distante et les variables distantes de la définition d'interface sont inclus dans la liste de signaux.

### Grammaire textuelle concrète

```

<interface definition> ::=
    {<package use clause>}*
    [<virtuality>] <interface heading>
    [<interface specialization>]
        <end> <entity in interface>* [<interface use list>]
    <interface closing>
    |
    {<package use clause>}*
    |
    [<virtuality>] <interface heading>
    |
    [<interface specialization>] <end>
    |
    {<package use clause>}*
    |
    [<virtuality>] <interface heading>
    |
    [<interface specialization>] <left curly bracket>
        <entity in interface>* [<interface use list>]
    <right curly bracket>

<interface closing> ::=
    endinterface [<interface name>] <end>

<interface heading> ::=
    interface <interface name>
        [<formal context parameters>] [<virtuality constraint>]

<entity in interface> ::=
    <signal definition>
    |
    <interface variable definition>
    |
    <interface procedure definition>
    |
    <exception definition>

<interface use list> ::=
    use <signal list> <end>

<interface variable definition> ::=
    dcl <remote variable name> { , <remote variable name>* <sort> <end>

<interface procedure definition> ::=
    procedure <remote procedure name> <procedure signature> <end>

<interface reference> ::=
    [<virtuality>]
    interface <interface identifier> <type reference properties>

```

La liste <signal list> dans une définition <interface definition> doit contenir uniquement des identificateurs <signal identifier>, <remote procedure identifier>, <remote variable identifier> et <signal list identifier>. Si un identificateur <signal list identifier> fait partie de la liste <signal list>, il doit également respecter cette restriction.

Les paramètres <formal context parameters> doivent contenir uniquement des paramètres <signal context parameter>, <remote procedure context parameter>, <remote variable context parameter>, <sort context parameter> ou <exception context parameter>.

Le contexte de définition des entités définies dans l'interface (<entity in interface>) est l'unité de portée de l'interface et les entités définies sont visibles lorsque l'interface est visible.

## Grammaire graphique concrète

<interface reference area> ::=  
    <type reference area>

La zone <type reference area> qui fait partie d'une zone <interface reference area> doit avoir un en-tête <graphical type reference heading> qui contient un nom <interface name>.

## Sémantique

La sémantique de <virtuality> est définie en 8.3.2.

L'adjonction d'un identificateur <interface identifier> dans une liste <signal list> signifie que les identificateurs de signal, de procédure distante et de variable distante faisant partie de la définition <interface definition> sont inclus dans la liste <signal list>.

## Modèle

Les interfaces sont implicitement définies par les définitions d'agent, de sa machine à états et de type d'agent. L'interface implicitement définie porte le même nom que l'agent ou le type d'agent qui l'a définie.

NOTE 1 – Etant donné que chaque agent et chaque type d'agent possède une interface définie implicitement avec le même nom, chaque interface définie explicitement doit toujours avoir un nom différent de chaque agent et type d'agent défini dans le même domaine de visibilité, afin d'éviter des collisions de noms.

L'interface définie par une machine à états contient dans sa spécialisation <interface specialization> toutes les interfaces données dans la liste de signaux entrants associée aux accès explicites ou implicites de la machine à états. L'interface contient également dans sa liste <interface use list> tous les signaux, variables distantes et procédures distantes donnés dans la liste des signaux entrants associée aux accès explicites ou implicites de la machine à états.

L'interface définie par un agent ou un type d'agent contient dans sa spécialisation <interface specialization> l'interface définie par l'état composite représentant sa machine à états.

L'interface définie par un agent fondé sur le type contient dans sa spécialisation <interface specialization> l'interface définie par son type.

NOTE 2 – Pour alléger le texte, il a été convenu d'utiliser la phrase "la sorte de pid de l'agent A" à la place de "la sorte de pid définie par l'interface implicitement définie par l'agent A" lorsque aucune confusion n'est possible.

### 12.1.3 Spécialisation des types de données

La spécialisation permet de définir un type de données fondé sur un autre (super) type. La sorte définie par la spécialisation est considérée comme une sous-sort de la sorte définie par le type de base. La sorte définie par le type de base est une supersorte de la sorte définie par la spécialisation.

## Grammaire textuelle concrète

<data type specialization> ::=  
    **inherits** <data type type expression> [<renaming>] [**adding**]

<interface specialization> ::=  
    **inherits** <interface type expression> { , <interface type expression> }\* [**adding**]

<renaming> ::=  
    ( <rename list> )

<rename list> ::=  
    <rename pair> { , <rename pair> }\*

<rename pair> ::=  
    <operation name> <equals sign> <base type operation name>  
    |  
    <literal name> <equals sign> <base type literal name>

L'identificateur *Data-type-identifieur* dans la définition *Data-type-définition* représentée par la définition <data type definition> dans laquelle est contenue une spécialisation <data type specialization> (ou <interface specialization>) identifie le type de données représenté par l'expression <data type type expression> dans sa spécialisation <data type specialization> (voir également 8.1.2).

Une définition *Interface-définition* peut contenir une liste d'identificateurs *Data-type-identifieur*. L'interface désignée par une définition *Interface-définition* est une spécialisation de toutes les interfaces désignées par les identificateurs *Data-type-identifieur*.

Le contenu qui résulte d'une définition d'interface spécialisée (<interface specialization>) se compose du contenu des supertypes suivi du contenu de la définition spécialisée. Cela signifie que l'ensemble des signaux, des procédures distantes et des variables distantes de la définition spécialisée constitue l'union des éléments donnés dans la définition spécialisée proprement dite et des éléments des supertypes. L'ensemble de définitions qui en résulte doit respecter les règles données en 6.3.

Le constructeur <data type constructor> doit être du même genre que le constructeur <data type constructor> utilisé dans la définition <data type definition> de la sorte référencée par l'expression <data type type expression> dans la spécialisation <data type specialization>. C'est-à-dire que si le constructeur <data type constructor> utilisé dans un supertype (direct ou indirect) était une liste <literal list> (<structure definition>, <choice definition>), le constructeur <data type constructor> doit également être une liste <literal list> (<structure definition>, <choice definition>).

Le renommage peut être utilisé pour modifier le nom de littéraux, d'opérateurs et de méthodes hérités dans le type de données dérivé.

Tous les noms <literal name> et tous les noms <base type literal name> d'une liste <rename list> doivent être distincts.

Tous les noms <operation name> et tous les noms <base type operation name> d'une liste <rename list> doivent être distincts.

Un nom <base type operation name> spécifié dans une liste <rename list> doit être une opération ayant un nom <operation name> défini dans la définition de type de données définissant le type <base type> de l'expression <data type type expression>.

### Sémantique

La compatibilité de sorte détermine le moment où une sorte peut être utilisée à la place d'une autre sorte et le moment où elle ne peut pas être utilisée à la place d'une autre sorte. Cette relation est utilisée pour les affectation (voir 12.3.3), pour le transfert de paramètre (voir 12.2.7 et 9.4), pour la re-déclaration de types de paramètre lors de l'héritage (voir 12.1.2) et pour les paramètres de contexte réels (voir 8.1.2).

Soient T et V deux sortes. V est compatible avec T si et seulement si:

- a) V et T sont la même sorte;
- b) V est directement compatible avec T;
- c) T a été définie par un type d'objet ou une interface et, pour une sorte U quelconque, V est compatible avec U et U est compatible avec T.

NOTE 1 – La compatibilité de sorte est transitive uniquement pour les sortes définies par des types d'objet ou des interfaces, elle n'est pas transitive pour les sortes définies par des types de valeur.

Soient T et V, deux sortes. V est directement compatible avec T si et seulement si:

- a) V est désignée par une sorte <basic sort> et T est une sorte d'objet et T est une supersorte de V;
- b) V est désignée par une sorte <anchored sort> de la forme **this** T;
- c) V est désignée par une sorte <reference sort> de la forme **object** T;

- d) T est désignée par une sorte <reference sort> de la forme **object** V;
- e) V est désignée par une sorte <expanded sort> de la forme **value** T;
- f) T est désignée par une sorte <expanded sort> de la forme **value** V;
- g) V est désignée par une sorte <pid sort> (voir 12.1.2) et T est une supersorte de V.

### Modèle

Le modèle de la spécialisation en 8.2.14 est utilisé et complété comme suit.

Un type de données spécialisé est fondé sur un autre type de données (base) au moyen d'une définition <data type definition> en association avec une spécialisation <data type specialization>. La sorte définie par la spécialisation est distincte de la sorte définie par le type de base.

Si la sorte définie par le type de base comporte des littéraux définis, les noms des littéraux sont hérités en tant que noms pour les littéraux de la sorte définie par le type spécialisé, sauf si le renommage de littéral a été effectué pour ce littéral. Le renommage de littéral a été effectué pour un littéral si le nom du littéral du type de base apparaît comme le second nom d'une paire <rename pair>, auquel cas le littéral est renommé d'après le premier nom de cette paire.

Si le type de base comporte des opérateurs ou des méthodes définis, les noms des opérateurs sont hérités en tant que noms pour les opérateurs ou les méthodes de la sorte définie, et sont soumis aux restrictions indiquées en 8.3.1, sauf si l'opérateur ou la méthode a été déclaré comme privé (voir 12.1.9.3) ou si le renommage de l'opération a été effectué pour cet opérateur ou cette méthode. Le renommage d'opération a été effectué pour un opérateur ou une méthode si le nom d'opération hérité apparaît comme le second nom d'une paire <rename pair>, auquel cas l'opérateur ou la méthode est renommé d'après le premier nom de cette paire.

Lorsque plusieurs opérateurs ou méthodes du type <base type> de l'expression <sort type expression> ont le même nom que le nom <base type operation name> dans une paire <rename pair>, tous ces opérateurs ou méthodes sont renommés.

Dans chaque occurrence d'une sorte <anchored sort> dans le type spécialisé, la sorte <basic sort> est remplacée par la sous-sorte.

Les sortes d'argument et le résultat d'un opérateur ou d'une méthode hérité sont les mêmes que ceux de l'opérateur ou de la méthode correspondant du type de base, si ce n'est que dans chaque <argument> contenant une sorte <anchored sort> dans l'opérateur ou la méthode hérité, la sorte <basic sort> est remplacée par la sous-sorte. Pour les méthodes virtuelles héritées, <argument virtuality> est ajoutée à un <argument> contenant une sorte <anchored sort>, si elle n'est pas déjà présente.

NOTE 2 – Conformément au modèle de spécialisation décrit en 8.2.14, un opérateur est hérité uniquement si sa signature contient au moins une sorte <anchored sort> ou si le renommage a été effectué.

## 12.1.4 Opérations

### Grammaire abstraite

<i>Dynamic-operation-signature</i>	=	<i>Operation-signature</i>
<i>Static-operation-signature</i>	=	<i>Operation-signature</i>
<i>Operation-signature</i>	::	<i>Operation-name</i> <i>Formal-argument*</i> [ <i>Result</i> ]
<i>Operation-name</i>	=	<i>Name</i>
<i>Formal-argument</i>	=	<i>Virtual-argument</i>   <i>Nonvirtual-argument</i>
<i>Virtual-argument</i>	::	<i>Argument</i>
<i>Nonvirtual-argument</i>	::	<i>Argument</i>
<i>Argument</i>	=	<i>Sort-reference-identifier</i>



La notion de compatibilité de sorte est étendue aux signatures *Operation-signature*. Une signature *Operation-signature* S1 a une sorte compatible avec la sorte d'une signature *Operation-signature* S2 lorsque:

- a) S1 et S2 possèdent le même nombre d'arguments Formal-arguments;
- b) pour chaque argument *Virtual-argument* A de S1, la sorte identifiée par son identificateur *Sort-reference-identifiser* est compatible avec la sorte identifiée par l'identificateur *Sort-reference-identifiser* de l'argument correspondant de S2;
- c) pour chaque argument *Nonvirtual-argument* A de S1, la sorte identifiée par son identificateur *Sort-reference-identifiser* est la même que la sorte identifiée par l'identificateur *Sort-reference-identifiser* de l'argument correspondant de S2.

### Grammaire textuelle concrète

```

<operation signatures> ::=
    [<operator list>] [<method list>]

<operator list> ::=
    operators <operation signature> { <end> <operation signature> } * <end>

<method list> ::=
    methods <operation signature> { <end> <operation signature> } * <end>

<operation signature> ::=
    <operation preamble>
    { <operation name> | <name class operation> }
    [<arguments>] [<result>] [<raises>]

<operation preamble> ::=
    [<virtuality>] [<visibility>] | [<visibility>] [<virtuality>]

<operation name> ::=
    <operation name>
    | <quoted operation name>

<arguments> ::=
    ( <argument> { , <argument> } * )

<argument> ::=
    [<argument virtuality>] <formal parameter>

<formal parameter> ::=
    <parameter kind> <sort>

<argument virtuality> ::=
    virtual

<result> ::=
    <result sign> <sort>

```

Dans une signature *Operation-signature*, chaque identificateur *Sort-reference-identifiser* de l'argument *Formal-argument* est représenté par une sorte <sort> d'argument et le résultat *Result* est représenté par la sorte <sort> de résultat. Une sorte <sort> dans un argument <argument> qui contient <argument virtuality> représente un argument *Virtual-argument*; autrement, la <sort> de l'<argument> représente un argument *Nonvirtual-argument*.

Le nom *Operation-name* est unique à l'intérieur de l'unité de portée de définition dans la syntaxe abstraite, même si le nom <operation name> correspondant peut ne pas être unique. Le nom unique *Operation-name* provient:

- a) du nom <operation name>; plus
- b) de la liste (éventuellement vide) d'identificateurs de sorte d'argument; plus
- c) de l'identificateur de sorte de résultat; plus
- d) de l'identificateur de sorte de la définition de type de données dans laquelle le nom <operation name> est défini.

Le nom <quoted operation name> permet d'obtenir des noms d'opérateur et de méthode qui ont une forme syntaxique particulière. La syntaxe particulière est telle que, par exemple, les opérations arithmétiques et booléennes peuvent avoir leur forme syntaxique habituelle. C'est-à-dire que l'utilisateur peut écrire "(1 + 1) = 2" au lieu d'avoir à employer par exemple `equal(add(1,1),2)`.

Si la signature <operation signature> est contenue dans une liste <operator list>, la signature <operation signature> représente une signature *Static-operation-signature* et la signature <operation signature> ne doit pas contenir <virtuality> ou <argument virtuality>.

Si la signature <operation signature> est contenue dans une liste <method list> et que <virtuality> n'est pas présente, la signature <operation signature> représente une signature *Static-operation-signature* et aucun des <argument> ne doit contenir <argument virtuality>.

Si la signature <operation signature> est contenue dans une liste <method list> et que <virtuality> est présente, la signature <operation signature> représente une signature *Dynamic-operation-signature*. Dans ce cas, un ensemble de signatures *Dynamic-operation-signature* est formé qui se compose de la signature *Dynamic-operation-signature* représentée par la signature <operation signature> et de tous les éléments de l'ensemble de signatures de la méthode correspondante dans le supertype avec un nom *Operation-name* obtenu à partir du même nom <operation name>, en tenant compte du renommage et de telle manière que la signature *Operation-signature* ait une sorte compatible avec celle de la signature *Operation-signature* dans le supertype, si elle existe.

Cet ensemble doit être fermé dans le sens suivant: pour deux signatures *Operation-signatures*  $S_i$  et  $S_j$  quelconques dans l'ensemble de signatures *Operation-signatures*, la signature unique *Operation-signature*  $S$  telle que:

- a)  $S$  a une sorte compatible avec celle de  $S_i$  et de  $S_j$ ;
- b) pour toute signature *Operation-signature*  $S_k$  dont la sorte est compatible avec celle de  $S_i$  et de  $S_j$ , la sorte de  $S_k$  est également compatible avec celle de  $S$ ,

appartient également à l'ensemble de signatures *Dynamic-operation-signature*.

Cette condition permet de s'assurer que l'ensemble de signatures *Dynamic-operation-signature* forme un treillis et garantit qu'une signature *Operation-signature* correspondante unique et la meilleure possible peut être trouvée lors de l'interprétation d'une application d'opération (voir 12.2.7). Si l'ensemble de signatures *Dynamic-operation-signature* ne satisfait pas cette condition, la spécification <sdl specification> est illégale.

NOTE – La spécialisation d'un type peut nécessiter l'adjonction de signatures *Operation-signature* supplémentaires dans la liste <method list> pour satisfaire la condition.

Le résultat <result> dans une signature <operation signature> peut être omis uniquement si la signature <operation signature> est apparue dans une liste <method list>.

<argument virtuality> est légale uniquement si <virtuality> contient les mots clés **virtual** ou **redefined**.

### Sémantique

Les formes indiquées pour les opérateurs ou les méthodes infixes ou monadiques constituent des noms valides pour les opérateurs ou les méthodes.

Un opérateur ou une méthode a une sorte de résultat qui est la sorte identifiée par le résultat.

### Modèle

Si la signature <operation signature> est contenue dans une liste <method list>, il s'agit d'une syntaxe dérivée et elle est transformée comme suit: un <argument> est construit à partir du mot clé **virtual**, si <virtuality> était présente, du genre <parameter kind>**in/out** et de l'identificateur <sort

identifier> de la sorte définie par la définition englobante <data type definition>. S'il n'y avait pas d'<arguments>, les <arguments> sont formés à partir de l'<argument> construit et insérés dans la signature <operation signature>. S'il y a des <arguments>, l'<argument> construit est ajouté au début de la liste d'origine d'<argument> dans les <arguments>.

Si la sorte <sort> d'un <argument> est une sorte <anchored sort>, l'<argument> contient implicitement <argument virtuality>. Si une signature <operation signature> contient les mots clés **redefined** dans <virtuality>, pour chaque <argument> dans la signature <operation signature> correspondante du type de base, si cet <argument> contient (implicitement ou explicitement) <argument virtuality>, alors l'<argument> correspondant dans la signature <operation signature> contient également implicitement <argument virtuality>.

Un <argument> sans genre explicite <parameter kind> a le genre implicite <parameter kind> **in**.

### 12.1.5 Quelconque (Any)

Chaque type de valeur ou d'objet est (directement ou indirectement) un sous-type du type d'objet abstrait Any. Lorsqu'une variable est déclarée être de la sorte Any, les éléments de données appartenant à une quelconque sorte de valeur ou d'objet peuvent être affectés à cette variable.

Any est implicitement défini par la définition <data type definition> suivante, où Boolean est la sorte booléenne prédéfinie:

```

abstract object type Any
operators
  equal      ( this Any, this Any ) -> Boolean;
  clone      ( this Any           ) -> this Any;
methods
  virtual is_equal ( this Any           ) -> Boolean;
  virtual copy      ( this Any           ) -> this Any;
endobject type Any;

```

NOTE 1 – Dans la mesure où tous les constructeurs redéfinissent implicitement les méthodes virtuelles du type de données Any, ces méthodes ne peuvent pas être explicitement redéfinies dans une définition <data type definition>.

De plus, chaque définition <object data type definition> introduisant une sorte nommée S implique des signatures *Operation-signature*, ce qui équivaut à inclure la définition explicite dans les signatures <operation signature> suivantes dans la liste <operator list>:

```

Null      -> this S;
Make      -> this S;

```

ainsi que la signature <operation signature> suivante dans la liste <method list>:

```

virtual finalized -> <<S>> Null;

```

#### *Grammaire textuelle concrète*

Any du type de donnée peut être qualifié par le Predefined **package**.

#### *Sémantique*

Les opérateurs et les méthodes définis par Any sont disponibles pour tout type de valeur ou d'objet.

Chaque définition <object data type definition> ajoute un élément de donnée unique désignant une référence qui n'a pas encore été associée à une valeur. L'opérateur Null retourne cet élément de donnée. Toute tentative d'obtention d'une valeur associée à partir de l'objet retourné par Null déclenche l'exception prédéfinie InvalidReference (voir D.3.16).

L'opérateur *Make* introduit par une définition <object data type definition> crée un nouvel élément, non initialisé, de la sorte <result sort> de l'opérateur *Make*. Chaque définition <object data type definition> fournit une définition appropriée pour l'opérateur *Make*.

L'opérateur *equal* compare deux valeurs pour l'égalité (lorsqu'elle est définie pour un type de valeur) ou compare deux valeurs référencées par des objets pour l'égalité (lorsqu'elle est définie par un type d'objet). Soient X et Y, les résultats de son paramètre réel *Expressions*:

- a) si X ou Y est Null, le résultat est la valeur booléenne prédéfinie Vrai (true) si l'autre est également Null, et la valeur booléenne prédéfinie Faux (false) si un seul est Null; autrement
- b) si la sorte dynamique de Y n'est pas compatible avec la sorte dynamique de X, le résultat est la valeur booléenne prédéfinie Faux (false); autrement
- c) le résultat est obtenu par l'interprétation de *x.is\_equal(y)*, où *x* et *y* représentent respectivement X et Y.

L'opérateur *clone* crée un nouvel élément de données appartenant à la sorte de son paramètre réel et initialise ce nouvel élément de données en appliquant une copie à cet élément, en fonction du paramètre réel d'origine. Après application du *clone*, le nouvel élément de données est égal au paramètre réel. Soit Y, le résultat de son paramètre réel *Expression*, le *clone* de l'opérateur est défini comme suit:

- a) si Y est Null, le résultat est Null; autrement
- b) si la sorte de X est une sorte d'objet, soit X le résultat de l'interprétation de l'opérateur *Make* pour le type de données ayant défini la sorte de Y. Le résultat est obtenu par l'interprétation de *x.copy(y)*, où *x* et *y* représentent respectivement X et Y; autrement
- c) si la sorte de X est une sorte de valeur, soit X un élément arbitraire de la sorte de X. Le résultat est obtenu par l'interprétation de *x.copy(y)*, où *x* et *y* représentent respectivement X et Y.

La méthode *is\_equal* compare **this** au paramètre réel, composante par composante, s'il y en a. En général, pour que le résultat de la méthode *is\_equal* soit la valeur booléenne prédéfinie Vrai (true), ni **this** ni le paramètre réel ne doivent être Null, et la sorte du paramètre réel doit être compatible avec celle de **this**.

Des définitions de type de données peuvent redéfinir *is\_equal*, pour prendre en compte les différences sémantiques de leurs sortes correspondantes. Les constructeurs de type redéfinissent implicitement *is\_equal* comme suit. Soient X et Y, les résultats de son paramètre réel *Expressions*,

- a) si la sorte de X a été construite au moyen d'une liste <literal list>, le résultat est la valeur booléenne prédéfinie Vrai (true) si X et Y ont la même valeur;
- b) si la sorte de X a été construite au moyen d'une définition <structure definition>, le résultat est la valeur booléenne prédéfinie Vrai (true) si, pour chaque composante de X, cette composante est égale à la composante correspondante de Y, comme déterminé par l'interprétation d'une expression *Equality-expression*, avec ces composantes comme les opérands, en omettant les composantes de X et Y qui ont été définies comme facultatives et qui ne sont pas présentes.

La méthode *copy* copie le paramètre réel dans **this**, composante par composante, s'il y en a. Chaque constructeur de type de données ajoute une méthode qui redéfinit la méthode de *copie*. En général, ni **this** ni le paramètre réel ne doivent être Null et la sorte du paramètre réel doit être compatible avec celle de **this**. Chaque redéfinition de *copy* doit satisfaire à la condition postérieure selon laquelle, après application de la méthode de *copie*, **this** est égal au paramètre réel.

Les définitions de type de données peuvent redéfinir la méthode de *copie* pour tenir compte des différences sémantiques de leurs sortes correspondantes. Les constructeurs de type redéfinissent automatiquement la méthode de *copie* comme suit. Soient X et Y, les résultats de son paramètre réel *Expressions*,

- a) si la sorte de X a été construite au moyen d'une liste <literal list>, Y est alors copié dans X;
- b) si la sorte de X a été construite au moyen d'une définition <structure definition>, pour chaque composante de X, la composante correspondante de Y est alors copiée dans cette composante de X par l'interprétation de *xc.modify(yc)* – où *xc* représente la composante de X, *modify* est la méthode de modification de champ pour cette composante et *yc* représente la composante correspondante de Y, en omettant les composantes de X et Y qui ont été définies comme facultatives et qui ne sont pas présentes.

NOTE 2 – L'interprétation de la méthode de modification implique une affectation du paramètre réel dans le paramètre formel et, en conséquence, un appel récursif de la méthode de copie (voir 12.3.3).

#### *Modèle*

Si une définition <data type definition> ne contient pas de spécialisation <data type specialization>, il s'agit d'une abréviation pour une définition <data type definition> avec une spécialisation <data type specialization>

**inherits** Any;

### **12.1.6 Pid et sortes de pid**

Chaque interface est (directement ou indirectement) un sous-type de l'interface Pid. Lorsqu'une variable est déclarée comme étant de la sorte Pid, les éléments de données appartenant à une sorte de pid quelconque peuvent être affectés à cette variable.

#### *Grammaire textuelle concrète*

La Pid de type de données peut être qualifié par le prédéfini **package**.

#### *Sémantique*

La sorte Pid contient un unique élément de données désigné par le littéral Null. Null représente une référence qui n'est associée avec aucun agent.

Une définition *Interface-definition*, représentée par une définition <interface definition> n'ayant pas de spécialisation <interface specialization>, contient uniquement un identificateur *Data-type-identifier* désignant l'interface Pid.

Un élément d'une sorte de pid introduit par une interface implicitement définie par une définition d'agent, est associé avec une référence à l'agent par l'interprétation d'un nœud *Create-request-node* (voir 11.13.2).

Chaque interface ajoute une opération de vérification de compatibilité qui, selon le signal, détermine si:

- a) le signal est défini ou utilisé dans l'interface;
- b) si le signal est contenu dans un accès vers l'intérieur relié à la machine à états de l'agent qui a implicitement défini l'interface;
- c) la vérification de compatibilité est concluante pour une sorte de pid définie par une interface contenue dans sa spécialisation <interface specialization>.

Si cette condition n'est pas remplie, l'exception prédéfinie InvalidReference (voir D.3.16) doit être déclenchée. La vérification de compatibilité est définie de manière similaire pour les variables distantes (voir 10.6) et les procédures distantes (voir 10.5).

NOTE – Une sorte de pid peut être affectée de manière polymorphique (voir 12.3.3).

### **12.1.7 Constructeurs de type de données**

Les constructeurs de type de données spécifient le contenu de la sorte d'un type de données, soit par énumération des éléments qui constituent la sorte, ou par collecte de tous les éléments de données qui peuvent être obtenus par la construction d'un tuple à partir d'éléments de sortes données.

## Grammaire textuelle concrète

```
<data type constructor> ::=
    <literal list>
    | <structure definition>
    | <choice definition>
```

### 12.1.7.1 Littéraux

Le constructeur de type de données littérales spécifie le contenu de la sorte d'un type de données par l'énumération des éléments (éventuellement extrêmement nombreux) de la sorte. Le constructeur de type de données littérales définit implicitement les opérations qui permettent la comparaison entre les éléments de la sorte. Les éléments d'une sorte littérale sont appelés littéraux.

## Grammaire abstraite

```
Literal-signature      :: Literal-name
                        Result
Literal-name          = Name
```

## Grammaire textuelle concrète

```
<literal list> ::=
    [<visibility>] literals <literal signature> { , <literal signature> } * <end>

<literal signature> ::=
    <literal name>
    | <name class literal>
    | <named number>

<literal name> ::=
    <literal name>
    | <string name>

<named number> ::=
    <literal name> <equals sign> <Natural simple expression>
```

Dans une signature *Literal-signature*, le résultat *Result* est la sorte introduite par la définition <data type definition> définissant la signature <literal signature>.

Le nom *Literal-name* est unique à l'intérieur de l'unité de portée de définition dans la syntaxe abstraite, même si le nom correspondant <literal name> peut ne pas être unique. Le nom unique *Literal-name* provient:

- du nom <literal name>; plus
- de l'identificateur de sorte de la définition de type de données dans laquelle le nom <literal name> est défini.

NOTE – La chaîne <character string> est une des unités lexicales, la chaîne <bit string> et la chaîne <hex string>.

Chaque résultat de l'expression <Natural simple expression> apparaissant dans un nombre <named number> doit être unique parmi toutes les signatures <literal signature> de la liste <literal list>.

## Sémantique

Une liste <literal list> définit une sorte par l'énumération de tous les éléments de l'ensemble. Chaque élément de la sorte est représenté par une signature *Literal-signature*.

Les littéraux formés à partir de la chaîne <character string> sont utilisés pour les sortes de données prédéfinies Charstring (chaîne de caractères) (voir D.3.4) et Character (caractère) (voir D.3.2). Ils entretiennent également des relations particulières avec les expressions <regular expression> (voir 12.1.9.1). Les littéraux formés à partir de la chaîne <bit string> et de la chaîne <hex string>

sont également utilisés pour la sorte de données des entiers prédéfinie (voir D.3.5). Ces littéraux peuvent également être définis pour d'autres utilisations.

Une liste <literal list> redéfinit les opérations héritées (directement ou indirectement) de Any, comme décrit en 12.1.5.

La signification de <visibility> dans la liste <literal list> est expliquée au 12.1.9.3.

### Modèle

Un nom <literal name> dans une liste <literal list> est une syntaxe dérivée pour un nombre <named number> contenant le nom <literal name> et contenant une expression <Natural simple expression> désignant la valeur naturelle non négative la plus faible possible n'apparaissant dans aucune autre signature <literal signature> de la liste <literal list>. Les noms <literal name> sont remplacés par les nombres <named number> un par un de la gauche vers la droite.

Une liste de littéraux est une syntaxe dérivée pour la définition d'opérateurs qui établissent l'ordre des éléments dans la sorte définie par la liste <literal list>:

- a) opérateurs de comparaison de deux éléments de données selon l'ordre établi;
- b) opérateurs permettant de renvoyer le premier élément de données, le dernier élément de données, l'élément de données suivant et l'élément de données précédent dans l'ordre;
- c) un opérateur donnant la position de chaque élément de données dans l'ordre.

Une définition <data type definition> introduisant une sorte appelée S au moyen d'une liste <literal list> indique un ensemble de signatures *Static-operation-signatures* équivalent aux définitions explicites dans la liste <operator list> suivante:

```
"<" ( this S, this S ) -> Boolean;
">" ( this S, this S ) -> Boolean;
"<=" ( this S, this S ) -> Boolean;
">=" ( this S, this S ) -> Boolean;
first          -> this S;
last           -> this S;
succ ( this S ) -> this S;
pred ( this S ) -> this S;
num ( this S ) -> Natural;
```

où Boolean est la sorte booléenne prédéfinie et Natural est la sorte naturelle prédéfinie.

Les signatures <literal signature> dans une définition <data type definition> sont désignées dans l'ordre croissant des expressions <Natural simple expression>. Par exemple:

```
literals C = 3, A, B;
```

indique  $A < B$  et  $B < C$ .

Les opérateurs de comparaison "<" (">", "<=", ">=") représentent la comparaison normalisée inférieur à (supérieur à, inférieur ou égal à, supérieur ou égale à) entre les expressions <Natural simple expression> de deux littéraux. L'opérateur first renvoie le premier élément de données dans l'ordre (le littéral ayant l'expression <Natural simple expression> la plus faible). L'opérateur last renvoie le dernier élément de données dans l'ordre (le littéral ayant l'expression <Natural simple expression> la plus élevée). L'opérateur pred renvoie l'élément de données précédent, s'il existe, sinon, elle renvoie le dernier élément de données. L'opérateur succ renvoie l'élément de données suivant, s'il existe, sinon, elle renvoie le premier élément de données. L'opérateur num renvoie la valeur naturelle correspondant à l'expression <Natural simple expression> du littéral.

Si la signature <literal signature> est une expression <regular expression>, il s'agit d'une abréviation pour l'énumération d'un ensemble (éventuellement infini) de noms <literal name>, comme décrit en 12.1.9.1.

### 12.1.7.2 Types de données de structure

Le constructeur de types de données de structure spécifie le contenu d'une sorte en formant le produit cartésien d'un ensemble de sortes données. Les éléments d'une sorte de structure sont appelés structures. Le constructeur de types de données de structure définit implicitement les opérations de construction des structures à partir des éléments des sortes de composante, des opérations de projection permettant d'accéder aux éléments des composantes d'une structure, ainsi qu'à partir des opérations de mise à jour des éléments des composantes d'une structure.

#### *Grammaire textuelle concrète*

```
<structure definition> ::=
    [<visibility>] struct [<field list>] <end>

<field list> ::=
    <field> { <end> <field> } *

<field> ::=
    <fields of sort>
    | <fields of sort> optional
    | <fields of sort> <field default initialization>

<fields of sort> ::=
    [<visibility>] <field name> { , <field name> } * <field sort>

<field default initialization> ::=
    default <constant expression>

<field sort> ::=
    <sort>
```

Chaque nom <field name> d'une sorte de structure doit être différent de tout autre nom <field name> de la même définition <structure definition>.

#### *Sémantique*

Une définition <structure definition> définit une sorte de structure dont les éléments sont tous les tuples pouvant être construits à partir d'éléments de données appartenant aux sortes figurant dans la liste <field list>. Un élément d'une sorte de structure comporte autant d'éléments de composante qu'il y a de champs <field> dans la liste <field list>, bien qu'un champ ne soit pas forcément associé à un élément de données, si le champ <field> correspondant a été déclaré avec le mot clé **optional** ou s'il n'a pas encore été initialisé.

Une définition <structure definition> redéfinit les opérations héritées (directement ou indirectement) de Any, comme décrit en 12.1.5.

La signification de <visibility> dans les champs <fields of sort> et dans la définition <structure definition> est expliquée au 12.1.9.3.

#### *Modèle*

Une liste <field list> contenant un champ <field> avec une liste de noms <field name> dans un champ <fields of sort> est une syntaxe concrète dérivée où ce champ <field> est remplacé par une liste de champs <field> séparés par l'extrémité <end>, de sorte que chaque champ <field> de cette liste est créé à partir d'une copie du champ <field> d'origine et de la substitution d'un nom <field name> pour la liste de noms <field name>, à la place de chaque nom <field name> dans la liste.

Une définition de structure est une syntaxe dérivée pour la définition:

- a) d'un opérateur, Make, pour la création de structures;
- b) de méthodes permettant de modifier des structures et d'accéder à des éléments de données de composantes de structures;



- c) de méthodes d'essai permettant de déceler la présence d'éléments de données de composante facultatifs dans les structures.

Les <arguments> pour l'opérateur Make contiennent une liste de sortes <field sort> figurant dans la liste de champs dans l'ordre dans lequel elles apparaissent. La sorte <sort> de résultat pour l'opérateur Make est l'identificateur de sorte de la sorte de structure. L'opérateur Make crée une nouvelle structure et associe chaque champ au résultat du paramètre formel correspondant. Si le paramètre réel a été omis dans l'application de l'opérateur Make, le champ correspondant n'obtient aucune valeur; c'est-à-dire qu'il devient "non défini".

Une définition <structure definition> introduisant une sorte appelée S indique un ensemble de signatures *Dynamic-operation-signatures* équivalent aux définitions explicites de la liste <method list> suivante, pour chaque champ <field> de sa liste <field list>:

```
virtual field-modify-operation-name ( <field sort> ) -> S;  
virtual field-extract-operation-name -> <field sort>;  
field-presence-operation-name -> Boolean;
```

où Boolean est la sorte booléenne prédéfinie et la sorte <field sort> est la sorte du champ.

Le nom de la méthode indiquée pour modifier un champ, *field-modify-operation-name*, est le nom de champ concaténé avec "Modify". La méthode indiquée pour modifier un champ associe le champ avec le résultat de son argument *Expression*. Lorsque la sorte <field sort> est une sorte <anchored sort>, cette association est effectuée uniquement si la sorte dynamique de l'argument *Expression* est compatible avec la sorte <field sort> de ce champ. Autrement, l'exception prédéfinie UndefinedField (voir D.3.16) est déclenchée.

Le nom de la méthode indiquée pour accéder à un champ, *field-extract-operation-name*, est le nom de champ concaténé avec "Extract". La méthode permettant d'accéder à un champ renvoie l'élément de données associé à ce champ. Si, lors de l'interprétation, un champ d'une structure est "non défini", l'application de cette méthode pour accéder à ce champ entraîne le déclenchement de l'exception prédéfinie UndefinedField.

Le nom de la méthode indiquée pour déceler la présence d'un élément de données de champ, *field-presence-operation-name*, est le nom de champ concaténé avec "Present". A l'aide de cette méthode, on obtient la valeur booléenne prédéfinie Faux (false) si ce champ est "non défini", sinon on obtient la valeur booléenne prédéfinie Vrai (true). Une méthode permettant de déceler la présence d'un élément de données de champ est définie uniquement si ce champ <field> contient le mot clé **optional**.

Si un champ <field> est défini avec une initialisation <field default initialization>, il s'agit d'une syntaxe dérivée pour la définition de ce champ <field> comme champ facultatif. Lorsqu'une structure de cette sorte est créée et qu'aucun argument réel n'est fourni pour le champ par défaut, une modification immédiate du champ par l'expression associée <constant expression> après la création est ajoutée.

### 12.1.7.3 Type de données de choix

Un constructeur de type de données de choix est une abréviation permettant de définir un type de structure dont toutes les composantes sont facultatives, elle permet également de s'assurer que chaque élément de données de structure comportera toujours un seul et unique élément de données de composante. Le type de données de choix simule ainsi une sorte qui correspond à la somme disjointe des éléments des sortes de composante.

*Grammaire textuelle concrète*

```
<choice definition> ::=  
    [<visibility>] choice [<choice list>] <end>
```

```

<choice list> ::=
    <choice of sort> { <end> <choice of sort> }*

<choice of sort> ::=
    [<visibility>] <field name> <field sort>

```

Chaque nom <field name> d'une sorte de choix doit être différent de tout autre nom <field name> de la même définition <choice definition>.

### Sémantique

Une définition <choice definition> redéfinit les opérations héritées (directement ou indirectement) de Any, comme décrit en 12.1.5.

La signification de <visibility> dans la sorte <choice of sort> et dans la définition <choice definition> est expliquée au 12.1.9.3.

### Modèle

Une définition de type de données contenant une définition <choice definition> est une syntaxe dérivée et est transformée lors des étapes suivantes: soit le nom *Choice-name* correspondant au nom <data type name> de la définition de type de données d'origine:

- une définition <value data type definition> ayant un nom anonyme, *anon* et une définition <structure definition> en tant que constructeur de type, sont ajoutées. Dans la définition <value data type definition>, pour chaque <choice of sort>, un champ <field> est construit et contient les champs <fields of sort> équivalents avec le mot clé **optional**;
- une définition <value data type definition> ayant un nom anonyme, *anonPresent*, est ajoutée avec la liste <literal list> contenant tous les noms <field name> de la liste <choice list> comme noms <literal name>. L'ordre des littéraux est le même que celui dans lequel les noms <field name> ont été spécifiés;
- une définition <data type definition> ayant un nom anonyme, *anonChoice*, est construite comme suit:

```

object type anonChoice
struct
    protected Present anonPresent;
    protected Choice anon;
endobject type anonChoice;

```

si la définition de type de données d'origine avait défini une sorte d'objet. Sinon, la définition <data type definition> est une définition <value data type definition>;

- une définition <data type definition> est construite comme suit:

```

object type Choice-name inherits anonChoice (anonMake = Make,
    anonPresentModify = PresentModify,
    anonPresentExtract = PresentExtract,
    anonChoiceModify = ChoiceModify,
    anonChoiceExtract = ChoiceExtract )
adding
    operations
endobject type Choice-name;

```

si la définition de type de données d'origine avait défini un type d'objet, et où *operations* est <operations>, comme défini ci-dessous. Sinon, la définition <data type definition> est une définition <value data type definition>. Le renommage <renaming> renomme les opérations mentionnées, héritées de *anonChoice* avec des noms anonymes;

- e) pour chaque choix <choice of sort>, une signature <operation signature> est ajoutée à la liste <operator list> des *opérations* représentant un opérateur indiqué pour la création d'éléments de données:

*field-name* ( *field-sort* ) -> *Choice-name*;

où le nom *field-name* est le nom <field name> et où la sorte *field-sort* est la sorte <field sort> dans le choix <choice of sort>. L'opérateur indiqué pour créer des éléments de données crée une nouvelle structure en appelant *anonMake*, en initialisant le champ Choix (Choice) avec une structure nouvellement créée, initialisée avec le nom <field name>, et en affectant le littéral correspondant au nom <field name> au champ Présent (Present);

- f) pour chaque choix <choice of sort>, des signatures <operation signature> sont ajoutées à la liste <method list> des *opérations* représentant les méthodes indiquées pour modifier et accéder aux éléments de données:

**virtual** *field-modify* ( *field-sort* ) -> *Choice-name*;

**virtual** *field-extract* -> *field-sort*;

*field-present* -> Boolean;

où *field-extract* est le nom de la méthode indiquée par *anon* pour accéder au champ correspondant, *field-modify* est le nom de la méthode indiquée par *anon* pour modifier ce champ où *field-present* est le nom de la méthode indiquée par *anon* pour déceler la présence d'un élément de données de champ. Les appels à *field-extract* et à *field-present* sont retransmis au champ Choix. Les appels à *field-modify* attribuent une structure nouvellement créée, initialisée avec le nom <field name>, au champ Choix et attribuent le littéral correspondant au nom <field name> au champ Présent;

- g) une signature <operation signature> est ajoutée à la liste <operator list> des *opérations* représentant un opérateur indiqué pour obtenir la sorte d'élément de données actuellement présent dans le champ Choix:

PresentExtract ( *Choice-name* ) -> *anonPresent*;

PresentExtract renvoie la valeur associée au champ Présent.

### 12.1.8 Comportement des opérations

Une définition <data type definition> permet l'adjonction d'opérations dans un type de données. Le comportement des opérations peut être défini de manière semblable aux appels de procédure de retour de valeur. Toutefois, les opérations d'un type de données ne doivent pas avoir accès ni modifier l'état global des files d'attente d'entrée des agents dans lesquels elles sont appelées. Elles ne contiennent donc qu'une seule transition.

#### Grammaire textuelle concrète

<operation definitions> ::=

```
{
  <operation definition>
  |
  <textual operation reference>
  |
  <external operation definition> }*
```

<operation definition> ::=

```
{<package use clause>}*
<operation heading> <end>
  <entity in operation>*
  <operation body>
  { endoperator | endmethod } [ [<qualifier>] <operation name> ] <end>
  |
  {<package use clause>}*
  <operation heading>
  [ <end> <entity in operation>+ ] <left curly bracket>
  <statement list>
  <right curly bracket>
```

```

<operation heading> ::=
    { operator | method } <operation preamble> [<qualifier>] <operation name>
    [<formal operation parameters>]
    [<operation result>] [<raises>]

<operation identifier> ::=
    [<qualifier>] <operation name>

<formal operation parameters> ::=
    ( <operation parameters> {, <operation parameters> } * )

<operation parameters> ::=
    [<argument virtuality>] <parameter kind> <variable name> {, <variable name>} * <sort>

<entity in operation> ::=
    <data definition>
    | <variable definition>
    | <exception definition>
    | <select definition>
    | <macro definition>

<operation body> ::=
    [<on exception>] <start> { <free action> | <exception handler> } *

<operation result> ::=
    <result sign> [<variable name>] <sort>

<textual operation reference> ::=
    <operation heading> referenced <end>

<external operation definition> ::=
    <operation heading> external <end>

```

Les mots clés **operator** ou **method** utilisés dans l'en-tête <operation heading> doivent être mis en correspondance avec la définition <operation definition> associée par **endoperator** et **endmethod**, respectivement.

Il n'existe pas de syntaxe graphique correspondante pour la définition <external operation definition>.

Les paramètres <formal operation parameters> et le résultat <operation result> dans la référence <textual operation reference> et dans la définition <external operation definition> peuvent être omis s'il n'existe pas d'autre référence <textual operation reference> ou d'autre définition <external operation definition>, respectivement, à l'intérieur de la même sorte ayant le même nom. Dans ce cas, les paramètres <formal operation parameters> et le résultat <operation result> sont obtenus à partir de la signature <operation signature>.

Pour chaque signature <operation signature>, au plus une définition <operation definition> correspondante peut être donnée.

Le corps <operation body>, ainsi que les instructions <statement> dans la définition <operation definition>, ne doivent contenir ni une expression <imperative expression> ni un identificateur <identifieur> défini à l'extérieur de la définition englobante <operation definition> ou du diagramme <operation diagram> respectivement, excepté pour les identificateurs <synonym identifier>, les <operation identifier>, les <literal identifier> et les sortes <sort>.

Si une exception peut être déclenchée dans une opération dans laquelle aucun gestionnaire d'exception n'est actif avec la clause d'exception correspondante (c'est-à-dire que l'exception n'est pas gérée), <raises> doit mentionner cette exception. Une exception est considérée comme n'étant pas traitée dans une opération s'il n'y a pas de débit de contrôle potentiel dans l'opération produisant l'exception et qu'aucun gestionnaire d'exception activé dans ce débit de contrôle ne gère l'exception.

On considère que la liste de noms <variable name> constitue un lien plus étroit que la liste des paramètres <operation parameters> à l'intérieur de <formal operation parameters>.

## Grammaire graphique concrète

```
<operation diagram> ::=
    <frame symbol> contains {
        <operation heading>
        { <operation text area>* <operation graph area> } set }
    [ is associated with <package use area> ]

<operation graph area> ::=
    [ <on exception association area> ] <procedure start area>
    { <in connector area> | <exception handler area> }*

<operation text area> ::=
    <text symbol> contains
    {
        <data definition>
        | <variable definition>
        | <macro definition>
        | <exception definition>
        | <select definition> }*
```

La zone <package use area> doit être placée en tête du symbole <frame symbol>.

Le départ <start> dans le diagramme <operation diagram> ne doit pas contenir <virtuality>.

Dans la mesure où il n'y a pas de grammaire graphique pour les définitions de sortes, un diagramme <operation diagram> peut être utilisé uniquement par le biais d'une référence <textual operation reference>.

Pour chaque diagramme <operation diagram>, il doit y avoir une signature <operation signature> dans la même unité de portée ayant le même nom <operation name>, ayant par position les mêmes sortes <sort> d'argument que spécifiées dans les paramètres <formal operation parameters> (s'ils existent) et ayant la même sorte <sort> de résultat que spécifiée dans le résultat <operation result>. S'il n'existe qu'une seule opération avec l'identificateur <operation identifieur> ou le nom <operation name> donné, les sortes <sort> d'argument dans les paramètres <formal operation parameters> peuvent être omises. De même, la sorte <sort> peut être omise dans le résultat <operation result>.

Pour chaque signature <operation signature>, au plus un diagramme <operation diagram> correspondant peut être donnée.

Le corps <operation body>, ainsi que les instructions <statement> dans la définition <operation definition>, ne doivent contenir ni une expression <imperative expression> ni un identificateur <identifieur> défini à l'extérieur de la définition englobante <operation definition> ou du diagramme <operation diagram> respectivement, excepté pour les identificateurs <synonym identifieur>, les <operation identifieur>, les <literal identifieur> et les sortes <sort>.

### Sémantique

Un opérateur est un constructeur des éléments de la sorte identifiée par le résultat. Il doit toujours renvoyer une valeur ou un objet nouvellement construit. En revanche, une méthode peut renvoyer un objet existant.

Un opérateur ne doit pas modifier des objets qui sont accessibles, en suivant des références à partir des paramètres réels ou en suivant les paramètres réels proprement dits. Un objet est considéré comme modifié dans un opérateur s'il existe un débit de contrôle potentiel à l'intérieur de l'opérateur entraînant cette modification.

Une définition d'opération est une unité de portée définissant ses propres données et variables qui peuvent être manipulées à l'intérieur du corps <operation body>.

Si un en-tête <operation heading> débute par le mot clé **operator**, la définition <operation definition> définit le comportement d'un opérateur. Si l'en-tête <operation heading> débute par le mot clé **method**, la définition <operation definition> définit le comportement d'une méthode.

Les variables introduites dans les paramètres <formal operation parameters> sont des variables locales de l'opérateur ou de la méthodes et peuvent être modifiées à l'intérieur du corps <operation body>.

Une définition <external operation definition> est un opérateur ou une méthode dont le comportement n'est pas inclus dans la description du SDL. De manière conceptuelle, il est supposé qu'un comportement est attribué à une définition <external operation definition> et qu'elle est transformée en une définition <operation definition> faisant partie de la transformation du système générique (voir Annexe F).

### *Modèle*

Pour chaque définition <operation definition> qui n'a pas une signature correspondante <operation signature>, une signature <operation signature> est construite.

Une définition <operation definition> est transformée respectivement en une définition <procedure definition> ou un diagramme <procedure diagram>, portant un nom anonyme, ayant des paramètres <procedure formal parameters> dérivés des paramètres <formal operation parameters> et ayant un résultat <result> dérivé du résultat <operation result>. Le corps <procedure body> est obtenu à partir du corps <operation body> s'il était présent, ou, si la définition <operation definition> contient une liste <statement list>, le résultat de cette transformation est une définition <procedure definition> (voir 9.4). Après application du *Modèle* de la définition <procedure definition>, le départ virtuel inséré par ce *Modèle* est remplacé par un départ sans <virtuality>.

Un diagramme <operation diagram> est transformé en un diagramme <procedure diagram> de manière similaire.

La définition *Procedure-definition* correspondant à la définition résultante <procedure definition>, ou au diagramme <procedure diagram>, est associée à la signature *Operation-signature* représentée par la signature <operation signature>.

Si la définition <operation definition>, ou le diagramme <operation diagram> définit une méthode, alors, lors de la transformation en une définition <procedure definition> ou en un diagramme <procedure diagram>, un paramètre initial avec le genre <parameter kind> **in/out** est inséré dans les paramètres <formal operation parameters>, avec pour sorte <argument sort> la sorte définie par la définition <data type definition> qui constitue l'unité de portée dans laquelle apparaît la définition <operation definition>. Le nom <variable name> dans les paramètres <formal operation parameters> pour ce paramètre inséré est un nom anonyme nouvellement formé.

NOTE – Il n'est pas possible de spécifier une définition <operation definition> pour une signature <literal signature>.

Si une quelconque définition <operation definition> contient du texte informel, l'interprétation d'expressions impliquant l'application de l'opérateur correspondant ou de la méthode correspondante n'est pas définie formellement par le SDL mais peut être déterminée à partir du texte informel par l'interprète. Si du texte informel est spécifié, il n'a pas été donné de spécification formelle complète en SDL.

## **12.1.9 Constructions de définition de données supplémentaires**

Le présent sous-paragraphe introduit des constructions supplémentaires qui peuvent être utilisées pour les données.

### **12.1.9.1 Classe de nom**

Une classe de nom est une notation abrégée permettant d'écrire un ensemble (éventuellement infini) de noms de littéral ou d'opérateur défini par une expression régulière.

Un littéral `<name class literal>` est une autre façon de spécifier un nom `<literal name>`. Une opération `<name class operation>` est une autre façon de spécifier un nom `<operation name>` d'une opération d'annulation.

### *Grammaire textuelle concrète*

```

<name class literal> ::=
    nameclass <regular expression>

<name class operation> ::=
    <operation name> in nameclass <regular expression>

<regular expression> ::=
    <partial regular expression> { [or ] <partial regular expression> }*

<partial regular expression> ::=
    <regular element> [ <Natural literal name> | <plus sign> | <asterisk> ]

<regular element> ::=
    ( <regular expression> )
    | <character string>
    | <regular interval>

<regular interval> ::=
    <character string> <colon> <character string>

```

Les noms formés par l'expression `<regular expression>` doivent respecter les règles lexicales relatives aux noms ou la chaîne `<character string>`, la chaîne `<hex string>` ou la chaîne `<bit string>`. (voir 6.1).

Les chaînes `<character string>` dans un intervalle `<regular interval>` doivent avoir une longueur de un, sans compter les `<apostrophe>` de tête et de fin.

Une opération `<name class operation>` peut être utilisée uniquement dans une signature `<operation signature>`. Une signature `<operation signature>` contenant une opération `<name class operation>` doit apparaître uniquement dans une liste `<operator list>` et ne doit pas contenir d'`<arguments>`.

Lorsqu'un nom contenu dans l'ensemble de noms équivalent d'une opération `<name class operation>` apparaît comme le nom `<operation name>` dans une applicaiton `<operation application>`, il ne doit pas comporter de paramètres `<actual parameters>`.

L'ensemble de noms équivalent d'une classe de nom est défini comme l'ensemble de noms qui respecte la syntaxe spécifiée par l'expression `<regular expression>`. Les ensembles de noms équivalents pour les expressions `<regular expression>` contenues dans une définition `<data type definition>` ne doivent pas se chevaucher.

### *Modèle*

Le littéral `<name class literal>` est l'équivalent de cet ensemble de noms dans la syntaxe abstraite. Lorsqu'une opération `<name class operation>` est utilisée dans une signature `<operation signature>`, un ensemble de signatures `<operation signature>` est créé en substituant chaque nom dans l'ensemble de noms équivalent pour l'opération `<name class operation>` dans la signature `<operation signature>`.

Une expression `<regular expression>` qui est une liste d'expressions `<partial regular expression>` sans un **or**, spécifie que les noms peuvent être formés à partir des caractères définis par la première expression `<partial regular expression>`, suivis des caractères définis par la seconde expression `<partial regular expression>`.

Lorsqu'un **or** est spécifié entre deux expressions `<partial regular expression>`, les noms sont alors formés à partir de la première ou de la deuxième de ces expressions `<partial regular expression>`. **or** constitue un lien plus étroit qu'une simple mise en séquence.

Si un élément <regular element> est suivi d'un nom <Natural literal name>, l'expression <partial regular expression> équivaut à l'élément <regular element> répété autant de fois que cela est spécifié par le nom <Natural literal name>.

Si un élément <regular element> est suivi d'un '\*', l'expression <partial regular expression> équivaut à l'élément <regular element> répété zéro fois ou plus.

Si un élément <regular element> est suivi d'un signe <plus sign>, l'expression <partial regular expression> équivaut à l'élément <regular element> répété une ou plusieurs fois.

Un élément <regular element> qui est une expression <regular expression> entre crochets définit les séquences de caractères décrites dans l'expression <regular expression>.

Un élément <regular element> qui est une chaîne <character string> définit la séquence de caractères donnée dans la chaîne de caractères (sans guillemets).

Un élément <regular element> qui est un intervalle <regular interval> définit tous les caractères spécifiés par l'intervalle <regular interval> comme séquences de caractères possibles. Les caractères définis par l'intervalle <regular interval> sont tous les caractères supérieurs ou égaux au premier caractère et inférieurs ou égaux au deuxième caractère selon la définition de la sorte Character (voir D.2).

Les noms générés par un littéral <name class literal> sont définis dans l'ordre alphabétique d'après la relation d'ordre de la sorte character. Les caractères sont considérés en majuscules ou en minuscules, et un préfixe réel d'un mot est considéré inférieur au mot complet.

NOTE – Des exemples sont présentés dans l'Annexe D.

### 12.1.9.2 Mappage de classe de nom

Un mappage de classe de nom est une abréviation utilisée pour définir un nombre (éventuellement infini) de définitions d'opérations couvrant tous les noms dans une opération <name class operation>. Le mappage de classe de nom permet de définir le comportement des opérateurs et des méthodes définis par une opération <name class operation>. Un mappage de classe de nom se produit lorsqu'un nom <operation name> apparaît dans une opération <name class operation> à l'intérieur d'une signature <operation signature> de la définition englobante <data type definition> est utilisé dans des définitions <operation definitions> ou des diagrammes <operation diagram>.

Un terme orthographique dans un mappage de classe de nom correspond à la chaîne de caractères qui contient l'orthographe du nom. Grâce à ce mécanisme, les opérations Charstring (chaîne de caractères) peuvent être utilisées pour définir les mappages de classe de nom.

#### *Grammaire textuelle concrète*

<spelling term> ::= **spelling** (<operation name>)

Un terme <spelling term> est une syntaxe concrète légale uniquement à l'intérieur d'une définition <operation definition> ou d'un diagramme <operation diagram>, si un mappage de classe de nom s'est produit.

#### *Modèle*

Un mappage de classe de nom est une abréviation pour un ensemble de définitions <operation definition> ou un ensemble de diagrammes <operation diagram>. L'ensemble de définitions <operation definition> est obtenu à partir d'une définition <operation definition> par la substitution de chaque nom dans l'ensemble de noms équivalent de l'opération <name class operation> correspondante pour chaque occurrence du nom <operation name> dans la définition <operation definition>. L'ensemble dérivé de définitions <operation definition> contient



toutes les définitions <operation definition> possibles qui peuvent être générées de cette façon. La même procédure est suivie pour dériver un ensemble de diagrammes <operation diagram>.

Les définitions <operation definition> et les diagrammes <operation diagram> dérivés sont considérés comme légaux même s'il n'est pas permis d'utiliser un nom <string name> en tant que nom <operation name> dans la syntaxe concrète.

Les définitions <operation definition> dérivées sont ajoutées aux définitions <operation definitions> (si elles existent) dans la même définition <data type definition>. Les diagrammes <operation diagram> dérivés sont ajoutés à la liste de diagrammes dans laquelle la définition <operation definition> d'origine est apparue.

Si une définition <operation definition> ou si un diagramme <operation diagram> contient un ou plusieurs termes <spelling term>, chaque terme <spelling term> est remplacé par un littéral Charstring (voir D.3.4).

Si, lors de la transformation décrite ci-dessus, le nom <operation name> dans le terme <spelling term> a été remplacé par un nom <operation name>, le terme <spelling term> est une abréviation pour un littéral Charstring obtenu à partir du nom <operation name>. La chaîne Charstring contient l'orthographe du nom <operation name>.

Si, lors de la transformation décrite ci-dessus, le nom <operation name> dans le terme <spelling term> a été remplacé par un nom <string name>, le terme <spelling term> est une abréviation pour une chaîne Charstring obtenue à partir du nom <string name>. La chaîne Charstring contient l'orthographe du nom <string name>.

### 12.1.9.3 Visibilité réduite

#### *Grammaire textuelle concrète*

<visibility> ::=

**public | protected | private**

La visibilité <visibility> ne doit pas précéder une liste <literal list>, une définition <structure definition> ou une définition <choice definition> dans une définition <data type definition> contenant une spécialisation <data type specialization>. La visibilité <visibility> ne doit pas être utilisée dans une signature <operation signature> qui redéfinit une signature d'opération héritée.

#### *Sémantique*

<visibility> contrôle la visibilité d'un nom de littéral ou d'un nom d'opération.

Lorsqu'une liste <literal list> est précédée de <visibility>, celle-ci s'applique à toutes les signatures <literal signature>. Lorsqu'une définition <structure definition> ou <choice definition> est précédée d'une visibilité <visibility>, celle-ci s'applique à toutes les signatures <operation signatures> indiquées.

Lorsqu'un champ <fields of sort> ou un choix <choice of sort> est précédé d'une visibilité <visibility>, celle-ci s'applique à toutes les signatures <operation signatures> indiquées.

Si une signature <literal signature> ou <operation signature> contient le mot clé **private** dans <visibility>, le nom *Operation-name* obtenu à partir de cette signature <operation signature> est alors visible uniquement à l'intérieur de la portée de la définition <data type definition> qui contient la signature <operation signature>. Lorsqu'une définition <data type definition> contenant une telle signature <operation signature> est spécialisée, le nom <operation name> dans la signature <operation signature> est implicitement renommé par un nom anonyme. A chaque fois qu'apparaît ce nom <operation name> à l'intérieur des définitions <operation definitions> ou des diagrammes <operation diagram> correspondant à cette signature <operation signature>, il est renommé avec le

même nom anonyme lorsque la signature <operation signature> et la définition d'opération correspondante sont héritées par spécialisation.

NOTE 1 – Par conséquent, on ne peut utiliser l'opérateur ou la méthode défini(e) par cette signature <operation signature> que dans des applications d'opération à l'intérieur de la définition de type de données qui a défini cette signature <operation signature> à l'origine, mais il ou elle ne peut pas être utilisé dans un autre sous-type.

Si une signature <literal signature> ou <operation signature> contient le mot clé **protected** dans <visibility>, le nom *Operation-name* obtenu à partir de cette signature <operation signature> est visible uniquement à l'intérieur de la portée de la définition <data type definition> qui contient cette signature <operation signature>.

NOTE 2 – Parce que les opérateurs et les méthodes sont copiés dans le corps du sous-type, il est possible d'accéder à l'opérateur ou la méthode défini par cette signature <operation signature> à l'intérieur de la portée de toute définition <data type definition> qui est un sous-type de la définition <data type definition> qui a défini cette signature <operation signature> à l'origine

NOTE 3 – Si une signature <literal signature> ou <operation signature> ne contient pas <visibility>, le nom *Operation-name* obtenu à partir de cette signature <operation signature> est visible partout où le nom <sort name> défini dans la définition englobante <data type definition> est visible.

### Modèle

Si une signature <literal signature> ou <operation signature> contient le mot clé **public** dans <visibility>, il s'agit d'une syntaxe dérivée pour une signature ne comportant pas de protection.

#### 12.1.9.4 Syntypes

Un syntype spécifie un sous-ensemble des éléments d'une sorte. Un syntype utilisé comme une sorte a la même sémantique que la sorte référencée par le syntype, sauf en ce qui concerne les vérifications visant à montrer que les éléments de données appartiennent au sous-ensemble spécifié des éléments de la sorte.

#### Grammaire abstraite

<i>Syntype-identifiant</i>	=	<i>Identifiant</i>
<i>Syntype-définition</i>	::	<i>Syntype-nom</i> <i>Parent-sort-identifiant</i> <i>Range-condition</i>
<i>Syntype-nom</i>	=	<i>Nom</i>
<i>Parent-sort-identifiant</i>	=	<i>Sort-identifiant</i>

#### Grammaire textuelle concrète

```

<syntype> ::=
    <syntype identifiant>

<syntype définition> ::=
    {<package use clause>}*
    syntype <syntype nom> <equals sign> <parent sort identifiant>
        [<default initialization>] [<constraint> <end>]
        [<syntype closing> <end> ]
    |
    {<package use clause>}*
    |
    <type preamble> <data type heading> [<data type specialization>]
    {
        <constraint> <end>
        |
        <end> <data type définition body> <constraint> <end> <data type closing> <end>
        |
        <left curly bracket> <data type définition body> <constraint> <end>
        <right curly bracket> }

<syntype closing> ::=
    endsyntype [<syntype nom>]

<parent sort identifiant> ::=
    <sort>
  
```

Un <syntype> est une alternative pour une <sort>.

Une définition <syntype definition> avec les mots clés **value type** ou **object type** est la syntaxe dérivée définie ci-après.

Une définition <syntype definition> avec le mot clé **syntype** dans la syntaxe concrète correspond à une définition *Syntype-definition* dans la syntaxe abstraite.

Lorsqu'un identificateur <syntype identifieur> est utilisé comme une sorte <sort> dans les <arguments> lors de la définition d'une opération, la sorte pour les arguments correspondants *Formal-argument* est l'identificateur *Parent-sort-identifieur* du syntype.

Lorsqu'un identificateur <syntype identifieur> est utilisé comme résultat d'une opération, la sorte du résultat Result est l'identificateur *Parent-sort-identifieur* du syntype.

Lorsqu'un identificateur <syntype identifieur> est utilisé comme qualificateur pour un nom, le qualificateur *Qualifier* est l'identificateur *Parent-sort-identifieur* du syntype.

Si le mot clé **syntype** est utilisé et que la contrainte <constraint> est omise, les identificateurs <syntype identifieur> pour le syntype sont représentés dans la grammaire abstraite comme les identificateurs *Parent-sort-identifieur*.

### Sémantique

Une définition de syntype définit un syntype qui se réfère à un identificateur de sorte et à une contrainte. Spécifier un identificateur de syntype revient à spécifier un identificateur de la sorte parente du syntype, sauf en ce qui concerne les cas suivants:

- a) affectation à une variable déclarée avec un syntype (voir 12.3.3);
- b) sortie d'un signal si une des sortes spécifiées pour le signal est un syntype (voir 10.3 et 11.13.4);
- c) appel d'une procédure où une des sortes spécifiées pour les variables du paramètre in de la procédure est un syntype (voir 9.4 et 11.13.3);
- d) création d'un agent lorsqu'une des sortes spécifiées pour les paramètres d'agent est un syntype (voir 9.3 et 11.13.2);
- e) entrée d'un signal et une des variables qui est associée avec l'entrée, a une sorte qui est un syntype (voir 11.2.2);
- f) appel d'une application d'opération qui comporte un syntype défini comme une sorte d'argument ou une sorte de résultat (voir 12.2.7);
- g) clause d'initialisation ou de réinitialisation ou expression active sur un temporisateur, une des sortes dans la définition de temporisateur étant un syntype (voir 11.15 et 12.3.4.4);
- h) définition de variable distante ou de procédure distante si l'une des sortes pour la dérivation de signaux implicites est un syntype (voir 10.5 et 10.6);
- i) paramètre de contexte formel de procédure avec un paramètre in/out ou out dans la signature <procedure signature> correspondant à un paramètre de contexte réel où le paramètre formel correspondant ou le paramètre in/out ou out dans la signature <procedure signature> est un syntype;
- j) <any expression>, pour laquelle le résultat sera dans l'intervalle (voir 12.3.4.5);
- k) déclenchement d'une exception si l'une des sortes spécifiées pour l'exception est un syntype (voir 11.12.2.5).

Lorsqu'un syntype est spécifié en termes d'identificateur <syntype identifieur>, les deux syntypes ne doivent pas être mutuellement définis.

Un syntype a une sorte, qui est la sorte identifiée par l'identificateur de la sorte parente donnée dans la définition de syntype.

Un syntype a une condition *Range-condition* qui pose une contrainte sur la sorte. Si une condition d'intervalle est utilisée, la sorte est contrainte à l'ensemble des éléments de données spécifiés par les constantes de la définition de syntype. Si une contrainte de taille est utilisée, la sorte est contrainte à contenir les éléments de données fournis par la contrainte de taille.

### Modèle

Une définition <syntype definition> avec les mots clés **value type** ou **object type** peut être distinguée d'une définition <data type definition> par l'inclusion d'une condition <constraint>. Une telle définition <syntype definition> est une abréviation d'introduction de définition <data type definition> avec un nom anonyme, suivie par une définition <syntype definition> avec le mot clé **syntype**, fondée sur cette sorte portant un nom anonyme et incluant <constraint>.

## 12.1.9.5 Contrainte

### Grammaire abstraite

```

Range-condition      :: Condition-item-set
Condition-item       = Open-range | Closed-range
Open-range           :: Operation-identifier
                    Constant-expression
Closed-range         :: Open-range
                    Open-range

```

### Grammaire textuelle concrète

```

<constraint> ::=
    constants ( <range condition> )
    |
    <size constraint>

<range condition> ::=
    <range> { , <range> } *

<range> ::=
    <closed range>
    |
    <open range>

<closed range> ::=
    <constant> <colon> <constant>

<open range> ::=
    <constant>
    |
    {
    |   <equals sign>
    |   <not equals sign>
    |   <less than sign>
    |   <greater than sign>
    |   <less than or equals sign>
    |   <greater than or equals sign> } <constant>

<size constraint> ::=
    size ( <range condition> )

<constant> ::=
    <constant expression>

```

le symbole "<" ne doit être utilisé que dans la syntaxe concrète de la condition <range condition> si ce symbole a été défini avec une signature <operation signature>:

```
"<" ( P, P ) -> <<package Predefined>>Boolean;
```

où P est la sorte du syntype, et de même pour les symboles ("<=", ">", ">=", respectivement). Ces symboles représentent un identificateur *Operation-identifier*.

Un intervalle <closed range> doit seulement être utilisé si le symbole "<=" est défini avec une signature <operation signature>:

"<="( P, P ) -> <<package Predefined>>Boolean;

où P est la sorte du syntype.

Une expression <constant expression> dans une condition <range condition> doit avoir la même sorte que la sorte du syntype.

Une contrainte <size constraint> ne doit être utilisée que dans la syntaxe concrète de la condition <range condition> si le symbole Length (longueur) a été défini avec une signature <operation signature>:

Length ( P ) -> <<package Predefined>>Natural;

où P est la sorte du syntype.

### Sémantique

Une contrainte définit une vérification d'intervalle. Une vérification d'intervalle est utilisée quand un syntype a une sémantique additionnelle à la sorte du syntype [voir 12.3.1, 12.1.9.4 et les cas où les syntypes ont des sémantiques différentes – voir les sous-paragraphes auxquels on fait référence aux points a) à k) du 12.1.9.4, *Sémantique*]. Une vérification d'intervalle est également utilisée pour déterminer l'interprétation d'une décision (voir 11.13.5).

La vérification d'intervalle est l'application de l'opération formée à partir de la condition d'intervalle ou de la contrainte de taille. Pour les vérifications d'intervalle de syntype, l'application de cette opération doit être équivalente à la valeur booléenne prédéfinie Vrai (true), dans le cas contraire, l'exception prédéfinie OutOfRange est déclenchée (voir D.3.16). La vérification d'intervalle est obtenue comme suit:

- a) chaque intervalle <open range> ou <closed range> dans la condition <range condition> a un intervalle *Open-range* (le booléen prédéfini **or**) ou *Closed-range* (le booléen prédéfini **and**) correspondant dans l'élément *Condition-item*;
- b) un intervalle <open range> de la forme constante <constant> est équivalent à un intervalle <open range> de la forme = constante <constant>.
- c) pour une expression donnée, A:
  - 1) un intervalle <open range> de la forme = <constant>, /= <constant>, < <constant>, <less than or equals sign> <constant>, > <constant> et <greater than or equals sign> <constant>, a des sous-expressions dans la vérification d'intervalle de la forme A = <constant>, A /= <constant>, A < constante <constant>, A <less than or equals sign> <constant>, A > <constant> et A <greater than or equals sign> <constant> respectivement;
  - 2) un intervalle <closed range> de la forme première constante <constant>: deuxième constante <constant> a une sous-expression dans la vérification d'intervalle de la forme première constante <constant> <less than or equals sign> A **and** A <less than or equals sign> deuxième constante <constant> où **and** correspond au booléen prédéfini **and**.
  - 3) une contrainte <size constraint> a une sous-expression dans la vérification d'intervalle de la forme Length(A) = condition <range condition>;
- d) il y a une opération booléenne prédéfinie **or** pour l'opération distribuée sur tous les éléments de données dans l'ensemble *Condition-item-set*. La vérification d'intervalle est l'expression formée à partir de l'union booléenne prédéfinie **or** de tous les sous-termes obtenus à partir de la condition <range condition>.

Si un syntype est spécifié sans une contrainte <constraint>, la vérification d'intervalle est la valeur booléenne prédéfinie Vrai (true).

### 12.1.9.6 Définition de synonyme

Un synonyme donne un nom à une expression constante qui représente un des éléments de données d'une sorte.

#### *Grammaire textuelle concrète*

```
<synonym definition> ::=
    synonym <synonym definition item> { , <synonym definition item> } * <end>
<synonym definition item> ::=
    <internal synonym definition item>
    | <external synonym definition item>
<internal synonym definition item> ::=
    <synonym name> [<sort>] <equals sign> <constant expression>
<external synonym definition item> ::=
    <synonym name> <predefined sort> = external
```

Dans la syntaxe concrète, l'expression <constant expression> désigne une expression *Constant-expression* dans la syntaxe abstraite, conformément à la définition donnée en 12.2.1.

Si une sorte <sort> est spécifiée, le résultat de l'expression <constant expression> a une sorte statique de la sorte <sort>. Il doit être possible pour l'expression <constant expression> d'avoir cette sorte.

Si la sorte de l'expression <constant expression> ne peut être déterminée de manière unique, il faut alors spécifier une sorte dans la définition <synonym definition>.

L'expression <constant expression> ne doit pas se référer directement ou indirectement (par l'intermédiaire d'un autre synonyme) au synonyme décrit dans la définition <synonym definition>.

Un élément <external synonym definition item> définit un synonyme <synonym> dont le résultat n'est pas défini dans une spécification. De manière conceptuelle, il est supposé qu'un résultat est attribué à un élément <external synonym definition item> et que celui-ci est transformé en un élément <internal synonym definition item> faisant partie de la transformation du système générique (voir Annexe F).

#### *Sémantique*

Le résultat que représente le synonyme est déterminé par le contexte dans lequel la définition de synonyme apparaît.

Si la sorte de l'expression constante ne peut être déterminée uniquement dans le contexte du synonyme, la sorte est alors indiquée par la sorte <sort>.

Un synonyme a un résultat, qui est le résultat de l'expression constante dans la définition de synonyme.

Un synonyme a une sorte, qui est la sorte de l'expression constante dans la définition de synonyme.

## 12.2 Utilisation passive des données

La façon dont les sortes, les littéraux, les opérateurs, les méthodes et les synonymes sont interprétés dans les expressions est définie dans les sous-paragraphes suivants.

### 12.2.1 Expressions

#### *Grammaire abstraite*

```
Expression = Constant-expression
            | Active-expression
Constant-expression = Literal
                    | Conditional-expression
                    | Equality-expression
                    | Operation-application
```

<i>Active-expression</i>		<i>Range-check-expression</i>
	=	<i>Variable-access</i>
		<i>Conditional-expression</i>
		<i>Operation-application</i>
		<i>Equality-expression</i>
		<i>Imperative-expression</i>
		<i>Range-check-expression</i>
		<i>Value-returning-call-node</i>

### Grammaire textuelle concrète

Pour simplifier la description, aucune distinction n'est établie entre la syntaxe concrète de l'expression *Constant-expression* et de l'expression *Active-expression*.

<expression> ::=		<operand>
		<create expression>
		<value returning procedure call>
<operand> ::=		<operand0>
		<operand> <implies sign> <operand0>
<operand0> ::=		<operand1>
		<operand0> { <b>or</b>   <b>xor</b> } <operand1>
<operand1> ::=		<operand2>
		<operand1> <b>and</b> <operand2>
<operand2> ::=		<operand3>
		<operand2> { <greater than sign>
		<greater than or equals sign>
		<less than sign>
		<less than or equals sign>
		<b>in</b> } <operand3>
		<range check expression>
		<equality expression>
<operand3> ::=		<operand4>
		<operand3> { <plus sign>   <hyphen>   <concatenation sign> } <operand4>
<operand4> ::=		<operand5>
		<operand4> { <asterisk>   <solidus>   <b>mod</b>   <b>rem</b> } <operand5>
<operand5> ::=		[ <hyphen>   <b>not</b> ] <primary>
<primary> ::=		<operation application>
		<literal>
		( <expression> )
		<conditional expression>
		<spelling term>
		<extended primary>
		<active primary>
		<synonym>
<active primary> ::=		<variable access>
		<imperative expression>
<expression list> ::=		<expression> { , <expression> }*





est une syntaxe dérivée pour:

<quotation mark> <infix operation name> <quotation mark> ( <expression>, <expression> )

où <quotation mark> <infix operation name> <quotation mark> représente un nom *Operation-name*.

De même,

<monadic operation name> <expression>

est une syntaxe dérivée pour:

<quotation mark> <monadic operation name> <quotation mark> ( <expression> )

où <quotation mark> <monadic operation name> <quotation mark> représente un nom *Operation-name*.

### 12.2.2 Littéral

*Grammaire abstraite*

*Literal* :: *Literal-identifïer*  
*Literal-identifïer* = *Identifïer*

L'identificateur *Literal-identifïer* désigne une signature *Literal-signature*.

*Grammaire textuelle concrète*

<literal> ::= <literal identifïer>

<literal identifïer> ::= [<qualifïer>] <literal name>

Chaque fois qu'un identificateur <literal identifïer>, se trouve spécifié, on obtient l'unique nom *Literal-name* dans l'identificateur *Literal-identifïer* de la même façon, avec la sorte de résultat obtenue à partir du contexte. Un identificateur *Literal-identifïer* est obtenu à partir du contexte (voir 6.3) de manière à ce que si l'identificateur <literal identifïer> est surchargé (c'est-à-dire que le même nom est utilisé pour plusieurs littéraux ou opérateurs), le nom *Literal-name* identifie alors un littéral visible avec le même nom et la sorte de résultat cohérente avec le littéral. Deux littéraux avec le même nom <name> mais avec des sortes de résultat différentes, ont des noms *Literal-names* différents.

Il doit être possible de lier chaque identificateur <literal identifïer> non qualifié à exactement un identificateur *Literal-identifïer* défini qui satisfait les conditions dans la construction dans laquelle l'identificateur <literal identifïer> est utilisé.

Chaque fois qu'un qualificateur <qualifïer> d'un identificateur <literal identifïer> contient un élément <path item> avec le mot clé **type**, le nom <sort name> après ce mot clé ne fait pas partie du qualificateur *Qualifïer* de l'identificateur *Literal-identifïer* mais est utilisé pour obtenir le nom *Name* unique de l'identificateur *Identifïer*. Dans ce cas, le qualificateur *Qualifïer* est formé à partir de la liste des éléments <path item> précédant le mot clé **type**.

*Sémantique*

Un *Littéral* renvoie l'élément de données unique correspondant à sa signature *Literal-signature*.

La sorte du littéral <literal> est le résultat *Result* dans sa signature *Literal-signature*.

### 12.2.3 Synonyme

*Grammaire textuelle concrète*

<synonym> ::= <synonym identifïer>

## Sémantique

Un synonyme est une notation abrégée permettant de désigner une expression définie ailleurs.

### Modèle

Un synonyme `<synonym>` représente l'expression `<constant expression>` décrite par la définition `<synonym definition>` identifiée par l'identificateur `<synonym identifieur>`. Un identificateur `<identifieur>` utilisé dans l'expression `<constant expression>` représente un identificateur *Identifieur* dans la syntaxe abstraite, selon le contexte de la définition `<synonym definition>`.

### 12.2.4 Primaire étendu

Un primaire étendu est une notation syntaxique abrégée. Toutefois, à l'exception de sa forme syntaxique spéciale, un primaire étendu n'a pas de propriétés spéciales et désigne une opération et son ou ses paramètres.

#### Grammaire textuelle concrète

```
<extended primary> ::=
    <indexed primary>
    | <field primary>
    | <composite primary>

<indexed primary> ::=
    <primary> ( <actual parameter list> )
    | <primary> <left square bracket> <actual parameter list> <right square bracket>

<field primary> ::=
    <primary> <exclamation mark> <field name>
    | <primary> <full stop> <field name>
    | <field name>

<field name> ::=
    <name>

<composite primary> ::=
    [<qualifier>] <composite begin sign> <actual parameter list> <composite end sign>
```

### Modèle

Un primaire `<indexed primary>` est une syntaxe concrète dérivée pour:

```
<primary> <full stop> Extract ( <actual parameter list> )
```

La syntaxe abstraite est déterminée à partir de cette expression concrète conformément au 12.2.1.

Un primaire `<field primary>` est une syntaxe concrète dérivée pour:

```
<primary> <full stop> field-extract-operation-name
```

où le nom *field-extract-operation-name* est formé à partir de la concaténation du nom de champ et de "Extract" dans cet ordre. La syntaxe abstraite est déterminée à partir de cette expression concrète conformément au 12.2.1. La transformation selon ce modèle est réalisée avant la modification de la signature des méthodes en 12.1.4.

Lorsque le primaire `<field primary>` a la forme `<field name>`, il s'agit d'une syntaxe dérivée pour:

```
this ! <field name>
```

Un primaire `<composite primary>` est une syntaxe concrète dérivée pour:

```
<qualifier> Make ( <actual parameter list> )
```

si n'importe quels paramètres réels étaient présents, ou sinon:

```
<qualifier> Make
```

et où le qualificateur <qualifier> est inséré uniquement s'il était présent dans le primaire <composite primary>. La syntaxe abstraite est déterminée à partir de cette expression concrète conformément au 12.2.1.

### 12.2.5 Expression d'égalité

#### *Grammaire abstraite*

```

Equality-expression      ::   First-operand
                               Second-operand
First-operand           =   Expression
Second-operand         =   Expression
  
```

Une expression *Equality-expression* représente l'égalité des références ou des valeurs de son opérande *First-operand* et son opérande *Second-operand*.

#### *Grammaire textuelle concrète*

```

<equality expression> ::=
    <operand2> { <equals sign> | <not equals sign> } <operand3>
  
```

Une expression <equality expression> est une syntaxe concrète légale uniquement si l'un de ses opérandes a une sorte compatible avec celle de l'autre opérande.

#### *Sémantique*

L'interprétation de l'expression *Equality-expression* est effectuée par l'interprétation de son opérande *First-operand* et de son opérande *Second-operand*.

Si, après l'interprétation, les deux opérandes sont des objets, l'expression *Equality-expression* désigne l'égalité de référence. Elle renvoie la valeur booléenne prédéfinie Vrai (true) si et seulement si les deux opérandes sont Null ou renvoient au même élément de données objet.

Si, après l'interprétation, les deux opérandes sont des pids, l'expression *Equality-expression* désigne une identité d'agent. Elle renvoie la valeur booléenne prédéfinie Vrai (true) si et seulement si les deux opérandes sont Null ou renvoient à la même instance d'agent.

Si, après l'interprétation, l'un des deux opérandes est une valeur, l'expression *Equality-expression* désigne l'égalité de valeurs.

- a) Si la sorte dynamique de l'opérande *First-operand* est compatible avec la sorte dynamique de l'opérande *Second-operand*, l'expression <equality expression> renvoie le résultat de l'application de l'opérateur égal à l'opérande *First-operand* et à l'opérande *Second-operand*, où égal est l'identificateur *Operation-identifïer* représenté par l'identificateur <operation identifïer> dans l'application <operation application>:

```
equal(<operand2>, <operand3>)
```

- b) Autrement, l'expression <equality expression> renvoie le résultat de l'application de l'opérateur égal à l'opérande *Second-operand* et à l'opérande *First-operand*, où égal est l'identificateur *Operation-identifïer* représenté par l'identificateur <operation identifïer> dans l'application <operation application>:

```
equal(<operand3>, <operand2>)
```

La forme syntaxique concrète:

```
<operand2> <not equals sign> <operand3>
```

est la syntaxe concrète dérivée de:

```
not ( <operand2> = <operand3> )
```

où **not** est une opération du type de données booléen prédéfini.

## 12.2.6 Expression conditionnelle

### Grammaire abstraite

*Conditional-expression* ::= *Boolean-expression*  
*Consequence-expression*  
*Alternative-expression*  
*Boolean-expression* = *Expression*  
*Consequence-expression* = *Expression*  
*Alternative-expression* = *Expression*

Une *expression conditionnelle* est une *expression* qui est interprétée comme une *expression de conséquence* ou comme une *expression alternative*.

La sorte de l'expression *Consequence-expression* doit être la même que la sorte de l'expression *Alternative-expression*.

### Grammaire textuelle concrète

```
<conditional expression> ::=  
    if <Boolean expression>  
    then <consequence expression>  
    else <alternative expression>  
    fi  
  
<consequence expression> ::=  
    <expression>  
  
<alternative expression> ::=  
    <expression>
```

La sorte de l'expression <consequence expression> doit être la même que celle de l'expression <alternative expression>.

### Sémantique

Une expression conditionnelle représente une *Expression* qui est interprétée comme une expression *Consequence-expression* ou comme une expression *Alternative-expression*.

Si l'expression *Boolean-expression* est la valeur booléenne prédéfinie Vrai (true), l'expression *Alternative-expression* n'est pas interprétée. Si l'expression *Boolean-expression* est la valeur booléenne prédéfinie Faux (false), l'expression *Consequence-expression* n'est pas interprétée.

Une expression conditionnelle a une sorte, qui est la sorte de l'expression de conséquence (et également celle de l'expression alternative).

Le résultat de l'expression conditionnelle est le résultat de l'interprétation de l'expression *Consequence-expression* ou de l'expression *Alternative-expression*.

La sorte statique d'une expression conditionnelle est la sorte statique de l'expression *Consequence-expression* (qui est également la sorte de l'expression *Alternative-expression*). La sorte dynamique de l'expression conditionnelle est la sorte dynamique du résultat de l'interprétation de l'expression conditionnelle.

## 12.2.7 Application d'opération

### Grammaire abstraite

*Operation-application* ::= *Operation-identifïer*  
*[Expression]\**  
*Operation-identifïer* = *Identifïer*

L'identificateur *Operation-identifïer* désigne une signature *Operation-signature*, soit une signature *Static-operation-signature*, soit une signature *Dynamic-operation-signature*. Chaque *Expression* de la liste d'*Expressions* après l'identificateur *Operation-identifïer* doit avoir une sorte compatible (par

sa position) avec la sorte correspondante de la liste d'arguments *Formal-argument* de la signature *Operation-signature*.

A chaque signature *Operation-signature* est associée une définition *Procedure-definition*, comme décrit en 12.1.8.

Chaque *Expression* correspondant par sa position à un paramètre *Inout-parameter* ou *Out-parameter* dans la définition *Procedure-definition* associée à la signature *Operation-signature*, doit être un identificateur *Variable-identifieur* ayant le même identificateur *Sort-reference-identifieur* que la sorte correspondante (par sa position) de la liste d'arguments *Formal-argument* de la signature *Operation-signature*.

### Grammaire textuelle concrète

<operation application> ::=

    <operator application>  
    |    <method application>

<operator application> ::=

    <operation identifieur> [<actual parameters>]

<method application> ::=

    <primary> <full stop> <operation identifieur> [<actual parameters>]

Chaque fois qu'un identificateur <operation identifieur> est spécifié, le nom unique *Operation-name* dans l'identificateur *Operation-identifieur* est obtenu de la même manière. La liste des sortes d'argument est obtenue à partir des paramètres réels et la sorte de résultat est obtenue à partir du contexte (voir 6.3). Donc, si le nom <operation name> est surchargé (c'est-à-dire que le même nom est utilisé pour plus d'un littéral ou d'une opération), le nom *Operation-name* identifie alors une opération visible avec le même nom et la sorte de résultat cohérente avec l'application d'opération. Deux opérations ayant le même nom <name> mais ayant un ou plusieurs arguments ou sortes de résultat différents, ont des noms *Operation-names*.

Il doit être possible de lier chaque identificateur <operation identifieur> non qualifié à exactement un identificateur *Operation-identifieur* défini qui satisfait les conditions dans la construction dans laquelle l'identificateur <operation identifieur> est utilisé.

Lorsque l'application d'opération présente la forme syntaxique:

    <operation identifieur> [<actual parameters>]

lors de la dérivation de l'identificateur *Operation-identifieur* à partir du contexte, la forme:

**this** <full stop> <operation identifieur> [<actual parameters>]

est également considérée. Le modèle donné en 12.3.1 est appliqué avant de tenter la résolution par le contexte.

Chaque fois qu'un qualificatif <qualifier> d'un identificateur <operation identifieur> contient un élément <path item> avec le mot clé **type**, le nom <sort name> après ce mot clé ne fait pas partie du qualificatif *Qualifier* de l'identificateur *Operation-identifieur* mais est utilisé pour obtenir le nom unique *Name* de l'identificateur *Identifieur*. Dans ce cas, le qualificatif *Qualifier* est formé à partir de la liste des éléments <path item> précédant le mot clé **type**.

Si toutes les <expression> dans la liste entre parenthèses d'<expression> sont des expressions <constant expression>, l'application <operation application> représente une expression *Constant-expression*, comme défini en 12.2.1.

Une application <method application> est une syntaxe concrète légale uniquement si l'identificateur <operation identifieur> représente une méthode.

Une <expression> dans <actual parameters> correspondant à un paramètre *Inout-parameter* ou *Out-parameter* dans la définition *Procedure-définition* associée à la signature *Operation-signature*, ne peut être omises et doit être un accès <variable access> ou un primaire <extended primary>.

Si **this** est utilisé, l'identificateur <operation identifier> doit désigner un opérateur ou une méthode englobante.

NOTE – <actual parameters> peut être omis dans une application <operation application> si tous les paramètres réels ont été omis.

### Sémantique

La résolution par le contexte (voir 6.3) permet de garantir qu'une opération est choisie de manière à ce que les types des arguments réels aient des sortes compatibles par paires avec celles des types des arguments formels.

Une application d'opération avec un identificateur *Operation-identifiant* qui désigne une signature *Static-operation-signature* est interprétée par transfert de l'interprétation dans la définition *Procedure-définition* associée à la signature *Operation-signature* et ce graphe de procédure est interprété (l'explication est présentée en 9.4).

Une application d'opération avec un identificateur *Operation-identifiant* qui désigne une signature *Dynamic-operation-signature*, est interprétée par les étapes suivantes:

- a) les paramètres réels sont interprétés;
- b) si le résultat d'un paramètre réel correspondant à un argument *Virtual-argument* était Null, l'exception prédéfinie *InvalidReference* est déclenchée;
- c) toutes les signatures *Dynamic-operation-signatures* sont regroupées dans un ensemble tel que la signature d'opération formée à partir d'un nom *Operation-name* issu du nom <operation name> dans l'identificateur <operation identifier> et les sortes dynamiques du résultat de l'interprétation des paramètres réels sont compatibles avec la signature candidate *Dynamic-operation-signature*;
- d) la signature unique *Dynamic-operation-signature*, dont la sorte est compatible avec celles de toutes les autres signatures *Dynamic-operation-signatures* dans cet ensemble, est choisie;
- e) l'interprétation est transférée dans la définition *Procedure-définition* associée à la signature *Operation-signature* choisie et ce graphe de procédure est interprété (l'explication est présentée en 9.4).

L'existence d'une telle signature est garantie par l'exigence selon laquelle l'ensemble de signatures *Dynamic-operation-signatures* doit former un treillis (voir 12.1.2).

La liste des *Expressions* de paramètre réel dans une application *Operation-application* est interprétée dans l'ordre donné de gauche à droite avant d'interpréter l'opération proprement dite.

Si une <expression> dans <actual parameters> est omise, le paramètre formel correspondant n'a pas d'élément de données associé; c'est-à-dire qu'il est "non défini".

Si une sorte d'argument de la signature d'opération est un syntype, la vérification d'intervalle définie au 12.1.9.5 est appliquée au résultat de l'*Expression*. Si la vérification d'intervalle est la valeur booléenne prédéfinie Faux (false) lors de l'interprétation, l'exception prédéfinie *OutOfRange* (voir D.3.16) est déclenchée.

L'interprétation de la transition contenant l'application <operation application> se poursuit lorsque l'interprétation de la procédure appelée est finie. Le résultat de l'application d'opération est le résultat obtenu par l'interprétation de la définition de la procédure référencée.

Si la sorte de résultat de la signature d'opération est un syntype, la vérification d'intervalle définie au 12.1.9.5 est appliquée au résultat de l'application d'opération. Si la vérification d'intervalle est la

valeur booléenne prédéfinie Faux (false) lors de l'interprétation, l'exception prédéfinie `OutOfRangeException` (voir D.3.16) est déclenchée.

Une application `<operation application>` a une sorte, qui est la sorte du résultat obtenu par l'interprétation de la procédure associée.

#### Modèle

La forme syntaxique concrète:

```
<expression> <full stop> <operation identifieur> [<actual parameters>]
```

est la syntaxe concrète dérivée de:

```
<operation identifieur> new-actual-parameters
```

où *new-actual-parameters* représente des paramètres `<actual parameters>` contenant uniquement une `<expression>`, si `<actual parameters>` était absent; sinon, *new-actual-parameters* est obtenu en insérant l'`<expression>` avant la première expression facultative dans les paramètres `<actual parameters>`.

### 12.2.8 Expression de vérification d'intervalle

#### Grammaire abstraite

```
Range-check-expression :: Range-condition Expression
```

#### Grammaire textuelle concrète

```
<range check expression> ::=  
    <operand2> in type { <sort identifieur> <constraint> | <sort> }
```

La sorte de l'opérande `<operand2>` doit être la même que la sorte identifiée par l'identificateur `<sort identifieur>` ou par la sorte `<sort>`.

#### Sémantique

Une expression *Range-Check-Expression* est une expression de la sorte booléenne prédéfinie dont le résultat est Vrai (true) si le résultat de l'*Expression* remplit la condition *Range-condition* correspondant `<constraint>` telle que définie au 12.1.9.5, autrement, le résultat est Faux (false).

#### Modèle

La spécification d'une sorte `<sort>` est une syntaxe dérivée permettant de spécifier la contrainte `<constraint>` du type de données qui a défini la sorte `<sort>`. Si ce type de données n'a pas été défini avec une contrainte `<constraint>`, l'expression `<range check expression>` n'est pas évaluée et l'expression `<range check expression>` est une syntaxe dérivée permettant de spécifier la valeur booléenne prédéfinie Vrai (true).

### 12.3 Utilisation active des données

Le présent sous-paragraphe définit l'utilisation des données et des variables déclarées, les méthodes d'interprétation des expressions impliquant une variable et les expressions impératives permettant d'obtenir des résultats auprès du système sous-jacent.

Une variable a une sorte et un élément de données associé de cette sorte. L'élément de données associé à une variable peut être modifié si un nouvel élément de données est affecté à la variable. On peut utiliser l'élément de données associé à la variable dans une expression en accédant à la variable.

Une expression quelconque contenant une variable est considérée comme "active" car l'élément de données obtenu en interprétant l'expression peut varier selon le dernier élément de données affecté à la variable. Le résultat de l'interprétation d'une expression active dépendra de l'état actuel du système.

### 12.3.1 Définition de variable

Une variable a un élément de données associé ou elle est "non définie".

#### Grammaire abstraite

*Variable-definition* :: *Variable-name*  
*Sort-reference-identif*  
[ *Constant-expression* ]  
*Variable-name* = *Name*

Si l'expression *Constant-expression* est présente, elle doit être de la même sorte que l'identificateur *Sort-reference-identif* désigné.

#### Grammaire textuelle concrète

<variable definition> ::= **dcl** [**exported**] <variables of sort> {, <variables of sort> }\* <end>  
<variables of sort> ::= <variable name> [<exported as>] {, <variable name> [<exported as>]}\*  
<sort> [ <is assigned sign> <constant expression> ]  
<exported as> ::= **as** <remote variable identifier>

L'attribut <exported as> ne peut être utilisé que pour une variable ayant **exported** dans sa définition <variable definition>. Deux variables exportées dans un agent ne peuvent pas mentionner le même identificateur <remote variable identifier>.

L'expression *Constant-expression* est représentée par:

- cette expression <constant expression> si elle est donnée dans la définition <variable definition>;
- autrement, si le type de données qui a défini la sorte <sort> a une initialisation <default initialization>, elle est représentée par l'expression <constant expression> de l'initialisation <default initialization>.

Sinon, l'expression *Constant-expression* est absente.

#### Sémantique

Lorsqu'une variable est créée et que l'expression *Constant-expression* est présente, la variable est associée avec le résultat de l'expression *Constant-expression*. Autrement, la variable n'a pas d'élément de données associé; c'est-à-dire qu'elle est "non définie".

Si l'identificateur *Sort-reference-identif* est un identificateur *Syntype-identif*, l'expression *Constant-expression* est présente et le résultat de l'expression *Constant-expression* ne remplit pas la condition *Range-condition*, l'exception prédéfinie *OutOfRangeException* est déclenchée (voir D.3.16).

Le mot clé **exported** permet d'utiliser une variable en tant que variable exportée conformément au 10.6.

### 12.3.2 Accès de variable

#### Grammaire abstraite

*Variable-access* = *Variable-identif*

#### Grammaire textuelle concrète

<variable access> ::= <variable identifier>  
| **this**

**this** doit apparaître uniquement dans les définitions de méthodes.



## Sémantique

L'interprétation d'un accès de variable donne l'élément de données associé à la variable identifiée.

Un accès de variable a une sorte statique, qui est la sorte de la variable identifiée par l'accès de variable. Il a une sorte dynamique qui est celle de l'élément de données associé à la variable identifiée.

Un accès de variable a un résultat qui est le dernier élément de données associé à la variable. Si la variable est "non définie", un nœud Raise-node pour l'exception prédéfinie `UndefinedVariable` (voir D.3.16) est interprété.

## Modèle

Un accès `<variable access>` utilisant le mot clé **this** est remplacé par le nom anonyme introduit comme le nom du paramètre principal des `<arguments>`, conformément au 12.1.8.

### 12.3.3 Affectation et tentative d'affectation

Une affectation crée une association entre la variable et le résultat de l'interprétation d'une expression. Lors d'une tentative d'affectation, cette association est créée uniquement si les sortes dynamiques de la variable et de l'expression sont compatibles.

#### Grammaire abstraite

*Assignment* :: *Variable-identifiant*  
*Expression*

*Assignment-attempt* :: *Variable-identifiant*  
*Expression*

Dans une affectation *Assignment*, la sorte de l'*Expression* doit être compatible avec celle de l'identificateur *Variable-identifiant*.

Dans une tentative *Assignment-attempt*, la sorte de l'identificateur *Variable-identifiant* doit être compatible avec celle de l'*Expression*.

Si la variable est déclarée avec un *Syntaxe* et si l'*Expression* est une expression *Constant-expression*, la vérification d'intervalle définie au 12.1.9.5 et appliquée à l'*Expression* doit être la valeur booléenne prédéfinie `Vrai` (`true`).

#### Grammaire textuelle concrète

`<assignment> ::=`  
`<variable> <is assigned sign> <expression>`

`<variable> ::=`  
`<variable identifiant>`  
`| <extended variable>`

Si la `<variable>` est un identificateur `<variable identifiant>`, l'`<expression>` en syntaxe concrète représente alors l'*Expression* en syntaxe abstraite. Une variable `<extended variable>` est une syntaxe dérivée et est remplacée au niveau de la syntaxe concrète comme défini en 12.3.3.1 avant de tenir compte de la relation avec la syntaxe abstraite.

Si l'identificateur `<variable identifiant>` a été déclaré avec une sorte d'objet et si la sorte de l'`<expression>` est une supersorte (directe ou indirecte) de la sorte de l'identificateur `<variable identifiant>`, l'affectation `<assignment>` représente une tentative *Assignment-attempt*. Autrement, l'affectation `<assignment>` représente une affectation *Assignment*.

## Sémantique

Une affectation *Assignment* est interprétée comme créant une association entre la variable identifiée dans l'affectation et le résultat de l'expression contenue dans l'affectation. L'association antérieure de la variable est perdue.

La méthode utilisée pour établir cette association dépend de la sorte de l'identificateur <variable identifieur> et de la sorte de l'<expression>:

- a) si l'identificateur <variable identifieur> a une sorte de valeur, le résultat de l'*Expression* est copié dans la valeur actuellement associée à l'identificateur *Variable-identifieur* par l'interprétation de la méthode de *copie* définie dans la définition de type de données ayant introduit la sorte de l'identificateur <variable identifieur>, en prenant l'identificateur *Variable-identifieur* et l'*Expression* comme des paramètres réels. Si l'*Expression* est Null, l'exception prédéfinie *InvalidReference* (voir D.3.16) est déclenchée;
- b) si l'identificateur <variable identifieur> a une sorte d'objet et si le résultat de l'*Expression* est un objet, l'identificateur *Variable-identifieur* est associé à l'objet qui est le résultat de l'*Expression*. Il n'est pas admis que la sorte de l'<expression> soit un syntype qui restreint les éléments de la sorte de l'identificateur <variable identifieur>;
- c) si l'identificateur <variable identifieur> a une sorte d'objet et si le résultat de l'*Expression* est une valeur, le clone du résultat de l'*Expression* est construit en interprétant l'opérateur du *clone* défini par la définition de type de données qui a introduit la sorte de l'identificateur <variable identifieur>, en prenant l'*Expression* comme le paramètre réel. L'identificateur *Variable-identifieur* est associé à la valeur clonée au moyen d'une référence. Il n'est pas admis que la sorte de l'<expression> soit un syntype qui restreint les éléments de la sorte de l'identificateur <variable identifieur>;
- d) si l'identificateur <variable identifieur> a une sorte de pid et si le résultat de l'*Expression* est un pid, l'identificateur *Variable-identifieur* est associé au pid qui est le résultat de l'*Expression*.

Si la variable est déclarée avec un syntype, la vérification d'intervalle définie au 12.1.9.5 est appliquée à l'expression. Si cette vérification d'intervalle renvoie la valeur booléenne prédéfinie Faux (false), l'exception prédéfinie *OutOfRange* (voir D.3.16) est déclenchée.

Lorsqu'une tentative *Assignment-attempt* est interprétée, si la sorte dynamique de l'*Expression* est compatible avec celle de l'identificateur *Variable-identifieur*, une affectation impliquant l'identificateur *Variable-identifieur* et l'*Expression* est interprétée. Autrement, l'identificateur *Variable-identifieur* est associé à Null.

NOTE – Il est possible de déterminer la sorte dynamique d'une *Expression* au moyen d'une tentative d'affectation.

### 12.3.3.1 Variable étendue

Une variable étendue est une notation syntaxique abrégée; toutefois, à l'exception de forme syntaxique spéciale, une variable étendue n'a pas de propriétés spéciales et désigne une opération et ses paramètres.

#### *Grammaire textuelle concrète*

```
<extended variable> ::=
    <indexed variable>
    | <field variable>

<indexed variable> ::=
    <variable> ( <actual parameter list> )
    | <variable> <left square bracket> <actual parameter list> <right square bracket>

<field variable> ::=
    <variable> <exclamation mark> <field name>
    | <variable> <full stop> <field name>
```

## Modèle

<indexed variable> is derived concrete syntax for:

<variable> <is assigned sign> <variable> <full stop> Modify ( *expressionlist* )

où *expressionlist* est construite en ajoutant <expression> à la liste <actual parameter list>. La grammaire abstraite est déterminée à partir de cette expression concrète conformément au 12.2.1. La même méthode s'applique à la seconde forme de variable <indexed variable>.

La forme syntaxique concrète:

<variable> <exclamation mark> <field name> <is assigned sign> <expression>

est la syntaxe concrète dérivée de:

<variable> <full stop> *field-modify-operation-name* ( <expression> )

où le nom *field-modify-operation-name* est formé à partir de la concaténation du nom de champ et de "Modify". La syntaxe abstraite est déterminée à partir de cette expression concrète conformément au 12.2.1. Le même modèle s'applique à la seconde forme de variable <field variable>.

### 12.3.3.2 Initialisation par défaut

Une initialisation par défaut permet l'initialisation de toutes les variables d'une sorte spécifiée avec le même élément de données, lorsque les variables sont créées.

#### *Grammaire textuelle concrète*

<default initialization> ::=

**default** [<virtuality>] [<constant expression>] [<end>]

Une définition <data type definition> ou <syntype definition> doit contenir au plus une initialisation <default initialization>.

L'expression <constant expression> peut être omise si <virtuality> est **redefined** ou **finalized**.

#### *Sémantique*

Une initialisation par défaut peut être ajoutée aux opérations <operations> d'une définition de type de données. Une initialisation par défaut spécifie que n'importe quelle variable déclarée avec la sorte introduite par la définition de type de données ou la définition de syntype est au départ associée au résultat de l'expression <constant expression>.

## Modèle

Une initialisation par défaut est une abréviation pour spécifier une initialisation explicite pour toutes les variables déclarées de la sorte <sort>, mais dont la définition <variable definition> ne comporte pas d'expression <constant expression>.

Si la définition <syntype definition> ne donne pas d'initialisation <default initialization>, le syntype a alors l'initialisation <default initialization> de l'identificateur <parent sort identifier> à condition que son résultat soit dans l'intervalle.

Une initialisation <default initialization> implicite de valeur Null est donnée à toute variable définie par une définition <object data type definition>, excepté si une initialisation <default initialization> était présente dans la définition <object data type definition>.

Toute sorte de pid est traitée comme s'il lui avait été implicitement donné une initialisation <default initialization> de valeur Null.

Si l'expression <constant expression> est omise dans une initialisation par défaut redéfinie, l'initialisation explicite n'est pas ajoutée.

### 12.3.4 Expressions impératives

Les expressions impératives obtiennent des résultats de l'état du système sous-jacent.

Les transformations décrites dans les *Modèles* du présent sous-paragraphe sont effectuées au même moment que le développement de l'import. Une étiquette attachée à une action dans laquelle apparaît une expression impérative est déplacée à la première tâche insérée pendant la transformation décrite. Si plusieurs expressions impératives apparaissent dans une expression, les tâches sont insérées dans le même ordre que l'ordre d'apparition des expressions impératives dans l'expression.

#### *Grammaire abstraite*

<i>Imperative-expression</i>	=	<i>Now-expression</i>
		<i>Pid-expression</i>
		<i>Timer-active-expression</i>
		<i>Any-expression</i>

#### *Grammaire textuelle concrète*

```
<imperative expression> ::=
    <now expression>
    | <import expression>
    | <pid expression>
    | <timer active expression>
    | <any expression>
    | <state expression>
```

Les expressions impératives sont des expressions permettant l'accès à l'horloge du système, aux résultats des variables importées, au pid associé à un agent, aux variables visibles, aux états des temporisateurs ou permettant de fournir des éléments de données non spécifiés.

#### 12.3.4.1 Expression maintenant (Now)

##### *Grammaire abstraite*

*Now-expression* :: ()

##### *Grammaire textuelle concrète*

```
<now expression> ::=
    now
```

##### *Sémantique*

L'expression maintenant (now) est une expression qui permet d'accéder à l'horloge du système pour déterminer le temps absolu du système.

L'expression maintenant représente une expression demandant la valeur actuelle de l'horloge du système indiquant le temps. L'origine et l'unité de temps dépendent du système, tout comme la question de savoir si l'on obtient la même valeur lorsque deux occurrences **now** se présentent dans la même transition. Toutefois, il est toujours vrai que:

**now** <= **now**;

Une expression maintenant a la sorte temps.

#### 12.3.4.2 Expression d'import

##### *Grammaire textuelle concrète*

La syntaxe concrète d'une expression d'import est définie au 10.6.

##### *Sémantique*

En plus de la sémantique définie au 10.6, une expression d'import est interprétée comme un accès de variable (voir 12.3.2) à la variable implicite pour l'expression d'import.

## Modèle

L'expression d'import a une syntaxe implicite pour l'importation du résultat définie au 10.6 et comporte également un accès *Variable-access* implicite de la variable implicite pour l'import dans le contexte où l'expression <import expression> apparaît.

L'utilisation de l'expression <import expression> dans une expression est une abréviation pour insérer une tâche juste avant l'action, où se produit l'expression qui affecte à une variable implicite le résultat de l'expression <import expression> et utilise ensuite cette variable implicite dans l'expression. Si l'expression <import expression> se produit plusieurs fois dans une expression, une variable doit être utilisée pour chaque occurrence.

### 12.3.4.3 Expression Pid

#### Grammaire abstraite

<i>Pid-expression</i>	=	<i>Self-expression</i>
		<i>Parent-expression</i>
		<i>Offspring-expression</i>
		<i>Sender-expression</i>
<i>Self-expression</i>	::	()
<i>Parent-expression</i>	::	()
<i>Offspring-expression</i>	::	()
<i>Sender-expression</i>	::	()

#### Concrete textual grammar

```
<pid expression> ::=
    self
    | parent
    | offspring
    | sender
<create expression> ::=
    <create request>
```

#### Sémantique

Une expression pid permet d'accéder à l'une des variables anonymes implicites **self**, **parent**, **offspring** ou **sender** (voir paragraphe 9, *Modèle*). L'expression pid **self**, **parent**, **offspring** ou **sender** a un résultat qui est le dernier pid associé à la variable implicite correspondante telle que définie au paragraphe 9.

La sorte dynamique d'une expression <pid expression> est la sorte dynamique de son résultat.

Une expression pid **parent**, **offspring** ou **sender** a une sorte statique, qui est Pid.

Si une expression <create expression> inclut un identificateur <agent identifier>, elle a une sorte statique, qui est la sorte de pid de l'agent désigné par l'identificateur <agent identifier>. Si une expression <create expression> inclut un identificateur <agent type identifier>, elle a une sorte statique, qui est la sorte de pid du type d'agent identifié par l'identificateur <agent type identifier>. Si l'expression <create expression> inclut **this**, elle a une sorte statique qui est la sorte de pid de l'agent ou du type d'agent dans lequel l'expression de création apparaît. Si l'expression <create expression> inclut **this** et qu'elle apparaît dans un contexte situé à l'extérieur d'un agent ou d'un type d'agent (par exemple dans une procédure globale), elle a alors la sorte statique Pid.

Une expression **self** a une sorte statique qui est le pid de l'agent ou du type d'agent dans lequel apparaît l'expression **self**. Si elle apparaît dans un contexte qui se situe à l'extérieur d'un agent ou d'un type d'agent (par exemple dans une procédure globale), elle a la sorte statique Pid.

## Modèle

L'utilisation de l'expression <create expression> dans une expression est une abréviation pour insérer une demande de création juste avant une action où se produit l'expression <create expression>, suivie d'une affectation de l'expression **offspring** à une variable anonyme déclarée implicitement ayant la même sorte que la sorte statique de l'expression <create expression>. La variable implicite est ensuite utilisée dans l'expression. Si l'expression <create expression> se produit plusieurs fois dans une expression, une variable distincte doit être utilisée pour chaque occurrence. Dans ce cas, l'ordre des demandes de création et des affectations de variables insérées est le même que celui des expressions <create expression>.

Si l'expression <create expression> contient un identificateur <agent type identifier>, les transformations appliquées à une instruction de création contenant un identificateur <agent type identifier> sont également appliquées à l'instruction de création implicite résultant de la transformation d'une expression <create expression> (voir 11.13.2).

### 12.3.4.4 Expression active de temporisateur

#### Grammaire abstraite

*Timer-active-expression* :: *Timer-identifier*  
*Expression\**

Les sortes de la liste d'*Expression* dans l'expression *Timer-active-expression* doivent correspondre par leur position à la liste d'identificateurs *Sort-reference-identifier* suivant immédiatement le nom *Timer-name* (11.15) identifié par l'identificateur *Timer-identifier*.

#### Grammaire textuelle concrète

<timer active expression> ::= **active** ( <timer identifier> [ ( <expression list> ) ] )

#### Sémantique

Une expression active de temporisateur est une expression de la sorte booléenne prédéfinie qui a le résultat Vrai (true) si le temporisateur identifié par l'identificateur de temporisateur, et positionné avec les mêmes résultats que ceux indiqués par la liste d'expressions (le cas échéant), est actif (voir 11.15). Dans le cas contraire, l'expression active de temporisateur a la valeur Faux (false). Les expressions sont interprétées dans l'ordre indiqué.

Si une sorte spécifiée dans une définition de temporisateur est un syntype, la vérification d'intervalle définie en 12.1.9.5 et appliquée à l'expression correspondante dans une liste <expression list> doit être la valeur booléenne prédéfinie Vrai (true); dans le cas contraire, l'exception prédéfinie *OutOfRangeException* (voir D.3.16) est déclenchée.

### 12.3.4.5 Expression quelconque (Any expression)

L'expression *Any-expression* est utile pour modéliser le comportement lorsqu'un élément de données spécifique peut donner lieu à une surspécification. On ne peut pas tirer des hypothèses concernant les autres résultats retournés par l'interprétation d'une expression *Any-expression* à partir d'un résultat retourné par une expression *Any-expression*.

#### Grammaire abstraite

*Any-expression* :: *Sort-reference-identifier*

#### Grammaire textuelle concrète

<any expression> ::= **any** ( <sort> )

La <sort> doit contenir des éléments.

## Sémantique

Une expression *Any-expression* retourne un élément non spécifié de la sorte ou du syntype désigné par l'identificateur *Sort-reference-identif*, si cette sorte ou syntype est une sorte de valeur. Si l'identificateur *Sort-reference-identif* désigne un identificateur *Syntype-identif*, le résultat sera dans l'intervalle de ce syntype. Si la sorte ou le syntype désigné(e) par l'identificateur *Sort-reference-identif* est une sorte d'objet ou une sorte de pid, l'expression *Any-expression* renvoie Null.

### 12.3.4.6 Expression d'état

#### Grammaire textuelle concrète

```
<state expression> ::=  
state
```

#### Sémantique

Une expression d'état désigne une valeur de chaîne de caractères contenant l'orthographe de l'état entré le plus récemment.

#### Modèle

L'expression <state expression> est une syntaxe dérivée pour un littéral Charstring qui contient l'orthographe du nom de l'état entré le plus récemment de l'unité de portée englobante la plus proche. Si un tel état n'existe pas, l'expression <state expression> désigne la chaîne vide (''). La construction est transformée avec l'état suivant <dash nextstate> (voir Annexe F).

### 12.3.5 Appel de procédure renvoyant une valeur

La grammaire abstraite et les contraintes sémantiques statiques pour un appel de procédure renvoyant une valeur sont présentées en 11.13.3.

#### Grammaire textuelle concrète

```
<value returning procedure call> ::=  
[ call ] <procedure call body>  
| [ call ] <remote procedure call body>
```

Le mot clé **call** ne peut pas être omis si l'appel <value returning procedure call> est syntaxiquement ambigu avec une opération (ou une variable) ayant le même nom suivi d'une liste de paramètres.

NOTE 1 – Cette ambiguïté n'est pas résolue par le contexte.

Un appel <value returning procedure call> ne doit pas se produire dans l'expression <Boolean expression> d'une zone <continuous signal area>, d'un signal <continuous signal>, d'une zone <enabling condition area> ou d'une condition <enabling condition>.

L'identificateur <procedure identif> dans un appel <value returning procedure call> doit identifier une procédure ayant un résultat <procedure result>.

Une <expression> dans <actual parameters> correspondant à un paramètre formel **in/out** ou **out** ne peut être omise et doit être un identificateur <variable identif>.

Une fois le *Modèle* de **this** appliqué, l'identificateur <procedure identif> doit désigner une procédure contenant une transition de départ.

Si **this** est utilisé, l'identificateur <procedure identif> doit désigner une procédure englobante.

Le corps <procedure call body> représente un nœud *Value-returning-call-node*, dans lequel l'identificateur *Procedure-identif* est représenté par l'identificateur <procedure identif> et la liste d'*Expressions* est représentée par la liste des paramètres réels. Le corps <remote procedure call body> représente un nœud *Value-returning-call-node*, dans lequel l'identificateur *Procedure-identif* contient uniquement l'identificateur *Procedure-identif* de la procédure

implicitement définie par le *Modèle* ci-dessous. La sémantique du nœud *Value-returning-call-node* est présentée en 11.13.3.

### *Modèle*

Si l'identificateur `<procedure identifier>` n'est pas défini à l'intérieur de l'agent englobant, l'appel de procédure est transformé en un appel d'un sous-type local de la procédure créé implicitement.

**this** implique que lorsque la procédure est spécialisée, l'identificateur `<procedure identifier>` est remplacé par l'identificateur de la procédure spécialisée.

Lorsque l'appel `<value returning procedure call>` contient un corps `<remote procedure call body>`, la procédure suivante, avec un nom anonyme désigné par *RPCcall*, est implicitement définie. *RPCsort* est la sorte `<sort>` dans le résultat `<procedure result>` de la définition de procédure désignée par l'identificateur `<procedure identifier>`.

```
procedure RPCcall -> RPCsort;  
  start ;  
  return call <remote procedure call body>;  
endprocedure ;
```

NOTE 2 – Cette transformation n'est pas appliquée à nouveau à la définition de procédure implicite.

## 13 Définition de système générique

Afin de répondre à divers besoins, une spécification de système peut avoir des parties optionnelles et des paramètres de système dont les résultats ne sont pas spécifiés. Une telle spécification de système est appelée générique, sa propriété générique est spécifiée aux moyens de synonymes externes (qui sont analogues aux paramètres formels d'une définition de procédure). La spécification d'un système générique est adaptée en choisissant un sous-ensemble convenable et en donnant un élément de données à chaque paramètre de système. La spécification de système qui en résulte ne contient pas de synonymes externes, et est appelée spécification de système spécifique.

Une définition de système générique est une définition de système qui contient un synonyme défini par un élément `<external synonym definition item>` (voir 12.1.9.6), une opération décrite par une définition `<external operation definition>` (voir 12.1.8), une procédure décrite par une définition `<external procedure definition>` (voir 9.4) ou du texte `<informal text>` dans une option de transition (voir 13.2). Une définition de système particulière est créée à partir d'une définition générique de système en donnant des résultats pour les éléments `<external synonym definition item>`, comme fournir le comportement pour des définitions `<external operation definition>` et des définitions `<external procedure definition>`, et en transformant le texte `<informal text>` en construction formelle. La manière d'effectuer cette opération et la relation avec la grammaire abstraite ne font pas partie de la définition de langage.



## 13.1 Définition optionnelle

### Grammaire textuelle concrète

<select definition> ::=

```
select if ( <Boolean simple expression> ) <end>
{
  <agent type definition>
  <agent type reference>
  <agent definition>
  <channel definition>
  <signal definition>
  <signal list definition>
  <remote variable definition>
  <remote procedure definition>
  <data definition>
  <data type reference>
  <composite state type definition>
  <composite state type reference>
  <state partitioning>
  <textual typebased state partition definition>
  <timer definition>
  <variable definition>
  <procedure definition>
  <procedure reference>
  <channel to channel connection>
  <select definition>
  <macro definition>
  <exception definition> }+
endselect <end>
```

Les seuls noms visibles dans une expression <Boolean simple expression> d'une définition <select definition> sont des noms de synonymes externes définis à l'extérieur de toutes les définitions <select definition> et zones <option area> et des littéraux et des opérations des types de données définis à l'intérieur du paquetage Predefined, comme défini dans l'Annexe D.

Une définition <select definition> ne peut contenir que les définitions syntaxiquement autorisées à cet endroit.

### Grammaire graphique concrète

<option area> ::=

```
<option symbol> contains
{ select if ( <Boolean simple expression> )
  {
    <agent type diagram>
    <agent type reference area>
    <agent area>
    <channel definition area>
    <agent text area>
    <procedure text area>
    <composite state type diagram>
    <composite state type reference area>
    <state partition area>
    <procedure area>
    <create line area>
    <option area> } + }
```

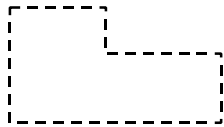
<option symbol> ::=

```
{ <dashed line symbol> }-set
```

<dashed line symbol> ::=

```
-----
```

Le symbole <option symbol> doit former un polygone en pointillés ayant des angles droits, par exemple:



Un symbole <option symbol> contient logiquement la totalité d'un symbole graphique unidimensionnel quelconque coupé par sa frontière (c'est-à-dire avec un point d'extrémité à l'extérieur).

Une zone <option area> peut apparaître n'importe où, sauf à l'intérieur d'une zone <state machine graph area> et d'une zone <type state machine graph area>. Une zone <option area> ne peut contenir que les zones et les diagrammes qui sont syntaxiquement autorisés à cet endroit.

### Sémantique

Si le résultat de l'expression <Boolean simple expression> est la valeur booléenne prédéfinie Faux (false), toutes les constructions contenues dans la définition <select definition> ou dans le symbole <option symbol> ne sont pas sélectionnées. Dans l'autre cas, ces constructions sont sélectionnées.

### Modèle

La définition <select definition> et la zone <option area> sont supprimées à la transformation et remplacées par les constructions sélectionnées qu'elles contiennent, s'il y en a. Tous les connecteurs connectés à une zone dans des zones <option area> non sélectionnées sont également supprimés.

## 13.2 Chaîne de transition optionnelle

### Grammaire textuelle concrète

```
<transition option> ::=  
    alternative <alternative question> <end>  
        <decision body>  
    endalternative
```

```
<alternative question> ::=  
    <simple expression>  
    | <informal text>
```

Chaque expression <constant expression> dans une réponse <answer> d'un corps <decision body> doit être une expression <simple expression>. Les réponses <answer> dans le corps <decision body> d'une option <transition option> doivent s'exclure mutuellement. Si la question <alternative question> est une <expression>, la condition *Range-condition* des réponses <answer> dans le corps <decision body> doit être de la même sorte que la question <alternative question>.

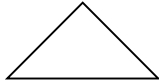
Il y a une ambiguïté syntaxique entre texte <informal text> et chaîne <character string> dans une question <alternative question> et les réponses <answer> contenues dans le corps <decision body>. Si la question <alternative question> et toutes les réponses <answer> sont des chaînes <character string>, elles sont toutes interprétées en tant que texte <informal text>. Si la question <alternative question> ou une quelconque réponse <answer> est une chaîne <character string> et que cela ne correspond pas au contexte de l'option de transition, la chaîne <character string> désigne un texte <informal text>.

Aucune réponse <answer> dans les parties <answer part> d'un corps <decision body> d'une option <transition option> ne peut être omise.

### Grammaire graphique concrète

```
<transition option area> ::=  
    <transition option symbol> contains <alternative question>  
    is followed by <graphical decision body>
```

<transition option symbol> ::=



Les symboles <flow line symbol> dans <graphical decision body> sont reliés au bas du symbole <transition option symbol>.

Les symboles <flow line symbol> issus d'un symbole <transition option symbol> peuvent avoir un trajet de départ commun.

Les réponses <graphical answer> dans le corps <graphical decision body> doivent s'exclure mutuellement.

### *Sémantique*

Les constructions dans une partie <graphical answer part> sont sélectionnées si la réponse <answer> contient le résultat de la question <alternative question>. Si aucune des réponses <answer> ne contient le résultat de la question <alternative question>, ce sont les constructions dans la partie <graphical else part> qui sont sélectionnées.

Si aucune partie <graphical else part> n'est fournie et aucun des trajets sortants n'est sélectionné, la sélection n'est pas valable.

### *Modèle*

Si une option <transition option> n'est pas terminale, elle constitue une syntaxe dérivée pour une option <transition option> dans laquelle toutes les parties <answer part> et les parties <else part> ont inséré dans leur <transition>:

- a) un branchement <join> à l'instruction <terminator statement> suivante, si l'option de transition est la dernière instruction <action statement> dans une chaîne <transition string>;
- b) sinon un branchement <join> à la première instruction <action statement> qui suit l'option de transition.

L'option <transition option> terminale est définie en 11.13.5.

L'option <transition option> et la zone <transition option area> sont supprimées à la transformation et remplacées par les constructions sélectionnées contenues.

## ANNEXE A

### Index des non-terminaux

Les non-terminaux ci-après (en anglais) sont définis à dessein et ne sont pas utilisés: <macro call>, <page>, <comment area>, <text extension area>, <sdl specification>.

- <abstract>, **54**
  - use in syntax, 45
  - use in text, 54, 127
- <action 1>, **119**
  - use in syntax, 119
- <action 2>, **119**
  - use in syntax, 119
- <action statement>, **119**
  - use in syntax, 119
  - use in text, 105, 110, 122, 132, 133, 135, 137, 148, 195
- <active primary>, **177**
  - use in syntax, 177
  - use in text, 177
- <actual context parameter list>, **61**
  - use in syntax, 61
- <actual context parameter>, **61**
  - use in syntax, 61
  - use in text, 51, 61
- <actual context parameters>, **61**
  - use in syntax, 50, 67
  - use in text, 50, 56, 62, 67
- <actual parameter list>, **126**
  - use in syntax, 126, 179, 187
  - use in text, 126, 179, 187
- <actual parameters>, **126**
  - use in syntax, 53, 94, 103, 121, 124, 126, 128, 129, 182
  - use in text, 104, 121, 124, 127, 128, 130, 131, 169, 182, 183, 191
- <agent additional heading>, **75**
  - use in syntax, 45, 75, 81
  - use in text, 81
- <agent area>, **77**
  - use in syntax, 77, 90, 193
  - use in text, 78
- <agent body area>, **76**
  - use in syntax, 76
  - use in text, 80
- <agent body>, **75**
  - use in syntax, 75
  - use in text, 75, 80, 110
- <agent constraint>, **63**
  - use in syntax, 63
- <agent context parameter>, **63**
  - use in syntax, 61
  - use in text, 46
- <agent definition>, **75**
  - use in syntax, 38, 43, 44, 75, 193
  - use in text, 32, 76, 80, 105, 109
- <agent diagram content>, **76**
  - use in syntax, 81, 82, 83
  - use in text, 78
- <agent diagram>, **76**
  - use in syntax, 38, 43, 77
  - use in text, 32, 36, 76, 78
- <agent formal parameters>, **75**
  - use in syntax, 49, 75, 112, 114
  - use in text, 32, 46, 63, 76, 81, 121, 254
- <agent instantiation>, **75**
  - use in syntax, 82, 83
  - use in text, 76
- <agent reference area>, **77**
  - use in syntax, 38, 77
  - use in text, 76, 78
- <agent reference>, **75**
  - use in syntax, 44, 75, 77
  - use in text, 76, 78
- <agent signature>, **63**
  - use in syntax, 62, 63
  - use in text, 62, 63
- <agent structure>, **75**
  - use in syntax, 81, 82, 83
  - use in text, 75
- <agent text area>, **77**
  - use in syntax, 45, 76, 193
  - use in text, 37, 54, 78
- <agent type additional heading>, **45**
  - use in syntax, 46, 47
  - use in text, 46
- <agent type area>, **78**
  - use in syntax, 77
  - use in text, 78
- <agent type body area>, **45**
  - use in syntax, 45
  - use in text, 46
- <agent type body>, **45**
  - use in syntax, 45
  - use in text, 46, 122
- <agent type constraint>, **62**
  - use in syntax, 62
- <agent type context parameter>, **62**
  - use in syntax, 61
- <agent type definition>, **44**
  - use in syntax, 40, 43, 75, 193
  - use in text, 32, 80, 126, 129
- <agent type diagram content>, **45**
  - use in syntax, 46, 47, 48
- <agent type diagram>, **45**
  - use in syntax, 43, 45, 78, 193
  - use in text, 32, 36, 45
- <agent type reference area>, **45**
  - use in syntax, 45, 74, 78, 193
  - use in text, 45
- <agent type reference>, **45**
  - use in syntax, 40, 41, 75, 77, 193
- <agent type structure>, **44**
  - use in syntax, 46, 47
- <aggregation aggregate end bound kind>, **72**
  - use in syntax, 72

- <aggregation aggregate end bound symbol>, **73**
  - use in syntax, 73
- <aggregation not bound kind>, **72**
  - use in syntax, 72
- <aggregation not bound symbol>, **73**
  - use in syntax, 73
- <aggregation part end bound kind>, **72**
  - use in syntax, 72
- <aggregation part end bound symbol>, **73**
  - use in syntax, 73
- <aggregation two ends bound kind>, **72**
  - use in syntax, 72
- <aggregation two ends bound symbol>, **73**
  - use in syntax, 73
- <algorithm action statement>, **135**
  - use in syntax, 133
  - use in text, 136
- <algorithm answer part>, **137**
  - use in syntax, 137
  - use in text, 137
- <algorithm else part>, **137**
  - use in syntax, 137
- <alphanumeric>, **25**
  - use in syntax, 25, 26
- <alternative expression>, **181**
  - use in syntax, 181
  - use in text, 181, 234
- <alternative question>, **194**
  - use in syntax, 194
  - use in text, 194, 195
- <alternative statement>, **136**
  - use in syntax, 136, 137
  - use in text, 136, 137
- <ampersand>, **27**
  - use in syntax, 27
- <anchored sort>, **150**
  - use in syntax, 150
  - use in text, 150, 151, 155, 158, 163
- <answer part>, **131**
  - use in syntax, 131
  - use in text, 132, 133, 137, 194, 195
- <answer>, **131**
  - use in syntax, 131, 132
  - use in text, 34, 132, 133, 194, 195
- <any expression>, **191**
  - use in syntax, 188
  - use in text, 133, 173
- <apostrophe>, **27**
  - use in syntax, 25, 26
  - use in text, 25, 37, 169
- <argument virtuality>, **156**
  - use in syntax, 156, 166
  - use in text, 71, 155, 156, 157, 158
- <argument>, **156**
  - use in syntax, 156
  - use in text, 71, 155, 156, 157, 158
- <arguments>, **156**
  - use in syntax, 156
  - use in text, 158, 163, 169, 172, 185
- <assignment statement>, **135**
  - use in syntax, 133
  - use in text, 135, 136, 139
- <assignment>, **186**
  - use in syntax, 125, 135
  - use in text, 34, 105, 136, 148, 186
- <association area>, **73**
  - use in syntax, 45
  - use in text, 74
- <association end area>, **73**
  - use in syntax, 73
  - use in text, 74
- <association end bound kind>, **72**
  - use in syntax, 72
- <association end bound symbol>, **73**
  - use in syntax, 73
- <association end>, **72**
  - use in syntax, 72
  - use in text, 72, 73
- <association kind>, **72**
  - use in syntax, 72
- <association not bound kind>, **72**
  - use in syntax, 72
- <association not bound symbol>, **73**
  - use in syntax, 73
- <association symbol>, **73**
  - use in syntax, 73
  - use in text, 35
- <association two ends bound kind>, **72**
  - use in syntax, 72
- <association two ends bound symbol>, **73**
  - use in syntax, 73
- <association>, **72**
  - use in syntax, 40, 75
  - use in text, 72, 73, 74
- <asterisk connect list>, **117**
  - use in syntax, 117
  - use in text, 118
- <asterisk exception handler list>, **143**
  - use in syntax, 143
  - use in text, 143, 144
- <asterisk exception stimulus list>, **147**
  - use in syntax, 147
  - use in text, 143, 147
- <asterisk input list>, **104**
  - use in syntax, 104
  - use in text, 104, 105, 108
- <asterisk save list>, **108**
  - use in syntax, 108
  - use in text, 104, 108, 109
- <asterisk state list>, **101**
  - use in syntax, 101, 103
  - use in text, 101, 103, 144
- <asterisk>, **27**
  - use in syntax, 25, 26, 101, 104, 108, 117, 143, 147, 169, 176
  - use in text, 22, 70, 114
- <attribute properties area>, **58**
  - use in syntax, 57
  - use in text, 58
- <attribute property>, **54**
  - use in syntax, 54, 58
  - use in text, 54, 58
- <axiomatic operation definitions>, **231**
  - use in syntax, 231
  - use in text, 231, 233

- <axioms>, **231**
  - use in syntax, 231
  - use in text, 233, 235, 236
- <base type>, **50**
  - use in syntax, 50
  - use in text, 50, 51, 56, 59, 61, 62, 67, 93, 154, 155
- <basic sort>, **150**
  - use in syntax, 150
  - use in text, 150, 151, 155
- <basic state area>, **102**
  - use in syntax, 101
- <basic state>, **101**
  - use in syntax, 101
- <basic type reference area>, **57**
  - use in syntax, 56
  - use in text, 56
- <behaviour properties area>, **58**
  - use in syntax, 57
  - use in text, 58
- <behaviour property>, **55**
  - use in syntax, 54, 58
  - use in text, 54, 55, 58
- <bit string>, **25**
  - use in syntax, 25, 31
  - use in text, 22, 161, 169
- <block definition>, **82**
  - use in syntax, 75
  - use in text, 31, 39, 52, 81
- <block diagram>, **82**
  - use in syntax, 76
  - use in text, 39, 77, 81, 82
- <block heading>, **82**
  - use in syntax, 82
- <block reference area>, **82**
  - use in syntax, 77
  - use in text, 43
- <block reference>, **82**
  - use in syntax, 75
  - use in text, 43
- <block symbol>, **82**
  - use in syntax, 52, 60, 81, 82
  - use in text, 51
- <block type definition>, **47**
  - use in syntax, 44
  - use in text, 47
- <block type diagram>, **47**
  - use in syntax, 45
  - use in text, 47
- <block type heading>, **47**
  - use in syntax, 47
- <block type reference area>, **47**
  - use in syntax, 45
  - use in text, 47
- <block type reference>, **47**
  - use in syntax, 45
- <block type symbol>, **47**
  - use in syntax, 46, 57, 58
- <Boolean axiom>, **234**
  - use in syntax, 231
- <break statement>, **139**
  - use in syntax, 134
  - use in text, 138, 139, 140
- <call statement>, **136**
  - use in syntax, 133
  - use in text, 136
- <channel definition area>, **90**
  - use in syntax, 77, 193
  - use in text, 78, 91, 92
- <channel definition>, **89**
  - use in syntax, 44, 75, 193
  - use in text, 80, 81, 91, 92, 254
- <channel endpoint>, **89**
  - use in syntax, 89
  - use in text, 89, 90, 92
- <channel identifiers>, **91**
  - use in syntax, 91
  - use in text, 91, 92
- <channel path>, **89**
  - use in syntax, 89
  - use in text, 81, 90, 91
- <channel symbol>, **90**
  - use in syntax, 90
  - use in text, 35, 90, 92
- <channel to channel connection>, **91**
  - use in syntax, 75, 193
  - use in text, 90, 91, 92
- <character string>, **25**
  - use in syntax, 25, 31, 35, 37, 169
  - use in text, 22, 25, 28, 30, 37, 132, 161, 169, 170, 194
- <choice definition>, **164**
  - use in syntax, 160
  - use in text, 154, 164, 171
- <choice list>, **164**
  - use in syntax, 164
  - use in text, 164
- <choice of sort>, **164**
  - use in syntax, 164
  - use in text, 164, 165, 171
- <circumflex accent>, **27**
  - use in syntax, 27
- <class symbol>, **57**
  - use in syntax, 57
  - use in text, 57, 58
- <closed range>, **174**
  - use in syntax, 173
  - use in text, 174, 175
- <colon>, **27**
  - use in syntax, 26, 27, 51, 52, 53, 63, 115, 131, 137, 169, 174
- <comma>, **27**
  - use in syntax, 27
- <comment area>, **37**
  - use in syntax, 196
  - use in text, 28
- <comment symbol>, **37**
  - use in syntax, 37
  - use in text, 35, 37
- <comment>, **37**
  - use in syntax, 37
  - use in text, 28, 37
- <commercial at>, **27**
  - use in syntax, 27
- <communication constraints>, **94**
  - use in syntax, 94, 97, 129
  - use in text, 94, 96, 99, 129

- <composite begin sign>, **26**
  - use in syntax, 26, 179
- <composite end sign>, **26**
  - use in syntax, 26, 179
- <composite special>, **26**
  - use in syntax, 25
  - use in text, 22, 254
- <composite state application area>, **103**
  - use in syntax, 101
- <composite state application>, **103**
  - use in syntax, 101
  - use in text, 101, 103, 104
- <composite state area>, **111**
  - use in syntax, 43, 115
  - use in text, 32, 36, 100, 101
- <composite state body area>, **113**
  - use in syntax, 49, 113
- <composite state body>, **112**
  - use in syntax, 48, 112
  - use in text, 110, 112
- <composite state graph area>, **113**
  - use in syntax, 111
- <composite state graph>, **112**
  - use in syntax, 111
- <composite state heading>, **112**
  - use in syntax, 112, 113
- <composite state list>, **103**
  - use in syntax, 103
- <composite state reference>, **112**
  - use in syntax, 115
  - use in text, 103
- <composite state text area>, **113**
  - use in syntax, 49, 113, 115
  - use in text, 37, 54
- <composite state type constraint>, **66**
  - use in syntax, 66
  - use in text, 66
- <composite state type context parameter>, **66**
  - use in syntax, 61
- <composite state type definition>, **48**
  - use in syntax, 40, 43, 75, 85, 112, 193
  - use in text, 32, 111
- <composite state type diagram>, **49**
  - use in syntax, 43, 45, 113, 193
  - use in text, 32, 36, 49, 50
- <composite state type heading>, **49**
  - use in syntax, 48, 49
- <composite state type reference area>, **49**
  - use in syntax, 45, 113, 193
  - use in text, 49, 50
- <composite state type reference>, **49**
  - use in syntax, 40, 75, 85, 112, 193
- <composite state type signature>, **66**
  - use in syntax, 66
- <composite state type symbol>, **49**
  - use in syntax, 57, 58
- <composite state>, **111**
  - use in syntax, 43, 115
  - use in text, 32, 53, 100, 111, 112, 113, 114, 117, 118, 121
- <composition composite end bound kind>, **72**
  - use in syntax, 72
- <composition composite end bound symbol>, **73**
  - use in syntax, 73
  - use in text, 74
- <composition not bound kind>, **72**
  - use in syntax, 72
- <composition not bound symbol>, **73**
  - use in syntax, 73
  - use in text, 74
- <composition part end bound kind>, **72**
  - use in syntax, 72
- <composition part end bound symbol>, **73**
  - use in syntax, 73
  - use in text, 74
- <composition two ends bound kind>, **72**
  - use in syntax, 72
- <composition two ends bound symbol>, **73**
  - use in syntax, 73
  - use in text, 74
- <compound statement>, **134**
  - use in syntax, 125, 133
  - use in text, 32, 33, 88, 125, 134, 140
- <concatenation sign>, **26**
  - use in syntax, 25, 26, 176
  - use in text, 29
- <conditional equation>, **233**
  - use in syntax, 231
  - use in text, 232
- <conditional expression>, **181**
  - use in syntax, 177
  - use in text, 234
- <connect area>, **118**
  - use in syntax, 103
- <connect association area>, **103**
  - use in syntax, 103
- <connect list>, **117**
  - use in syntax, 117, 118
  - use in text, 118
- <connect part>, **117**
  - use in syntax, 103
  - use in text, 118
- <connection definition area>, **115**
  - use in syntax, 115
- <consequence expression>, **181**
  - use in syntax, 181
  - use in text, 181, 234
- <consequence statement>, **136**
  - use in syntax, 136
  - use in text, 136, 137
- <constant expression>, **177**
  - Duration
    - use in syntax, 141
    - use in text, 142
  - use in syntax, 162, 174, 175, 177, 184, 187
  - use in text, 61, 65, 71, 111, 133, 145, 164, 174, 175, 177, 179, 182, 185, 187, 188, 194
- <constant>, **174**
  - use in syntax, 174
  - use in text, 133, 174, 175
- <constraint>, **173**
  - use in syntax, 172, 184
  - use in text, 172, 173, 175, 184
- <context parameters end>, **61**
  - use in syntax, 61

- <context parameters start>, **61**
  - use in syntax, 61
- <continuous expression>, **107**
  - use in syntax, 106, 107
- <continuous signal area>, **107**
  - use in syntax, 107
  - use in text, 107, 191
- <continuous signal association area>, **107**
  - use in syntax, 102, 103
- <continuous signal>, **106**
  - use in syntax, 101, 103
  - use in text, 107, 191
- <create body>, **126**
  - use in syntax, 126
- <create expression>, **189**
  - use in syntax, 176
  - use in text, 177, 190
- <create line area>, **77**
  - use in syntax, 41, 77, 193
- <create line endpoint area>, **77**
  - use in syntax, 77
- <create line symbol>, **78**
  - use in syntax, 77
  - use in text, 35, 78
- <create request area>, **126**
  - use in syntax, 119
  - use in text, 126
- <create request symbol>, **126**
  - use in syntax, 126
- <create request>, **126**
  - use in syntax, 119, 135, 189
  - use in text, 79, 127
- <dash nextstate>, **121**
  - use in syntax, 121
  - use in text, 113, 121, 122, 191
- <dashed association symbol>, **37**
  - use in syntax, 37
  - use in text, 35, 37
- <dashed block symbol>, **52**
  - use in syntax, 52
- <dashed line symbol>, **193**
  - use in syntax, 193
- <dashed process symbol>, **53**
  - use in syntax, 53
- <dashed state symbol>, **115**
  - use in syntax, 115
- <data definition>, **150**
  - use in syntax, 40, 41, 75, 77, 85, 87, 112, 113, 166, 167, 193
- <data symbol>, **57**
  - use in syntax, 57
  - use in text, 57
- <data type closing>, **151**
  - use in syntax, 151, 172
  - use in text, 152
- <data type constructor>, **160**
  - use in syntax, 151
  - use in text, 152, 154
- <data type definition body>, **151**
  - use in syntax, 151, 172
- <data type definition>, **151**
  - object
    - use in text, 151, 152, 158, 188
  - use in syntax, 150, 151
  - use in text, 32, 33, 57, 150, 152, 154, 155, 158, 160, 161, 162, 164, 165, 168, 169, 170, 171, 173, 187, 233
  - value
    - use in text, 151, 152, 164, 165, 255
- <data type heading>, **151**
  - use in syntax, 151, 172
  - use in text, 152
- <data type reference area>, **152**
  - use in syntax, 41, 45, 74, 113
  - use in text, 152
- <data type reference>, **151**
  - use in syntax, 40, 41, 75, 77, 85, 87, 112, 113, 193
- <data type specialization>, **154**
  - use in syntax, 151, 172
  - use in text, 154, 155, 160, 171
- <decimal digit>, **25**
  - use in syntax, 25
- <decision area>, **132**
  - use in syntax, 119
  - use in text, 132
- <decision body>, **131**
  - use in syntax, 131, 194
  - use in text, 132, 133, 137, 194
- <decision statement body>, **137**
  - use in syntax, 137
- <decision statement>, **137**
  - use in syntax, 133
  - use in text, 137
- <decision symbol>, **132**
  - use in syntax, 132
  - use in text, 132
- <decision>, **131**
  - use in syntax, 119
  - use in text, 34, 119, 132, 133, 137, 139
- <default initialization>, **187**
  - use in syntax, 151, 172
  - use in text, 185, 187, 188
- <definition selection list>, **40**
  - use in syntax, 40
  - use in text, 33, 42
- <definition selection>, **40**
  - use in syntax, 40
  - use in text, 33, 41, 42
- <definition>, **43**
  - use in syntax, 42
  - use in text, 43
- <delaying channel symbol 1>, **90**
  - use in syntax, 90
- <delaying channel symbol 2>, **90**
  - use in syntax, 90
- <dependency symbol>, **42**
  - use in syntax, 42, 78
  - use in text, 35
- <destination>, **129**
  - use in syntax, 94
  - use in text, 94, 97, 98, 100, 129, 130, 131
- <diagram in package>, **41**
  - use in syntax, 41
- <diagram>, **43**
  - use in syntax, 42
  - use in text, 38, 43



- <dollar sign>, **27**
  - use in syntax, 27
- <drawing kind>, **36**
  - use in syntax, 36
  - use in text, 36
- <else part>, **131**
  - use in syntax, 131
  - use in text, 132, 133, 137, 195
- <empty statement>, **140**
  - use in syntax, 133
  - use in text, 140
- <enabling condition area>, **107**
  - use in syntax, 105, 109
  - use in text, 191
- <enabling condition symbol>, **107**
  - use in syntax, 107
- <enabling condition>, **107**
  - use in syntax, 104, 109
  - use in text, 191
- <end>, **37**
  - use in syntax, 29, 30, 39, 40, 46, 47, 48, 49, 51, 52, 53, 54, 58, 59, 61, 72, 75, 81, 82, 83, 85, 86, 89, 91, 92, 93, 94, 97, 100, 101, 103, 104, 106, 107, 108, 109, 110, 112, 114, 115, 116, 117, 119, 131, 134, 135, 136, 138, 139, 140, 141, 142, 143, 144, 147, 151, 152, 153, 156, 161, 162, 164, 166, 172, 175, 184, 187, 193, 194, 231, 235
  - use in text, 37, 125, 136, 163
- <endpoint constraint>, **60**
  - use in syntax, 60
  - use in text, 60
- <entity in agent>, **75**
  - use in syntax, 44, 45, 75
- <entity in composite state>, **112**
  - use in syntax, 48, 49, 112, 114
- <entity in data type>, **151**
  - use in syntax, 151
- <entity in interface>, **153**
  - use in syntax, 152
  - use in text, 153
- <entity in operation>, **166**
  - use in syntax, 166
- <entity in package>, **40**
  - use in syntax, 39
- <entity in procedure>, **85**
  - use in syntax, 85
- <equality expression>, **180**
  - use in syntax, 176
  - use in text, 180
- <equals sign>, **27**
  - use in syntax, 25, 26, 27, 93, 154, 161, 172, 174, 175, 180
- <equation>, **231**
  - use in syntax, 231, 235
  - use in text, 230, 235, 236
- <error term>, **234**
  - use in syntax, 231
  - use in text, 235
- <exception constraint>, **65**
  - use in syntax, 65
- <exception context parameter>, **65**
  - use in syntax, 61
  - use in text, 153
- <exception definition item>, **142**
  - use in syntax, 142
  - use in text, 56
- <exception definition>, **142**
  - use in syntax, 40, 41, 75, 77, 85, 87, 112, 151, 153, 166, 167, 193
- <exception handler area>, **143**
  - use in syntax, 45, 77, 87, 113, 144, 167
  - use in text, 144
- <exception handler body area>, **144**
  - use in syntax, 143
- <exception handler list>, **143**
  - use in syntax, 143
  - use in text, 143, 144
- <exception handler symbol>, **143**
  - use in syntax, 143, 144
  - use in text, 144
- <exception handler>, **143**
  - use in syntax, 45, 75, 86, 112, 166
  - use in text, 121, 122, 143, 144, 146
- <exception property>, **56**
  - use in syntax, 55
  - use in text, 56
- <exception raise>, **124**
  - use in syntax, 104, 124
- <exception statement>, **140**
  - use in syntax, 133
  - use in text, 140
- <exception stimulus list>, **147**
  - use in syntax, 140, 147
  - use in text, 140, 143, 147, 148
- <exception stimulus>, **147**
  - use in syntax, 147
  - use in text, 147, 148
- <exclamation mark>, **27**
  - use in syntax, 27, 187
  - use in text, 187
- <exit transition area>, **118**
  - use in syntax, 118
- <exit transition>, **117**
  - use in syntax, 117
- <expanded sort>, **150**
  - use in syntax, 150
  - use in text, 151, 155
- <export>, **97**
  - use in syntax, 119, 135
  - use in text, 98, 100
- <exported as>, **184**
  - use in syntax, 184
  - use in text, 185
- <exported>, **55**
  - use in syntax, 54, 56
  - use in text, 55, 56
- <expression list>, **177**
  - use in syntax, 141, 190
  - use in text, 190
- <expression statement>, **136**
  - use in syntax, 133
- <expression>, **176**
  - Boolean
    - use in syntax, 107, 136, 138, 181
    - use in text, 136, 137, 138, 139, 191

Boolean  
   use in text, 108  
 constant  
   use in syntax, 177, 231  
 pid  
   use in syntax, 129  
   use in text, 94, 97, 98, 100, 129, 130  
 Time  
   use in syntax, 141  
   use in text, 141, 142  
 use in syntax, 123, 126, 132, 134, 137, 138, 177, 181, 186  
 use in text, 34, 88, 118, 123, 124, 126, 127, 128, 130, 135, 137, 138, 139, 177, 178, 182, 183, 186, 187, 191, 194, 230  
 <extended primary>, **179**  
   use in syntax, 177  
   use in text, 128, 177, 182  
 <extended variable>, **187**  
   use in syntax, 186  
   use in text, 186  
 <external channel identifiers>, **91**  
   use in syntax, 82, 83, 91  
   use in text, 47, 48, 49, 82, 84, 91, 92  
 <external operation definition>, **166**  
   use in syntax, 166  
   use in text, 70, 166, 168, 192  
 <external procedure definition>, **86**  
   use in syntax, 85  
   use in text, 86, 192  
 <external synonym definition item>, **175**  
   use in syntax, 175  
   use in text, 175, 192  
 <extra heading>, **36**  
   use in syntax, 36  
   use in text, 36  
 <field default initialization>, **162**  
   use in syntax, 162  
   use in text, 164  
 <field list>, **162**  
   use in syntax, 162  
   use in text, 163  
 <field name>, **179**  
   use in syntax, 179, 187  
   use in text, 179, 187  
 <field primary>, **179**  
   use in syntax, 179  
   use in text, 179  
 <field property>, **55**  
   use in syntax, 54  
   use in text, 55  
 <field sort>, **162**  
   use in syntax, 162, 164  
   use in text, 163, 165  
 <field variable>, **187**  
   use in syntax, 187  
   use in text, 105, 148, 187  
 <field>, **162**  
   use in syntax, 162  
   use in text, 55, 163, 164  
 <fields of sort>, **162**  
   use in syntax, 162  
   use in text, 163, 164, 171  
 <finalization statement>, **138**  
   use in syntax, 138  
   use in text, 137, 138, 139  
 <flow line symbol>, **122**  
   use in syntax, 122, 132  
   use in text, 35, 110, 122, 132, 194  
 <formal context parameter list>, **61**  
   use in syntax, 61  
   use in text, 57  
 <formal context parameter>, **61**  
   use in syntax, 61  
   use in text, 32, 46, 51, 62, 86, 93, 152  
 <formal context parameters>, **61**  
   use in syntax, 45, 49, 57, 85, 92, 151, 152  
   use in text, 46, 57, 61, 62, 93, 152, 153  
 <formal name>, **29**  
   use in syntax, 30  
   use in text, 31  
 <formal operation parameters>, **166**  
   use in syntax, 166  
   use in text, 166, 167, 168, 254  
 <formal parameter>, **156**  
   use in syntax, 63, 65, 86, 156  
   use in text, 55, 56  
 <formal variable parameters>, **85**  
   use in syntax, 85  
   use in text, 32  
 <frame symbol>, **77**  
   use in syntax, 36, 38, 41, 46, 47, 48, 49, 81, 82, 83, 86, 113, 115, 117, 167  
   use in text, 36, 42, 45, 49, 77, 82, 84, 87, 113, 117, 167  
 <free action>, **110**  
   use in syntax, 45, 75, 86, 112, 166  
   use in text, 110, 111  
 <full stop>, **27**  
   use in syntax, 25, 26, 27, 179, 182, 187  
   use in text, 179, 182, 183, 187  
 <gate constraint>, **59**  
   use in syntax, 59, 66  
   use in text, 59, 60, 93  
 <gate context parameter>, **66**  
   use in syntax, 61  
 <gate in definition>, **59**  
   use in syntax, 44, 45, 48, 49, 75, 112  
 <gate on diagram>, **60**  
   use in syntax, 47, 48, 49, 82, 83, 90, 113, 115  
   use in text, 45, 47, 48, 49, 50, 60, 78, 81, 92  
 <gate property area>, **60**  
   use in syntax, 45, 49, 52, 53, 82, 84, 90  
   use in text, 45, 50, 60, 78  
 <gate symbol 1>, **60**  
   use in syntax, 60  
 <gate symbol 2>, **60**  
   use in syntax, 60  
 <gate symbol>, **60**  
   use in syntax, 60  
   use in text, 60  
 <gate>, **59**  
   use in syntax, 52, 53, 59, 60, 66, 82, 84, 89, 90  
   use in text, 52, 53, 59, 78, 82, 84, 89, 90  
 <general text character>, **26**  
   use in syntax, 25, 26

<graphical answer part>, **132**  
   use in syntax, 132  
   use in text, 195

<graphical answer>, **132**  
   use in syntax, 132  
   use in text, 132, 194

<graphical decision body>, **132**  
   use in syntax, 132, 194  
   use in text, 194

<graphical else part>, **132**  
   use in syntax, 132  
   use in text, 132, 195

<graphical gate definition>, **60**  
   use in syntax, 60  
   use in text, 51, 60, 78

<graphical interface gate definition>, **60**  
   use in syntax, 60  
   use in text, 51, 60, 78, 129

<graphical package use area>, **42**  
   use in syntax, 42, 56, 77  
   use in text, 56, 78

<graphical point>, **115**  
   use in syntax, 115

<graphical state connection point>, **117**  
   use in syntax, 49, 113, 115  
   use in text, 117

<graphical system specification>, **38**  
   use in syntax, 38

<graphical task body>, **125**  
   use in syntax, 125  
   use in text, 125

<graphical type reference heading>, **57**  
   use in syntax, 57  
   use in text, 47, 48, 49, 57, 58, 87, 93, 152, 153

<graphical typebased agent definition>, **51**  
   use in syntax, 38, 77  
   use in text, 51

<graphical typebased block definition>, **52**  
   use in syntax, 51  
   use in text, 39, 52

<graphical typebased process definition>, **53**  
   use in syntax, 51  
   use in text, 39, 53

<graphical typebased state partition definition>, **115**  
   use in syntax, 115

<graphical typebased system definition>, **52**  
   use in syntax, 51  
   use in text, 52

<grave accent>, **27**  
   use in syntax, 27

<greater than or equals sign>, **26**  
   use in syntax, 25, 26, 174, 176  
   use in text, 174, 175

<greater than sign>, **27**  
   use in syntax, 25, 26, 27, 61, 174, 176

<handle area>, **147**  
   use in syntax, 144  
   use in text, 147, 148

<handle association area>, **144**  
   use in syntax, 144

<handle statement>, **140**  
   use in syntax, 140  
   use in text, 140

<handle symbol>, **147**  
   use in syntax, 147  
   use in text, 35, 147

<handle>, **147**  
   use in syntax, 143  
   use in text, 121, 140, 143, 147, 148

<heading area>, **36**  
   use in syntax, 36  
   use in text, 36

<heading>, **36**  
   use in syntax, 36  
   use in text, 36

<hex string>, **25**  
   use in syntax, 25, 31  
   use in text, 22, 161, 169

<history dash nextstate>, **121**  
   use in syntax, 121  
   use in text, 121

<history dash sign>, **26**  
   use in syntax, 26, 121

<hyphen>, **27**  
   use in syntax, 25, 26, 27, 54, 55, 121, 176

<iconized type reference area>, **58**  
   use in syntax, 56  
   use in text, 56, 58

<identifier>, **31**

- agent
  - use in syntax, 62, 89, 126, 129
  - use in text, 127, 129, 131, 190
- agent type
  - use in syntax, 63, 72, 126
  - use in text, 62, 63, 127, 190
- block
  - use in syntax, 52
- block type
  - use in syntax, 47
- channel
  - use in syntax, 91, 129
  - use in text, 81, 91, 92
- composite state type
  - use in syntax, 49, 66
  - use in text, 66
- data type
  - use in syntax, 72, 151
- drawing
  - use in syntax, 36
- exception
  - use in syntax, 85, 124, 147
  - use in text, 94, 104, 143, 147
- gate
  - use in syntax, 129
  - use in text, 129
- interface
  - use in syntax, 59, 60, 66, 72, 93, 153
  - use in text, 60, 66, 78, 93, 153
- literal
  - use in text, 235
- package
  - use in syntax, 40, 41
  - use in text, 40, 41
- procedure
  - use in syntax, 63, 86, 128
  - use in text, 63, 94, 128, 138, 191, 192

process  
   use in syntax, 53  
   use in text, 127  
 process type  
   use in syntax, 47  
 remote procedure  
   use in syntax, 85, 93, 94  
   use in text, 86, 93, 94, 104, 153  
 remote variable  
   use in syntax, 93, 97, 184  
   use in text, 93, 97, 98, 104, 153, 185  
 signal  
   use in syntax, 64, 93, 129  
   use in text, 64, 93, 105, 106, 109, 129, 131, 153  
 signal list  
   use in syntax, 93  
   use in text, 42, 93, 104, 105, 153  
 sort  
   use in syntax, 150, 184  
   use in text, 63, 65, 151, 157, 184  
 state  
   use in syntax, 89  
 state partition  
   use in syntax, 115  
 synonym  
   use in syntax, 179  
   use in text, 65, 167, 179  
 syntype  
   use in syntax, 172  
   use in text, 63, 172, 173  
 system type  
   use in syntax, 46  
 timer  
   use in syntax, 93, 94, 141, 190  
   use in text, 93, 94, 104, 105, 129, 141  
 timer  
   use in text, 94  
 use in syntax, 50, 59, 61, 68  
 use in text, 33, 34, 41, 50, 59, 68, 93, 94, 166, 167,  
   179, 231, 254  
 value  
   use in syntax, 235  
   use in text, 231, 234, 235, 236  
 variable  
   use in syntax, 97, 138, 185, 186  
   use in text, 98, 105, 139, 148, 186, 191  
 <if statement>, **136**  
   use in syntax, 133  
   use in text, 137  
 <imperative expression>, **188**  
   use in syntax, 177  
   use in text, 107, 108, 166, 167  
 <implicit text symbol>  
   use in syntax, 36  
   use in text, 36  
 <implies sign>, **26**  
   use in syntax, 25, 26, 176  
 <import expression>, **97**  
   use in syntax, 188  
   use in text, 97, 98, 99, 100, 108, 120, 146, 189  
 <in connector area>, **110**  
   use in syntax, 45, 77, 87, 113, 167  
   use in text, 110, 122  
 <in connector symbol>, **110**  
   use in syntax, 110, 122  
 <indexed primary>, **179**  
   use in syntax, 179  
   use in text, 179  
 <indexed variable>, **187**  
   use in syntax, 187  
   use in text, 105, 148, 187  
 <infix operation name>, **25**  
   use in syntax, 25  
   use in text, 178  
 <informal text>, **35**  
   use in syntax, 125, 131, 132, 194  
   use in text, 29, 132, 192, 194  
 <inherited agent definition>, **51**  
   use in syntax, 77  
   use in text, 51  
 <inherited block definition>, **52**  
   use in syntax, 51  
   use in text, 52  
 <inherited gate symbol 1>, **60**  
   use in syntax, 60  
 <inherited gate symbol 2>, **60**  
   use in syntax, 60  
 <inherited gate symbol>, **60**  
   use in syntax, 60  
   use in text, 60, 68  
 <inherited process definition>, **53**  
   use in syntax, 51  
   use in text, 53  
 <inherited state partition definition>, **115**  
   use in syntax, 115  
 <initial number>, **76**  
   use in syntax, 76  
   use in text, 76  
 <inner entry point>, **115**  
   use in syntax, 115  
 <input area>, **105**  
   use in syntax, 102  
   use in text, 105, 106  
 <input association area>, **102**  
   use in syntax, 102, 103  
 <input list>, **104**  
   use in syntax, 104, 105  
   use in text, 104, 105, 106, 107, 109  
 <input part>, **104**  
   use in syntax, 101, 103  
   use in text, 95, 96, 97, 99, 101, 104, 105, 106, 109  
 <input symbol>, **105**  
   use in syntax, 105, 109  
   use in text, 35, 105  
 <interaction area>, **77**  
   use in syntax, 45, 76  
   use in text, 78  
 <interface closing>, **152**  
   use in syntax, 152  
 <interface constraint>, **66**  
   use in syntax, 66  
 <interface context parameter>, **66**  
   use in syntax, 61  
 <interface definition>, **152**  
   use in syntax, 150  
   use in text, 32, 33, 34, 56, 57, 150, 151, 153, 160

- <interface heading>, **152**
  - use in syntax, 152
- <interface procedure definition>, **153**
  - use in syntax, 153
  - use in text, 56
- <interface reference area>, **153**
  - use in syntax, 41, 74
  - use in text, 153
- <interface reference>, **153**
  - use in syntax, 40, 41, 75, 77
- <interface specialization>, **154**
  - use in syntax, 152
  - use in text, 153, 154, 160
- <interface use list>, **153**
  - use in syntax, 55, 152
  - use in text, 56, 152, 153
- <interface variable definition>, **153**
  - use in syntax, 153
  - use in text, 55
- <interface variable property>, **55**
  - use in syntax, 54
  - use in text, 55
- <internal input symbol>, **105**
  - use in syntax, 105
  - use in text, 35, 105
- <internal output symbol>, **130**
  - use in syntax, 130
  - use in text, 35, 130
- <internal synonym definition item>, **175**
  - use in syntax, 175
  - use in text, 175
- <is assigned sign>, **26**
  - use in syntax, 26, 134, 138, 141, 184, 186
  - use in text, 187
- <join>, **122**
  - use in syntax, 119
  - use in text, 111, 122, 133, 139, 195
- <kernel heading>, **36**
  - use in syntax, 36
- <keyword>
  - use in syntax, 25
  - use in text, 22, 29
- <label>, **110**
  - use in syntax, 119, 139
  - use in text, 110, 111, 139, 140
- <labelled statement>, **139**
  - use in syntax, 133
  - use in text, 139, 140
- <left curly bracket>, **27**
  - use in syntax, 27, 85, 134, 137, 151, 152, 166, 172
  - use in text, 22
- <left parenthesis>, **27**
  - use in syntax, 26, 27
- <left square bracket>, **27**
  - use in syntax, 27, 179, 187
  - use in text, 22
- <less than or equals sign>, **26**
  - use in syntax, 25, 26, 174, 176
  - use in text, 174, 175
- <less than sign>, **27**
  - use in syntax, 25, 26, 27, 61, 174, 176
- <letter>, **25**
  - use in syntax, 25
- use in text, 29
- <lexical unit>, **25**
  - use in syntax, 30, 36
  - use in text, 28, 29, 30
- <linked type identifier>, **72**
  - use in syntax, 72
  - use in text, 73
- <linked type reference area>, **74**
  - use in syntax, 73
  - use in text, 74
- <literal equation>, **235**
  - use in syntax, 231
  - use in text, 234, 235, 236
- <literal identifier>, **178**
  - use in syntax, 178
  - use in text, 54, 167, 178, 231, 235
- <literal list>, **161, 235**
  - use in syntax, 160
  - use in text, 154, 159, 161, 162, 164, 171, 255
- <literal name>, **161**
  - base type
    - use in syntax, 154
    - use in text, 154
  - use in syntax, 36, 154, 161, 178
  - use in text, 154, 161, 162, 168
- <literal quantification>, **235**
  - use in syntax, 235
  - use in text, 235, 236
- <literal signature>, **161**
  - use in syntax, 65, 161, 235
  - use in text, 65, 161, 162, 168, 171
- <literal>, **178**
  - use in syntax, 177
  - use in text, 178
- <local variables of sort>, **134**
  - use in syntax, 134
  - use in text, 135
- <local>, **54**
  - use in syntax, 54, 56
  - use in text, 55, 56
- <loop body statement>, **138**
  - use in syntax, 138
  - use in text, 137, 138, 139
- <loop break statement>, **138**
  - use in syntax, 134
  - use in text, 134, 139
- <loop clause>, **138**
  - use in syntax, 138
  - use in text, 137, 138, 139
- <loop continue statement>, **138**
  - use in syntax, 134
  - use in text, 134, 139
- <loop statement>, **138**
  - use in syntax, 133
  - use in text, 134, 137, 138, 139
- <loop step>, **138**
  - use in syntax, 138
  - use in text, 137, 138, 139
- <loop variable definition>, **138**
  - use in syntax, 138
  - use in text, 138, 139

- <loop variable indication>, **138**
  - use in syntax, 138
  - use in text, 137, 138, 139
- <lowercase keyword>, **28**
  - use in text, 29, 254
- <lowercase letter>, **25**
  - use in syntax, 25
- <macro actual parameter>, **30**
  - use in syntax, 30
  - use in text, 30, 31
- <macro body>, **30**
  - use in syntax, 29
  - use in text, 29, 30
- <macro call body>, **30**
  - use in syntax, 30
- <macro call>, **30**
  - use in syntax, 196
  - use in text, 30, 31
- <macro definition>, **29**
  - use in syntax, 40, 41, 43, 75, 77, 85, 87, 112, 113, 166, 167, 193
  - use in text, 30, 43
- <macro formal parameter>, **30**
  - use in syntax, 30
  - use in text, 30, 31, 33, 254
- <macro formal parameters>, **30**
  - use in syntax, 29
- <macro parameter>, **30**
  - use in syntax, 29
- <maximum number>, **76**
  - use in syntax, 76
  - use in text, 76
- <merge area>, **122**
  - use in syntax, 119
  - use in text, 92, 122
- <merge symbol>, **122**
  - use in syntax, 122
- <method application>, **182**
  - use in syntax, 182
  - use in text, 182
- <method list>, **156**
  - use in syntax, 156
  - use in text, 157, 158, 163, 165
- <monadic operation name>, **25**
  - use in syntax, 25
  - use in text, 178
- <multiplicity>, **72**
  - use in syntax, 72, 73
  - use in text, 72
- <name class literal>, **169**
  - use in syntax, 161
  - use in text, 168, 169, 170
- <name class operation>, **169**
  - use in syntax, 65, 156
  - use in text, 168, 169, 170
- <name>, **25**
  - agent
    - use in syntax, 63
  - agent type
    - use in syntax, 62
  - association
    - use in syntax, 72, 73
  - block
    - use in syntax, 52, 82
- block type
  - use in syntax, 47
  - use in text, 47
- channel
  - use in syntax, 89, 90
  - use in text, 89, 91
- composite state
  - use in syntax, 112, 114
  - use in text, 111
- composite state type
  - use in syntax, 48, 49, 66
  - use in text, 49
- connector
  - use in syntax, 110, 122, 139
  - use in text, 33, 110, 122, 139, 140
- data type
  - use in syntax, 151
  - use in text, 150, 152, 164
- drawing
  - use in syntax, 36
- exception
  - use in syntax, 56, 65, 142, 234
  - use in text, 235
- exception handler
  - use in syntax, 143, 144
  - use in text, 143, 144, 146
- exception handler
  - use in text, 144
- field
  - use in syntax, 55, 162, 164
  - use in text, 55, 163, 164, 165
- gate
  - use in syntax, 59
  - use in text, 33, 60, 78
- interface
  - use in syntax, 66, 152
  - use in text, 150, 153
- literal
  - use in syntax, 161
  - use in text, 164
- macro
  - use in syntax, 29, 30
  - use in text, 30, 33
- Natural literal*
  - use in syntax, 36, 107, 169
  - use in text, 169
- operation
  - use in syntax, 55, 156, 169, 170, 235
  - use in text, 170, 236
- package
  - use in syntax, 39, 40
  - use in text, 42
- procedure
  - use in syntax, 56, 63, 85, 86
  - use in text, 87
- process
  - use in syntax, 53, 83, 84
- process type
  - use in syntax, 47
  - use in text, 48
- remote procedure
  - use in syntax, 94, 153

- remote variable
  - use in syntax, 55, 64, 97, 153
- role
  - use in syntax, 72, 73
  - use in text, 72, 74
- signal
  - use in syntax, 56, 64, 92
  - use in text, 93, 95, 98
- signal list
  - use in syntax, 93
- sort
  - use in syntax, 65
  - use in text, 171, 178, 182
- state
  - use in syntax, 53, 101, 103, 115, 121
  - use in text, 33, 101, 102, 103, 104, 114, 122, 144
- state entry point
  - use in syntax, 100, 117, 121
  - use in text, 100, 121
- state exit point
  - use in syntax, 117
  - use in text, 113
- synonym
  - use in syntax, 64, 175
- syntype
  - use in syntax, 172
- system
  - use in syntax, 51, 81
- system type
  - use in syntax, 46
  - use in text, 47
- timer
  - use in syntax, 56, 64, 141
- use in syntax, 25, 29, 30, 31, 32, 40, 57, 179
- use in text, 22, 29, 30, 32, 33, 34, 35, 41, 42, 43, 58, 98, 178, 182, 254
- value
  - use in syntax, 231, 235
  - use in text, 235
- variable
  - use in syntax, 54, 64, 76, 85, 134, 138, 166, 184
  - use in text, 55, 86, 88, 112, 135, 136, 139, 167, 168
- <named number>, **161**
  - use in syntax, 161
  - use in text, 65, 161
- <nextstate area>, **121**
  - use in syntax, 119
  - use in text, 35, 104
- <nextstate body>, **121**
  - use in syntax, 121
- <nextstate>, **121**
  - use in syntax, 119
  - use in text, 109, 121
- <noequality>, **233**
  - use in syntax, 231
  - use in text, 233
- <nondelaying channel symbol 1>, **90**
  - use in syntax, 90
  - use in text, 90
- <nondelaying channel symbol 2>, **90**
  - use in syntax, 90
  - use in text, 90
- <not asterisk or solidus>, **26**
  - use in syntax, 26
- <not equals sign>, **26**
  - use in syntax, 25, 26, 174, 180
  - use in text, 180
- <note text>, **26**
  - use in syntax, 26
- <note>, **26**
  - use in syntax, 25
  - use in text, 28, 29, 37
- <now expression>, **188**
  - use in syntax, 188
- <number of instances>, **76**
  - use in syntax, 52, 53, 75, 82, 83, 84
  - use in text, 76, 81
- <number of pages>, **36**
  - use in syntax, 36
- <number sign>, **27**
  - use in syntax, 27, 55
- <on exception area>, **144**
  - use in syntax, 144
  - use in text, 144
- <on exception association area>, **144**
  - use in syntax, 45, 77, 87, 95, 100, 102, 103, 105, 106, 107, 109, 113, 118, 123, 125, 126, 128, 130, 132, 143, 147, 167
  - use in text, 87, 145, 146
- <on exception>, **144**
  - use in syntax, 45, 75, 86, 100, 101, 103, 104, 106, 109, 112, 117, 119, 131, 143, 147, 166
  - use in text, 99, 121, 122, 145, 146
- <open range>, **174**
  - use in syntax, 173
  - use in text, 174
- <operand>, **176**
  - use in syntax, 176
  - use in text, 177
- <operand0>, **176**
  - use in syntax, 176
- <operand1>, **176**
  - use in syntax, 176
  - use in text, 177
- <operand2>, **176**
  - use in syntax, 176, 180, 184
  - use in text, 177, 180, 184
- <operand3>, **176**
  - use in syntax, 176, 180
  - use in text, 177, 180
- <operand4>, **176**
  - use in syntax, 176
  - use in text, 177
- <operand5>, **176**
  - use in syntax, 176
  - use in text, 177
- <operation application>, **182**
  - use in syntax, 136, 177
  - use in text, 87, 136, 169, 180, 182, 183, 230
- <operation body>, **166**
  - use in syntax, 166
  - use in text, 110, 166, 167, 168
- <operation definition>, **166**
  - use in syntax, 43, 166
  - use in text, 32, 55, 70, 133, 136, 166, 167, 168, 170

- <operation definitions>, **166**
  - use in syntax, 151, 231
  - use in text, 170, 171
- <operation diagram>, **167**
  - use in syntax, 43
  - use in text, 32, 36, 167, 168, 170, 171
- <operation graph area>, **167**
  - use in syntax, 167
  - use in text, 110
- <operation heading>, **166**
  - use in syntax, 166, 167
  - use in text, 166, 168
- <operation identifier>, **166**
  - use in syntax, 182
  - use in text, 54, 167, 180, 182, 183, 231
- <operation name>, **156**
  - base type
    - use in syntax, 154
    - use in text, 154, 155
  - use in syntax, 65, 154, 156, 166
  - use in text, 70, 154, 156, 157, 167, 168, 169, 170, 171, 182, 183
- <operation parameters>, **166**
  - use in syntax, 166
  - use in text, 167
- <operation preamble>, **156**
  - use in syntax, 156, 166
- <operation property>, **55**
  - use in syntax, 55
  - use in text, 55
- <operation result>, **166**
  - use in syntax, 166
  - use in text, 166, 167, 168
- <operation signature in constraint>, **65**
  - use in syntax, 65
- <operation signature>, **156**
  - use in syntax, 156
  - use in text, 70, 71, 157, 158, 165, 166, 167, 168, 169, 170, 171, 174, 233, 236
- <operation signatures>, **156**
  - use in syntax, 151, 231
  - use in text, 171
- <operation text area>, **167**
  - use in syntax, 167
  - use in text, 37, 54
- <operations>, **151, 231**
  - use in syntax, 151
  - use in text, 152, 165, 188, 231
- <operator application>, **182**
  - use in syntax, 182
- <operator list>, **156**
  - use in syntax, 156
  - use in text, 157, 158, 162, 165, 169, 233
- <option area>, **193**
  - use in syntax, 41, 193
  - use in text, 78, 193, 194
- <option symbol>, **193**
  - use in syntax, 193
  - use in text, 193, 194
- <ordering area>, **73**
  - use in syntax, 73
- <other character>, **27**
  - use in syntax, 26
- <other special>, **27**
  - use in syntax, 26
- <out connector area>, **122**
  - use in syntax, 119
  - use in text, 110, 122
- <out connector symbol>, **122**
  - use in syntax, 122
  - use in text, 35
- <outer entry point>, **115**
  - use in syntax, 115
- <output area>, **130**
  - use in syntax, 119
  - use in text, 131
- <output body>, **129**
  - use in syntax, 129, 130
  - use in text, 131
- <output symbol>, **130**
  - use in syntax, 130
  - use in text, 35
- <output>, **129**
  - use in syntax, 119, 135
  - use in text, 46, 129, 131
- <package definition>, **39**
  - use in syntax, 38, 40, 43
  - use in text, 32, 42, 43
- <package diagram>, **41**
  - use in syntax, 38, 41, 42, 43
  - use in text, 32, 36
- <package heading>, **39**
  - use in syntax, 39, 41
- <package interface>, **40**
  - use in syntax, 39
  - use in text, 33, 42
- <package reference area>, **41**
  - use in syntax, 38, 41, 42
  - use in text, 42
- <package reference>, **40**
  - use in syntax, 40, 41
- <package symbol>, **42**
  - use in syntax, 41
  - use in text, 42
- <package text area>, **41**
  - use in syntax, 41
  - use in text, 37, 54
- <package use area>, **41**
  - use in syntax, 41, 46, 47, 48, 49, 81, 82, 83, 86, 113, 115, 167
  - use in text, 42, 45, 49, 77, 87, 113, 167
- <package use clause>, **40**
  - use in syntax, 39, 41, 42, 46, 47, 48, 49, 81, 82, 83, 85, 112, 114, 151, 166
  - use in text, 33, 34, 40, 41, 42, 56
- <package>, **38**
  - use in syntax, 38, 42
  - use in text, 33, 34, 38, 40, 41, 42, 43
- <page number area>, **36**
  - use in syntax, 36
  - use in text, 36
- <page number>, **36**
  - use in syntax, 36
- <page>, **36**
  - use in syntax, 196
  - use in text, 36



- <parameter kind>, **85**
  - use in syntax, 85, 156, 166
  - use in text, 51, 63, 71, 88, 157, 158, 168
- <parameters of sort>, **76**
  - use in syntax, 75, 85
- <parent sort identifier>, **172**
  - use in syntax, 172
  - use in text, 42, 188
- <partial regular expression>, **169**
  - use in syntax, 169
  - use in text, 169
- <path item>, **32**
  - use in syntax, 31
  - use in text, 33, 35, 40, 178, 182
- <percent sign>, **27**
  - use in syntax, 27
- <pid expression>, **189**
  - use in syntax, 188
  - use in text, 190
- <pid sort>, **150**
  - use in syntax, 150
  - use in text, 151, 155
- <plain input symbol>, **105**
  - use in syntax, 105
  - use in text, 105
- <plain output symbol>, **130**
  - use in syntax, 130
  - use in text, 130
- <plus sign>, **27**
  - use in syntax, 25, 27, 55, 169, 176
  - use in text, 22, 169
- <point>, **115**
  - use in syntax, 115
- <primary>, **177**
  - constant
    - use in syntax, 61
    - use in text, 61
  - use in syntax, 176, 179, 182
  - use in text, 61, 179
- <priority input area>, **106**
  - use in syntax, 106
  - use in text, 106
- <priority input association area>, **106**
  - use in syntax, 102, 103
  - use in text, 106
- <priority input list>, **106**
  - use in syntax, 106
  - use in text, 105, 109
- <priority input symbol>, **106**
  - use in syntax, 106
  - use in text, 35
- <priority input>, **106**
  - use in syntax, 101, 103
  - use in text, 106
- <priority name>, **107**
  - use in syntax, 106, 107
- <private>, **55**
  - use in syntax, 55
  - use in text, 55
- <procedure area>, **86**
  - use in syntax, 86, 113, 193
- <procedure body>, **86**
  - use in syntax, 85
    - use in text, 88, 110, 168
- <procedure call area>, **128**
  - use in syntax, 119
  - use in text, 128
- <procedure call body>, **128**
  - use in syntax, 128, 136, 138, 191
  - use in text, 136, 138, 139, 192
- <procedure call symbol>, **128**
  - use in syntax, 95, 128
- <procedure call>, **128**
  - use in syntax, 119
  - use in text, 87, 94, 128, 136, 139
- <procedure constraint>, **63**
  - use in syntax, 63
  - use in text, 86
- <procedure context parameter>, **63**
  - use in syntax, 61
- <procedure definition>, **85**
  - use in syntax, 40, 41, 43, 75, 77, 85, 87, 112, 193
  - use in text, 32, 56, 86, 88, 112, 133, 136, 168
- <procedure diagram>, **86**
  - use in syntax, 43, 45, 86
  - use in text, 32, 36, 168
- <procedure formal parameters>, **85**
  - use in syntax, 85
  - use in text, 56, 168, 254
- <procedure graph area>, **87**
  - use in syntax, 86
  - use in text, 87, 110
- <procedure heading>, **85**
  - use in syntax, 85, 86
  - use in text, 88
- <procedure preamble>, **85**
  - use in syntax, 85
  - use in text, 68, 87, 88
- <procedure property>, **56**
  - use in syntax, 55
  - use in text, 56
- <procedure reference area>, **87**
  - use in syntax, 41, 45, 86
  - use in text, 87
- <procedure reference>, **86**
  - use in syntax, 40, 41, 75, 77, 85, 87, 113, 193
- <procedure result>, **85**
  - use in syntax, 85
  - use in text, 56, 68, 88, 96, 123, 191, 192
- <procedure signature in constraint>, **63**
  - use in syntax, 63
- <procedure signature>, **86**
  - use in syntax, 55, 56, 86, 94, 153
  - use in text, 55, 56, 173, 254
- <procedure start area>, **87**
  - use in syntax, 87, 167
  - use in text, 88
- <procedure start symbol>, **87**
  - use in syntax, 87
- <procedure symbol>, **87**
  - use in syntax, 57, 58
- <procedure text area>, **87**
  - use in syntax, 86, 193
  - use in text, 37, 54

<process definition>, **83**  
   use in syntax, 75  
   use in text, 39, 53, 83  
 <process diagram>, **83**  
   use in syntax, 76  
   use in text, 39, 77, 83, 84  
 <process heading>, **83**  
   use in syntax, 83  
 <process reference area>, **84**  
   use in syntax, 77  
 <process reference>, **83**  
   use in syntax, 75  
 <process symbol>, **84**  
   use in syntax, 53, 60, 84  
   use in text, 51  
 <process type definition>, **47**  
   use in syntax, 44  
   use in text, 48  
 <process type diagram>, **48**  
   use in syntax, 45  
   use in text, 48  
 <process type heading>, **47**  
   use in syntax, 47, 48  
 <process type reference area>, **48**  
   use in syntax, 45  
   use in text, 48  
 <process type reference>, **47**  
   use in syntax, 45  
 <process type symbol>, **48**  
   use in syntax, 57, 58  
 <protected>, **55**  
   use in syntax, 55, 235  
   use in text, 55  
 <provided expression>, **107**  
   use in syntax, 107  
   use in text, 107, 108  
 <public>, **55**  
   use in syntax, 55  
   use in text, 55  
 <qualifier begin sign>, **26**  
   use in syntax, 26, 31  
   use in text, 254  
 <qualifier end sign>, **26**  
   use in syntax, 26, 31  
   use in text, 254  
 <qualifier>, **31**  
   use in syntax, 31, 39, 40, 46, 47, 48, 49, 57, 82, 83,  
     85, 112, 114, 166, 178, 179  
   use in text, 33, 34, 40, 41, 42, 43, 58, 178, 179, 180,  
     182, 254  
 <quantification>, **231**  
   use in syntax, 231  
   use in text, 231, 232  
 <quantified equations>, **231**  
   use in syntax, 231  
   use in text, 230, 231  
 <question mark>, **27**  
   use in syntax, 27  
 <question>, **132**  
   use in syntax, 131, 132  
   use in text, 34, 132  
 <quotation mark>, **27**  
   use in syntax, 25, 27  
   use in text, 178  
 <quoted operation name>, **25**  
   use in syntax, 25, 156  
   use in text, 22, 29, 32, 35, 157  
 <raise area>, **124**  
   use in syntax, 119  
 <raise body>, **124**  
   use in syntax, 124  
 <raise statement>, **136**  
   use in syntax, 134  
   use in text, 136  
 <raise symbol>, **124**  
   use in syntax, 124  
   use in text, 35  
 <raise>, **124**  
   use in syntax, 119, 136  
   use in text, 124  
 <raises>, **85**  
   use in syntax, 85, 86, 156, 166  
   use in text, 55, 56, 86, 96, 97, 167  
 <range check expression>, **184**  
   use in syntax, 176  
   use in text, 184  
 <range condition>, **173**  
   use in syntax, 72, 131, 137, 173, 174  
   use in text, 72, 133, 137, 174, 175  
 <range>, **173**  
   use in syntax, 173  
 <reference sort>, **150**  
   use in syntax, 150  
   use in text, 151, 155  
 <referenced definition>, **42**  
   use in syntax, 38  
   use in text, 33, 38, 43  
 <regular element>, **169**  
   use in syntax, 169  
   use in text, 169, 170  
 <regular expression>, **169**  
   use in syntax, 169  
   use in text, 161, 162, 169  
 <regular interval>, **169**  
   use in syntax, 169  
   use in text, 169, 170  
 <remote procedure call area>, **95**  
   use in syntax, 119  
 <remote procedure call body>, **94**  
   use in syntax, 94, 95, 191  
   use in text, 94, 177, 192  
 <remote procedure call>, **94**  
   use in syntax, 119  
   use in text, 94, 95, 120  
 <remote procedure context parameter>, **63**  
   use in syntax, 61  
   use in text, 153  
 <remote procedure definition>, **94**  
   use in syntax, 40, 41, 75, 77, 193  
   use in text, 63, 94, 95, 104  
 <remote procedure reject>, **104**  
   use in syntax, 104  
   use in text, 97, 104  
 <remote variable context parameter>, **64**  
   use in syntax, 61  
   use in text, 153

- <remote variable definition>, **97**
  - use in syntax, 40, 41, 75, 77, 193
  - use in text, 64, 97, 98
- <rename list>, **154**
  - use in syntax, 154
  - use in text, 154
- <rename pair>, **154**
  - use in syntax, 154
  - use in text, 155
- <renaming>, **154**
  - use in syntax, 154
  - use in text, 65, 165
- <reset clause>, **141**
  - use in syntax, 141
  - use in text, 141, 142
- <reset>, **141**
  - use in syntax, 119, 135
  - use in text, 141, 142
- <restricted equation>, **233**
  - use in syntax, 233
- <restriction>, **233**
  - use in syntax, 233
  - use in text, 235
- <result sign>, **26**
  - use in syntax, 26, 85, 156, 166
- <result>, **156**
  - use in syntax, 63, 65, 86, 156
  - use in text, 55, 56, 157, 168
- <return area>, **123**
  - use in syntax, 119
  - use in text, 88, 123
- <return statement>, **135**
  - use in syntax, 134
  - use in text, 136
- <return symbol>, **123**
  - use in syntax, 123
- <return>, **123**
  - use in syntax, 119, 135
  - use in text, 88, 113, 117, 118, 123, 135, 254
- <reverse solidus>, **27**
  - use in syntax, 27
- <right curly bracket>, **27**
  - use in syntax, 27, 85, 134, 137, 151, 152, 166, 172
  - use in text, 22
- <right parenthesis>, **27**
  - use in syntax, 26, 27
- <right square bracket>, **27**
  - use in syntax, 27, 179, 187
  - use in text, 22
- <save area>, **108**
  - use in syntax, 102
  - use in text, 109
- <save association area>, **102**
  - use in syntax, 102, 103
- <save list>, **108**
  - use in syntax, 108
  - use in text, 106, 108, 109
- <save part>, **108**
  - use in syntax, 101, 103
  - use in text, 95, 97, 101, 109
- <save symbol>, **108**
  - use in syntax, 108
- <scope unit kind>, **32**
  - use in syntax, 32
  - use in text, 33, 35
- <sdl specification>, **38**
  - use in syntax, 196
  - use in text, 18, 29, 30, 38, 40, 41, 81, 157
- <select definition>, **193**
  - use in syntax, 40, 41, 75, 77, 85, 87, 112, 113, 166, 167, 193
  - use in text, 193, 194
- <selected entity kind>, **40**
  - use in syntax, 40
  - use in text, 41, 42
- <semicolon>, **27**
  - use in syntax, 27, 37
- <set clause>, **141**
  - use in syntax, 141
  - use in text, 141, 142
- <set>, **141**
  - use in syntax, 119, 135
  - use in text, 141, 142
- <signal constraint>, **64**
  - use in syntax, 64
- <signal context parameter>, **64**
  - use in syntax, 61
  - use in text, 32, 153
- <signal definition item>, **92**
  - use in syntax, 92
  - use in text, 55, 56
- <signal definition>, **92**
  - use in syntax, 40, 41, 75, 77, 153, 193
  - use in text, 32, 56, 95, 98
- <signal list area>, **94**
  - use in syntax, 60, 90
  - use in text, 60, 90
- <signal list definition>, **93**
  - use in syntax, 40, 41, 75, 77, 193
  - use in text, 93
- <signal list item>, **93**
  - use in syntax, 93, 104
  - use in text, 45, 50, 56, 93, 104, 105
- <signal list symbol>, **94**
  - use in syntax, 94
- <signal list>, **93**
  - use in syntax, 59, 75, 89, 93, 94, 108, 153
  - use in text, 46, 59, 66, 90, 93, 95, 153
- <signal parameter property>, **55**
  - use in syntax, 54
  - use in text, 55
- <signal property>, **56**
  - use in syntax, 55
  - use in text, 56
- <signal reference area>, **93**
  - use in syntax, 41, 45
  - use in text, 93
- <signal reference>, **93**
  - use in syntax, 40, 41, 75, 77
- <signal signature>, **64**
  - use in syntax, 64
- <simple expression>, **177**
  - Boolean
    - use in syntax, 193
    - use in text, 193, 194

- Natural
  - use in syntax, 76, 161
  - use in text, 161, 162
- use in syntax, 194
- use in text, 177, 194
- <size constraint>, **174**
  - use in syntax, 173
  - use in text, 174, 175
- <solid association symbol>, **37**
  - use in syntax, 37, 102, 103, 105, 106, 107, 109, 115, 144
  - use in text, 35, 37, 102, 144
- <solid on exception association symbol>, **144**
  - use in syntax, 144
  - use in text, 35, 144
- <solidus>, **27**
  - use in syntax, 25, 26, 176
- <sort constraint>, **65**
  - use in syntax, 65
  - use in text, 65
- <sort context parameter>, **65**
  - use in syntax, 61
  - use in text, 32, 93, 152, 153
- <sort identifier>
  - use in text, 150
- <sort list>, **93**
  - use in syntax, 55, 56, 63, 64, 65, 66, 92, 141, 142
  - use in text, 55, 56
- <sort signature>, **65**
  - use in syntax, 65
- <sort>, **150**
  - predefined
    - use in syntax, 175
  - result
    - use in text, 158
  - use in syntax, 54, 55, 64, 65, 76, 85, 93, 97, 134, 138, 153, 156, 162, 166, 172, 175, 184, 191, 231, 235
  - use in text, 33, 42, 55, 62, 63, 65, 66, 88, 96, 139, 150, 151, 156, 158, 163, 167, 168, 172, 175, 184, 185, 188, 191, 192, 232, 235
- <space>, **28**
  - use in syntax, 26
  - use in text, 28, 29
- <special>, **26**
  - use in syntax, 25, 26
  - use in text, 22
- <specialization area>, **67**
  - use in syntax, 56
  - use in text, 56
- <specialization relation symbol>, **67**
  - use in syntax, 67
  - use in text, 35, 56, 67
- <specialization>, **67**
  - use in syntax, 49, 75, 85, 92
  - use in text, 51, 56, 62, 69, 93
- <specification area>, **38**
  - use in syntax, 38, 42
  - use in text, 36, 38
- <spelling term>, **170, 235**
  - use in syntax, 177
  - use in text, 170, 235, 236
- <spontaneous designator>, **109**
  - use in syntax, 109
- <spontaneous transition area>, **109**
  - use in syntax, 102
  - use in text, 110
- <spontaneous transition association area>, **102**
  - use in syntax, 102, 103
- <spontaneous transition>, **109**
  - use in syntax, 101, 103
  - use in text, 101, 110
- <start area>, **100**
  - use in syntax, 45, 77, 113
  - use in text, 100
- <start symbol>, **100**
  - use in syntax, 100
- <start>, **100**
  - use in syntax, 45, 75, 86, 112, 166
  - use in text, 88, 100, 112, 113, 114, 117, 121, 167
- <state aggregation area>, **115**
  - use in syntax, 111
- <state aggregation body area>, **115**
  - use in syntax, 49, 115
- <state aggregation body>, **114**
  - use in syntax, 49, 114
- <state aggregation heading>, **114**
  - use in syntax, 114, 115
- <state aggregation type heading>, **49**
  - use in syntax, 49
- <state aggregation>, **114**
  - use in syntax, 111
- <state area>, **101**
  - use in syntax, 45, 77, 87, 113, 119
  - use in text, 102, 104
- <state connection point symbol 1>, **117**
  - use in syntax, 117
  - use in text, 117
- <state connection point symbol 2>, **117**
  - use in syntax, 117
  - use in text, 117
- <state connection point symbol>, **117**
  - use in syntax, 115, 117
- <state connection points>, **116**
  - use in syntax, 48, 49, 112, 114
  - use in text, 112, 117
- <state entry point>, **117**
  - use in syntax, 115, 116
  - use in text, 117
- <state entry points>, **116**
  - use in syntax, 115, 116, 117
  - use in text, 117
- <state exit point>, **117**
  - use in syntax, 115, 117, 123
  - use in text, 117, 118, 123
- <state exit points>, **117**
  - use in syntax, 115, 116, 117
  - use in text, 117
- <state expression>, **191**
  - use in syntax, 188
  - use in text, 191, 254
- <state list token>, **103**
  - use in syntax, 103
  - use in text, 104

- <state list>, **101**
  - use in syntax, 101, 102
  - use in text, 101, 102, 103
- <state machine area>, **77**
  - use in syntax, 77, 90
  - use in text, 78, 90
- <state machine graph area>, **77**
  - use in syntax, 76
  - use in text, 110, 122, 194
- <state machine graph>, **75**
  - use in syntax, 75
- <state partition area>, **115**
  - use in syntax, 77, 115, 193
- <state partition connection>, **115**
  - use in syntax, 114
- <state partition reference area>, **115**
  - use in syntax, 115
- <state partitioning>, **115**
  - use in syntax, 44, 75, 114, 193
  - use in text, 75
- <state symbol>, **102**
  - use in syntax, 60, 102, 103, 115, 121
  - use in text, 102
- <state>, **101**
  - use in syntax, 45, 75, 86, 112
  - use in text, 86, 96, 97, 99, 101, 102, 103, 104, 106, 108, 109, 111, 112, 114, 121, 122
- <statement list>, **133**
  - use in syntax, 85, 125, 134, 166
  - use in text, 37, 88, 110, 125, 133, 134, 135, 139, 168
- <statement>, **133**
  - use in syntax, 133, 136, 137, 138, 139, 140
  - use in text, 134, 135, 137, 166, 167
- <statements>, **133**
  - use in syntax, 133
  - use in text, 134, 135, 139
- <stimulus>, **104**
  - use in syntax, 104, 106
  - use in text, 104, 105, 107, 108, 109
- <stop symbol>, **123**
  - use in syntax, 119
- <stop>, **123**
  - use in syntax, 119
  - use in text, 254
- <string name>, **31**
  - use in syntax, 161
  - use in text, 32, 170
- <structure definition>, **162**
  - use in syntax, 160
  - use in text, 154, 159, 163, 164, 171, 255
- <structure primary>, **179**
  - use in syntax, 179
  - use in text, 179, 180
- <symbol>, **36**
  - use in syntax, 36
  - use in text, 36
- <symbolic visibility>, **55**
  - use in syntax, 55, 73
- <synonym constraint>, **64**
  - use in syntax, 64
- <synonym context parameter>, **64**
  - use in syntax, 61
  - use in text, 152
- <synonym definition item>, **175**
  - use in syntax, 175
- <synonym definition>, **175**
  - use in syntax, 150, 151
  - use in text, 175, 179
- <synonym>, **179**
  - use in syntax, 177
  - use in text, 175, 179
- <syntype closing>, **172**
  - use in syntax, 172
- <syntype definition>, **172**
  - use in syntax, 150, 151
  - use in text, 150, 172, 173, 187, 188
- <syntype>, **172**
  - use in syntax, 150
  - use in text, 172
- <system definition>, **81**
  - use in syntax, 75
  - use in text, 39, 42, 52, 80, 81, 95
- <system diagram>, **81**
  - use in syntax, 76
  - use in text, 39, 77, 80, 81
- <system heading>, **81**
  - use in syntax, 81
- <system reference area>, **81**
  - use in syntax, 77
  - use in text, 81
- <system specification>, **38**
  - use in syntax, 38, 42
  - use in text, 38, 39, 41, 43
- <system type definition>, **46**
  - use in syntax, 44
  - use in text, 46, 47
- <system type diagram>, **46**
  - use in syntax, 45
  - use in text, 46
- <system type heading>, **46**
  - use in syntax, 46
- <system type reference area>, **46**
  - use in syntax, 45
  - use in text, 47
- <system type reference>, **46**
  - use in syntax, 45
- <system type symbol>, **46**
  - use in syntax, 57, 58
- <task area>, **125**
  - use in syntax, 119
  - use in text, 32, 33, 125, 133
- <task symbol>, **125**
  - use in syntax, 125
- <task>, **125**
  - use in syntax, 119
  - use in text, 33, 105, 125, 133, 136, 148
- <term>, **231**
  - Boolean
    - use in syntax, 234
    - use in text, 234
  - use in syntax, 231
  - use in text, 231
- <terminating statement>, **134**
  - use in syntax, 133
  - use in text, 134

- <terminator 1>, **119**
  - use in syntax, 119
- <terminator 2>, **119**
  - use in syntax, 119
- <terminator statement>, **119**
  - use in syntax, 119
  - use in text, 110, 119, 132, 133, 135, 195
- <text extension area>, **37**
  - use in syntax, 196
  - use in text, 28
- <text extension symbol>, **37**
  - use in syntax, 37
  - use in text, 35, 37
- <text symbol>, **38**
  - use in syntax, 41, 77, 87, 113, 167
  - use in text, 38, 86
- <text>, **26**
  - use in syntax, 37
  - use in text, 28, 37
- <textual endpoint constraint>, **59**
  - use in syntax, 59, 60
  - use in text, 59
- <textual gate definition>, **59**
  - use in syntax, 59
  - use in text, 32, 60
- <textual interface gate definition>, **59**
  - use in syntax, 59
  - use in text, 60, 129
- <textual operation reference>, **166**
  - use in syntax, 166
  - use in text, 70, 166, 167
- <textual system specification>, **38**
  - use in syntax, 38
- <textual task body>, **125**
  - use in syntax, 125
  - use in text, 105, 125, 148
- <textual typebased agent definition>, **51**
  - use in syntax, 38, 44, 75
  - use in text, 51, 76, 80, 89, 91
- <textual typebased block definition>, **52**
  - use in syntax, 51
  - use in text, 39, 52, 59
- <textual typebased process definition>, **53**
  - use in syntax, 51
  - use in text, 39, 53, 59
- <textual typebased state partition definition>, **53**
  - use in syntax, 115, 193
  - use in text, 89
- <textual typebased system definition>, **51**
  - use in syntax, 51
  - use in text, 52
- <tilde>, **27**
  - use in syntax, 27
- <timer active expression>, **190**
  - use in syntax, 188
- <timer constraint>, **64**
  - use in syntax, 64
- <timer context parameter>, **64**
  - use in syntax, 61
  - use in text, 46
- <timer default initialization>, **141**
  - use in syntax, 141
  - use in text, 141, 142
- <timer definition item>, **141**
  - use in syntax, 141
  - use in text, 56
- <timer definition>, **141**
  - use in syntax, 75, 77, 193
- <timer property>, **56**
  - use in syntax, 55
  - use in text, 56
- <transition area>, **119**
  - use in syntax, 87, 100, 105, 106, 107, 109, 110, 118, 132, 147
  - use in text, 120, 122, 147
- <transition option area>, **194**
  - use in syntax, 119
  - use in text, 195
- <transition option symbol>, **194**
  - use in syntax, 194
  - use in text, 194
- <transition option>, **194**
  - use in syntax, 119
  - use in text, 78, 119, 132, 194, 195
- <transition string area>, **119**
  - use in syntax, 119
  - use in text, 120
- <transition string>, **119**
  - use in syntax, 119
  - use in text, 110, 132, 133, 195
- <transition>, **119**
  - use in syntax, 100, 104, 106, 109, 110, 117, 131, 147
  - use in text, 95, 105, 109, 110, 119, 121, 132, 133, 135, 146, 148, 195
- <try statement>, **140**
  - use in syntax, 140
  - use in text, 140
- <type expression>, **50**
  - block
    - use in syntax, 52
    - use in text, 52
  - composite state
    - use in syntax, 53, 115
    - use in text, 53
  - data type
    - use in syntax, 154
    - use in text, 154
  - interface
    - use in syntax, 154
  - procedure
    - use in syntax, 128
  - process
    - use in syntax, 53
    - use in text, 53
  - sort
    - use in text, 155
  - system
    - use in syntax, 51
    - use in text, 52
  - use in syntax, 67
  - use in text, 50, 51, 56, 61, 62, 67, 86
- <type in agent area>, **45**
  - use in syntax, 41, 45, 76
- <type in composite state area>, **113**
  - use in syntax, 49, 113, 115

- <type preamble>, **45**
  - use in syntax, 47, 49, 57, 85, 86, 92, 93, 151, 172
  - use in text, 57
- <type reference area>, **56**
  - use in syntax, 46, 47, 48, 49, 67, 87, 93, 152, 153
  - use in text, 47, 48, 49, 55, 56, 67, 87, 93, 152, 153
- <type reference heading>, **57**
  - block type
    - use in syntax, 57, 58
  - composite state type
    - use in syntax, 57, 58
  - data type
    - use in syntax, 57
  - interface
    - use in syntax, 57
  - procedure
    - use in syntax, 57, 58
  - process type
    - use in syntax, 57, 58
  - signal
    - use in syntax, 57
  - system type
    - use in syntax, 57, 58
    - use in text, 58
- <type reference kind symbol>, **57**
  - use in syntax, 57
  - use in text, 57, 58
- <type reference properties>, **54**
  - use in syntax, 46, 47, 49, 86, 93, 151, 153
  - use in text, 54, 55
- <type state machine graph area>, **45**
  - use in syntax, 45
  - use in text, 110, 194
- <typebased block heading>, **52**
  - use in syntax, 52
- <typebased composite state>, **53**
  - use in syntax, 103
  - use in text, 53
- <typebased process heading>, **53**
  - use in syntax, 53
- <typebased state partition heading>, **115**
  - use in syntax, 53, 115
- <typebased system heading>, **51**
  - use in syntax, 51, 52
- <underline>, **27**
  - use in syntax, 25, 27
  - use in text, 29
- <unordered>, **235**
  - use in syntax, 235
  - use in text, 235
- <unquantified equation>, **231**
  - use in syntax, 231, 233
  - use in text, 231, 232
- <uppercase keyword>
  - use in text, 29, 254
- <uppercase letter>, **25**
  - use in syntax, 25
- use in syntax, 25
- <valid input signal set>, **75**
  - use in syntax, 44, 48, 49, 75, 77, 112, 113
  - use in text, 76, 79
- <value returning procedure call>, **191**
  - use in syntax, 176
  - use in text, 87, 128, 177, 191, 192
- <variable access>, **185**
  - use in syntax, 177
  - use in text, 128, 182, 185
- <variable context parameter>, **64**
  - use in syntax, 61
  - use in text, 46
- <variable definition statement>, **134**
  - use in syntax, 133
  - use in text, 135, 139
- <variable definition>, **184**
  - use in syntax, 75, 77, 85, 87, 112, 113, 166, 167, 193
  - use in text, 55, 86, 88, 98, 111, 112, 133, 134, 135, 136, 185, 188
- <variable definitions>, **133**
  - use in syntax, 133
  - use in text, 135
- <variable property>, **54**
  - use in syntax, 54
  - use in text, 55
- <variable>, **186**
  - use in syntax, 104, 147, 186, 187
  - use in text, 34, 104, 105, 147, 148, 186, 187
- <variables of sort>, **184**
  - use in syntax, 184
  - use in text, 88
- <vertical line>, **27**
  - use in syntax, 27
  - use in text, 22
- <via gate>, **89**
  - use in syntax, 89
  - use in text, 92
- <via path>, **129**
  - use in syntax, 94
  - use in text, 94, 129, 130, 131, 254
- <virtuality constraint>, **68**
  - use in syntax, 45, 49, 85, 92, 151, 152
  - use in text, 68, 69, 71
- <virtuality>, **68**
  - use in syntax, 36, 45, 49, 85, 87, 100, 104, 105, 106, 107, 108, 109, 117, 118, 147, 152, 153, 156, 187
  - use in text, 36, 68, 70, 71, 88, 93, 100, 106, 107, 109, 110, 148, 153, 157, 158, 167, 168, 187
- <visibility>, **171**
  - use in syntax, 72, 156, 161, 162, 164
  - use in text, 55, 161, 163, 164, 171
- <word>, **25**
  - use in syntax, 25
  - use in text, 29, 254

## ANNEXE B

### Réservée pour une utilisation future

## ANNEXE C

### Réservée pour une utilisation future

## ANNEXE D

### Données prédéfinies du SDL

#### D.1 Introduction

Dans le SDL, les données prédéfinies sont fondées sur des types de données abstraits qui sont définis essentiellement en termes de propriétés abstraites plutôt qu'en termes d'implémentation concrète. Bien que la définition d'un type de données abstrait donne un moyen possible d'implémenter ce type de données, il n'est pas obligatoire d'utiliser cette implémentation pour ce type de données abstrait, à condition de conserver le même comportement abstrait.

Les types de données prédéfinis, y compris la sorte booléenne qui définit les propriétés de deux littéraux "true" et "false", sont définis dans la présente annexe. Les deux *termes* booléens "true" et "false" ne doivent pas être définis (directement ou indirectement) comme des équivalents. Chaque expression booléenne constante qui est utilisée en dehors des définitions de type de données doit être interprétée comme "true" ou "false". S'il n'est pas possible de réduire une telle expression à "true" ou "false", la spécification est incomplète et permet plusieurs interprétations du type de données.

Les données prédéfinies sont définies dans un paquetage Predefined utilisé implicitement (voir 7.2). Ce paquetage est défini dans la présente annexe.

#### D.2 Notation

Pour des besoins de notation, la présente annexe étend la syntaxe concrète du SDL au moyen d'une description des propriétés des opérations abstraites ajoutées par une définition de type de données. Cette syntaxe supplémentaire est cependant utilisée à titre d'exemple uniquement et n'est pas une extension de la syntaxe définie dans la partie principale de la présente Recommandation. Une spécification utilisant la syntaxe définie dans la présente annexe n'est donc pas du SDL valide.

Les propriétés abstraites décrites ici ne spécifient pas une représentation spécifique des données prédéfinies. Au contraire, une interprétation doit être conforme à ces propriétés. Lorsqu'une expression <expression> est interprétée, l'évaluation de cette expression fournit une valeur (par exemple comme résultat d'une application <operation application>). Deux expressions E1 et E2 sont équivalentes si:

- a) il existe une équation <equation>  $E1 == E2$ ;
- b) une des équations obtenues à partir de l'ensemble donné d'équations <quantified equations> est  $E1 == E2$ ;
- c)
  - i) E1 est équivalente à EA;
  - ii) E2 est équivalente à EB;
  - iii) il existe une équation ou une équation obtenue des équations quantifiées de l'ensemble donné telle que  $EA == EB$ ;



- d) si, en remplaçant un sous-terme de E1 par un terme de la même classe que le sous-terme créant un terme E1A, il est possible de démontrer que E1A se trouve dans la même classe que E2.

Autrement, les deux expressions ne sont pas équivalentes.

Deux expressions équivalentes représentent la même valeur.

L'interprétation d'expressions est conforme à ces propriétés si deux expressions équivalentes représentent la même valeur et si deux expressions non équivalentes représentent des valeurs différentes.

### D.2.1 Axiomes

Les axiomes déterminent quels termes représentent la même valeur. A partir des axiomes dans une définition de type de données, on détermine les relations entre les valeurs des arguments et les valeurs des résultats des opérateurs et, donc on donne une signification aux opérateurs. Les axiomes sont donnés soit comme étant des axiomes booléens soit sous la forme d'équations algébriques d'équivalence.

Une opération définie par <axiomatic operation definitions> est traitée comme une définition complète en fonction d'une spécialisation. C'est-à-dire que lorsqu'un type de données défini par le paquetage Predefined est spécialisé et qu'une opération est redéfinie dans le type spécialisé, tous les axiomes mentionnant le nom de l'opération sont remplacés par la définition correspondante dans le type spécialisé.

#### *Grammaire textuelle concrète*

```

<axiomatic operation definitions> ::=
    axioms <axioms>

<axioms> ::=
    <equation> { <end> <equation> } * [ <end> ]

<equation> ::=
    <unquantified equation>
    | <quantified equations>
    | <conditional equation>
    | <literal equation>
    | <noequality>

<unquantified equation> ::=
    <term> == <term>
    | <Boolean axiom>

<term> ::=
    <constant expression>
    | <error term>

<quantified equations> ::=
    <quantification> ( <axioms> )

<quantification> ::=
    for all <value name> { , <value name> } * in <sort>

```

La présente annexe modifie <operations> (voir 12.1.1) comme décrit ci-dessous.

```

<operations> ::=
    <operation signatures>
    { <operation definitions> | <axiomatic operation definitions> }

```

Les définitions <axiomatic operation definitions> peuvent être utilisées uniquement pour décrire le comportement des opérateurs.

Un identificateur <identifieur> qui est un nom non qualifié apparaissant dans un terme <term> peut être:

- a) un identificateur <operation identifieur> (voir 12.2.7);
- b) un identificateur <literal identifieur> (voir 12.2.2);
- c) un identificateur <value identifieur> s'il y a une définition de ce nom dans une <quantification> d'équations <quantified equations> englobant le terme <term>, qui doit alors avoir une sorte adaptée à ce contexte; sinon
- d) un identificateur <value identifieur> qui a une équation quantifiée implicite pour l'équation <unquantified equation>.

Deux occurrences au moins du même identificateur de valeur non liée dans la même équation <unquantified equation> (ou si cette équation est contenue dans une équation <conditional equation>, à l'intérieur de cette équation <conditional equation>), indiquent une <quantification>

Dans une équation <unquantified equation>(ou si cette équation est contenue dans une équation <conditional equation> à l'intérieur de cette équation <conditional equation>), il doit y avoir exactement une sorte pour chaque identificateur de valeur quantifiée implicitement qui soit cohérente avec toutes ses utilisations.

### *Sémantique*

Un terme clos est un terme qui ne contient aucun identificateur de valeur. Un terme clos représente une valeur particulière connue. Pour chaque valeur dans une sorte, il existe au moins un terme clos qui représente cette valeur.

Chaque équation exprime l'équivalence algébrique de termes. Le membre de gauche et le membre de droite sont supposés être équivalents si bien que chaque fois qu'un terme apparaît, l'autre terme peut lui être substitué. Quand un identificateur de valeur apparaît dans une équation, il peut simultanément être substitué dans cette équation par le même terme pour chaque apparition de l'identificateur de valeur. Pour cette substitution, le terme peut être un terme clos quelconque de la même sorte que l'identificateur de valeur.

Les identificateurs de valeur sont introduits par les noms de valeur dans les équations quantifiées. Un identificateur de valeur est utilisé pour représenter toute valeur de données appartenant à la sorte de la quantification. Une équation est valide si la même valeur est simultanément substituée pour toute occurrence de l'identificateur de valeur dans l'équation quelle que soit la valeur choisie pour la substitution.

En général, il n'est ni nécessaire ni justifié d'établir une distinction entre le terme clos et le résultat de ce terme. Par exemple, le terme clos de l'élément du nombre entier représentant l'unité peut s'écrire "1". Il existe normalement plusieurs termes clos pour désigner le même élément de données, par exemple les termes clos entiers "0+1", "3-2" et "(7+5)/12", et on adopte d'ordinaire une forme simple du terme clos (dans ce cas "1") pour désigner l'élément de données.

Si un axiome quelconque contient un texte informel, l'interprétation des expressions n'est pas forcément définie par le SDL, mais peut être déterminée à partir du texte informel par l'interpréteur. Il est supposé que si un terme informel est spécifié, l'ensemble d'équations est connu comme étant incomplet et par conséquent une spécification formelle complète n'a pas été donnée en SDL.

Un nom de valeur est toujours introduit au moyen d'équations quantifiées et la valeur correspondante à un identificateur de valeur, qui est le nom de la valeur qualifiée par l'identificateur de sorte des équations quantifiées englobantes. Par exemple:

```
for all z in X (for all z in X ... )
```

introduit un seul identificateur de valeur appelé z de la sorte X.

Dans la syntaxe concrète des axiomes, il n'est pas permis de spécifier un qualificatif pour des identificateurs de valeur.

Chaque identificateur de valeur introduit au moyen d'équations quantifiées a une sorte identifiée dans les équations quantifiées par la sorte <sort>. La sorte des quantifications implicites est la sorte requise par le ou les contextes d'occurrence de l'identificateur non lié. Si les contextes d'un identificateur de valeur qui a une quantification implicite permettent différentes sortes, l'identificateur est lié à une sorte qui est cohérente avec toutes ses utilisations dans l'équation.

Un terme a une sorte, qui est la sorte de l'identificateur de valeur ou la sorte résultante de l'opérateur (littéral).

A moins que l'on puisse déduire à partir des équations que deux termes désignent la même valeur, chaque terme représente une valeur différente.

### D.2.2 Equations conditionnelles

Une équation conditionnelle permet la spécification d'équations qui ne sont valides que lorsque certaines restrictions existent. Ces restrictions sont écrites sous la forme d'équations simples.

#### *Grammaire textuelle concrète*

```
<conditional equation> ::=  
    <restriction> { , <restriction> }* ==><restricted equation>  
<restricted equation> ::=  
    <unquantified equation>  
<restriction> ::=  
    <unquantified equation>
```

#### *Sémantique*

Une équation conditionnelle définit que des termes prennent le même élément de données seulement quand tout identificateur de valeur dans les équations restreintes prend un élément de données dont on peut prouver à partir des autres équations qu'il satisfait la restriction.

La sémantique d'un système d'équations pour un type de données qui inclut des équations conditionnelles s'établit comme suit:

- a) on élimine la quantification en générant chaque équation possible de termes clos qui peut être déduite des équations quantifiées. Comme cette opération s'applique aussi bien à la quantification explicite qu'à la quantification implicite, on obtient un système d'équations non quantifiées concernant uniquement des termes clos;
- b) une équation conditionnelle est dite prouvable si l'on peut prouver que toutes les restrictions (en termes clos seulement) sont vraies en se basant sur des équations non quantifiées qui ne sont pas des équations restreintes. S'il existe une équation conditionnelle prouvable, elle est remplacée par l'équation restreinte correspondant à l'équation conditionnelle prouvable;
- c) s'il reste des équations conditionnelles dans le système d'équations et si aucune d'entre elles n'est une équation conditionnelle prouvable, on supprime ces équations conditionnelles; dans le cas contraire, on revient à b);
- d) le reste du système d'équations quantifiées définit la sémantique du type de données.

### D.2.3 Egalité

#### *Grammaire textuelle concrète*

```
<noequality> ::=  
    noequality
```

## Modèle

Toute définition <data type definition> introduisant une certaine sorte appelée S a la signature <operation signature> implicite suivante dans sa liste <operator list>, sauf si <noequality> est présent dans les <axioms>:

```
equal ( S, S ) -> Boolean;
```

où Boolean est la sorte booléenne prédéfinie.

Toute définition <data type definition> introduisant une sorte appelée S telle qu'elle contient uniquement des définitions <axiomatic operation definitions> dans une liste <operator list> comporte l'ensemble d'équations implicite:

```
for all a,b,c in S (
  equal(a, a) == true;
  equal(a, b) == equal(b, a);
  equal(a, b) and equal(b, c) ==> equal(a, c) == true;
  equal(a, b) == true ==> a == b;
```

et une équation <literal equation> implicite:

```
for all L1,L2 in S literals (
  spelling(L1) /= spelling(L2) ==> L1 = L2 == false;)
```

### D.2.4 Axiomes booléens

#### Grammaire textuelle concrète

<Boolean axiom> ::=  
                  <Boolean term>

#### Sémantique

Un axiome booléen est une affirmation de vérité qui est valable dans tous les cas pour le type de données défini.

## Modèle

Un axiome de la forme:

```
<Boolean term>;
```

est une syntaxe dérivée pour l'équation en syntaxe concrète:

```
<Boolean term> == << package Predefined/type Boolean >> true;
```

### D.2.5 Terme conditionnel

#### Sémantique

Une équation contenant un terme conditionnel est sémantiquement équivalente à un ensemble d'équations où tous les identificateurs de valeurs quantifiées dans le terme booléen ont été éliminés. Cet ensemble d'équations peut être formé en substituant simultanément dans toute l'équation de terme conditionnel chaque identificateur <value identifier> dans l'expression <conditional expression> par chaque terme clos de la sorte appropriée. Dans cet ensemble d'équations, l'expression <conditional expression> aura toujours été remplacée par un terme clos booléen. Dans ce qui suit, on fera référence à cet ensemble d'équations comme étant l'ensemble clos développé.

Une équation de terme conditionnel est équivalente à l'ensemble d'équations qui contient:

- a) pour chaque *equation* dans l'ensemble clos développé pour laquelle l'expression <conditional expression> est équivalente à "true", cette *equation* obtenue à partir de l'ensemble clos développé avec l'expression <conditional expression> remplacée par l'expression (fondamentale) <consequence expression>;
- b) pour chaque *equation* dans l'ensemble clos développé pour laquelle l'expression <conditional expression> est équivalente à "false", cette *equation* obtenue à partir de l'ensemble clos développé avec l'expression <conditional expression> remplacée par l'expression (fondamentale) <alternative expression>;

il faut remarquer que dans ce cas particulier une équation de la forme:

```
ex1 == if a then b else c fi;
```

est équivalente à la paire d'équations conditionnelles:

```
a == true ==> ex1 == b;  
a == false ==> ex1 == c;
```

## D.2.6 Terme d'erreur

Les erreurs sont utilisées afin de permettre aux propriétés des types de données d'être totalement définies même dans les cas où aucune signification particulière ne peut être donnée au résultat d'un opérateur.

### *Grammaire textuelle concrète*

```
<error term> ::=  
    raise <exception name>
```

Un terme <error term> ne doit pas être utilisé comme une partie d'une <restriction>.

Il ne doit pas être possible de déduire à partir de *equations* qu'un identificateur <literal identifier> est égal à un terme <error term>.

### *Sémantique*

Un terme peut être un terme erreur <error term> de telle sorte qu'il est possible de spécifier les circonstances dans lesquelles un opérateur produit une erreur. Si ces circonstances apparaissent lors de l'interprétation, l'exception avec le nom <exception name> est déclenchée.

## D.2.7 Littéraux non ordonnés

### *Grammaire textuelle concrète*

```
<unordered> ::=  
    unordered  
  
La présente annexe modifie la syntaxe concrète pour le constructeur de type de liste de littéraux (voir 12.1.7.1) comme suit.  
  
<literal list> ::=  
    [<protected>] literals [<unordered>]  
    <literal signature> { , <literal signature> }* <end>
```

### *Modèle*

Si <unordered> est utilisé, le *Modèle* en 12.1.7.1 n'est pas appliqué. Par conséquent, les opérations d'ordonnement "<", ">", "<=", ">=", first, last, pred, succ et num ne sont pas implicitement définies pour ce type de données.

## D.2.8 Equations de littéral

### Grammaire textuelle concrète

<literal equation> ::=

<literal quantification>  
( <equation> { <end> <equation> }\* [ <end> ] )

<literal quantification> ::=

**for all** <value name> { , <value name> }\* **in** <sort> **literals**  
| **for all** <value name> { , <value name> }\* **in** { <sort> | <value identifier> } **nameclass**

La présente annexe modifie la syntaxe concrète du terme <spelling term> (voir 12.1.9.2) comme suit.

<spelling term> ::=

**spelling** ( { <operation name> | <value identifier> } )

L'identificateur <value identifier> dans un terme <spelling term> doit être un identificateur <value identifier> défini par une quantification <literal quantification>.

### Sémantique

Le mappage de littéral est une notation abrégée permettant de définir un grand nombre (éventuellement infini) d'axiomes s'étendant à la totalité des littéraux d'une sorte ou à tous les noms d'une classe de noms. Grâce à ce mappage, les littéraux d'une sorte peuvent être mappés avec les valeurs de la sorte.

Le terme <spelling term> est utilisé dans les quantifications de littéral pour désigner la chaîne de caractères qui contient l'orthographe du littéral. Grâce à ce mécanisme, les opérateurs Charstring peuvent être utilisés pour définir les équations de littéral.

### Modèle

Une équation <literal equation> est une abréviation pour un ensemble d'axiomes <axioms> contenus dans une équation <literal equation>, les identificateurs <value identifier> définis par le nom <value name> dans la quantification <literal quantification> sont remplacés. Dans chaque équation <equation> obtenue, le même identificateur <value identifier> est remplacé, chaque fois qu'il apparaît, par le même identificateur <literal identifier> de la sorte <sort> de la quantification <literal quantification> (si des **literals** sont utilisés) ou par le même identificateur <literal identifier> de la classe de nom référencée (si une **nameclass** est utilisée). L'ensemble obtenu d'axiomes <axioms> renferme toutes les équations <equation> possibles qui peuvent être obtenues de cette manière.

Les axiomes <axioms> obtenus pour les équations <literal equation> sont ajoutés aux axiomes <axioms> (le cas échéant) définis après le mot clé **axioms**.

Si une quantification <literal quantification> contient un ou plusieurs termes <spelling term>, il faut alors remplacer les termes <spelling term> par les littéraux Charstring (voir D.3). La chaîne Charstring est utilisée pour remplacer l'identificateur <value identifier> une fois que l'équation <literal equation> contenant le terme <spelling term> est développée de la façon indiquée en 12.1.9.2, en utilisant l'identificateur <value identifier> à la place du nom <operation name>.

NOTE – Les équations de littéral n'affectent pas les opérateurs d'annulation définis dans les signatures <operation signature>.

### D.3 Paquetage Prédéfini (Predefined)

Dans les définitions suivantes, toutes les références aux noms définis dans le paquetage Predefined sont supposées traitées comme des préfixes de la qualification <<package Predefined>>. Pour une meilleure lisibilité, cette qualification est omise.

```
/* */
package Predefined
/*
```

#### D.3.1 Sorte booléenne (Boolean)

##### D.3.1.1 Définition

```
*/
value type Boolean
  literals true, false;
  operators
    "not" ( this Boolean )      -> this Boolean;
    "and" ( this Boolean, this Boolean ) -> this Boolean;
    "or"  ( this Boolean, this Boolean ) -> this Boolean;
    "xor" ( this Boolean, this Boolean ) -> this Boolean;
    "=>" ( this Boolean, this Boolean ) -> this Boolean;
  axioms
    not( true  ) == false;
    not( false ) == true ;
/* */
  true and true  == true ;
  true and false == false;
  false and true == false;
  false and false == false;
/* */
  true or true   == true ;
  true or false  == true ;
  false or true  == true ;
  false or false == false;
/* */
  true xor true   == false;
  true xor false  == true ;
  false xor true  == true ;
  false xor false == false;
/* */
  true => true    == true ;
  true => false   == false;
  false => true    == true ;
  false => false  == true ;
endvalue type Boolean;
/*
```

##### D.3.1.2 Utilisation

La sorte booléenne est utilisée pour représenter les valeurs Vrai (true) et Faux (false). On l'utilise souvent comme résultat d'une comparaison.

On utilise largement la sorte booléenne dans le langage SDL.

#### D.3.2 Sorte de caractère (Character)

##### D.3.2.1 Définition

```
*/
value type Character
  literals
    NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
    BS, HT, LF, VT, FF, CR, SO, SI,
    DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB,
    CAN, EM, SUB, ESC, IS4, IS3, IS2, IS1,
    ' ', '!', '"', '#', '$', '%', '&', ''',
    '(', ')', '*', '+', ',', '-', '.', '/',
    '0', '1', '2', '3', '4', '5', '6', '7',
```

```

'8', '9', ':', ';', '<', '=', '>', '?',
'@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
'X', 'Y', 'Z', '[', '\', ']', '^', '_',
`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
'x', 'y', 'z', '{', '|', '}', '~', DEL;
/* ''' est une apostrophe, ' ' est un espace, '~' est un tilde */
/* */
operators
chr ( Integer ) -> this Character;
/* "<", "<=", ">", ">=", et "num" sont définis de manière implicite (voir 12.1.7.1). */
axioms
for all a,b in Character (
for all i in Integer (
/* définition de Chr */
chr(num(a)) == a;
chr(i+128) == chr(i);
));
endvalue type Character;
/*

```

### D.3.2.2 Utilisation

La sorte de caractère est utilisée pour représenter des caractères dans l'alphabet international de référence [international reference alphabet (Recommandation T.50)].

### D.3.3 Sorte de chaîne (String)

#### D.3.3.1 Définition

```

/*
value type String < type Itemsort >
/* les chaînes sont "indexées" à partir de un */
operators
emptystring                -> this String;
mkstring ( Itemsort        ) -> this String;
make ( Itemsort            ) -> this String;
length ( this String       ) -> Integer;
first ( this String        ) -> Itemsort;
last ( this String         ) -> Itemsort;
"//" ( this String, this String ) -> this String;
extract ( this String, Integer ) -> Itemsort raise InvalidIndex;
modify ( this String, Integer, Itemsort ) -> this String;
substring ( this String, Integer, Integer ) -> this String raise InvalidIndex;
/* substring (s,i,j) donne une chaîne de longueur j à partir du ie élément */
remove ( this String, Integer, Integer ) -> this String;
/* remove (s,i,j) donne une chaîne avec une sous-chaîne de longueur j
commençant au ie élément enlevé */
axioms
for all e in Itemsort ( /*e - élément de Itemsort*/
for all s,s1,s2,s3 in String (
for all i,j in Integer (
/* les constructeurs sont emptystring, mkstring, et "/" */
/* égalités entre termes de constructeurs */
s // emptystring == s;
emptystring // s == s;
(s1 // s2) // s3 == s1 // (s2 // s3);
/* */
/* définition de length appliquée à tous les constructeurs */
<<type String>>length(emptystring) == 0;
<<type String>>length(mkstring(e)) == 1;
<<type String>>length(s1 // s2) == length(s1) + length(s2);
make(s) == mkstring(s);
/* */

```



```

/* définition de extract appliqué à tous les constructeurs,
   les cas d'erreur sont traités séparément */
extract(mkstring(e),1) == e;
i <= length(s1) ==> extract(s1 // s2,i) == extract(s1,i);
i > length(s1) ==> extract(s1 // s2,i) == extract(s2,i-length(s1));
i<=0 or i>length(s) ==> extract(s,i) == raise InvalidIndex;
/* */
/* définition de first et de last à l'aide d'autres opérations */
first(s) == extract(s,1);
last(s) == extract(s,length(s));
/* */
/* définition de substring(s,i,j) par induction de j,
   les cas d'erreur sont traités séparément */
i>0 and i-1<=length(s) ==>
    substring(s,i,0) == emptystring;
/* */
i>0 and j>0 and i+j-1<=length(s) ==>
    substring(s,i,j) == substring(s,i,j-1) // mkstring(extract(s,i+j-1));
/* */
i<=0 or j<0 or i+j-1>length(s) ==>
    substring(s,i,j) == raise InvalidIndex;
/* */
/* définition de modify à l'aide d'autres opérations */
modify(s,i,e) == substring(s,1,i-1) // mkstring(e) // substring(s,i+1,length(s)-i);
/* définition de remove */
remove(s,i,j) == substring(s,1,i-1) // substring(s,i+j,length(s)-i-j+1);
));
endvalue type String;
/*

```

### D.3.3.2 Utilisation

Les opérateurs `make`, `extract` et `modify` sont classiquement utilisés avec les abréviations définies en 12.2.4 et 12.3.3.1 afin d'accéder aux valeurs de chaînes et d'attribuer des valeurs à des chaînes.

## D.3.4 Sorte Charstring

### D.3.4.1 Définition

```

/*
value type Charstring
  inherits String < Character > ( ' ' = emptystring )
  adding
    operators ocs in nameclass
      ' ' ( ( ' ':'&' ) or ' ' or ( '(' : '~' ) ) + ' ' -> this Charstring;
/* chaînes de caractères de n'importe quelle longueur de tous caractères allant de
l'espace ' ' au tilde '~' */
axioms
  for all c in Character nameclass (
    for all cs, cs1, cs2 in ocs nameclass (
      spelling(cs) == spelling(c) ==> cs == mkstring(c);
/* la chaîne 'A' est formée à partir du caractère 'A' etc. */
      spelling(cs) == spelling(cs1) // spelling(cs2),
      length(spelling(cs2)) == 1 ==> cs == cs1 // cs2;
    ));
endvalue type Charstring;
/*

```

### D.3.4.2 Utilisation

```
/*
```

La sorte `Charstring` définit des chaînes de caractères.

Un littéral `Charstring` peut contenir des caractères imprimables et des espaces.

Un caractère non imprimable peut être utilisé en tant que chaîne à l'aide de `mkstring`, par exemple `mkstring(DEL)`.

Exemple:

```
synonym newline_prompt Charstring = mkstring(CR) // mkstring(LF) // '$>';
```

### D.3.5 Sorte Entier (Integer)

#### D.3.5.1 Définition

```
*/
value type Integer
  literals unordered nameclass (('0':'9')*) ('0':'9'));
  operators
    "-" ( this Integer          ) -> this Integer;
    "+" ( this Integer, this Integer ) -> this Integer;
    "-" ( this Integer, this Integer ) -> this Integer;
    "*" ( this Integer, this Integer ) -> this Integer;
    "/" ( this Integer, this Integer ) -> this Integer raise DivisionByZero;
    "mod" ( this Integer, this Integer ) -> this Integer raise DivisionByZero;
    "rem" ( this Integer, this Integer ) -> this Integer;
    "<" ( this Integer, this Integer ) -> Boolean;
    ">" ( this Integer, this Integer ) -> Boolean;
    "<=" ( this Integer, this Integer ) -> Boolean;
    ">=" ( this Integer, this Integer ) -> Boolean;
    power ( this Integer, this Integer ) -> this Integer;
    bs in nameclass ''' ( (('0' or '1')*'B') or (('0':'9') or ('A':'F'))*'H' )
      -> this Integer;
  axioms noequality
    for all a,b,c in Integer (
/* les constructeurs sont 0, 1, +, et le - unaire*/
/* égalités entre termes de constructeurs */
      (a + b) + c      == a + (b + c);
      a + b            == b + a;
      0 + a            == a;
      a + (- a)        == 0;
      (- a) + (- b)    == - (a + b);
      <<type Integer>> - 0 == 0;
      - (- a)          == a;
/* */
/* définition de "-" binaire à l'aide d'autres opérations */
      a - b            == a + (- b);
/* */
/* définition de "*" en l'appliquant à tous les constructeurs */
      0 * a            == 0;
      1 * a            == a;
      (- a) * b        == - (a * b);
      (a + b) * c      == a * c + b * c;
/* */
/* définition de "<" en l'appliquant à tous les constructeurs */
      a < b              == 0 < (b - a);
      <<type Integer>> 0 < 0      == false;
      <<type Integer>> 0 < 1      == true ;
      0 < a              == true ==> 0 < (- a) == false;
      0 < a and 0 < b == true ==> 0 < (a + b) == true ;
/* */
/* définition de ">", "equal", "<=", et ">=" à l'aide d'autres opérations */
      a > b              == b < a;
      equal(a, b)        == not(a < b or a > b);
      a <= b             == a < b or a = b;
      a >= b             == a > b or a = b;
/* */
/* définition de "/" à l'aide d'autres opérations */
      a / 0              == raise DivisionByZero;
      a >= 0 and b > a    == true ==> a / b == 0;
      a >= 0 and b <= a and b > 0 == true ==> a / b == 1 + (a-b) / b;
      a >= 0 and b < 0    == true ==> a / b == - (a / (- b));
      a < 0 and b < 0     == true ==> a / b == (- a) / (- b);
      a < 0 and b > 0     == true ==> a / b == - ((- a) / b);
/* */
```

```

/* définition de "rem" à l'aide d'autres opérations */
  a rem b == a - b * (a/b);
/* */
/* définition de "mod" à l'aide d'autres opérations */
  a >= 0 and b > 0          ==> a mod b == a rem b;
  b < 0                      ==> a mod b == a mod (- b);
  a < 0 and b > 0 and a rem b = 0 ==> a mod b == 0;
  a < 0 and b > 0 and a rem b < 0 ==> a mod b == b + a rem b;
  a mod 0 == raise DivisionByZero;
/* */
/* définition de la puissance power à l'aide d'autres opérations */
  power(a, 0)                == 1;
  b > 0 ==> power(a, b) == a * power(a, b-1);
  b < 0 ==> power(a, b) == power(a, b+1) / a; );
/* */
/* définition des littéraux */
  <<type Integer>> 2 == 1 + 1;
  <<type Integer>> 3 == 2 + 1;
  <<type Integer>> 4 == 3 + 1;
  <<type Integer>> 5 == 4 + 1;
  <<type Integer>> 6 == 5 + 1;
  <<type Integer>> 7 == 6 + 1;
  <<type Integer>> 8 == 7 + 1;
  <<type Integer>> 9 == 8 + 1;
/* */
/* littéraux autres que 0 à 9 */
  for all a,b,c in Integer nameclass (
    spelling(a) == spelling(b) // spelling(c),
    length(spelling(c)) == 1          ==> a == b * (9 + 1) + c;
  );
/* */
/* représentation hexadécimale et binaire de Integer */
  for all b in Bitstring nameclass (
    for all i in bs nameclass (
      spelling(i) == spelling(b)          ==> i == <<type Bitstring>>num(b);
    ));
endvalue type Integer;
/*

```

### D.3.5.2 Utilisation

La sorte Integer est utilisée pour les entiers mathématiques avec une notation décimale, hexadécimale ou binaire.

## D.3.6 Syntype Naturel (Natural)

### D.3.6.1 Définition

```

*/
syntype Natural = Integer constants >= 0 endsyntype Natural;
/*

```

### D.3.6.2 Utilisation

Le syntype naturel est utilisé lorsque seuls des entiers positifs sont nécessaires. Tous les opérateurs sont les opérateurs de integer mais, lorsqu'une valeur est utilisée comme paramètre ou si elle est attribuée, cette valeur est vérifiée. Une valeur négative est une erreur.

## D.3.7 Sorte Real

### D.3.7.1 Définition

```

*/
value type Real
  literals unordered nameclass
    ( ('0':'9')* ('0':'9') ) or ( ('0':'9')* '.' ('0':'9')+ );

```

```

operators
  "-" ( this Real          ) -> this Real;
  "+" ( this Real, this Real ) -> this Real;
  "-" ( this Real, this Real ) -> this Real;
  "*" ( this Real, this Real ) -> this Real;
  "/" ( this Real, this Real ) -> this Real raise DivisionByZero;
  "<" ( this Real, this Real ) -> Boolean;
  ">" ( this Real, this Real ) -> Boolean;
  "<=" ( this Real, this Real ) -> Boolean;
  ">=" ( this Real, this Real ) -> Boolean;
  float ( Integer          ) -> this Real;
  fix   ( this Real       ) -> Integer;

axioms noequality
  for all r,s in Real (
    for all a,b,c,d in Integer (
/* les constructeurs sont float et "/" */
/* des égalités entre termes de constructeurs permettent de toujours obtenir une forme
   float(a) / float(b) où b > 0 */
      r / float(0)                == raise DivisionByZero;
      r / float(1)                == r;
      c /= 0 ==> float(a) / float(b) == float(a*c) / float(b*c);
      b /= 0 and d /= 0 ==>
        (float(a) / float(b)) / (float(c) / float(d)) == float(a*d) / float(b*c);
/* */
/* définition de "-" unaire en l'appliquant à tous les constructeurs */
      - (float(a) / float(b))      == float(- a) / float(b);
/* */
/* définition de "+" en l'appliquant à tous les constructeurs */
      (float(a) / float(b)) + (float(c) / float(d)) ==float(a*d + c*b) / float(b*d);
/* */
/* définition de binary "-" à l'aide d'autres opérations */
      r - s                          == r + (- s);
/* */
/* définition de "*" en l'appliquant à tous les constructeurs */
      (float(a) / float(b)) * (float(c) / float(d)) == float(a*c) / float(b*d);
/* */
/* définition de "<" en l'appliquant à tous les constructeurs */
      b > 0 and d > 0 ==>
        (float(a) / float(b)) < (float(c) / float(d)) == a * d < c * b;
/* */
/* définition de ">", "equal", "<=", et ">=" à l'aide d'autres opérations */
      r > s          == s < r;
      equal(r, s) == not(r < s or r > s);
      r <= s        == r < s or r = s;
      r >= s        == r > s or r = s;
/* */
/* définition de fix en l'appliquant à tous les constructeurs */
      a >= b and b > 0 ==> fix(float(a) / float(b)) == fix(float(a-b) / float(b)) + 1;
      b > a and a >= 0 ==> fix(float(a) / float(b)) == 0;
      a < 0 and b > 0 ==> fix(float(a) / float(b)) == - fix(float(-a)/float(b)) - 1;));
/* */
      for all r,s in Real nameclass (
        for all i,j in Integer nameclass (
          spelling(r) == spelling(i)          ==> r == float(i);
/* */
          spelling(r) == spelling(i)          ==> i == fix(r);
/* */
          spelling(r) == spelling(i) // spelling(s),
          spelling(s) == '.' // spelling(j) ==> r == float(i) + s;
/* */
          spelling(r) == '.' // spelling(i),
          length(spelling(i)) == 1          ==> r == float(i) / 10;
/* */
          spelling(r) == '.' // spelling(i) // spelling(j),
          length(spelling(i)) == 1,
          spelling(s) == '.' // spelling(j) ==> r == (float(i) + s) / 10;
        ));
endvalue type Real;
/*

```

### D.3.7.2 Utilisation

La sorte `real` est utilisée pour représenter les nombres réels.

La sorte `real` peut représenter tous les nombres que l'on peut représenter comme résultat de la division d'un entier par un autre entier.

Les nombres que l'on ne peut représenter de cette manière (les nombres irrationnels – par exemple la racine carrée de 2) ne font pas partie de la sorte `real`. Cependant on peut généralement utiliser une approximation suffisamment précise pour des raisons techniques.

### D.3.8 Sorte Tableau (Array)

#### D.3.8.1 Définition

```
*/
value type Array < type Index; type Itemsort >
  operators
    make                               -> this Array ;
    make ( Itemsort                    ) -> this Array ;
    modify ( this Array, Index, Itemsort ) -> this Array ;
    extract( this Array, Index          ) -> Itemsort raise InvalidIndex;
  axioms
    for all item, itemi, itemj in Itemsort (
      for all i, j in Index (
        for all a, s in Array (
          <<type Array>>extract(make,i)                == raise InvalidIndex;
          <<type Array>>extract(make,item),i)           == item ;
          i = j ==> modify(modify(s,i,itemi),j,item) == modify(s,i,item);
          i = j ==> extract(modify(a,i,item),j)       == item ;
          i = j == false ==> extract(modify(a,i,item),j) == extract(a,j);
          i = j == false ==> modify(modify(s,i,itemi),j,itemj)==
                                                                modify(modify(s,j,itemj),i,itemi);
/*égalité*/
          <<type Array>>make(itemi) = make(itemj)      == itemi = itemj;
          a=s == true, i=j == true, itemi = itemj ==>
            modify(a,i,itemi) = modify(s,j,itemj)    == true;
/* */
          extract(a,i) = extract(s,i) == false ==> a = s == false;));
endvalue type Array;
/*
```

#### D.3.8.2 Utilisation

Un tableau peut être utilisé pour définir une sorte qui est indexée par une autre. Par exemple:

```
value type indexbychar inherits Array< Character, Integer >
endvalue type indexbychar;
```

définit un tableau contenant des entiers indexé par des caractères.

Les tableaux sont généralement utilisés en combinaison avec les formes d'abréviations de `make`, `modify` et `extract` définies en 12.2.4 et 12.3.3.1. Par exemple:

```
dcl charvalue indexbychar;
task charvalue := (. 12.);
task charvalue('A') := charvalue('B')-1;
```

### D.3.9 Vecteur

#### D.3.9.1 Définition

```
*/
value type Vector < type Itemsort; synonym MaxIndex >
  inherits Array< Indexsort, Itemsort >;
syntype Indexsort = Integer constants 1:MaxIndex endsyntype;
endvalue type Vector;
/*
```

## D.3.10 Sorte Powerset

### D.3.10.1 Définition

```
*/
value type Powerset < type Itemsort >
  operators
    empty                                -> this Powerset;
    "in"  ( Itemsort, this Powerset    ) -> Boolean;      /* est membre de */
    incl  ( Itemsort, this Powerset    ) -> this Powerset; /* inclure l'item dans
                                                                l'ensemble */
    del   ( Itemsort, this Powerset    ) -> this Powerset; /* supprimer l'item de
                                                                l'ensemble */
    "<"   ( this Powerset, this Powerset ) -> Boolean;      /* est un sous-ensemble
                                                                propre de */
    ">"   ( this Powerset, this Powerset ) -> Boolean;      /* est un super-ensemble
                                                                propre de */
    "<="  ( this Powerset, this Powerset ) -> Boolean;      /* est un sous-ensemble de */
    ">="  ( this Powerset, this Powerset ) -> Boolean;      /* est un super-ensemble de */
    "and" ( this Powerset, this Powerset ) -> this Powerset; /* intersection d'ensembles */
    "or"  ( this Powerset, this Powerset ) -> this Powerset; /* union d'ensembles */
    length ( this Powerset              ) -> Integer;
    take   ( this Powerset              ) -> Itemsort raise Empty;
  axioms
    for all i,j in Itemsort (
      for all p,ps,a,b,c in Powerset (
        /* les constructeurs sont empty et incl */
        /* égalités entre termes de constructeurs */
        incl(i,incl(j,p)) == incl(j,incl(i,p));
        i = j ==> incl(i,incl(j,p)) == incl(i,p);
        /* définition de "in" en l'appliquant à tous les constructeurs */
        i in <<type Powerset>>empty == false;
        i in incl(j,ps) == i=j or i in ps;
        /* définition de del en l'appliquant à tous les constructeurs */
        <<type Powerset>>del(i,empty) == empty;
        i = j ==> del(i,incl(j,ps)) == del(i,ps);
        i /= j ==> del(i,incl(j,ps)) == incl(j,del(i,ps));
        /* définition de "<" en l'appliquant à tous les constructeurs */
        a < <<type Powerset>>empty == false;
        <<type Powerset>>empty < incl(i,b) == true;
        incl(i,a) < b == i in b and del(i,a) < del(i,b);
        /* définition de ">" à l'aide d'autres opérations */
        a > b == b < a;
        /* définition de "=" en l'appliquant à tous les constructeurs */
        empty = incl(i,ps) == false;
        incl(i,a) = b == i in b and del(i,a) = del(i,b);
        /* définition de "<=" and ">=" à l'aide d'autres opérations */
        a <= b == a < b or a = b;
        a >= b == a > b or a = b;
        /* définition de "and" en l'appliquant à tous les constructeurs */
        empty and b == empty;
        i in b ==> incl(i,a) and b == incl(i,a and b);
        not(i in b) ==> incl(i,a) and b == a and b;
        /* définition de "or" en l'appliquant à tous les constructeurs */
        empty or b == b;
        incl(i,a) or b == incl(i,a or b);
        /* définition de length */
        length(<<type Powerset>>empty) == 0;
        i in ps ==> length(ps) == 1 + length(del(i, ps));
        /* définition de take */
        take(empty) == raise Empty;
        i in ps ==> take(ps) == i;
      ));
  endvalue type Powerset;
/*
```

### D.3.10.2 Utilisation

Les ensembles Powersets sont utilisés pour représenter des ensembles mathématiques. Par exemple:

```
value type Boolset inherits Powerset< Boolean > endvalue type Boolset;
```

peut être utilisé pour une variable qui peut être empty ou contain (true), (false) ou (true, false).

### D.3.11 Sort durée (Duration)

#### D.3.11.1 Définition

```
*/
value type Duration
  literals unordered nameclass ('0':'9')+ or (('0':'9')* '.' ('0':'9')+);
  operators
    protected duration ( Real          ) -> this Duration;
    "+" ( this Duration, this Duration ) -> this Duration;
    "-" ( this Duration, this Duration ) -> this Duration;
    "-" ( this Duration, this Duration ) -> this Duration;
    ">" ( this Duration, this Duration ) -> Boolean;
    "<" ( this Duration, this Duration ) -> Boolean;
    ">=" ( this Duration, this Duration ) -> Boolean;
    "<=" ( this Duration, this Duration ) -> Boolean;
    "*" ( this Duration, Real          ) -> this Duration;
    "*" ( Real, this Duration         ) -> this Duration;
    "/" ( this Duration, Real          ) -> Duration;
  axioms noequality
  /* le constructeur est duration(Real)*/
    for all a, b in Real nameclass (
      for all d, e in Duration nameclass (
        /* définition de "+" en l'appliquant à tous les constructeurs */
        duration(a) + duration(b) == duration(a + b);
        /* */
        /* définition de "-" unaire en l'appliquant à tous les constructeurs */
        - duration(a) == duration(-a);
        /* */
        /* définition de "-" binaire à l'aide d'autres opérations */
        d - e == d + (-e);
        /* */
        /* définition de "equal", ">", "<", ">=" et "<=" en l'appliquant à tous les constructeurs */
        equal(duration(a), duration(b)) == a = b;
        duration(a) > duration(b) == a > b;
        duration(a) < duration(b) == a < b;
        duration(a) >= duration(b) == a >= b;
        duration(a) <= duration(b) == a <= b;
        /* */
        /* définition de "*" en l'appliquant à tous les constructeurs */
        duration(a) * b == duration(a * b);
        a * d == d * a;
        /* */
        /* définition de "/" en l'appliquant à tous les constructeurs */
        duration(a) / b == duration(a / b);
        /* */
        spelling(d) == spelling(a) ==>
          d == duration(a);
      ));
endvalue type Duration;
/*
```

#### D.3.11.2 Utilisation

La sorte duration est utilisée pour la valeur devant être ajoutée au temps actuel pour régler les temporisateurs. Les littéraux de la sorte duration sont les mêmes que les littéraux de la sorte real. La signification d'une unité de durée dépend du système en cours de définition.

Les valeurs de durée peuvent être multipliées et divisées par des valeurs réelles.

## D.3.12 Sorte temps (Time)

### D.3.12.1 Définition

```
*/
value type Time
  literals unordered nameclass ('0':'9')+ or (('0':'9')* '.' ('0':'9')+);
  operators
    protected time ( Duration    ) -> this Time;
    "<" ( this Time, this Time ) -> Boolean;
    "<=" ( this Time, this Time ) -> Boolean;
    ">" ( this Time, this Time ) -> Boolean;
    ">=" ( this Time, this Time ) -> Boolean;
    "+" ( this Time, Duration ) -> this Time;
    "+" ( Duration, this Time ) -> this Time;
    "-" ( this Time, Duration ) -> this Time;
    "-" ( this Time, this Time ) -> Duration;
  axioms noequality
/* le constructeur est time */
    for all t, u in Time nameclass (
      for all a, b in Duration nameclass (
/* définition de ">", "equal" en l'appliquant à tous les constructeurs */
        time(a) > time(b)      == a > b;
        equal(time(a), time(b)) == a = b;
/* */
/* définition de "<", "<=", ">=" à l'aide d'autres opérations */
        t < u                    == u > t;
        t <= u                   == (t < u) or (t = u);
        t >= u                   == (t > u) or (t = u);
/* */
/* définition de "+" en l'appliquant à tous les constructeurs */
        time(a) + b              == time(a + b);
        a + t                    == t + a;
/* */
/* définition de "-" : Time, Duration à l'aide d'autres opérations */
        t - b                    == t + (-b);
/* */
/* définition de "-" : Time, Time en l'appliquant à tous les constructeurs */
        time(a) - time(b)       == a - b;
/* */
        spelling(a) == spelling(t) ==>
                                a == time(t);
    ));
endvalue type Time;
/*
```

### D.3.12.2 Utilisation

L'expression **now** renvoie une valeur de la sorte time. Une valeur de temps peut avoir une durée qui lui est ajoutée ou retranchée afin de fournir une autre valeur de temps. Une valeur de temps retranchée d'une autre valeur de temps fournit une durée. Les valeurs de temps sont utilisées pour régler les temps de durée de validité des temporisateurs.

L'origine du temps dépend de chaque système. Une unité de temps est la quantité de temps représenté en ajoutant une unité de durée à un temps.

## D.3.13 Sorte Sac (Bag)

### D.3.13.1 Définition

```
*/
value type Bag < type Itemsort >
  operators
    empty                    -> this Bag;
    "in" ( Itemsort, this Bag ) -> Boolean; /* est membre de */
    incl ( Itemsort, this Bag ) -> this Bag; /* inclure l'item dans l'ensemble */
    del ( Itemsort, this Bag ) -> this Bag; /* supprimer l'item de l'ensemble */
    "<" ( this Bag, this Bag ) -> Boolean; /* est un sous-sac propre de */
```



```

    ">" ( this Bag, this Bag ) -> Boolean; /* est un supersac propre de */
    "<=" ( this Bag, this Bag ) -> Boolean; /* est un sous-sac de */
    ">=" ( this Bag, this Bag ) -> Boolean; /* est un supersac de */
    "and" ( this Bag, this Bag ) -> this Bag; /* intersection de sacs */
    "or" ( this Bag, this Bag ) -> this Bag; /* union de sacs */
    length ( this Bag ) -> Integer;
    take ( this Bag ) -> Itemsort raise Empty;
axioms
    for all i,j in Itemsort (
        for all p,ps,a,b,c in Bag (
/* les constructeurs sont empty et incl */
/* égalités entre termes de constructeurs */
            incl(i,incl(j,p)) == incl(j,incl(i,p));
/* définition de "in" en l'appliquant à tous les constructeurs */
            i in <<type Bag>>empty == false;
            i in incl(j,ps) == i=j or i in ps;
/* définition de del en l'appliquant à tous les constructeurs */
            <<type Bag>>del(i,empty) == empty;
            i = j ==> del(i,incl(j,ps)) == ps;
            i /= j ==> del(i,incl(j,ps)) == incl(j,del(i,ps));
/* définition de "<" en l'appliquant à tous les constructeurs */
            a < <<type Bag>>empty == false;
            <<type Bag>>empty < incl(i,b) == true;
            incl(i,a) < b == i in b and del(i,a) < del(i,b);
/* définition de ">" à l'aide d'autres opérations */
            a > b == b < a;
/* définition de "=" en l'appliquant à tous les constructeurs */
            empty = incl(i,ps) == false;
            incl(i,a) = b == i in b and del(i,a) = del(i,b);
/* définition de "<=" and ">=" à l'aide d'autres opérations */
            a <= b == a < b or a = b;
            a >= b == a > b or a = b;
/* définition de "and" en l'appliquant à tous les constructeurs */
            empty and b == empty;
            i in b ==> incl(i,a) and b == incl(i,a and b);
            not(i in b) ==> incl(i,a) and b == a and b;
/* définition de "or" en l'appliquant à tous les constructeurs */
            empty or b == b;
            incl(i,a) or b == incl(i,a or b);
/* définition de length */
            length(<<type Bag>>empty) == 0;
            i in ps ==> length(ps) == 1 + length(del(i, ps));
/* définition de take */
            take(empty) == raise Empty;
            i in ps ==> take(ps) == i; ));
endvalue type Bag;
/*

```

### D.3.13.2 Utilisation

Les sacs sont utilisés pour représenter des ensembles multiples. Par exemple:

```
value type Boolset Bag< Boolean > endvalue type Boolset;
```

peut être utilisé pour une variable qui peut être empty ou contain (true), (false), (true, false) (true, true), (false, false),...

Les sacs sont utilisés pour représenter la construction SET OF de l'ASN.1.

## D.3.14 Sortes Bit et Bitstring de l'ASN.1

### D.3.14.1 Définition

```

*/
value type Bit
    inherits Boolean ( 0 = false, 1 = true );
    adding
    operators
        num ( this Bit ) -> Integer;
        bit ( Integer ) -> this Bit raise OutOfRange;

```

```

axioms
  <<type Bit>>num (0)           == 0;
  <<type Bit>>num (1)           == 1;
  <<type Bit>>bit (0)            == 0;
  <<type Bit>>bit (1)            == 1;
  for all i in Integer (
    i > 1 or i < 0 ==> bit (i) == raise OutOfRange;
  )
endvalue type Bit;
/* */
value type Bitstring
  operators
    bs in nameclass
      ''' ( (('0' or '1')*''B') or (('0':'9') or ('A':'F'))*''H') -> this Bitstring;
/*les opérateurs suivants qui sont les mêmes que ceux de String sauf que Bitstring
est indexé à partir de zéro*/
  is indexed from zero*/
  mkstring (Bit                               ) -> this Bitstring;
  make (Bit                                   ) -> this Bitstring;
  length ( this Bitstring                      ) -> Integer;
  first ( this Bitstring                       ) -> Bit;
  last ( this Bitstring                        ) -> Bit;
  "/" ( this Bitstring, this Bitstring         ) -> this Bitstring;
  extract ( this Bitstring, Integer            ) -> Bit raise InvalidIndex;
  modify ( this Bitstring, Integer, Bit       ) -> this Bitstring;
  substring ( this Bitstring, Integer, Integer) -> this Bitstring raise InvalidIndex;
/* substring (s,i,j) donne une chaîne de longueur j qui commence au ie élément */
  remove ( this Bitstring, Integer, Integer) -> this Bitstring;
/* remove (s,i,j) donne une chaîne avec une sous-chaîne de longueur j qui commence
au ie élément enlevé */
/*les opérateurs suivants sont spécifiques aux Bitstrings*/
  "not" ( this Bitstring                       ) -> this Bitstring;
  "and" ( this Bitstring, this Bitstring       ) -> this Bitstring;
  "or" ( this Bitstring, this Bitstring        ) -> this Bitstring;
  "xor" ( this Bitstring, this Bitstring       ) -> this Bitstring;
  "=>" ( this Bitstring, this Bitstring       ) -> this Bitstring
  num ( this Bitstring                        ) -> Integer;
  bitstring ( Integer                         ) -> this Bitstring raise OutOfRange;
  octet ( Integer                             ) -> this Bitstring raise OutOfRange;
axioms
/* Bitstring commence à l'indice 0 */
/* définition d'opérateurs ayant les mêmes noms que les opérateurs de String*/
  for all b in Bit ( /*b is bit in string*/
    for all s,s1,s2,s3 in Bitstring (
      for all i,j in Integer (
/* les constructeurs sont 'B', mkstring et "/" */
/* égalités entre termes de constructeurs */
        s // 'B == s;
        'B// s == s;
        (s1 // s2) // s3 == s1 // (s2 // s3);
/* définition de length en l'appliquant à tous les constructeurs */
        <<type Bitstring>>length('B) == 0;
        <<type Bitstring >>length(mkstring(b)) == 1;
        <<type Bitstring >>length(s1 // s2) == length(s1) + length(s2);
        make(s) == mkstring(s);
/* définition de extract en l'appliquant à tous les constructeurs,
avec les cas d'erreur traités séparément */
        extract(mkstring(b),0) == b;
        i < length(s1) ==> extract(s1 // s2,i) == extract(s1,i);
        i >= length(s1) ==> extract(s1 // s2,i) == extract(s2,i-length(s1));
        i<0 or i=>length(s) ==> extract(s,i) == raise InvalidIndex;
/* définition de first et last à l'aide d'autres opérations */
        first(s) == extract(s,0);
        last(s) == extract(s,length(s)-1);
/* définition de substring(s,i,j) par induction sur j,
les cas d'erreur sont traités séparément */
        i>=0 and i < length(s) ==>
          substring(s,i,0) == 'B;
/* */
        i>=0 and j>0 and i+j<=length(s) ==>
          substring(s,i,j) == substring(s,i,j-1) // mkstring(extract(s,i+j));

```

```

/* */
    i<0 or j<0 or i+j>length(s)      ==>
        substring(s,i,j) == raise InvalidIndex;
/* */
/* définition de modify à l'aide d'autres opérations */
    modify(s,i,b) == substring(s,0,i) // mkstring(b) // substring(s,i+1,length(s)-i);
/* définition de remove */
    remove(s,i,j) == substring(s,0,i) // substring(s,i+j,length(s)-i-j);
    ));
/* fin de la définition d'opérateurs de chaînes indexés à partir de zéro*/
/* */
/* définition de 'H et 'x'H en termes de 'B, 'xxxx'B pour les Bitstring*/
    <<type Bitstring>>'H == 'B;
    <<type Bitstring>>'0'H == '0000'B;
    <<type Bitstring>>'1'H == '0001'B;
    <<type Bitstring>>'2'H == '0010'B;
    <<type Bitstring>>'3'H == '0011'B;
    <<type Bitstring>>'4'H == '0100'B;
    <<type Bitstring>>'5'H == '0101'B;
    <<type Bitstring>>'6'H == '0110'B;
    <<type Bitstring>>'7'H == '0111'B;
    <<type Bitstring>>'8'H == '1000'B;
    <<type Bitstring>>'9'H == '1001'B;
    <<type Bitstring>>'A'H == '1010'B;
    <<type Bitstring>>'B'H == '1011'B;
    <<type Bitstring>>'C'H == '1100'B;
    <<type Bitstring>>'D'H == '1101'B;
    <<type Bitstring>>'E'H == '1110'B;
    <<type Bitstring>>'F'H == '1111'B;
/* */
/* définition des opérateurs spécifiques aux Bitstring */
    <<type Bitstring>>mkstring(0) == '0'B;
    <<type Bitstring>>mkstring(1) == '1'B;
/* */
    for all s, s1, s2, s3 in Bitstring (
        s = s == true;
        s1 = s2 == s2 = s1;
        s1 /= s2 == not ( s1 = s2 );
        s1 = s2 == true ==> s1 == s2;
        ((s1 = s2) and (s2 = s3)) ==> s1 = s3 == true;
        ((s1 = s2) and (s2 /= s3)) ==> s1 = s3 == false;
/* */
    for all b, b1, b2 in Bit (
        not('B) == 'B;
        not(mkstring(b) // s) == mkstring( not(b) ) // not(s);
/* définition de or */
/* la longueur de l'union (mise en or) de deux chaînes est la longueur maximale
des deux chaînes */
    'B or 'B == 'B;
    length(s) > 0 ==> 'B or s == mkstring(0) or s;
    s1 or s2 == s2 or s1;
    (b1 or b2) // (s1 or s2) == (mkstring(b1) // s1) or (mkstring(b2) // s2);
/* */
/* définition des opérateurs restants fondée sur "or" et "not" */
    s1 and s2 == not (not s1 or not s2);
    s1 xor s2 == (s1 or s2) and not(s1 and s2);
    s1 => s2 == not (s1 and s2);
    ));
/* */
/*définition de littéraux 'xxxxx'B*/
    for all s in Bitstring (
        for all b in Bit (
            for all i in Integer (
                <<type Bitstring>>num ('B) == 0;
                <<type Bitstring>>bitstring (0) == '0'B;
                <<type Bitstring>>bitstring (1) == '1'B;
                num (s // mkstring (b)) == num (b) + 2 * num (s);
                i > 1 ==> bitstring (i) == bitstring (i / 2) // bitstring (i mod 2);
                i >= 0 and i <= 255 ==> octet (i) == bitstring (i) or '00000000'B;

```

```

    i < 0                ==> bitstring (i) == raise OutOfRange;
    i < 0 or i > 255    ==> octet (i)    == raise OutOfRange;
  )))
/*définition de littéraux 'xxxxx'H */
  for all b1,b2,b3,h1,h2,h3 in bs nameclass (
    for all bs1, bs2, bs3, hs1, hs2, hs3 in Charstring (
      spelling(b1) = '''' // bs1 // ''B',
      spelling(b2) = '''' // bs2 // ''B',
      bs1 /= bs2                ==> b1 = b2    == false;
/* */
      spelling(h1) = '''' // hs1 // ''H',
      spelling(h2) = '''' // hs2 // ''H',
      hs1 /= hs2 ==> h1 = h2 == false;
      spelling(b1) = '''' // bs1 // ''B',
      spelling(b2) = '''' // bs2 // ''B',
      spelling(b3) = '''' // bs1 // bs2 // ''B',
      spelling(h1) = '''' // hs1 // ''H',
      spelling(h2) = '''' // hs2 // ''H',
      spelling(h3) = '''' // hs1 // hs2 // ''H',
      length(bs1) = 4,
      length(hs1) = 1,
      length(hs2) > 0,
      length(bs2) = 4 * length(hs2),
      h1 = b1                ==> h3 = b3    == h2 = b2;
/* */
/* connexion au générateur de String */
  for all b in Bit literals (
    spelling(b1) = '''' // bs1 // bs2 // ''B',
    spelling(b2) = '''' // bs2 // ''B',
    spelling(b)  = bs1                ==> b1    == mkstring(b) // b2;
  ));
endvalue type Bitstring;
/*

```

## D.3.15 Sortes Octet et Octetstring de l'ASN.1

### D.3.15.1 Définition

```

*/
syntype Octet = Bitstring size (8)
endsyntype Octet;
/* */
value type Octetstring
  inherits String < Octet > ( ''B = emptystring )
  adding
  operators
  os in nameclass
    '''' ( (((('0' or '1')8)*''B') or (((('0':'9') or ('A':'F'))2)*''H') )
    -> this Octetstring;
  bitstring ( this Octetstring ) -> Bitstring;
  octetstring ( Bitstring ) -> this Octetstring;
axioms
  for all b,b1,b2 in Bitstring (
    for all s in Octetstring (
      for all o in Octet(
        <<type Octetstring>> bitstring(''B)          == ''B;
        <<type Octetstring>> octetstring(''B)         == ''B;
        bitstring( mkstring(o) // s )              == o // bitstring(s);
/* */
    length(b1) > 0,
    length(b1) < 8,
    b2 == b1 or '00000000'B ==> octetstring(b1) == mkstring(b2);
/* */
    b == b1 // b2,
    length(b1) == 8                ==> octetstring(b) == mkstring(b1) // octetstring(b2);
  ));

```

```

/* */
  for all b1, b2 in Bitstring (
    for all o1, o2 in os nameclass (
      spelling( o1 ) = spelling( b1 ),
      spelling( o2 ) = spelling( b2 ) ==> o1 = o2 == b1 = b2
    ));
endvalue type Octetstring;
/*

```

### D.3.16 Exceptions prédéfinies

```

*/
exception
  OutOfRange,          /* une vérification d'intervalle a échoué. */
  InvalidReference,   /* Null a été incorrectement utilisé. Mauvais Pid pour ce
                       signal. */
  NoMatchingAnswer,   /* aucune réponse ne correspond dans une décision sans partie
                       else. */
  UndefinedVariable,  /* une variable "non définie" a été utilisée */
  UndefinedField,     /* un champ non défini d'un choix ou d'une struct a été accédée
                       */
  InvalidIndex,       /* on a accédé à une Chaîne (String) ou un Tableau (Array) avec
                       un indice incorrect. */
  DivisionByZero;     /* tentative de division entière ou réelle par zéro. */
  Empty;              /* aucun élément ne peut être retourné. */
/* */
endpackage Predefined;

```

## ANNEXE E

### Réservée aux exemples

## ANNEXE F

### Définition formelle

Etudes supplémentaires nécessaires pour le SDL 2000.

## APPENDICE I

### Etat de la Z.100, des documents et Recommandations associés

Le présent sous-paragraphe comporte à la liste de l'état de tous les documents associés concernant le SDL émis par l'UIT-T. La liste comprend toutes les parties des Recommandations Z.100, Z.105, Z.106, Z.107, Z.109 et tous les documents de méthodologie associés. Elle énumère également d'autres documents pertinents, tels que la Recommandation Z.110.

Cette liste doit être mise à jour par des moyens appropriés (par exemple un corrigendum) dès que des modifications du SDL sont convenues et que de nouveaux documents sont approuvés.

SDL 2000 est défini par les Recommandations suivantes approuvées par la Commission d'études 10 de l'UIT-T le 19 novembre 1999.

Z.100 (11/99) – SDL: langage de description et de spécification.

L'Annexe A de Z.100 – Index des non-terminaux en anglais.

L'Annexe D de Z.100 – Données prédéfinies de SDL.

*L'Annexe F est prévue pour novembre 2000.*

*Aucun plan spécifique au moment de l'approbation des Annexes B, C et E.*

Supplément 1 à Z.100 (05/97): SDL+ Méthodologie: mise à jour pour *SDL 2000 en cours*.

Z.105 (11/99) Langage SDL combiné avec des modules ASN.1 (SDL/ASN.1).

Z.106 (10/96) Format d'échange commun pour le langage de description et de spécification. *Mise à jour pour SDL 2000 en cours*.

Z.107 (11/99) Langage SDL avec notation ASN.1 incorporée.

Z.109 (11/99) Combinaison du langage SDL avec le langage de modélisation unifié (SDL/UML).

Z.110 (10/96) Critères d'utilisation des techniques de description formelle par l'UIT-T. *Mise à jour prévue pour novembre 2000*.

D'autres informations sur le langage SDL, y compris une liste de livres et publications s'y rapportant sont disponibles à l'adresse <http://www.sdl-forum.org>.

## APPENDICE II

### Directives concernant la maintenance du SDL

#### II.1 Maintenance du SDL

Le présent sous-paragraphe décrit la terminologie et les règles concernant la maintenance du SDL, convenues lors de la réunion de la Commission d'études du 10 novembre 1993, ainsi que la "procédure de demande de modification" associée.

##### II.1.1 Terminologie

- a) une **erreur** (*error*) est une inconsistance interne dans la Recommandation Z.100;
- b) une **correction textuelle** (*textual correction*) est une modification du texte ou des diagrammes de la Z.100 permettant de corriger des fautes de copiste ou de typographie;
- c) un **sujet ouvert** (*open item*) est un problème identifié mais non résolu. Un sujet ouvert peut être identifié par une demande de modification, par un accord de la Commission d'études ou par le groupe de travail;
- d) une **défaillance** (*deficiency*) est un sujet identifié pour lequel la sémantique du SDL n'est pas (clairement) définie par la Z.100;
- e) une **clarification** est une modification du texte ou des diagrammes de la Z.100 permettant de clarifier le texte ou les diagrammes qui pourraient être compris de manière ambiguë sans cette clarification. Il convient qu'une clarification tente de rapprocher la Z.100 de la sémantique du SDL telle qu'elle est comprise par la Commission d'études ou le groupe de travail;
- f) une **modification** est un changement apporté au texte ou aux diagrammes de la Z.100 qui modifie la sémantique du SDL;
- g) une **caractéristique sursitaire** (*decommitted feature*) est une caractéristique du SDL qui doit être supprimée du SDL lors de la prochaine révision de la Z.100;
- h) une **extension** est une nouvelle caractéristique, qui ne doit pas modifier la sémantique des caractéristiques définies dans la Z.100.

## II.1.2 Règles de maintenance

Dans le texte suivant, les références à la Z.100 doivent être considérées comme comprenant les annexes, appendices et suppléments, addendum, corrigendum ou guide d'implémentation, ainsi que les textes des Recommandations Z.105, Z.106, Z.107 et Z.109.

- a) lorsqu'une erreur ou une défaillance est décelée dans la Z.100, elle doit être corrigée ou clarifiée. Il convient que la correction d'une erreur implique des modifications aussi minimales que possible. Les corrections et les clarifications d'erreur seront insérées dans la liste principale des modifications de la Z.100 et prendront effet immédiatement;
- b) à l'exception des corrections d'erreur et des résolutions de sujets ouverts datant d'une période d'études précédente, il convient de considérer les modifications et les extensions uniquement comme le résultat d'une demande de modification soutenue par un groupe important d'utilisateurs. Après une demande de modification, il convient que la Commission d'études ou le groupe de travail enquête sur cette demande, en collaboration avec le groupe d'utilisateurs, de manière à clairement déterminer les besoins et les avantages, ainsi que la preuve qu'une des caractéristiques existantes du SDL n'est pas appropriée;
- c) les modifications et les extensions qui ne résultent pas de corrections d'erreurs doivent être largement présentées et l'opinion des utilisateurs et des développeurs doit être étudiée avant d'adopter la modification. Sauf si des circonstances particulières nécessitent l'implémentation de tels changements le plus tôt possible, ceux-ci ne seront pas recommandés avant la prochaine révision de la Z.100;
- d) jusqu'à ce qu'une version révisée de la Z.100 soit publiée, une liste principale des modifications de la Z.100 sera tenue à jour et couvrira la Z.100 et toutes ses annexes, à l'exception de la définition formelle. Les appendices, addenda, corrigenda, guides d'implémentation ou suppléments seront publiés tel que décidé par la Commission d'études. Pour assurer une diffusion efficace de la liste principale des modifications de la Z.100, elle sera publiée sous forme de Rapports COM et par des moyens électroniques appropriés;
- e) en ce qui concerne les défaillances dans la Z.100, il convient de consulter la définition formelle. Cela peut mener à une clarification ou à une correction qui est enregistrée dans la liste principale des modifications de la Z.100.

## II.1.3 Procédure de demande de modification

La procédure de demande de modification est destinée à permettre aux utilisateurs du SDL, au sein et à l'extérieur de l'UIT-T, de poser des questions concernant la signification exacte du SDL ou de la Z.100, de faire des suggestions de modifications du SDL ou de la Z.100 et à fournir des informations sur les modifications proposées. Le groupe d'experts du SDL doit publier les propositions de modification du SDL avant de les implémenter.

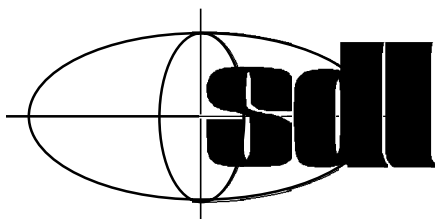
Il convient que les demandes de modification utilisent le formulaire de demande de modification (voir ci-dessous) ou fournissent les informations énumérées dans le formulaire. Il convient de clairement indiquer la nature de la modification (correction d'erreur, clarification, simplification, extension, modification ou caractéristique sursitaire). Il est également important d'indiquer le nombre d'utilisateurs soutenant la demande pour toute modification autre qu'une correction d'erreur.

Il convient d'étudier toutes les demandes de modification lors des réunions de la Commission d'études de l'UIT-T responsable de la Z.100. Pour les corrections ou les clarifications, les modifications peuvent être ajoutées dans la liste de corrections sans consulter les utilisateurs. Autrement, une liste des sujets ouverts est élaborée. Il convient de communiquer ces informations aux utilisateurs:

- sous forme de rapports de contribution de l'UIT-T;
- de courrier électronique dans les listes d'adresses SDL (telles que la liste informelle de l'UIT-T et [sdlnews@sdl-forum.org](mailto:sdlnews@sdl-forum.org));
- d'autres moyens convenus par les experts de la Commission d'études 10.

Il convient que les experts de la Commission d'études déterminent le niveau d'adhésion et d'opposition concernant chaque modification et évaluent les réactions des utilisateurs. Une modification ne sera inscrite dans la liste de modifications acceptées que s'il existe une forte adhésion de la part des utilisateurs et aucune objection sérieuse à cette proposition émise par seulement quelques utilisateurs. Enfin, toutes les modifications acceptées seront incorporées dans une version révisée de la Z.100. Il convient de souligner que tant que les modifications n'ont pas été incorporées et approuvées par la Commission d'études responsable de la Z.100, elles ne sont pas recommandées par l'UIT-T.





## Formulaire de demande de modification

Veuillez indiquer les éléments suivants		
Nature de la modification:	<input type="checkbox"/> correction d'erreur	<input type="checkbox"/> clarification
	<input type="checkbox"/> simplification	<input type="checkbox"/> extension
	<input type="checkbox"/> modification	<input type="checkbox"/> désengagement
Résumé succinct de la demande de modification:		
Brève justification de la demande de modification:		
Cette opinion est-elle partagée au sein de votre organisme:	<input type="checkbox"/> oui	<input type="checkbox"/> non
Avez-vous consulté d'autres utilisateurs:	<input type="checkbox"/> oui	<input type="checkbox"/> non
Combien d'utilisateurs représentez-vous?	<input type="checkbox"/> 1-5	<input type="checkbox"/> 6-10
	<input type="checkbox"/> 11-100	<input type="checkbox"/> plus de 100
Nom et adresse:		

*veuillez joindre d'autres feuillets d'explication si nécessaire*

SDL (Z.100) Rapporteur, c/o ITU-T, Place des Nations, CH-1211, Genève 20, Suisse.  
Fax: +41 22 730 5853, e-mail: [SDL.rapporteur@itu.int](mailto:SDL.rapporteur@itu.int)

## APPENDICE III

### Conversion systématique de SDL 92 en SDL 2000

Bien que toutes les spécifications du SDL 92 ne puissent être converties automatiquement en SDL 2000, une simple transformation devrait être suffisante dans de nombreux cas. La procédure suivante permet d'obtenir une conversion SDL/PR; une procédure similaire de transformation SDL/GR est également possible:

- 1) orthographe correcte concernant la casse et les nouveaux mots clés:
  - a) remplacer tous les mots clés avec le mot clé correspondant <lowercase keyword> (ou <uppercase keyword>);
  - b) remplacer tous les mots <word> contenant les caractères <national> tels que définis dans la Recommandation Z.100 (03/93) par un mot <word> unique;
  - c) remplacer tous les noms <name> par leur équivalent en lettres minuscules;
  - d) s'il y a un conflit entre les noms <name> et les mots clés <lowercase keyword>, remplacer le premier caractère par un caractère en majuscule.

Dans de nombreux cas, une procédure plus souple est possible, par exemple en utilisant systématiquement l'orthographe du nom <name> définissant l'occurrence de son identificateur <identifieur> correspondant. Ceci entraîne une modification sémantique uniquement si le nom d'un état est modifié et que ce nom est utilisé dans une expression <state expression>, comme décrit dans l'Addendum 1 du SDL 92;

- 2) remplacer tous les qualificatifs <qualifier> par le qualificatif <qualifier> correspondant du SDL 2000 (c'est-à-dire que la liste des éléments de trajet est toujours englobée dans les <composite special>, <qualifier begin sign> et <qualifier end sign>);
- 3) transformer toutes les utilisations des mots clés fpar et return dans les paramètres <agent formal parameters>, les paramètres <procedure formal parameters>, le retour <procedure return>, le paramètre <macro formal parameter>, les paramètres signature <formal operation parameters> et la signature <procedure signature> par la syntaxe SDL 2000 correspondante;
- 4) remplacer tous les acheminements de signaux par des définitions nodelay <channel definition>;
- 5) dans chaque bloc ne comportant pas d'acheminement des signaux ou de canaux, ajouter des canaux à chaque processus énumérant tous les signaux envoyés ou reçus par ce processus. Alternativement, ajouter des canaux implicites conformément au modèle relatif à l'acheminement des signaux implicites du SDL 92. Les spécifications concernant les canaux implicites tels qu'ils sont décrits dans l'Addendum 1 doivent également ajouter des accès aux blocs respectifs;
- 6) remplacer toutes les occurrences de subdivision de bloc. Le SDL 92 ne spécifiait pas la manière de sélectionner un sous-ensemble cohérent, cette étape peut donc nécessiter des connaissances extérieures. Une conversion partant du principe que la sous-structure devrait toujours être sélectionnée refléterait probablement l'utilisation type du SDL 92;
  - a) déplacer tous les blocs de la sous-structure directement dans le bloc conteneur;
  - b) s'il y a un conflit entre des entités du bloc et des entités de la sous-structure, renommer l'une des entités avec un nom unique;
  - c) régler tous les identificateurs des entités dans des blocs emboîtés pour utiliser le nouveau qualificatif;

- 7) remplacer toutes les actions de sortie utilisant `via all` par une liste d'actions de sortie. Si le trajet `<via path>` était un canal entre des ensembles d'instances de bloc, aucune transformation automatique n'est possible;
- 8) remplacer `service` et types de service respectivement par état composite et types d'état composite. Si les services ont des actions d'entrée qui se chevauchent (même si leurs ensembles d'entrée valides sont disjoints), celle des transitions dupliquées doit être éliminée. Éliminer tous les chemins de signaux entre les services; il convient que la sortie se rapportant à ces chemins de signaux se rapporte aux accès du type de processus. Remplacer `<stop>` par `<return>`. Les temporisateurs, les procédures exportées et les variables exportées d'un service doivent être définies dans un agent;
- 9) remplacer toutes les définitions de type de donnée impliquant des transformations de générateur par les définitions équivalentes utilisant des types paramétrés;
- 10) la transformation d'axiomes de données ne peut pas se faire automatiquement, mais il n'y a que quelques utilisateurs qui ont défini leurs propres types de données de manière axiomatique. Cependant, les cas suivants peuvent être facilement transformés:
  - a) les expressions de données prédéfinies restent valides, y compris l'utilisation de `String`, `Array` et `PowerSet`, après ajustement de la casse dans l'orthographe de leurs types;
  - b) une définition `newtype` avec des littéraux (sans opérateurs définis de manière axiomatique) peut être convertie en une définition `<value data type definition>` avec une liste `<literal list>`;
  - c) une définition `newtype` avec une propriété de structure peut être convertie en une définition `<value data type definition>` avec une définition `<structure definition>`.

Si une spécification en SDL 92 utilise des constructions qui ne peuvent pas être converties automatiquement en constructions SDL 2000 équivalentes, un contrôle rigoureux de cette spécification sera nécessaire si elle doit être conforme à la présente Recommandation.





## SÉRIES DES RECOMMANDATIONS UIT-T

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

**\*18356\***

Imprimé en Suisse  
Genève, 2000