UNION INTERNATIONALE DES TÉLÉCOMMUNICATIONS

UIT-T Z.100

SECTEUR DE LA NORMALISATION
DES TÉLÉCOMMUNICATIONS
DE L'UIT

(03/93)

LANGAGES DE PROGRAMMATION

LANGAGE DE DESCRIPTION ET DE SPÉCIFICATION DU CCITT

Recommandation UIT-T Z.100

(Antérieurement «Recommandation du CCITT»)

AVANT-PROPOS

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

La Conférence mondiale de normalisation des télécommunications (CMNT), qui se réunit tous les quatre ans, détermine les thèmes que les Commissions d'études de l'UIT-T doivent examiner et à propos desquels elles doivent émettre des Recommandations.

La Recommandation révisée UIT-T Z.100, élaborée par la Commission d'études X (1988-1993) de l'UIT-T, a été approuvée par la CMNT (Helsinki, 1-12 mars 1993).

_____

NOTES

1 Suite au processus de réforme entrepris au sein de l'Union internationale des télécommunications (UIT), le CCITT n'existe plus depuis le 28 février 1993. Il est remplacé par le Secteur de la normalisation des télécommunications de l'UIT (UIT-T) créé le 1er mars 1993. De même, le CCIR et l'IFRB ont été remplacés par le Secteur des radiocommunications.

Afin de ne pas retarder la publication de la présente Recommandation, aucun changement n'a été apporté aux mentions contenant les sigles CCITT, CCIR et IFRB ou aux entités qui leur sont associées, comme «Assemblée plénière», «Secrétariat», etc. Les futures éditions de la présente Recommandation adopteront la terminologie appropriée reflétant la nouvelle structure de l'UIT.

2 Dans la présente Recommandation, le terme «Administration» désigne indifféremment une administration de télécommunication ou une exploitation reconnue.

TABLE DES MATIÈRES

RÉSUMÉ

Champ d'application – Objectifs

La présente Recommandation définit le langage de description et de spécification du CCITT (SDL) (*specification and description language*) destiné à être utilisé dans les spécifications et descriptions non ambiguës des systèmes de télécommunication. Le champ d'application du SDL est précisé en 1.1.1. La présente Recommandation constitue un manuel de référence du langage.

Couverture

Le SDL fournit des concepts pour la description du comportement et des données, aussi bien que pour la structuration des 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 algébriques. Quant à la structuration, elle est fondée sur la décomposition hiérarchique et la hiérarchie des types. Ces fondements du SDL sont élaborés dans les paragraphes principaux 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 domaines d'application principaux pour lesquels le SDL a été conçu sont précisés en 1.1.2, mais le SDL convient généralement à la description des systèmes réactifs.

Etat/Stabilité

La présente Recommandation constitue un manuel de référence complet du langage, appuyé par des directives d'utilisation données dans l'Appendice I. L'Annexe F donne une définition formelle des règles sémantiques du langage.

Travaux associés

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

- A Index des mots clés non terminaux
- B Glossaire
- C Modèle algébrique initial
- D Données prédéfinies du SDL
- E Prévision des utilisations futures
- F Définition formelle

Il est également accompagné des appendices suivants:

- I Directives concernant la méthodologie du SDL
- II Bibliographie du SDL

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 dans l'Appendice III des références à des textes supplémentaires concernant le SDL, y compris des informations sur son utilisation industrielle.

Historique

Depuis 1976, le CCITT a recommandé différentes versions du SDL. La présente version est une révision de la Recommandation Z.100, UIT 1988.

Par comparaison avec le SDL défini dans la version de 1988, la version définie dans le présent texte a été étendue au domaine de la structuration orientée objet afin de s'adapter à la modélisation orientée objet des systèmes. D'autres extensions mineures ont été introduites, tout en veillant à ne pas rendre invalides les documents existants de 1988. Le paragraphe 1.6 donne les détails des modifications introduites.

Mots clés

Types de données abstraites, technique de description formelle, spécification fonctionnelle, représentation graphique, décomposition hiérarchique, orientation objet, technique de spécification, machine d'état.

LANGAGE DE DESCRIPTION ET DE SPÉCIFICATION DU CCITT

(Melbourne, 1988; révisée à Helsinki, 1993)

1 Introduction au SDL

Le texte du présent article n'est pas normatif; il vise plutôt à définir les conventions utilisées pour la description du SDL. L'utilisation du SDL dans cet article n'a qu'un caractère d'illustration. Les métalangages et conventions introduits ici ne visent que la description non ambiguë du SDL.

1.1 Introduction

Le but poursuivi, en recommandant l'utilisation du SDL (langage de description et de spécification), 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; et

b) la description d'un système est la description du comportement réel de celui-ci.

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 le texte qui suit est utilisé pour désigner à la fois le comportement souhaité et le comportement réel.

1.1.1 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 soumission des offres et le partage des commandes;

c) extensible pour permettre un développement ultérieur;

d) permettant l'application de plusieurs méthodologies de spécification et de conception de système, sans supposer *a priori* une méthodologie particulière quelconque.

1.1.2 Champ d'application

La présente Recommandation constitue le manuel de référence du SDL. L'Appendice I est un document relatif aux directives concernant la méthodologie et donne des exemples d'utilisation du SDL.

Le champ d'application principal du SDL est la description du comportement des systèmes en temps réel dans certains de leurs aspects. Ces applications comprennent:

a) le traitement des appels (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: alarme, relève automatique des dérangements, essais périodiques, etc.) dans les systèmes généraux de télécommunication;

c) la commande du système (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 au moyen de 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 spécifications d'une installation;

b) établir les spécifications d'un système;

c) établir des Recommandations du CCITT;

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 dans le détail);

g) décrire les essais d'un système;

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

1.1.3 Spécification d'un système

Le SDL décrit le comportement d'un système sous la forme de stimulus/réaction, étant admis que les stimuli aussi bien que les réactions sont des entités discrètes et contiennent de l'information. 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 étendue.

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.

1.2 Grammaires du 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 les deux des représentations concrètes de la même sémantique du SDL, elles sont équivalentes du point de vue sémantique. 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.

La Figure 1.1 montre les relations qui existent entre le SDL/PR, le SDL/GR, les grammaires concrètes et la grammaire abstraite.

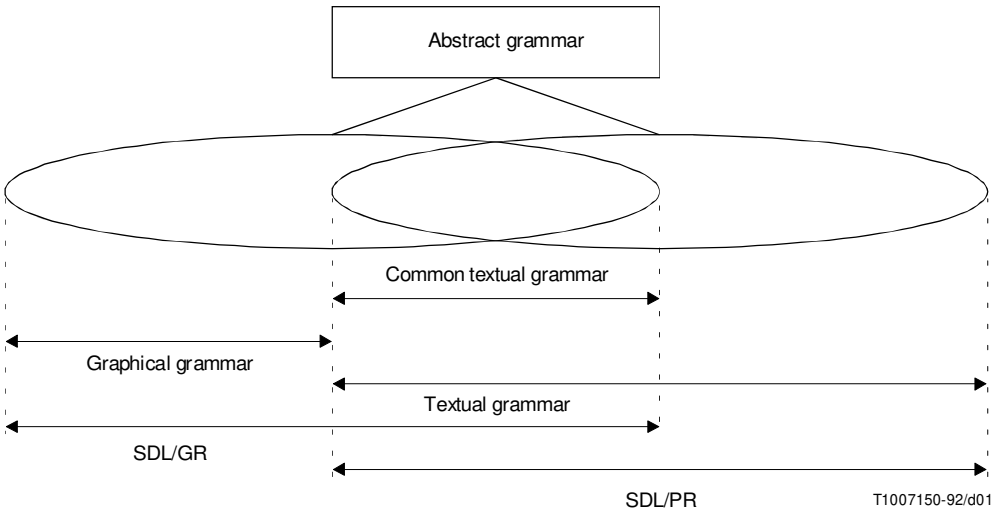


FIGURE 1.1/Z.100

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 héritera 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.

1.3 Définitions fondamentales

La présente Recommandation fait appel à des conventions et à des concepts généraux dont les définitions sont données dans les paragraphes suivants.

1.3.1 Type, définition et instance

Dans la présente Recommandation, les concepts de type, d'instance de type 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 1.2.

Dans le présent paragraphe, 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.

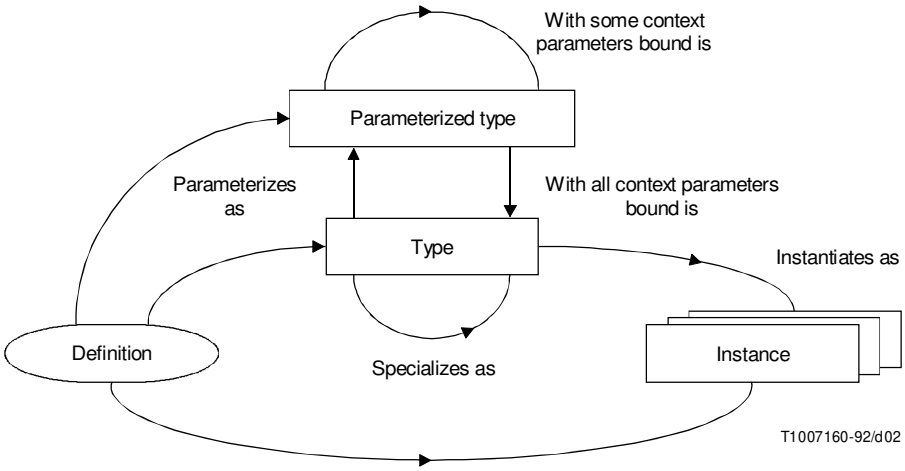


FIGURE 2.2/Z.100

Le concept du type

Les définitions introduisent des entités nommées qui sont soit des types, soit des instances. La définition d'un type définit toutes les propriétés qui lui sont associées. La définition d'une variable est un exemple de définition d'une instance. La définition d'un signal est un exemple de définition d'un 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 ou 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 qui peut être une instanciation 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 le type paramétré et le type non paramétré qui en découle.

Le type de données constitue une catégorie spéciale de type (voir 2.3 et 5).

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 . . .».

1.3.2 Environnement

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

On suppose qu'il y a une ou plusieurs instances de processus dans l'environnement et, par conséquent, les signaux circulant de l'environnement en direction du système ont des identités associées à ces instances de processus. Ces processus ont des valeurs PId différentes des autres valeurs PId du système (voir D.10).

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

1.3.3 Erreurs

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. Une interprétation d'une spécification de système qui conduit à une erreur indique que le comportement du système ne peut pas être déterminé à partir de la spécification.

1.4 Présentation

1.4.1 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 2.2.4).

d) *Sémantique* – Donnant une signification à une construction, confère les propriétés de cette construction, la façon avec laquelle 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 la correspondance avec les abréviations exprimées en termes de constructions en syntaxe concrète stricte précédemment définies.

f) *Exemples*.

1.4.2 Intitulés

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

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

La suite du présent 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 1.5.1.

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 correspond à la syntaxe abstraite définie par une autre section de texte numérotée.

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 seulement 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 au moyen de la Backus-Naur Form (BNF) étendue de la description de la syntaxe définie en 2.1/Z.200 (voir également le 1.5.2 de la présente Recommandation).

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 simple relation 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 sont significatives.

La syntaxe textuelle concrète qui ne constitue pas une forme abrégée (syntaxe dérivée modélisée par d'autres constructions SDL) 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 1.5.3.

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 simple relation entre les diagrammes de la grammaire graphique concrète et les définitions de la syntaxe abstraite. Lorsque le nom d'un non-terminal commence 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 commence par le mot *définition*, les deux règles représentent la même notion. Par exemple, <system diagram> dans la grammaire concrète correspond à *System-definition* dans la grammaire abstraite.

L'expansion dans la syntaxe concrète provenant de constructions telles les définitions référencées (2.4.1.3), les macros (4.2) et les mises en correspondance de littéraux (5.3.1.15) etc., doit être examinée avant la mise en correspondance entre la syntaxe concrète et la syntaxe abstraite. Ces expansions sont étudiées en détail en 7.

Sémantique

Des propriétés sont utilisées dans les règles de bonne formation qui font intervenir soit le type soit d'autres types qui se réfèrent à ce type.

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 *Signal-identifiers* d'entrée (dans l'ensemble des signaux d'entrée valides) apparaissent soit dans un *Save-signalset* ou dans un *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 l'article général traitant des identités en 2. 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» englobée des définitions de processus et éventuellement une définition de sous-structure de bloc.

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 la 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 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 1.3.3).

Modèle

Certaines constructions sont considérées comme étant une «syntaxe concrète dérivée» (ou une notation abrégée) 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.

Dans certains cas, une telle «syntaxe concrète dérivée», si elle est étendue, donnera lieu à une représentation immensément grande (éventuellement infinie). Néanmoins, la sémantique d'une telle spécification peut être déterminée.

Exemples

L'intitulé *Exemple(s)* contient des exemple(s).

1.5 Métalangages

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

Dans ce qui suit, on trouvera une introduction aux métalangages; des références renvoyant à des livres ou à des publications particulières de l'UIT seront données au besoin.

1.5.1 Le 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'une vue est view definition

View-definition :: *Variable-identifier*
Sort-reference-identifier

qui définit le domaine de l'objet composite (arborescence) appelé *View-definition*. Cet objet comporte deux sous-composantes qui à leur tour peuvent être des arborescences.

La définition en Méta IV

Process-identifier = *Identifier*

indique qu'un *Process-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 entiers

Exemple:

Number-of-instances :: *Intg* [*Intg*]

Le terme *Number-of-instances* désigne un domaine composite contenant une valeur entière obligatoire (*Intg*) et un entier facultatif ([*Intg*]) 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:

Destination = *Process-identifier* | *Service-identifier* | **ENVIRONMENT**

La *Destination* est soit un *Process-identifier*, soit un *Service-identifier* soit l'environnement qui est désigné par le mot clé **ENVIRONMENT**.

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 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 1.5.2) sont également utilisés dans la syntaxe abstraite: «*» 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:

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

Un *Process-graph* est constitué par un nœud *Process-start-node* et d'un ensemble de nœuds *State-node*.

1.5.2 Backus-Naur Form

Dans la Backus-Naur Form (BNF), un symbole terminal est soit celui qui n'est pas mis entre crochets angulaires (c'est-à-dire le signe inférieur à ou le signe supérieur à, <et>) ou est l'une des deux représentations <name> et <character string>. Notons que les deux terminaux spéciaux <name> et <character string> peuvent avoir également une sémantique telle que celle qui est définie ci-après.

Les crochets angulaires et le(s) mot(s) sont soit un symbole non-terminal ou l'un des deux terminaux <character string> ou <name>. 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:

<textual 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 symbole(s) non-terminaux et éventuellement un ou plusieurs symbole(s) terminaux. Par exemple <textual 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:

<block area> ::=
<block diagram>
| <existing typebased block definition>
| <graphical block reference>
| <graphical typebased block definition>

indique qu'une zone <block area> est soit une référence <graphical block reference>, un schéma <block diagram>, une définition <graphical typebased block definition> ou une définition <existing typebased block definition>.

Les éléments syntaxiques peuvent être regroupés au moyen d'accolades ({ et }), analogues aux parenthèses du Méta IV (voir 1.5.1). Un groupe entre accolades peut contenir une ou plusieurs barres verticales, indiquant des éléments syntaxiques possibles. Par exemple:

<block interaction area> ::=

{<block area> | <channel definition area>}+

La répétition 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 qu'une zone <block interaction area> contient au moins une zone <block area> ou <channel definition area> et peut contenir plusieurs autres zones <block area> et <channel definition area>.

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>.

1.5.3 Métalangage applicable à la grammaire graphique

En ce qui concerne la grammaire graphique, le métalangage décrit en 1.5.2 est complété avec les métasymboles 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:

{{<system text area>}* {<macro diagram>}* <block interaction area>
{<type in system area>}* }***set***

est un ensemble de zéro, d'une ou de plusieurs zones <system text area>; de zéro, d'un ou plusieurs diagrammes <macro diagram>, d'une zone <block interaction area> et de zéro, d'une ou de plusieurs zones <type in system area>.

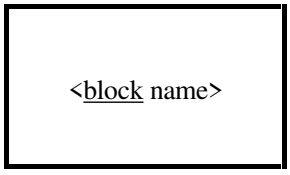
Tous les autres métasymboles sont des opérateurs infixes, ayant un symbole graphique non-terminal comme argument de gauche. L'argument de droite est soit un groupe d'éléments syntaxiques situés à l'intérieur d'accolades ou un seul élément syntaxique. Si le membre de droite d'une règle de production comporte 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 <text extension symbol> associé. Par exemple:

<graphical block reference> ::=
<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.

1.6 Différences avec la version SDL 88

Le langage défini dans la présente Recommandation est une extension de la Recommandation Z.100 publiée dans la version de 1988 du *Livre bleu*. Dans le présent paragraphe, on désignera par SDL 88 le langage défini dans le *Livre bleu*, alors que SDL 92 désignera le langage défini dans la présente Recommandation. Tous les efforts ont 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'aucune utilisation existante du SDL 88. De plus, les améliorations n'ont été acceptées que sur la base des besoins exprimés par plusieurs organes membres du CCITT.

Les extensions majeures ont été apportées dans le domaine de la programmation orientée objet. Alors que le SDL 88 est fondé sur les objets dans son modèle sous-jacent, on a ajouté quelques constructions de langage pour permettre au SDL 92 de prendre en charge de manière plus complète et uniforme le paradigme d'objet: (voir 2.4.1.2 et 6):

a) progiciels (2.4.1.2);

b) types de système, de bloc, de processus et de service (6.1.1);

c) (ensemble d') instances de système, de bloc, de processus et de service fondées sur les types (6.1.2);

d) paramétrisation des types au moyen de paramètres de contexte (6.2);

e) spécialisation des types et redéfinitions de types et transitions virtuels (6.3).

Les autres extensions sont: les transitions spontanées (2.6.6), le choix non déterministe (2.7.5), le symbole d'entrée et sortie interne dans le SDL/GR pour la compatibilité avec les diagrammes existants (2.9), un opérateur **any** non déterministe impératif (5.4.4.6), canal sans retard (2.5.1), appel de procédure distante (4.14), procédure retournant une valeur (5.4.5), entrée de champ de variable (2.6.4), définition d'opérateur (5.3.2), combinaison avec des descriptions de données externes (5.4.6), possibilités d'adressage externe en sortie (2.7.4), libre action dans les transitions (2.6.7), transition continue dans le même état avec la même priorité (4.11), connexions m:n de canaux et acheminement des signaux aux frontières de la structure (2.5.3). De plus, on a introduit un certain nombre de simplifications mineures de la syntaxe.

Il a été nécessaire dans peu de cas de modifier le SDL 88. Cela a été effectué à part 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 est également nécessaire si un document en SDL 88 contient des noms composés de mots qui sont des mots clés du SDL 92.

La sémantique de la construction **output** a été simplifiée; cela peut nuire à la validité de certaines utilisations spéciales de **output** dans les spécifications en SDL 88 (lorsqu'il n'existe pas de clause **to** et qu'il existe plusieurs accès possibles pour le signal). Quelques propriétés de la propriété d'égalité des sortes 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 contiennent des qualificateurs dans les expressions d'import ou introduisent plusieurs noms importés dans la même portée avec différentes sortes. Dans les cas (rares) où il est nécessaire de qualifier les variables importées pour résoudre la résolution par contexte, la correction 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 contiennent des qualificateurs dans les définitions de vues ou les expressions de vues. La correction consiste à supprimer ces qualificateurs. Il n'en résultera pas de modification dans la sémantique des expressions de vues, 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'est pas 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'agit là d'une modification conceptuelle mineure sans implications sur l'utilisation concrète (la syntaxe de l'acheminement de signal de service du SDL 88 est identique à celle de l'acheminement de signal 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** par exemple, peuvent paraître considérablement étendues, mais il faut noter que les extensions sont facultatives et leur utilisation n'est nécessaire que pour tirer profit de la puissance offerte par les extensions orientées objet (par exemple, 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.

2 Le SDL de base

2.1 Introduction

Un système SDL possède un ensemble de blocs. Les blocs sont connectés entre eux et à l'environnement par des canaux. A l'intérieur de chacun des blocs il y a un ou plusieurs processus. Ces processus communiquent entre eux par des signaux et sont supposés s'exécuter en parallèle.

L'article 2 a été subdivisé en neuf principaux sujets:

a) *Règles générales*

Les concepts de base tels les règles lexicales et les identificateurs, les règles de visibilité, les textes informels, la subdivision des diagrammes, les règles applicables aux dessins, les commentaires, les extensions de textes, les symboles de texte.

b) *Concepts de base concernant les données*

Les concepts de base concernant les données telles les valeurs, les variables, les expressions.

c) *Structure des systèmes*

Contient les concepts relatifs aux principes généraux de structuration du langage. Il s'agit des concepts de système, de bloc, de processus, de procédure.

d) *Communication*

Contient les mécanismes de communication tels le canal, l'acheminement du signal, le signal.

e) *Comportement*

Les constructions qui concernent le comportement d'un processus: règles de connectivité générales d'un processus ou graphe de procédure, définition de variable, départ, état, entrée, sauvegarde, transition spontanée, étiquette.

f) *Action*

Constructions actives telles les tâches, la création de processus, l'appel de procédure, la sortie, la décision.

g) *Temporisateurs*

Définition des temporisateurs et primitives des temporisateurs.

h) *Entrée et sortie intenses*

Abréviations pour assurer la compatibilité avec les anciennes versions du SDL.

i) *Exemples*

Il s'agit d'exemples concernant les autres points.

2.2 Règles générales

2.2.1 Règles lexicales

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

<lexical unit> ::=

<word>
| <character string>
| <special>
| <composite special>
| <note>
| <keyword>

<word> ::=

{<alphanumeric> | <full stop>}*
<alphanumeric>
{<alphanumeric> | <full stop>}*

<alphanumeric> ::=

<letter>
| <decimal digit>
| <national>

<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
| 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

<national> ::=

| ` | ¤ | @ | \
| <left square bracket>
| <right square bracket>
| <left curly bracket>
| <vertical line>
| <right curly bracket>
| <overline>
| <upward arrow head>

<left square bracket> ::= [

<right square bracket> ::=]

<left curly bracket> ::= {

<vertical line> ::= |

<right curly bracket> ::= }

<overline> ::= ~

<upward arrow head> ::= ^

<full stop> ::= .

<underline> ::= _

<character string> ::=

<apostrophe> {<alphanumeric>
| <other character> | <special>
| <full stop> | <underline>
| <space>
| <apostrophe><apostrophe>}* <apostrophe>

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

<text> ::=

{ <alphanumeric>
| <other character>
| <special>
| <full stop>
| <underline>
| <space>
| <apostrophe> }*

<apostrophe> ::= '

<other character> ::=

? | & | %

<special> ::=

+ | - | ! | / | > | * | (|) | " | , | ;
| < | = | :

<composite special> ::=

<< | >> | == | ==> | /= | <= | >= | // | := | =>
| -> | (. | .)

<note> ::=

/* <text> */

<keyword> ::=

active | **adding** | **all**
| **alternative** | **and** | **any**
| **as** | **atleast** | **axioms**
| **block** | **call** | **channel**
| **comment** | **connect** | **connection**
| **constant** | **constants** | **create**
| **dcl** | **decision** | **default**
| **else** | **endalternative** | **endblock**
| **endchannel** | **endconnection** | **enddecision**
| **endgenerator** | **endmacro** | **endnewtype**
| **endoperator** | **endpackage** | **endprocedure**
| **endprocess** | **endrefinement** | **endselect**
| **endservice** | **endstate** | **endsubstructure**
| **endsyntype** | **endsystem** | **env**
| **error** | **export** | **exported**
| **external** | **fi** | **finalized**
| **for** | **fpar** | **from**
| **gate** | **generator** | **if**
| **import** | **imported** | **in**
| **inherits** | **input** | **interface**
| **join** | **literal** | **literals**
| **macro** | **macrodefinition** | **macroid**
| **map** | **mod** | **nameclass**
| **newtype** | **nextstate** | **nodelay**
| **noequality** | **none** | **not**
| **now** | **offspring** | **operator**

	operators		**or**		**ordering**
	out		**output**		**package**
	parent		**priority**		**procedure**
	process		**provided**		**redefined**
	referenced		**refinement**		**rem**
	remote		**reset**		**return**
	returns		**revealed**		**reverse**
	save		**select**		**self**
	sender		**service**		**set**
	signal		**signallist**		**signalroute**
	signalset		**spelling**		**start**
	state		**stop**		**struct**
	substructure		**synonym**		**syntype**
	system		**task**		**then**
	this		**timer**		**to**
	type		**use**		**via**
	view		**viewed**		**virtual**
	with		**xor**		

Les caractères <national> sont représentés ci-dessus de la même façon que la version internationale de référence de l'Alphabet n° 5 du CCITT (Recommandation T.50). La responsabilité de la définition des représentations nationales de ces caractères relève des organismes nationaux de normalisation.

Les caractères de commande sont définis conformément à la Recommandation T.50. Une séquence de caractères de commande peut apparaître là où un <space> peut apparaître et a la même signification qu'un espace <space>. L'espace <space> représente le caractère espace de l'Alphabet n° 5 du CCITT.

L'occurrence d'un caractère de commande n'est pas significative dans un <informal text> ni dans une <note>. Un caractère de commande ne peut pas apparaître dans une chaîne de caractères si sa présence est significative. Dans ce cas, l'opérateur // et les lettres des caractères de commande doivent être utilisés.

Dans toutes les unités <lexical unit> à l'exception de <character string>, les <letter> sont toujours traitées comme des majuscules. (Le traitement des caractères <national> peut être défini par les organismes nationaux de normalisation.)

Une <lexical unit> se termine par le premier caractère qui ne peut pas faire partie de l'unité <lexical unit> conformément à la syntaxe spécifiée ci-dessus. Lorsqu'un caractère souligné <underline> est suivi d'un ou 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.

Le caractère / immédiatement suivi par le caractère * marque toujours le début d'une <note>. Le caractère * immédiatement suivi par le caractère / dans une <note> marque toujours la fin d'une <note>. Une <note> peut être insérée avant ou après toute unité <lexical unit>.

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

2.2.2 Règles de visibilité, noms et identificateurs

Grammaire abstraite

Identifier	::	*Qualifier Name*
Qualifier	=	*Path-item* +
Path-item	=	*System-qualifier* |
		Block-qualifier |
		Block-substructure-qualifier |
		Signal-qualifier |
		Process-qualifier |
		Service-qualifier |
		Procedure-qualifier |
		Sort-qualifier

System-qualifier :: *System-name*
Block-qualifier :: *Block-name*
Block-substructure-qualifier :: *Block-substructure-name*
Process-qualifier :: *Process-name*
Service-qualifier :: *Service-name*
Procedure-qualifier :: *Procedure-name*
Signal-qualifier :: *Signal-name*
Sort-qualifier :: *Sort-name*
Name :: *Token*

Grammaire textuelle concrète

<name>::=
<word> {<underline> <word> }*

<identifier> ::=
[<qualifier>] <name>

<qualifier> ::=
<path item> {/<path item>}*
| << <path item> {/<path item>}* >>

<path item> ::=
<scope unit kind> {<name> | <quoted operator> }

<scope unit kind> ::=
package
| **system type**
| **system**
| **block**
| **block type**
| **substructure**
| **process**
| **process type**
| **service**
| **service type**
| **procedure**
| **signal**
| **operator**
| **type**

Lorsqu'un caractère <underline> est suivi par un <word> dans un <name>, il est autorisé à spécifier un ou plusieurs caractères de commande ou espaces au lieu du caractère <underline>, pour autant que l'un des mots <word> englobant le caractère <underline> ne forme pas un <keyword>, par exemple A B désigne le même <name> que A_B. Cette règle ne s'applique pas à l'utilisation des <underline> et <space> dans des <character string>.

Toutefois, dans certains cas, l'absence de <underline> dans les noms <names> est ambiguë. Les règles suivantes s'appliquent donc:

1) les <underline> dans le <name> doivent être spécifiés explicitement;
2) lorsqu'un ou plusieurs <name> ou <identifier> peuvent être suivis directement par une <sort> (par exemple les définitions <variable definition>, <view definition>) les <underline> dans ces <name> ou <identifier> doivent être spécifiés explicitement;
3) lorsqu'une <data definition> contient des <generator transformations>, les <underline> du <sort name> suivant le mot clé **newtype** doivent être spécifiés explicitement;
4) les <underline> de <process name> d'un <process context parameter> avec une contrainte <process constraint> étant uniquement <process identifier> doivent être spécifiés explicitement;

5) les <underline> de <sort> dans <procedure result> doivent être spécifiés explicitement.

L'opérateur <quoted operator> n'est applicable que lorsque <scope unit kind> est opérationnel. Voir 5.3.2.

Il n'y a pas de syntaxe abstraite correspondante pour la classe <scope unit kind> indiquée par l'ensemble, le type de système, le type de bloc, le type de processus, le type de service ou l'opérateur.

Le <qualifier> 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.

Il est permis d'omettre certains <path item> les plus à gauche ou bien tout le <qualifier>. Lorsque le <name> désigne une entité de la classe d'entité contenant des variables, des synonymes, des littéraux et des opérateurs (voir la *sémantique* ci-après), l'association du <name> avec une définition doit pouvoir être résolue par le contexte réel. Dans d'autres cas, l'<identifier> 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'<identifier> est le même que la partie la plus à droite du <qualifier> complet désignant cette unité de portée. Si l'<identifier> ne contient pas de <qualifier>, la nécessité de mise en correspondance des <qualifier> n'est pas applicable.

La qualification par un supertype rend visibles les noms des types virtuels dans un supertype qui sont autrement cachés par la redéfinition dans le sous-type (voir 6.3.2). Il s'agit là des noms dans le supertype qui peuvent être qualifiés par un supertype.

Si un <qualifier> peut être compris à la fois comme qualifiant par une portée d'encadrement ou qualifiant par un supertype, il désigne une portée d'encadrement.

Un sous-signal doit être qualifié par ses signaux parents, si aucun autre signal visible (sans paramètres de contexte) n'existe à cet endroit qui porte le même <name>.

La résolution par contexte est possible dans les cas suivants:

a) l'unité de portée dans laquelle le <name> est utilisé n'est pas une définition <partial type definition> et contient une définition ayant ce <name>. Le <name> sera lié à cette définition;

b) l'unité de portée dans laquelle le <name> est utilisé ne contient pas de définition ayant ce <name> ou l'unité de portée est une définition <partial type definition>, et il existe exactement une définition visible d'une entité qui a le même <name> et à laquelle le <name> peut être lié sans violer aucune des propriétés statiques (compatibilité de sorte, etc.) de la construction dans laquelle le <name> apparaît. Le <name> sera lié à cette définition.

Seuls les identificateurs visibles peuvent être utilisés, à l'exception de l'<identifier> utilisé à la place d'un <name> dans une définition référencée (c'est-à-dire une définition extraite de la définition <system definition>).

Sémantique

Les unités de portée sont définies par le schéma suivant:

Grammaire textuelle concrète	*Grammaire graphique concrète*
<package definition>	<package diagram>
<textual system definition>	<system diagram>
<system type definition>	<system type diagram>
<block definition>	<block diagram>
<block type definition>	<block type diagram>
<process definition>	<process diagram>
<process type definition>	<process type diagram>
<service definition>	<service diagram>
<service type definition>	<service type diagram>
<procedure definition>	<procedure diagram>
<block substructure definition>	<block substructure diagram>
<channel substructure definition>	<channel substructure diagram>
<partial type definition>	
<operator definition>	<operator diagram>
<sort context parameter>	
<signal definition>	
<signal context parameter>	

Une liste de définitions est associée à une unité de portée. Chaque définition définit une entité appartenant à une certaine classe d'entité et ayant un nom associé. La liste contient les définitions <gate definition>, les paramètres <formal context parameter>, les paramètres <formal parameter>, les paramètres <formal variable parameter> et les définitions des sous-structures contenues dans l'unité de portée. Pour une <partial type definition>, la liste de définitions associée comprend les <operator signature>, les <literal signature> et toutes <operator signature> et <literal signature> provenant d'une sorte parente, d'un générateur d'instance ou imposé par l'utilisation d'abréviations telles que <specialization> ou <ordering>.

Bien que les <quoted operator>, les <operator name> avec un point d'exclamation <exclamation> et les <character string> 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>. Toutefois, les <state name>, les <connector name>, les <gate name> qui se présentent dans les définitions du chemin de signal et des canaux, les <generator formal name>, les <value identifier> dans les équations, les <macro formal name> et les <macro name> ont des règles de visibilité particulières et ne peuvent par conséquent être qualifiés. Les noms <state name> et <connector name> ne sont pas visibles à l'extérieur d'un <body> unique. D'autres règles de visibilité particulières sont expliquées dans les paragraphes concernés.

Chaque entité est dite avoir son contexte de définition dans l'unité de portée qui la définit. Les entités sont référencées au moyen d'<identifier>.

Le <qualifier> dans un <identifier> spécifie uniquement le contexte de définition du <name>.

Les classes d'entités sont les suivantes:

a) progiciels;
b) système;
c) types de système;
d) types de bloc;
e) blocs;
f) canal, acheminement de signaux, portes;
g) sous-structures de bloc, sous-structures de canal;
h) signaux, temporisateurs;
i) types de processus;
j) processus;
k) types de service;
l) services;
m) procédures, procécures distantes;
n) variables (y compris les paramètres formels), synonymes, littéraux, opérateurs;
o) variables distantes;
p) sortes;
q) générateurs;
r) listes de signaux;
s) vue.

Un <identifier> est dit être visible dans une unité de portée

a) si la partie nom de l'<identifier> a son contexte de définition dans cette unité de portée et l'<identifier>
    1) ne figure pas dans une <gate definition>, dans un <actual context parameter>, dans des <parameters of sort>, dans des <formal variable parameters> ou dans une <specialization>, ou
    2) désigne un <formal context parameter>; ou
b) s'il est visible dans l'unité de portée qui définit cette unité de portée; ou
c) si l'unité de portée contient une <partial type definition> dans laquelle l'<identifier> est défini; ou
d) si l'unité de portée contient une <signal definition> dans laquelle l'<identifier> se trouve défini; ou
e) si l'unité de portée possède une <package reference clause> a travers laquelle l'<identifier> est rendu visible (comme défini en 2.4.1.2).

On ne peut avoir deux définitions dans la même unité de portée et appartenant à la même classe d'entités portant le même <name>. Il existe cependant une exception unique: les définitions des opérateurs et de littéraux dans la même <partial type definition>. Ces opérateurs et littéraux peuvent avoir le même <name> avec différentes <argument sort> ou différentes sortes de <result>.

Dans la grammaire textuelle concrète, le nom ou l'identificateur optionnel dans une définition après le mot 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.).

2.2.3 Texte informel

Grammaire abstraite

Informal-text :: ...

Grammaire textuelle concrète

<informal text> ::=
<character string>

Sémantique

Lorsqu'un texte informel est utilisé dans une spécification SDL de système, il signifie que le texte n'est pas en SDL formel, c'est-à-dire que le SDL ne fournit pas de sémantique. La sémantique du texte informel peut être définie par d'autres moyens.

2.2.4 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, c'est-à-dire le <channel symbol>, le <signal route symbol>, le <create line symbol>, le <flow line symbol>, le <solid association symbol> et le <dashed association symbol>, qui peuvent se couper. Il n'existe pas d'association logique entre les symboles qui se coupent.

Le métasymbole ***is followed by*** implique un <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 <flow line symbol> entre dans un autre <flow line symbol>, dans un <out-connector symbol> ou dans une <nextstate area>. Dans d'autres cas, ces flèches sont facultatives sur les <flow line symbol>. Les <flow line symbol> sont horizontaux ou verticaux.

On peut utiliser des images symétriques verticales des <input symbol>, des <output symbol>, des <internal input symbol>, des <internal output symbol>, des <priority input symbol>, des <comment symbol> et des <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.

2.2.5 Subdivision des diagrammes

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

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

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

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

<kernel heading> ::=
[<virtuality>] [**exported**]
<diagram kind> {<diagram name> | <diagram identifier>}

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

<page number> ::=
<literal name>

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

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

L'utilisateur peut choisir les <frame symbol> imposés par la limite du support sur lequel les diagrammes sont reproduits.

Afin d'avoir une séparation nette entre la <heading area> et la <page number area>, le symbole <implicit text symbol> n'est pas matérialisé, il est sous-entendu. La <heading area> est placée au coin supérieur gauche du <frame symbol>, la <page number area> est située au coin supérieur droit du <frame symbol>. Le <heading> et la <syntactical unit> dépendent du type de diagramme.

Les <additional heading> doivent être présentés sur la première page du diagramme, mais ils sont facultatifs sur les pages suivantes. Les <heading> et les <diagram kind> sont élaborés pour les diagrammes particuliers dans les paragraphes individuels de la présente Recommandation. La présente Recommandation ne définit pas les <additional heading>.

La <virtuality> désigne la virtualité du type défini par le diagramme (voir 6.3.2) et **exported** indique si une procédure donnée est exportée en tant que procédure distante (voir 4.14).

2.2.6 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>.

Des exemples sont donnés aux Figures 2.10.1 à 2.10.4.

La syntaxe concrète de la deuxième forme est:

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

<comment> ::=
comment <character string>

Un exemple est donné à la Figure 2.10.2.

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 <dashed association symbol> doit être reliée au milieu du segment vertical du <comment symbol>.

Un <comment symbol> peut être relié à tout symbole graphique au moyen d'un <dashed association symbol>. Le <comment symbol> est considéré comme étant 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.

Le <text> dans *la grammaire graphique concrète* correspond à <character string> dans *la grammaire textuelle concrète* sans les <apostrophe> de fermeture.

Un exemple est donné à la Figure 2.10.4.

2.2.7 Extension de texte

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

<text extension symbol> ::=
<comment symbol>

<solid association symbol> ::=

———————

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

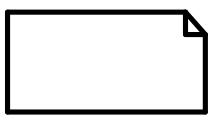
Un <text extension symbol> peut être relié à tout symbole graphique au moyen d'un <solid association symbol>. Le <text extension symbol> est considéré comme étant un symbole fermé en complétant (par la pensée) le rectangle.

Le texte contenu dans le <text extension symbol> est la suite du texte dans le symbole graphique et est considéré comme étant contenu dans ce symbole.

2.2.8 Symbole de texte

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

<text symbol> ::=



2.3 Concepts de base concernant les données

Le concept de données dans le SDL est exposé dans le détail en 5; plus précisément en ce qui concerne la terminologie du SDL pour ce qui est des données, des concepts de définition de nouvelles sortes et les données prédéfinies.

2.3.1 Définitions des types de données

Les données dans le SDL sont principalement abordées sous l'aspect sortes. Une sorte définit un ensemble de valeurs, un ensemble d'opérateurs qui peut être appliqué à ces valeurs, et un ensemble de règles algébriques (équations) qui définissent le comportement de ces opérateurs lorsqu'ils sont appliqués aux valeurs. Les valeurs, les opérateurs et les règles algébriques définissent tous ensemble les propriétés de la sorte. Ces propriétés sont définies par les définitions des sortes.

Le SDL permet la définition de toute sorte qui est nécessaire, y compris les mécanismes de composition (type composite) avec pour seule contrainte qu'une telle définition puisse être spécifiée de manière formelle. En revanche, pour les langages de programmation, il y a des considérations de mise en œuvre qui nécessitent que l'ensemble des sortes disponibles et, en particulier, les mécanismes de composition mis à disposition soient limités (tableau, structure, etc.).

2.3.2 Variables

Les variables sont des objets qui peuvent être associés à une valeur par une affectation. Quand on accède à la variable, on renvoie la valeur associée.

2.3.3 Valeurs et littéraux

Un ensemble de valeurs avec certaines caractéristiques est appelé sorte. Les opérateurs sont définis à partir et vers les valeurs des sortes. Par exemple, l'application de l'opérateur somme («+») à partir et vers les valeurs de la sorte entière est valable, tandis que la somme de la sorte booléenne ne l'est pas.

Chaque valeur appartient à une sorte unique, c'est-à-dire que les sortes n'ont jamais de valeurs en commun.

Pour la plupart des sortes, il y a des formes littérales qui décrivent les valeurs de la sorte (par exemple, pour les entiers «2» est utilisé de préférence à «1 + 1»). Il peut y avoir plusieurs littéraux qui décrivent la même valeur (par exemple, 12 et 012 peuvent être utilisés pour décrire la même valeur entière). 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 littéraux; par exemple, une valeur composite n'a souvent pas de littéraux en propre mais a des valeurs qui sont définies par des opérations de composition sur les valeurs de ses composantes.

2.3.4 Expressions

Une expression décrit une valeur. Si une expression ne contient pas de variable ou d'opérateur obligatoire, par exemple, si elle est un littéral d'une sorte donnée, chaque occurrence de l'expression décrira toujours la même valeur. Une expression qui contient des variables ou des opérateurs obligatoires peut être interprétée comme différentes valeurs durant l'interprétation d'un système SDL selon la valeur associée aux variables.

2.4 Structure du système

2.4.1 Organisation des spécifications SDL

2.4.1.1 Cadre d'application

Une <sdl specification> peut être décrite comme une <system definition> monolithique 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 progiciels qui peuvent être indépendants. Une <referenced definition> est une définition qui a été déplacée de son contexte de définition pour avoir une vue d'ensemble à l'intérieur d'une description de système. Elle est similaire à une définition de macro (voir 4.2) mais elle est «insérée» en une place unique (le contexte de définition) utilisant une référence.

Grammaire concrète

<sdl specification> ::=

<package list> [<system definition> {<referenced definition>}*]

2.4.1.2 Progiciel

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

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

Les définitions à l'intérieur d'un progiciel sont rendues visibles à un système ou à un autre progiciel par une *clause de référence de progiciel* (*package reference clause*).

Grammaire textuelle concrète

```
<package list> ::=
                        { <package> {<referenced definition>}* }*

<package> ::=
                        <package definition> | <package diagram>

<package definition> ::=
                        {<package reference clause>}*
                        package <package name>
                              [<interface>] <end>
                              {<entity in package>}*
                        endpackage [<package name>] <end>

<entity in package> ::=
                        <system type definition>
                      | <textual system type reference>
                      | <block type definition>
                      | <textual block type reference>
                      | <process type definition>
                      | <textual process type reference>
                      | <procedure definition>
                      | <remote procedure definition>
                      | <textual procedure reference>
                      | <signal definition>
                      | <signal list definition>
                      | <service type definition>
                      | <textual service type reference>
                      | <select definition>
                      | <remote variable definition>
                      | <data definition>
                      | <macro definition>

<package reference clause> ::=
                        use <package name> [ / <definition selection list> ] <end>

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

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

<entity kind> ::=
                        system type
                      | block type
                      | process type
                      | service type
                      | signal
                      | procedure
                      | newtype
                      | signallist
```

| **generator**
| **synonym**
| **remote**

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

newtype est également utilisé pour sélectionner un nom de type synonyme dans un progiciel. La classe <entity kind> **remote** est utilisée pour la sélection d'une définition de variable distante.

Grammaire graphique concrète

<package diagram> ::=
<package reference area>
is associated with
<frame symbol> ***contains***
{ <package heading>
{{<package text area>}*
{<diagram in package> }*} ***set*** }

<package reference area> ::=
<text symbol> ***contains*** {<package reference clause>}*

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

<package text area> ::=
<text symbol> ***contains***
{ <entity in package>}*

<diagram in package> ::=
<system type diagram>
| <system type reference>
| <type in system area>
| <macro diagram>
| <option area>

La <package reference area> doit être placée en tête du <frame symbol>.

Sémantique

Pour chaque <package name> mentionné dans une <package reference clause>, un <package> correspondant doit exister. Si ce <package> ne fait pas partie d'une <package list>, il doit exister un mécanisme d'accès au <package> référencé, comme s'il s'agissait d'une partie textuelle de la <package list>. La présente Recommandation ne définit pas ce mécanisme.

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

Le nom d'une entité définie à l'intérieur d'un <package> est visible pour un autre <package> si et seulement si:

1) le <package> ou la <system definition> possède une <package reference clause> qui mentionne le <package> et la <definition selection list> de la <package reference clause> est omise ou le nom est mentionné dans une <definition selection>; et
2) l'<interface> du <package> définissant le nom est omise, ou bien le nom est mentionné dans l'<interface>.

La résolution par contexte dans les expressions tiendra également compte des sortes dans les progiciels qui ne sont pas rendues visibles par une <package reference clause>. Les signaux qui ne sont pas rendus visibles dans une clause **use** peuvent faire partie d'une liste de signaux via un <signal list identifier> rendu visible dans une clause **use**. Ces signaux pourront alors affecter l'ensemble complet de signaux d'entrée valides d'un processus ou d'un service.

Si un nom dans une <definition selection> désigne une <sort>, la <definition selection> désigne aussi implicitement tous les littéraux et opérateurs définis pour la <sort> ou l'identificateur <parent sort identifier> dans le cas d'un type synonyme.

Si un nom dans une <definition selection> désigne un signal, la <definition selection> désigne aussi implicitement tous les sous-signaux de ce signal.

La référence aux noms ayant leur occurrence de définition dans un <package> se fait au moyen d'identificateurs <identifier> ayant le **package** <package name> comme le <path item> le plus à gauche. Toutefois, le <path item> peut être omis si le <package name> désigne le progiciel englobant, si la paire (le nom, le genre d'entité) est visible uniquement d'un seul progiciel, ou si la résolution par contexte s'applique.

Le genre <entity kind> dans une <definition selection> désigne le genre d'entité de <name>. Toute paire (<entity kind>,<name>) doit être distincte à l'intérieur d'une <definition selection list>. Pour une <definition selection> dans une <interface>, le genre <entity kind> peut être omis uniquement s'il n'y a pas d'autre nom ayant son occurrence de définition directement dans le <package>. Pour une <definition selection> dans une <package reference clause>, <entity kind> peut être omis si et seulement si exactement une entité de ce nom est mentionnée dans n'importe quelle <definition selection list> pour le progiciel, ou si le progiciel n'a pas de <definition selection list> et contient directement une définition unique de ce nom.

Une définition <system definition> et toute définition <package definition> possède une <package reference clause> implicite:

use `Predefined`;

où Predefined désigne un progiciel contenant les données prédéfinies selon l'Annexe D.

Modèle

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

Une <sdl specification> avec une définition <system definition> et une <package list> non vide correspond à une <system definition> où

1) toutes les occurrences du même nom (y compris l'occurrence de définition) d'une entité définie dans un <package> ont été renommées par le même nom unique et anonyme;

2) toutes les définitions à l'intérieur des <package> ont été incluses au niveau du système;

3) toutes les occurrences de **package** <package name> dans les qualificateurs ont été remplacées par **system** <system name>, où <system name> est le nom utilisé pour la définition <system definition>.

La transformation est expliquée en détail en 7. La relation entre <system definition> et *Grammaire abstraite* est définie en 2.4.2.

2.4.1.3 Définitions référencées

Grammaire concrète

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

<system definition> ::=
{<textual system definition> | <system diagram>}

<definition> ::=

<system type definition>
| <block definition>
| <block type definition>
| <process definition>
| <process type definition>
| <service definition>
| <service type definition>
| <procedure definition>
| <block substructure definition>
| <channel substructure definition>
| <macro definition>
| <operator definition>

<diagram> ::=

<system type diagram>
| <block diagram>
| <block type diagram>
| <process diagram>
| <process type diagram>
| <service diagram>
| <service type diagram>
| <procedure diagram>
| <block substructure diagram>
| <channel substructure diagram>
| <macro diagram>
| <operator diagram>

Pour chaque <referenced definition>, sauf pour des <macro definition> et des <macro diagram>, on doit avoir une référence dans le <package> ou la <system definition> associé.

Dans une <referenced definition>, un <identifier> est placé immédiatement après le(s) mot(s) clé(s) initial (initiaux). Pour chaque référence, il doit exister une <referenced definition> avec le même genre d'entité que la référence, dont le <qualifier>, s'il est présent, désigne un trajet à partir de l'unité de portée englobant la référence jusqu'à la référence. Si deux <referenced definition> du même genre d'entité ont le même nom, le <qualifier> d'un des identificateurs <identifier> ne doit pas constituer la partie la plus à gauche de l'autre <qualifier>.

On ne peut pas spécifier un <qualifier> dans l'<identifier> après le mot clé initial pour les définitions qui ne sont pas des <referenced definition> (c'est-à-dire qu'un <name> doit être spécifié pour les définitions normales).

Sémantique

Avant d'établir les propriétés d'une définition <system definition>, chaque référence est remplacée par la <referenced definition> correspondante. Dans cette substitution, l'<identifier> de la <referenced definition> est remplacé par le <name> dans la référence.

Modèle

La relation entre la définition <referenced definition> et la *grammaire abstraite* est donnée en 7.

2.4.2 *Système*

Grammaire abstraite

System-definition	::	*System-name*
		Block-definition-**set**
		Channel-definition-**set**
		Signal-definition-**set**
		Data-type-definition
		Syntype-definition-**set**
System-name	=	*Name*

Une définition *System-definition* a un nom qui peut être utilisé dans les qualificateurs.

La définition *System-definition* doit au moins contenir une définition *Block-definition*.

Les définitions de tous les signaux, canaux, types de données, types de synonymes utilisés dans l'interface avec l'environnement et entre les blocs du système sont contenues dans la définition *System-definition*.

Grammaire textuelle concrète

<textual system definition> ::=
{<package reference clause>}*
{ **system** <system name><end>
{<entity in system>}+
endsystem [<system name>] <end>
| <textual typebased system definition> }

<entity in system> ::=
<block definition>
| <textual block reference>
| <textual typebased block definition>
| <channel definition>
| <signal definition>
| <signal list definition>
| <select definition>
| <macro definition>
| <remote variable definition>
| <data definition>
| <textual block type reference>
| <block type definition>
| <process type definition>
| <textual process type reference>
| <procedure definition>
| <textual procedure reference>
| <remote procedure definition>
| <service type definition>
| <textual service type reference>

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

Un exemple de <textual system definition> est donné à la Figure 2.10.5.

Grammaire graphique concrète

<system diagram> ::=
[<package reference area>]
is associated with
{<frame symbol> ***contains***
{<system heading>
{{<system text area>}*
{<macro diagram>}*
<block interaction area>
{<type in system area>}*} ***set*** }
| <graphical typebased system definition>}

<frame symbol> ::=

<system heading> ::=
system <system name>

<system text area> ::=
<text symbol> ***contains***
{ <signal definition>
| <signal list definition>
| <remote variable definition>
| <data definition>
| <remote procedure definition>
| <macro definition>
| <select definition>}*

<block interaction area> ::=
{ <block area> | <channel definition area>}+

<block area> ::=
<graphical block reference>
| <block diagram>
| <graphical typebased block definition>
| <existing typebased block definition>

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

<block symbol> ::=

La zone <package reference area> doit être placée en haut du symbole de cadre du système.

*Block-definition-***set** dans la *grammaire abstraite* correspond aux zones <block area>, et *Channel-definition-***set** correspond aux zones <channel definition area>.

Un exemple de <system diagram> est donné à la Figure 2.10.6.

Sémantique

Une *System-definition* est la représentation en SDL d'une spécification ou de la description d'un système.

Un système est séparé de son environnement par une frontière de système et contient un ensemble de blocs. La communication entre le système et l'environnement, ou entre les blocs intérieurs au système ne peut se faire qu'au moyen de signaux. A l'intérieur d'un système, ces signaux sont véhiculés dans des canaux. Les canaux relient les blocs entre eux ou à la frontière du système.

Avant d'interpréter une *System-definition*, on choisit un sous-ensemble cohérent (voir 3.2.1). Ce sous-ensemble est appelé une instance de la *System-definition*. Une instance de système est une instanciation d'un type de système défini par une *System-definition*. L'interprétation d'une instance d'une *System-definition* est réalisée par une machine abstraite SDL qui en conséquence donne la sémantique aux concepts du SDL. Pour interpréter une instance d'une *System-definition*, il faut:

a) initialiser le temps du système;

b) interpréter les blocs et les canaux qui leur sont reliés et qui sont contenus dans le sous-ensemble de subdivision cohérent choisi.

2.4.3 *Bloc*

Grammaire abstraite

Block-definition	::	*Block-name*
		Process-definition-**set**
		Signal-definition-**set**
		Channel-to-route-connection-**set**
		Signal-route-definition-**set**
		Data-type-definition
		Syntype-definition-**set**
		[*Block-substructure-definition*]
Block-name	=	*Name*

A moins qu'une *Block-definition* ne contienne une *Block-substructure-definition*, on doit avoir au moins une *Process-definition* à l'intérieur du bloc.

Il est possible de subdiviser les blocs en spécifiant la *Block-substructure-definition*; cet aspect du langage est étudié en 3.2.2.

Grammaire textuelle concrète

```
<block definition> ::=
                    block {<block name>|<block identifier>} <end>
                        { <channel to route connection> | <entity in block>}*
                        [       <block substructure definition>
                            |   <textual block substructure reference>]
                    endblock [ <block name> | <block identifier> ] <end>

<entity in block> ::=
                        <signal definition>
                    |   <signal list definition>
                    |   <process definition>
                    |   <textual process reference>
                    |   <textual typebased process definition>
                    |   <signal route definition>
                    |   <macro definition>
                    |   <remote variable definition>
                    |   <data definition>
                    |   <select definition>
                    |   <process type definition>
                    |   <textual process type reference>
```

| <block type definition>
| <textual block type reference>
| <textual procedure reference>
| <procedure definition>
| <remote procedure definition>
| <service type definition>
| <textual service type reference>

<textual process reference> ::=
process <process name> [<number of process instances>] **referenced** <end>

Un exemple de <block definition> est montré à la Figure 2.10.7.

Grammaire graphique concrète

<block diagram> ::=
<frame symbol>
contains {<block heading>
{ {<block text area>}* {<macro diagram>}*
{ <type in block area> }*
[<process interaction area>] [<block substructure area>]}***set*** }
is associated with {<channel identifiers>}*

Les identificateurs <channel identifier> identifient des canaux reliés à des acheminements de signaux dans le <block diagram>. Les identificateurs <channel identifier> sont placés à l'extérieur du <frame symbol>, à proximité de l'extrémité du trajet du signal arrivant au <frame symbol>. Si le <block diagram> ne contient pas une zone <process interaction area>, il doit alors contenir une zone <block substructure area>.

<block heading> ::=
block {<block name> | <block identifier> }

<block text area> ::=
<system text area>

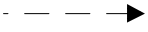
<process interaction area> ::=
{ <process area>
| <create line area>
| <signal route definition area>}+

<process area> ::=
<graphical process reference>
| <process diagram>
| <graphical typebased process definition>
| <existing typebased process definition>

<graphical process reference> ::=
<process symbol> ***contains***
{<process name> [<number of process instances>]}

<process symbol> ::=

<create line area> ::=
<create line symbol>
is connected to {<process area> <process area>}

<create line symbol> ::=


La tête de flèche placée sur le symbole <create line symbol> indique la zone <process area> sur laquelle le processus de création a lieu. Les symboles <create line symbol> sont facultatifs, mais s'ils sont utilisés, il doit exister dans le processus à l'extrémité de départ du symbole <create line symbol> une demande de création du processus à l'extrémité de la tête de flèche du symbole <create line symbol>. Cette règle s'applique après transformation de la zone <option area>.

NOTE – Cette règle peut être appliquée avant ou après transformation de l'option <transition option>.

Si une définition <block definition> ou <block type definition> utilisée dans une définition <textual typebased block definition> contient des définitions <signal route definition> et <textual typebased process definition>, alors chaque accès des définitions <process type definition> des définitions <textual typebased process definition> doivent être connectées à un acheminement de signal.

La même règle s'applique pour les définitions <process definition> et <process type definition> contenant des définitions <signal route definition> et <textual typebased service definition>.

Un exemple de <block diagram> est donné à la Figure 2.10.8.

Sémantique

Une définition de bloc contient une ou plusieurs définitions de processus d'un système et/ou une sous-structure de bloc.

Un bloc assure une interface statique de communication par laquelle les processus peuvent communiquer avec d'autres processus. De plus, elle donne la portée pour les définitions de processus.

Interpréter un bloc consiste à créer les instances initiales des processus dans le bloc.

2.4.4 Processus

Une définition de processus définit un grand ensemble arbitraire d'instances de processus.

Grammaire abstraite

Process-definition	::	*Process-name* *Number-of-instances* *Process-formal-parameter* * *Procedure-definition***-set** *Signal-definition***-set** *Data-type-definition* *Syntype-definition***-set** *Variable-definition***-set** *View-definition***-set** *Timer-definition***-set** *Process-graph* | *Service-decomposition*
Number-of-instances	::	*Intg* [*Intg*]
Process-name	=	*Name*
Process-graph	::	*Process-start-node* *State-node***-set**
Process-formal-parameter	::	*Variable-name* *Sort-reference-identifier*
Service-decomposition	::	*Service-definition***-set** *Signal-route-definition***-set** *Signal-route-to-route-connection***-set**

Si une définition *Process-definition* a une décomposition *Service-decomposition*, il ne doit pas exister de *Timer-definitions*. La décomposition *Service-decomposition* doit contenir au moins une *Service-definition*.

Grammaire textuelle concrète

<process definition> ::=
process {<process name> | <process identifier> }
[<number of process instances>] <end>
[<formal parameters> <end>] [<valid input signal set>]
{ <entity in process>
| <signal route to route connection> }*
[<process body>]
endprocess [<process name> | <process identifier>] <end>

<entity in process> ::=
<signal definition>
| <signal list definition>
| <textual procedure reference>
| <procedure definition>
| <remote procedure definition>
| <imported procedure specification>
| <macro definition>
| <remote variable definition>
| <data definition>
| <variable definition>
| <view definition>
| <select definition>
| <imported variable specification>
| <timer definition>
| <signal route definition>
| <service definition>
| <textual service reference>
| <textual typebased service definition>
| <service type definition>
| <textual service type reference>

<textual procedure reference> ::=
<procedure preamble>
procedure <procedure name> **referenced** <end>

<textual service reference> ::=
service <service name> **referenced** <end>

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

<process body> ::=
<start> {<state> | <free action>} *

<formal parameters> ::=
fpar <parameters of sort> {**,** <parameters of sort>}*

<parameters of sort> ::=
<variable name> {**,** <variable name>}* <sort>

<number of process instances> ::=
([<initial number>][,[<maximum number>]])

<initial number> ::=
<Natural simple expression>

<maximum number> ::=
<Natural simple expression>

Une définition <service definition>, une référence <textual service reference> ou une définition <textual typebased service definition> peut être présente uniquement si <process body> est omis.

Une définition <process definition> ne peut contenir des définitions <signal route definition> que si la définition <block definition> ou <textual typebased block definition> qui l'englobe contient des définitions <signal route definition>.

Le nombre initial d'instances et le nombre maximal d'instances contenus dans le nombre *Number-of-instances* sont tirés du nombre <number of process 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 process instances> utilisé dans la dérivation est le suivant:

a) s'il n'y a pas de <textual process reference> pour le processus, le nombre <number of process instances> dans la définition <process definition> ou dans la définition <textual typebased process definition> est utilisé. S'il ne contient pas un nombre <number of process instances>, on utilise le nombre <number of process instances> où le nombre <initial number > et le nombre <maximum number> sont omis;

b) si à la fois le nombre <number of process instances> dans la définition <process definition> et le nombre <number of process instances> dans une référence <textual process reference> sont omis, on utilise le nombre <number of process instances> où sont omis à la fois le nombre <initial number> et le nombre <maximum number>;

c) si soit le nombre <number of process instances> dans la définition <process definition> soit le nombre <number of process instances> dans une référence <textual process reference> est omis, le nombre <number of process instances> est celui qui est présent;

d) si à la fois le nombre <number of process instances> dans la définition <process definition> et le nombre <number of process instances> dans une référence <textual process reference> sont spécifiés, les deux nombres <number of process instances> doivent être égaux lexicalement et ce nombre <number of process instances> est utilisé.

Une relation analogue s'applique au nombre <number of process instances> spécifié dans le diagramme <process diagram> et dans la référence <graphical process reference> 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.

L'utilisation de l'ensemble <valid input signal set> est précisée en 2.5.2, *modèle*.

Un exemple de définition <process definition> est donné à la Figure 2.10.9.

Grammaire graphique concrète

<process diagram> ::=

<frame symbol>

contains {<process heading>

{ {<process text area>}*

{<macro diagram>}*

{<type in process area>}*

{<process graph area> | <service interaction area> } }***set*** }

[***is associated with*** {<signal route identifiers>}*]

L'identificateur <signal route identifier> identifie le trajet externe d'un signal relié à un trajet de signal dans le diagramme <process diagram>. Il est placé à l'extérieur du symbole <frame symbol> à proximité de l'extrémité du trajet du signal au symbole <frame symbol>.

<process text area> ::=

<text symbol> ***contains*** {
[<valid input signal set>]
{<signal definition>
| <signal list definition>
| <variable definition>
| <view definition>
| <imported variable specification>
| <imported procedure specification>
| <remote procedure definition>
| <remote variable definition>
| <data definition>
| <macro definition>
| <timer definition>
| <select definition> }* }

<process heading> ::=

process {<process name> | <process identifier>}
[<number of process instances> [<end>]]
[<formal parameters>]

<process graph area> ::=

<start area> { <state area> | <in-connector area> }*

<service interaction area> ::=

{ <service area>
| <signal route definition area> }+

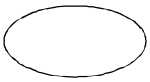
<service area> ::=

<graphical service reference>
| <service diagram>
| <graphical typebased service definition>
| <existing typebased service definition>

<graphical service reference> ::=

<service symbol> ***contains*** <service name>

<service symbol> ::=



Un exemple de diagramme de processus est donné à la Figure 2.10.10.

Sémantique

La suite de ce paragraphe s'applique lorsque la définition *Process-definition* contient un graphe *Process-graph*. La sémantique supplémentaire concernant le cas où la définition *Process-definition* contient des définitions *Service-definitions* est traitée en 2.4.5.

Une définition de processus définit un ensemble de processus. Plusieurs instances du même processus peuvent exister au même moment et s'exécuter de manière asynchrone et en parallèle entre elles et avec des instances d'autres ensembles de processus dans le système.

Dans le nombre *Number-of-instances,* la première valeur représente le nombre d'instances du processus qui existe à la création du système, la deuxième valeur représente le nombre maximal d'instances simultanées du type de processus.

Une instance de processus est une machine à états finis étendue communicante qui assure un certain nombre d'actions, appelées transitions, suite à la réception d'un signal donné, chaque fois qu'elle se trouve dans un certain état. La réalisation de la transition aboutit à l'attente du processus dans un autre état, qui n'est pas nécessairement différent de l'état d'origine.

Le concept de machine à états finis a été étendu pour couvrir les aspects suivants:

a) variables et branchement: l'état qui résulte après une transition, outre le signal qui a été à l'origine de la transition, peut être affecté par des décisions prises concernant les variables connues du processus;

b) transition spontanée: il est possible d'activer une transition de manière spontanée sans qu'aucun signal n'ait été reçu. L'activation est non déterministe et, par conséquent, elle rend instable l'état auquel elle est attachée;

c) sauvegarde: la construction de sauvegarde permet de spécifier pour le traitement des signaux un ordre différent de leur ordre d'arrivée dans la file.

Lorsqu'un système est créé, les processus initiaux sont créés dans un ordre arbitraire. La communication des signaux entre les processus ne commence que lorsque tous les processus initiaux ont été créés. Les paramètres formels de ces processus initiaux n'ont pas de valeurs associées, c'est-à-dire qu'ils sont initialisés avec une valeur non définie.

Les instances de processus existent à partir du moment où un système est créé ou peuvent être créées par une des actions de demande de création qui lance les processus à interpréter; leur interprétation commence lorsque l'action de départ est interprétée; des actions d'arrêt peuvent arrêter leur existence.

Les signaux reçus par les instances de processus sont appelés signaux d'entrée, et les signaux envoyés par les instances de processus sont appelés signaux de sortie.

Les signaux peuvent être traités par une instance de processus seulement lorsque celle-ci se trouve dans un certain état. L'ensemble complet de signaux d'entrée valides est l'union de l'ensemble des signaux se trouvant dans tous les trajets des signaux conduisant à l'ensemble des instances désigné par la définition du processus, de l'ensemble <valid input signal set>, des signaux d'entrée implicites introduits par les concepts supplémentaires des paragraphes 4.10 - 4.14 et des signaux de temporisation.

Un accès d'entrée unique est associé à chaque instance de processus. Lorsqu'un signal d'entrée parvient au processus, il est appliqué à l'accès d'entrée de l'instance de processus.

Le processus peut être soit mis en attente, en occupant un état, soit en activité, en effectuant une transition. Pour chaque état, il existe un ensemble de signaux de mise en réserve (voir également 2.6.5). Dans le cas d'attente dans un état, le premier signal entrant dont l'identificateur ne figure pas dans l'ensemble des signaux de sauvegarde est extrait de la file d'attente, et traité par le processus. Toute transition peut également être lancée en tant que transition spontanée indépendamment de la présence de tout signal dans la file.

L'accès d'entrée peut retenir un nombre quelconque de signaux, de sorte que plusieurs signaux entrants sont mis dans une file d'attente pour le processus. 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 processus est créé, on lui attribue un accès d'entrée vide, et il y a alors création de variables locales.

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 de processus est dynamiquement créée.

Pour toutes les instances de processus, on peut utiliser quatre expressions produisant une valeur de PId (voir D.10): **self**, **parent**, **offspring** et **sender**. Elles donnent un résultat pour:

a) l'instance de processus (**self**);

b) l'instance du processus créateur (**parent**);

c) l'instance de processus la plus récente créée par l'instance du processus (**offspring**);

d) l'instance de processus en provenance de laquelle le dernier signal entrant a été utilisé (**sender**) (voir également 2.6.4).

Ces expressions sont expliquées dans le détail en 5.4.4.3.

self, **parent**, **offspring** et **sender** peuvent être utilisés dans des expressions à l'intérieur des instances de processus.

Pour toutes les instances de processus qui se trouvent présentes au moment de l'initialisation du système, l'expression prédéfinie **parent** présente toujours la valeur Null.

Pour toutes les instances de processus nouvellement créées, les expressions prédéfinies **sender** et **offspring** ont la valeur Null.

2.4.5 Service

Le concept de service offre une alternative au graphe de processus au travers d'un ensemble de services. Dans de nombreuses situations, les définitions de service peuvent diminuer la complexité globale et augmenter la lisibilité par comparaison avec l'utilisation d'un corps de processus. De plus, chaque définition de service peut définir un comportement partiel du processus, ce qui peut être utile dans certaines applications.

Grammaire abstraite

Service-definition :: *Service-name*
Procedure-definition-**set**
Data-type-definition
Syntype-definition-**set**
Variable-definition-**set**
View-definition-**set**
Timer-definition-**set**
Service-graph

Service-name = *Name*

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

Service-start-node :: *Transition*

Concrete textual grammar

<service definition> ::=
service {<service name> | <service identifier>} <end>
[<valid input signal set>]
{<entity in service>}*
<service body>
endservice [{<service name> | <service identifier>}] <end>

<entity in service> ::=
<variable definition>
| <data definition>
| <view definition>
| <imported variable specification>
| <imported procedure specification>
| <select definition>
| <macro definition>
| <procedure definition>
| <textual procedure reference>
| <timer definition>

<service body> ::=
<process body>

Les différentes définitions <service definition> ou <textual typebased definition> dans une définition <process definition> ne doivent ni révéler le même nom de variable, ni exporter ou importer le même nom de variable, ni exporter ou importer la même procédure distante.

Si une variable ou une procédure importée est définie dans le processus englobant, ou si une variable ou une procédure importée est spécifiée dans le processus englobant, la variable ou procédure définie ou spécifiée ne doit pas l'être avec le même nom que le nom <remote variable name> ou <remote procedure name> correspondant dans l'une des définitions <service definition> ou <textual typebased service definition> et elle ne peut être utilisée qu'en tant qu'exportée ou importée dans un seul service. Les signaux implicites pour une variable exportée sont ajoutés à un service arbitraire si

elle n'est utilisée dans aucun service. Une procédure exportée définie dans le processus englobant doit être mentionnée en tant qu'exportée que dans un seul service uniquement.

Les ensembles complets des signaux d'entrée valides des services à l'intérieur d'un processus doivent être disjoints. L'ensemble complet des signaux d'entrées valides d'un service est l'union de l'ensemble <valid input signal set> de sa définition <service definition>, de l'ensemble des signaux convoyés sur les acheminements des signaux entrants du service, y compris les signaux d'entrée implicites introduits par les concepts supplémentaires des paragraphes 4.10-4.14 et des signaux de temporisation.

Grammaire graphique concrète

<service diagram> ::=
<frame symbol> ***contains***
{<service heading>
{ {<service text area> }*
{<graphical procedure reference>}*
{<procedure diagram>}*
{<macro diagram>}*
<service graph area> } ***set*** }

<service heading> ::=
service {<service name> | <service identifier>}

<service text area> ::=
<text symbol> ***contains***
{ <variable definition>
| <data definition>
| <timer definition>
| <view definition>
| <imported variable specification>
| <imported procedure specification>
| <select definition>
| <macro definition> }*

<service graph area> ::=
<process graph area>

Sémantique

Dans une instance de processus, il existe une instance de service pour chaque définition *Service-definition* dans la définition *Process-definition*. Les instances de service sont des composantes de l'instance de processus et ne peuvent être manipulées comme des objets séparés. Ils partagent l'accès d'entrée et les expressions **self**, **parent**, **offspring** et **sender** de l'instance de processus.

Une instance de service est une machine à états.

Lorsqu'une instance de processus est créée, les noeuds *Service-start-nodes* sont exécutés dans un ordre arbitraire. Aucun état d'aucun service n'est interprété avant que tous les noeuds *Service-start-nodes* ne soient terminés. Un noeud *Service-start-node* est considéré comme terminé lorsque l'instance de service entre pour la première fois dans un noeud *State-node* (éventuellement à l'intérieur d'une procédure) ou interprète un noeud *Stop-node*.

A un instant donné, un seul service exécute une *Transition*. Lorsque le service exécutant atteint un état, le signal suivant à l'accès d'entrée (qui n'est pas sauvegardé par le service, sinon le service serait capable de le traiter) est donné au service qui est capable de le traiter.

Lorsqu'un service cesse d'exister, les signaux d'entrée liés à ce service sont écartés. Lorsque tous les services cessent d'exister, l'instance de processus cesse d'exister.

Exemple

Voir 2.10.

2.4.6 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

Procedure-definition	::	*Procedure-name* *Procedure-formal-parameter** *Procedure-definition*-**set** *Data-type-definition* *Syntype-definition*-**set** *Variable-definition*-**set** *Procedure-graph*
Procedure-name	=	*Name*
Procedure-formal-parameter	=	*In-parameter* | *Inout-parameter*
In-parameter	::	*Variable-name* *Sort-reference-identifier*
Inout-parameter	::	*Variable-name* *Sort-reference-identifier*
Procedure-graph	::	*Procedure-start-node* *State-node*-**set**
Procedure-start-node	::	*Transition*

Grammaire textuelle concrète

```
<procedure definition>  ::=
                        <procedure preamble>
                        procedure { <procedure name> | <procedure identifier>}
                              [<formal context parameters>]
                              [ <virtuality constraint>]
                              [<specialization>]  <end>
                              [<procedure formal parameters> <end>]
                              [<procedure result> <end>]
                              {       <entity in procedure>
                                 |    <select definition>
                                 |    <macro definition> }*
                              [ <procedure body> ]
                        endprocedure [<procedure name> | <procedure identifier>] <end>

<procedure preamble> ::=
                        [<virtuality>][exported  [ as <remote procedure identifier> ]]

<procedure formal parameters> ::=
                        fpar <formal variable parameters>
                              {, <formal variable parameters> }*

<formal variable parameters> ::=
                        <parameter kind>  <parameters of sort>
```

<parameter kind> ::=
[**in/out** | **in**]

<procedure result> ::=
returns [<variable name>] <sort>

<entity in procedure> ::=
<data definition>
| <variable definition>
| <textual procedure reference>
| <procedure definition>

<procedure body> ::=
<start> {<state> | <free action>}*
| {<state> | <free action>}+

Une procédure exportée ne peut pas avoir de paramètres de contexte formel et sa portée englobante doit être un type de processus, une définition de processus, un type de service ou une définition de service.

Dans une définition <procedure definition>, la définition <variable definition> ne peut contenir les noms <variable name> **revealed**, **exported**, **revealed exported**, et **exported revealed** (voir 2.6.1).

L'attribut exporté est hérité par n'importe quel sous-type d'une procédure. Une procédure exportée virtuelle doit contenir **exported** dans toutes les redéfinitions. Le type virtuel, y compris la procédure, est décrit en 6.3.2. 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 implique qu'il s'agit de l'identificateur <remote procedure identifier> du supertype.

Deux procédures exportées dans un processus, y compris les services possibles, ne peuvent pas mentionner le même identificateur <remote procedure identifier>.

Un exemple d'une définition <procedure definition> est donné à la Figure 2.10.11.

Grammaire graphique concrète

<procedure diagram> ::=
<frame symbol> ***contains*** {<procedure heading>
{ { <procedure text area>
| <procedure area>
| <macro diagram> }*
<procedure graph area> }***set*** }

<procedure area> ::=
<graphical procedure reference>
| <procedure diagram>

<procedure text area> ::=
<text symbol> ***contains***
{ <variable definition>
| <data definition>
| <select definition>
| <macro definition> }*

<graphical procedure reference> ::=
<procedure symbol> ***contains***
{ <procedure preamble>
<procedure name> }

<procedure symbol> ::=

<procedure heading> ::=

<procedure preamble>
procedure {<procedure name>| <procedure identifier> }
[<formal context parameters>]
[<virtuality constraint>]
[<specialization>]
[<procedure formal parameters>]
[<procedure result>]

<procedure graph area> ::=

[<procedure start area>]
{<state area> | <in-connector area> }*

<procedure start area> ::=

<procedure start symbol>
contains { [<virtuality>] }
is followed by <transition area>

<procedure start symbol> ::=

Un exemple de diagramme <procedure diagram> est montré à la Figure 2.10.12.

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 préambule <procedure preamble> implique que la procédure peut être appelée en tant que procédure distante, conformément au modèle donné en 4.14.

Une variable de procédure est une variable locale à la procédure qui ne peut apparaître, être visible, être exportée ou être importée. 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 <procedure call> 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 la valeur de l'expression donnée par le paramètre effectif correspondant s'il est présent. Autrement, la variable ne prend aucune valeur, c'est-à-dire qu'elle devient «non définie»;

b) un paramètre formel n'ayant pas d'attribut explicite, a un attribut **in** implicite;

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 dans l'expression des paramètres effectifs. Le nom *Variable-name* contenu est utilisé tout au long de l'interprétation du graphe *Procedure-graph* lorsqu'on se réfère à la valeur de la variable ou lorsque l'on affecte une nouvelle valeur à la variable;

e) la *Transition* contenue dans le nœud *Procedure-start-node* est interprétée.

Les nœuds du graphe de procédure sont interprétés de la même manière que les nœuds équivalents d'un graphe de processus ou de service, c'est-à-dire que la procédure a le même ensemble complet de signaux d'entrée valides que le processus englobant, et les mêmes accès d'entrée que l'instance du processus englobant qui l'a appelée, soit directement ou indirectement.

Modèle

La spécification d'un résultat <procedure result> conduit à des paramètres <procedure formal parameters> avec un paramètre variable formel supplémentaire attaché ayant l'attribut **in/out,** un nouveau nom distinct (à moins qu'il ne soit explicitement spécifié dans le résultat <procedure result>) comme <variable name> et la <sort> du <procedure result> comme <sort>.

Lorsqu'un corps <procedure body> ou qu'une zone <procedure graph area> contient <return> ou <return area> dans une <expression>, la procédure doit avoir au moins un paramètre formel ayant un genre <parameter kind> **in/out** ou un résultat <procedure result>.

Toute occurrence de <return> ou <return area> dans une <expression> à l'intérieur du corps <procedure body> ou de la zone <procedure graph area> est remplacée par une tâche <task> contenant une assignation de l'<expression> contenue à droite de la variable **in/out,** suivie par un <return> ou une zone <return area> sans aucune <expression>.

La transformation s'effectue après *Generic system* (voir 4.3) et avant l'appel <value returning procedure call>.

Une zone <procedure start area> qui contient <virtuality> est appelée zone de départ de procédure virtuelle. Cette zone est décrite avec plus de détail en 6.3.3.

2.5 Communication

2.5.1 Canal

Grammaire abstraite

Channel-definition	::	*Channel-name* [**NODELAY**] *Channel-path* [*Channel-path*]
Channel-path	::	*Originating-block* *Destination-block* *Signal-identifier***-set**
Originating-block	=	*Block-identifier* | **ENVIRONMENT**
Destination-block	=	*Block-identifier* | **ENVIRONMENT**
Block-identifier	=	*Identifier*
Signal-identifier	=	*Identifier*
Channel-name	=	*Name*

Une extrémité du canal doit au moins être un bloc. Si les deux extrémités sont des blocs, les identificateurs *Block-identifiers* doivent être différents.

La ou les extrémité(s) des blocs doivent être définies dans la même unité de portée que celles où est défini le canal.

NODELAY indique le fait que le canal ne cause pas de retard.

Grammaire textuelle concrète

<channel definition> ::=
channel <channel name> [**nodelay**]
<channel path>
[<channel path>]
[<channel substructure definition>
| <textual channel substructure reference>]
endchannel [<channel name>] <end>

<channel path> ::=
from <channel endpoint>
to <channel endpoint> **with** <signal list><end>

<channel endpoint> ::=
{<block identifier> | **env** }[**via** <gate>]

Lorsque deux trajets <channel path> sont définis, le sens de l'un doit être opposé par rapport au sens de l'autre.

L'accès <gate> doit être spécifié si et seulement si:

a) <channel endpoint> désigne une connexion à une définition <textual typebased block definition>, auquel cas l'accès <gate> doit être directement défini dans le type de bloc de ce bloc; ou

b) **env** est spécifié et le canal est défini dans une sous-structure de bloc d'une définition <block type definition>, auquel cas l'accès <gate> doit être défini dans ce type de bloc.

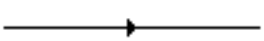
Grammaire graphique concrète

<channel definition area> ::=
<channel symbol>
is associated with {<channel name>
{ [{<channel identifiers> | <block identifier> | <gate> }]
<signal list area> [<signal list area>] }***set*** }
is connected to { <block area> {<block area> | <frame symbol>}
[<channel substructure association area>] }***set***

Les identificateurs <channel identifiers> peuvent être spécifiés si et seulement si le diagramme englobant est un diagramme <block substructure diagram> directement englobé dans une définition <block definition>. L'identificateur <block identifier> peut être spécifié si et seulement si le diagramme englobant est un diagramme <channel substructure diagram>. Les identificateurs <channel identifiers> identifient des canaux externes reliés au diagramme <block substructure diagram> délimité par le symbole <frame symbol>. L'identificateur <block identifier> identifie un bloc externe comme étant une extrémité de canal pour le diagramme <channel substructure diagram> délimité par le symbole <frame symbol>. Lorsque le symbole <channel symbol> est relié au symbole <frame symbol>, l'accès <gate> qui lui est associé est l'accès défini pour ce type de bloc au moyen d'un accès <gate> ou d'une contrainte <graphical gate constraint> associée au diagramme du type de bloc.

<channel symbol> ::=
<channel symbol 1>
| <channel symbol 2>
| <channel symbol 3>
| <channel symbol 4>
| <channel symbol 5>

<channel symbol 1> ::=



<channel symbol 2> ::=

<channel symbol 3> ::=

<channel symbol 4> ::=

<channel symbol 5> ::=

Pour chaque flèche placée sur le symbole <channel symbol>, il doit y avoir 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 <channel symbol 4> et <channel symbol 5> sont placées aux extrémités du canal et indiquent que le canal ne cause pas de retard.

Sémantique

Un canal est un moyen de transport pour les signaux. Un canal peut être considéré comme étant un ou deux trajets de canal unidirectionnels indépendants entre deux blocs ou entre un bloc et son environnement.

L'ensemble *Signal-identifier*-**set** dans chaque trajet *Channel-path* de la définition *Channel-definition* contient les signaux qui peuvent être acheminés sur ce trajet.

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 au point d'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'un des canaux ou l'un des trajets de signaux qui se trouvent connectés au canal.

Plusieurs canaux peuvent exister entre deux points d'extrémités. Le même type de signal peut être acheminé sur des canaux différents.

Si une extrémité d'un canal est une définition <textual typebased block definition>, l'autre extrémité peut être la même définition <textual typebased block definition>. Le nombre <number of block instances> doit alors être supérieur à un pour la définition <textual typebased block definition>.

Modèle

Un canal dont les deux extrémités sont des accès d'une définition <textual typebased block definition> représente un moyen d'acheminement à partir de chacun des blocs de l'ensemble vers tous les autres.

Un canal avec une seule extrémité étant un accès d'une définition <textual typebased block definition> représente des canaux individuels partant ou arrivant à chacun des blocs de l'ensemble. Les canaux individuels auront tous les mêmes propriétés relatives au retard que le canal étendu.

2.5.2 Acheminement du signal

Grammaire abstraite

Signal-route-definition :: *Signal-route-name*
Signal-route-path
[*Signal-route-path*]

Signal-route-path :: *Origin*
Destination
Signal-identifier-**set**

Origin = *Process-identifier* |
Service-identifier |
ENVIRONMENT

Destination = *Process-identifier* |
Service-identifier |
ENVIRONMENT

Signal-route-name = *Name*

Process-identifier = *Identifier*

Service-identifier = *Identifier*

Un des points extrêmes au moins du trajet d'acheminement du signal doit être un identificateur *Process-identifier* ou *Service-identifier*.

Si les deux points extrêmes sont des processus, les identificateurs *Process-identifier* doivent être différents. Si les deux points extrêmes sont des services, les identificateurs *Service-identifier* doivent être différents.

Le ou les point(s) extrême(s) du processus ou du service doivent être définis dans la même unité de portée que celle de l'acheminement du signal.

Grammaire textuelle concrète

<signal route definition> ::=
signalroute <signal route name>
<signal route path>
[<signal route path>]

<signal route path>::=
from <signal route endpoint> **to** <signal route endpoint>
with <signal list> <end>

<signal route endpoint> ::=
{<process identifier> | <service identifier> | **env** }
[**via** <gate>]

Lorsque deux <signal route path> sont définis, l'un doit être en sens opposé à l'autre.

L'accès <gate> doit être spécifié si et seulement si:

a) <signal route endpoint> désigne une connexion à une définition <textual typebased process definition> ou <textual typebased service definition>, auquel cas l'accès <gate> doit être directement défini dans le type de processus ou de service respectivement; ou

b) **env** est spécifié et le trajet d'acheminement du signal est défini dans un type de bloc ou un type de processus, auquel cas l'accès <gate> doit être défini dans ce type de bloc ou type de processus respectivement.

Grammaire graphique concrète

<signal route definition area> ::=

<signal route symbol>

is associated with {<signal route name>

{[<channel identifiers> | <external signal route identifiers> |

<gate>]

<signal list area> [<signal list area>] }***set*** }

is connected to

{{<process area> | <service area>}

{<process area> | <service area> | <frame symbol>} }***set***

<signal route symbol> ::=

<signal route symbol 1> | <signal route symbol 2>

<signal route symbol 1> ::=

<signal route symbol 2> ::=

Un symbole d'acheminement de signal comprend une tête de flèche à une extrémité (une direction) ou une tête de flèche à chaque extrémité (deux directions) pour montrer la direction du flot des signaux.

Pour chaque flèche située sur le symbole <signal route symbol>, il doit y avoir 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.

Lorsque le symbole <signal route symbol> est relié au symbole <frame symbol> dans le cas d'un diagramme <block type diagram>, l'accès <gate> qui lui est associé est l'accès défini pour ce type de bloc au moyen d'un accès <gate> ou d'une contrainte <graphical gate constraint> associée au diagramme <block type diagram>. Lorsque le symbole <signal route symbol> est relié au symbole <frame symbol> dans le cas d'un diagramme <process type diagram>, l'accès <gate> qui lui est associé est l'accès défini pour ce type de processus au moyen d'un accès <gate> ou d'une contrainte <graphical gate constraint> associée au diagramme <process type diagram>.

Sémantique

Un acheminement de signal est un moyen de transport pour les signaux. Un acheminement de signal peut être considéré comme étant un ou deux trajets d'acheminement de signal unidirectionnels et indépendants entre deux ensembles d'instances de processus désignés chacun par une définition de processus ou entre un ensemble d'instances de processus et l'environnement du bloc englobant, ou entre deux services, ou entre un service et l'environnement du processus englobant.

Les signaux acheminés par des acheminements de signaux sont délivrés au point extrême de destination.

Un acheminement de signaux n'introduit aucun retard dans le transport des signaux.

Lorsqu'une instance de signal est envoyée à une instance du même ensemble d'instances de processus, l'interprétation du nœud *Output-node* implique que le signal est appliqué directement dans l'accès d'entrée du processus destinataire.

Plusieurs trajets de signaux peuvent exister entre deux points extrêmes. Le même type de signal peut être acheminé sur différents acheminements de signaux.

Modèle

Si une définition <block definition> ou <block type definition> contient des définitions <signal route definition>, l'ensemble <valid input signal set> dans une définition <process definition>, s'il y en a un, n'a pas besoin de contenir des signaux dans des acheminements de signaux conduisant à l'ensemble d'instances de processus.

Si une définition <process type definition> ou <process definition> contient des services, l'ensemble <valid input signal set> est formé par l'union des signaux d'entrée aux accès (dans le cas de <process type definition>) ou des signaux dans les acheminements de signaux conduisant au processus (dans le cas de <process definition>), des signaux d'entrée sur les acheminements des signaux conduisant aux services et des ensembles <valid input signal set> des services. Cet ensemble est appelé l'ensemble implicite <valid input signal set>. Si l'ensemble <valid input signal set> pour un processus est spécifié explicitement, il doit être un sous-ensemble de l'ensemble implicite <valid input signal set>.

Si une définition <process definition> ou <process type definition> contient des définitions <signal route definition>, il n'est pas nécessaire que l'ensemble <valid input signal set> dans une définition <service definition>, s'il existe, contienne les signaux sur les acheminements de signaux conduisant au service.

Si une définition <block definition> ne contient aucune définition <signal route definition>, toutes les définitions <process definition> dans cette portée doivent (implicitement ou explicitement) contenir un ensemble <valid input signal set>. Dans ce cas, les définitions <signal route definition> et les connexions <channel to route connection> sont tirées de l'ensemble <valid input signal set>, des sorties <output> et des canaux qui se terminent à la frontière des blocs.

Si une définition <block type definition> ne contient aucune définition <signal route definition>, toutes les définitions <process definition> dans cette portée doivent (implicitement ou explicitement) contenir un ensemble <valid input signal set>. Dans ce cas, les définitions <signal route definition> sont tirées de l'ensemble <valid input signal set>, des sorties <output> et des listes <signal list> aux accès de la définition <block type definition>.

Les signaux correspondant à une direction donnée entre deux ensembles d'instances de processus dans l'acheminement de signal implicite sont constitués par l'intersection des signaux spécifiés dans l'ensemble <valid input signal set> du processus de destination et les signaux mentionnés dans une sortie du processus d'origine. Si l'un des points extrêmes est l'environnement, l'ensemble d'entrée/ensemble de sortie est constitué par les signaux acheminés par le canal ou l'accès (dans le cas d'un type de bloc) dans la direction donnée.

Si une définition de processus est une <textual typebased process definition>, la dérivation n'est pas fondée sur les ensembles <valid input signal set> et les sorties <output>, mais sur les listes <signal list> des signaux entrant et sortant des accès du type de processus.

Si une définition <process definition> ou <process type definition> ne contient aucune définition <signal route definition>, toutes les définitions <service definition> dans cette portée doivent contenir un ensemble <valid input signal set>. Dans ce cas, les définitions <signal route definition> et les connexions <signal route to route connection> sont tirées de l'ensemble <valid input signal set>, des sorties <output> et des acheminements de signaux externes qui se terminent à la frontière du processus ou aux accès du type de processus.

Les signaux correspondant à une direction donnée entre deux services dans l'acheminement de signal implicite sont constitués par l'intersection des signaux spécifiés dans l'ensemble <valid input signal set> du service de destination et les signaux mentionnés dans une sortie du service d'origine. Si l'un des points extrêmes est l'environnement, l'ensemble d'entrée/ensemble de sortie est constitué par les signaux acheminés par l'acheminement de signal externe ou l'accès (dans le cas d'un type de bloc) dans la direction donnée.

Si une définition de service est une <textual typebased service definition>, la dérivation n'est pas fondée sur les ensembles <valid input signal set> et les sorties <output>, mais sur les listes <signal list> des signaux entrant et sortant des accès du type de service.

2.5.3 Connexion

Grammaire abstraite

Channel-to-route-connection	::	*Channel-identifier*-**set** *Signal-route-identifier*-**set**
Signal-route-identifier	=	*Identifier*
Signal-route-to-route-connection	::	*External-signal-route-identifier*-**set** *Signal-route-identifier*-**set**
External-signal-route-identifier	=	*Identifier*

D'autres constructions sont données en 3.

Les identificateurs *Channel-identifier* dans un ensemble *Channel-identifier*-**set** dans une connexion *Channel-to-route-connection* doivent dénoter des canaux connectés au bloc englobant.

Chaque identificateur *Channel-identifier* connecté au bloc englobant doit être mentionné dans une et une seule connexion *Channel-to-route-connection.* Chaque identificateur *Signal-route-identifier* dans une connexion *Channel-to-route-connection* doit être défini dans le même bloc où la connexion *Channel-to-route-connection* se trouve définie et doit avoir la frontière de ce bloc située à l'un de ses points extrêmes. Chaque

identificateur *Signal-route-identifier* défini dans le bloc qui l'entoure et qui a son environnement comme étant l'un de ses points extrêmes, doit être mentionné dans une et une seule connexion *Channel-to-route-connection*.

Pour une direction donnée, l'union des ensembles d'identificateurs *Signal-identifier* dans les acheminements de signaux dans une connexion *Channel-to-route-connection* doit être égale à l'union des ensembles d'identificateurs *Signal-identifier* acheminés par les identificateurs *Channel-identifiers* dans la même connexion *Channel-to-route-connection* et correspondant à la même direction.

Les identificateurs *External-signal-route-identifiers* d'un ensemble *External-signal-route-identifier*-**set** dans une connexion *Signal-route-to-route-connection* doivent désigner des acheminements de signaux connectés au processus englobant.

Chaque identificateur *External-signal-route-identifier* connecté au processus englobant doit être mentionné dans une et une seule connexion *Signal-route-to-route-connection.* Chaque identificateur *Signal-route-identifier* dans une connexion *Signal-route-to-route-connection* doit être défini dans le même processus que la connexion *Signal-route-to-route-connection* et doit avoir la frontière de ce processus comme l'une de ses extrémités. Chaque identificateur *Signal-route-identifier* défini dans le processus qui l'entoure et qui a son environnement comme extrémité doit être mentionné dans une et une seule connexion *Signal-route-to-route-connection.*

Pour une direction donnée, l'union des ensembles d'identificateurs *Signal-identifier* dans les acheminements de signaux dans une connexion *Signal-route-to-route-connection* doit être égale à l'union des ensembles d'identificateurs *Signal-identifier* acheminés par les identificateurs *External-signal-route-identifiers* dans la même connexion *Signal-route-to-route-connection* et correspondant à la même direction.

Grammaire textuelle concrète

<channel to route connection> ::=
connect <channel identifiers>
and <signal route identifiers> <end>
<channel identifiers> ::=
<channel identifier> {, <channel identifier>}*
<signal route identifiers> ::=
<signal route identifier> {, <signal route identifier>}*
<signal route to route connection> ::=
connect <external signal route identifiers>
and <signal route identifiers> <end>
<external signal route identifiers> ::=
<signal route identifier>
{, <signal route identifier>}*

Grammaire graphique concrète

Sur le plan graphique, l'élément connexion est représenté par les identificateurs <channel identifiers> et <external signal route identifiers> associés au trajet de signal et contenus dans la zone <signal route definition area> (voir 2.5.2 *grammaire graphique concrète*).

Aucun identificateur <signal route identifier> ou <channel identifier> ne peut être mentionné plus d'une fois dans les connexions d'un diagramme.

2.5.4 Signal

Grammaire abstraite

Signal-definition	::	*Signal-name* *Sort-reference-identifier** [*Signal-refinement*]
Signal-name	=	*Name*

Grammaire textuelle concrète

<signal definition>::=

signal

<signal definition item>

{,<signal definition item>}* <end>

<signal definition item> ::=

<signal name>

[<formal context parameters>]

[<specialization>]

[<sort list>][<signal refinement>]

<sort list> ::=

(<sort> {, <sort>}*)

Un paramètre <formal context parameter> des paramètres <formal context parameters> doit être un paramètre <sort context parameter>. Le type <base type> comme partie de la <specialization> doit être un <signal identifier>.

Sémantique

Une instance de signal est un flot d'informations entre des processus; 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 processus, elle se dirige toujours vers un processus 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.

2.5.5 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 et les signaux de temporisation.

Grammaire textuelle concrète

<signal list definition> ::=

signallist <signal list name> = <signal list><end>

<signal list> ::=

<signal list item> { **,** <signal list item>}*

<signal list item> ::=

<signal identifier> | (<signal list identifier>) | <timer identifier>

La liste <signal list> qui est établie en remplaçant tous les identificateurs <signal list identifier> dans la liste par la liste des identificateurs <signal identifier> ou <timer identifier> qu'ils désignent, correspond à un ensemble *signal-identifier*-**set** dans la *grammaire abstraite*.

La liste <signal list> ne doit pas contenir l'identificateur <signal list identifier> défini par la définition <signal list definition> soit directement ou indirectement (via un autre identificateur <signal list identifier>).

Grammaire graphique concrète

<signal list area> ::=

<signal list symbol> ***contains*** <signal list>

<signal list symbol> ::=

2.6 Comportement

2.6.1 Variables

Une variable a une valeur de sorte associée ou elle est «non définie».

2.6.1.1 Définition de variable

Grammaire abstraite

Variable-definition	::	*Variable-name* *Sort-reference-identifier* [*Ground-expression*] [**REVEALED**]
Variable-name	=	*Name*

Si l'expression *Ground-expression* est présente, elle doit être de la même sorte que l'identificateur *Sort-reference-identifier* désigné.

Grammaire textuelle concrète

<variable definition> ::=
dcl
[revealed | exported | revealed exported | exported revealed]
<variables of sort> {**,** <variables of sort> }*<end>

<variables of sort> ::=
<variable name> [<exported as>] {**,** <variable name> [<exported as>]}*
<sort> [:= <ground expression>]

<exported as> ::=
as <remote variable identifier>

L'attribut <exported as> ne peut être utilisé que pour une variable ayant l'attribut **exported**. Deux variables exportées dans un processus, y compris les services possibles, ne peuvent pas mentionner le même identificateur <remote variable identifier>.

Sémantique

Lorsqu'une variable est créée et que l'expression *Ground-expression* est présente, la variable est associée avec la valeur de l'expression *Ground-expression*. Autrement, la variable n'a pas de valeur associée, c'est-à-dire qu'elle est «non définie».

Si l'identificateur *Sort-reference-identifier* est un identificateur *Syntype-identifier*, et que l'expression *Ground-expression* est présente, et sa valeur n'est pas conforme à la condition *Range-condition*, le système se trouve dans un état d'erreur et son comportement n'est pas spécifié.

L'attribut **revealed** permet à tous les autres processus du même bloc de lire la valeur de la variable, à condition qu'ils aient été dotés d'une définition <view definition> avec le même nom et la même sorte.

L'attribut **exported** permet d'utiliser une variable en tant que variable exportée conformément au 4.13.

Modèle

L'expression *Ground-expression* est représentée par:

a) une expression <ground expression> si celle-ci est donnée dans la définition <variable definition>;

b) sinon, l'expression <ground expression> de l'initialisation <default initialization> si la <sort> possède une telle initialisation.

Autrement, l'expression *Ground-expression* n'est pas présente.

2.6.1.2 Définition de visibilité

Grammaire abstraite

View-definition :: *View-name*
Sort-reference-identifier

Dans la définition *Block-definition* englobante, il doit exister au moins une définition *Process-definition* qui contient une définition **REVEALED** *Variable-definition* avec le même identificateur *Sort-reference-identifier* et le même nom *Variable-name* que le nom *Name* de *View-name*.

Grammaire textuelle concrète

<view definition> ::=
viewed
<view name> {**,** <view name>}* <sort>
{**,**<view name>{, <view name>}*<sort>}* <end>

Sémantique

Le mécanisme de visibilité permet à une instance de processus de voir la valeur de la variable vue de manière continue comme si elle était définie localement. Cette instance de processus n'a cependant aucun droit de la modifier.

2.6.2 Départ

Grammaire abstraite

Process-start-node :: *Transition*

Grammaire textuelle concrète

<start> ::=
start [<virtuality>] <end> <transition>

Grammaire graphique concrète

<start area> ::=
<start symbol>
contains { [<virtuality>] }
is followed by <transition area>

<start symbol> ::=

Sémantique

La *Transition* du nœud *Process-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 6.3.3.

2.6.3 Etat

Grammaire abstraite

State-node :: *State-name*
Save-signalset
Input-node-**set**
Spontaneous-transition-**set**

State-name = *Name*

Les nœuds *State-nodes* dans un graphe *Process-graph*, *Service-graph* ou *Procedure-graph* doivent avoir des noms *State-names* respectifs.

Pour chaque nœud *State-node*, tous les identificateurs *Signal-identifiers* (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-identifiers* dans l'ensemble *Input-node*-**set** doivent être distincts.

Grammaire textuelle concrète

<state> ::=
state <state list> <end>
{ <input part>
| <priority input>
| <save part>
| <spontaneous transition>
| <continuous signal>}*
[**endstate** [<state name>] <end>]

<state list> ::=
{<state name> { **,** <state name> }*}
| <asterisk state list>

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 implicite de signal. 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>.

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

<state area> ::=
<state symbol> ***contains*** <state list> ***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-nodes*.

Les symboles <solid association symbol> issus d'un symbole <state symbol> peuvent avoir un trajet de départ commun.

Une zone <state area> doit contenir un nom <state name> si elle coïncide avec une zone <nextstate area>.

Sémantique

Un état représente une condition particulière dans laquelle un processus peut traiter une instance de signal en exécutant une transition. Si l'état n'a ni transitions spontanées, ni signaux continus, et s'il n'y a pas d'instances de signaux en attente à l'accès d'entrée autres que celles mentionnées dans la sauvegarde, le processus attend dans l'état jusqu'à ce qu'une instance de signal soit reçue.

Dans un état qui contient des transitions *Spontaneous-transitions*, le processus peut à tout instant entrer dans la *Transition* de l'une des transitions *Spontaneous-transitions*.

Modèle

Lorsque la liste <state list> d'un certain état <state> contient plusieurs noms <state name>, une copie de l'état <state> est créée pour chacun de ces <state name>. Ensuite l'état <state> est remplacé par ces copies.

2.6.4 Entrée

Grammaire abstraite

Input-node	::	*Signal-identifier* [*Variable-identifier*]* *Transition*
Variable-identifier	=	*Identifier*

La longueur de l'identificateur [*Variable-identifier*]* doit être égale au nombre d'identificateurs *Sort-reference-identifiers* de la définition *Signal-definition* désigné par l'identificateur *Signal-identifier*.

Les sortes de variables doivent correspondre par leur position aux sortes des valeurs qui peuvent être acheminées par le signal.

Grammaire textuelle concrète

```
<input part>  ::=
                        <basic input part>
                |       <remote procedure input transition>

<basic input part> ::=
                        input [<virtuality>] <input list> <end>
                        [<enabling condition>]<transition>

<input list>  ::=
                        <asterisk input list>
                |       <stimulus> { ,<stimulus> }*

<stimulus>  ::=
                        { <signal identifier> | <timer identifier>}
                        [( [<variable>] {,[<variable>]}* )]
```

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 (identificateurs <timer identifier>) sont également représentés par l'identificateur *Signal-identifier*. 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 2.8.

Les virgules peuvent être omises après la dernière <variable> dans un <stimulus>.

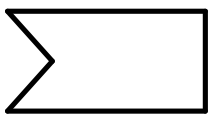
Grammaire graphique concrète

```
<input area> ::=
                        <basic input area>
                |       <remote procedure input area>

<basic input area> ::=
                        <input symbol> contains  { [<virtuality>] <input list> }
                        is followed by {[<enabling condition area>] <transition area>}

<input symbol> ::=
                        <plain input symbol>
                |       <internal input symbol>

<plain 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> contenus dans un symbole <input symbol> donne le nom de l'un des nœuds *Input-nodes* que ce symbole <input symbol> représente.

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 le processus. Aux variables associées à l'entrée, on affecte les valeurs acheminées par le signal utilisé.

Les valeurs seront assignées aux variables de gauche à droite. Si aucune variable n'est associée à l'entrée pour une sorte spécifiée dans le signal, la valeur de cette sorte est écartée. S'il n'y a pas de valeur associée à une sorte spécifiée dans le signal, la valeur correspondante devient «non définie».

A l'expression **sender** du processus récepteur est donnée la valeur PId du processus d'origine, acheminée par l'instance du signal.

Les instances de signal circulant de l'environnement vers une instance de processus dans le système achemineront toujours une valeur PId différente de toutes les valeurs dans le système.

Modèle

Lorsque la liste des <stimulus> d'une partie <input part> donnée 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.

Lorsque 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 <task body> une instruction <assignment statement> pour chacune des variables <variable>, affectant à la <variable> la valeur de la nouvelle variable correspondante. Les valeurs seront affectées dans l'ordre de gauche à droite de la liste des variables <variable>. Cette tâche <task> devient la première action <action> dans la <transition>.

Une partie <basic input part> ou une zone <basic input area> qui contient <virtuality> est appelée transition d'entrée virtuelle. Les transitions d'entrée virtuelles sont décrites en détail en 6.3.3.

2.6.5 Sauvegarde

Une sauvegarde spécifie un ensemble d'identificateurs de signaux dont les instances ne peuvent pas être traitées par le processus dans 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> ::=
<basic save part>
| <remote procedure save>

<basic save part> ::=
save [<virtuality>] <save list> <end>

<save list> ::=
{<signal list> | <asterisk save list>}

Une liste <save list> représente l'ensemble *Signal-identifier*-**set**. La liste <asterisk save list> est une notation abrégée expliquée en 4.7.

Grammaire graphique concrète

<save area> ::=
<basic save area>
| <remote procedure save area>

<basic save area> ::=
<save symbol> ***contains*** { [<virtuality>] <save list> }

<save symbol> ::=

Sémantique

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 partie <basic save part> ou une zone <basic save area> qui contient <virtuality> est appelée sauvegarde virtuelle. Cette dernière est décrite en 6.3.3.

2.6.6 Transition spontanée

Une transition spontanée spécifie une transition d'état sans aucune réception de signal.

Grammaire abstraite

Spontaneous-transition :: *Transition*

Grammaire textuelle concrète

<spontaneous transition> ::=
input [<virtuality>] <spontaneous designator> <end>
[<enabling condition>] <transition>

<spontaneous designator> ::=
none

Grammaire graphique concrète

<spontaneous transition area> ::=
<input symbol> ***contains***
{ [<virtuality>] <spontaneous designator> }
is followed by
{ [<enabling condition area>] <transition area> }

Sémantique

Une transition spontanée permet l'activation d'une transition sans qu'aucun stimuli ne soit présenté au processus. L'activation d'une transition spontanée ne dépend pas de la présence d'instances de signaux au port d'entrée du processus. Il n'y a pas de priorité entre les transitions activées par réception de signal et les transitions spontanées.

L'expression **sender** est **self** après activation d'une transition spontanée.

Modèle

Une transition <spontaneous transition> ou une zone <spontaneous transition area> qui contient <virtuality> est appelée transition spontanée virtuelle. Elle est décrite en 6.3.3.

2.6.7 Etiquette

Grammaire textuelle concrète

<label> ::=

<connector name> :

<free action> ::=

connection

<transition>

[**endconnection** [<connector name>] <end>]

<body> est utilisé comme un non-terminal qui convient aux règles applicables aux processus, aux services et aux procédures. Il ne fait pas partie de la grammaire textuelle concrète.

Tous les noms <connector name> définis dans un corps <body> doivent être distincts.

Une étiquette représente le point d'entrée d'un transfert de contrôle à partir des instructions de branchement correspondantes avec les mêmes noms <connector name> dans le même corps <body>.

Les transferts de contrôle ne sont autorisés que pour les étiquettes à l'intérieur du même corps <body>. Les règles relatives aux corps <body> d'un supertype et de sa spécialisation sont énoncées en 6.3.1.

Si la chaîne <transition string> de la <transition> de l'action <free action> est non vide, la première instruction <action statement> doit avoir une étiquette <label> sinon, l'instruction <terminator statement> doit avoir une étiquette <label>.

Lorsqu'il est présent, le nom <connector name> terminant l'action <free action> doit être le même que le nom <connector name> dans cette étiquette <label>.

Grammaire graphique concrète

<in-connector area> ::=

<in-connector symbol> ***contains*** <connector name> ***is followed by***

<transition area>

<in-connector symbol> ::=

Une zone <in-connector area> représente la continuation d'un symbole <flow line symbol> à partir d'une zone <out-connector area> correspondante avec le même nom <connector name>, dans la même zone <process graph area> ou la même zone <procedure graph area>.

Sémantique

La <transition> ou la zone <transition area> contenue est représentée dans la grammaire abstraite en appliquant <join> au nom <connector name> (voir 2.6.8.2.2).

2.6.8 Transition

2.6.8.1 Corps de transition

Grammaire abstraite

Transition :: *Graph-node* *
(*Terminator* | *Decision-node*)

Graph-node ::
Task-node |
Output-node |
Create-request-node |
Call-node |
Set-node |
Reset-node

Terminator ::
Nextstate-node |
Stop-node |
Return-node

Grammaire textuelle concrète

<transition> ::=
{<transition string> [<terminator statement>] }
| <terminator statement>

<transition string> ::=
{<action statement>}+

<action statement> ::=
[<label>] <action> <end>

<action> ::=
<task>
| <output>
| <create request>
| <decision>
| <transition option>
| <set>
| <reset>
| <export>
| <procedure call>
| <remote procedure call>

<terminator statement> ::=
[<label>] <terminator> <end>

<terminator> ::=
<nextstate>
| <join>
| <stop>
| <return>

Lorsque le terminateur <terminator> d'une <transition> est omis, la dernière action dans la <transition> doit contenir une décision <decision> terminale (voir 2.7.5) ou une option <transition option> terminale, sauf pour toutes les <transition> contenues dans les décisions <decision> et les options <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> }

<transition string area> ::=
{ <task area>
| <output area>
| <set area>
| <reset area>
| <export 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 le processus.

La zone <transition area> correspond à *Transition* et la zone <transition string area> correspond au *Graph-node**.

Sémantique

Une transition réalise une séquence d'actions. Pendant la transition, les données du processus peuvent être manipulées et des signaux peuvent être envoyés. La transition s'arrêtera lorsque le processus entrera dans un état, ou lors d'un arrêt ou d'un retour ou d'un transfert de contrôle à une autre transition.

2.6.8.2 Terminateur de transition

2.6.8.2.1 Etat suivant

Grammaire abstraite

Nextstate-node :: *State-name*

Le nom *State-name* spécifié dans un état suivant doit porter le même nom que l'état à l'intérieur du même graphe *Process-graph, Service-graph* ou *Procedure-graph.*

Grammaire textuelle concrète

<nextstate>::=
nextstate <nextstate body>

<nextstate body> ::=
{<state name> | <dash nextstate>}

Grammaire graphique concrète

<nextstate area> ::=
<state symbol> ***contains*** <nextstate body>

Sémantique

Un état suivant représente un terminateur d'une transition. Il spécifie l'état qu'aura le processus, le service ou la procédure lors de l'achèvement d'une transition.

2.6.8.2.2 Branchement

Un branchement modifie le flot dans un corps <body> en exprimant que l'instruction <action statement> suivante qui doit être interprétée est celle qui contient le même nom <connector name>.

Grammaire textuelle concrète

<join> ::=
join <connector name>

On doit avoir exactement un seul nom <connector name> correspondant à un branchement <join> à l'intérieur du même corps <body>. La règle concernant le corps <process type body> est énoncée en 6.3.1.

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 <process graph area>, on doit avoir une seule et unique zone <in-connector area> dans cette zone <process graph area>. La règle concernant la zone <process type graph area> est énoncée en 6.3.1.

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

Dans la syntaxe abstraite, un branchement <join> ou une zone <out-connector area> provient de la chaîne <transition string> dans laquelle la première instruction <action statement> ou zone d'action a le même nom <connector name> associé.

2.6.8.2.3 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 immédiat du processus ou du service qui l'interprète.

Dans le cas d'un processus, cela signifie que les signaux bloqués à l'accès d'entrée sont supprimés et que les variables et les temporisateurs créés pour le processus, l'accès d'entrée et le processus cessent d'exister.

Dans le cas d'un service, cela signifie que les signaux du service seront écartés jusqu'à ce que l'instance de processus cesse d'exister.

2.6.8.2.4 Retour

Grammaire abstraite

Return-node :: ( )

Un nœud *Return-node* doit être situé à l'intérieur d'un graphe *Procedure-graph.*

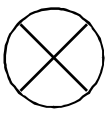
Grammaire textuelle concrète

<return> ::=
return [<expression>]

Grammaire graphique concrète

<return area> ::=
<return symbol>
[***is associated with*** <expression>]

<return symbol> ::=



L'<expression> dans un retour <return> ou dans une zone <return area> est permise si et seulement si la portée qui l'englobe est un opérateur ou une procédure ayant un résultat <procedure result>.

Sémantique

Un nœud *Return-node* est interprété de la manière suivante:

a) toutes les variables créées par l'interprétation du nœud *Procedure-start-node* cesseront d'exister;

b) l'interprétation du graphe *Procedure-graph* est terminée et l'instance de procédure cesse d'exister;

c) désormais, l'interprétation du processus, du service ou de la procédure appelant continue au nœud suivant l'appel.

Modèle

Le modèle de retour avec une expression est défini en 2.4.6.

2.7 Action

2.7.1 Tâche

Grammaire abstraite

Task-node :: *Assignment-statement* |
Informal-text

Grammaire textuelle concrète

<task> ::=
task <task body>

<task body> ::=
{<assignment statement>{**,**<assignment statement>}*}
| {<informal text> {,<informal text>}*}

Grammaire graphique concrète

<task area> ::=
<task symbol> ***contains*** <task body>

<task symbol> ::=

Sémantique

L'interprétation d'un nœud *Task-node* est l'interprétation de l'instruction *Assignment-statement* ou l'interprétation du texte *Informal-text*.

Modèle

Un corps <task body> peut contenir plusieurs instructions <assignment statement> ou textes <informal text>. Dans ce cas, il s'agit d'une syntaxe dérivée pour spécifier une séquence de tâches <task>, une pour chaque instruction <assignment statement> ou pour chaque texte <informal text> de façon que l'ordre d'origine dans lequel ils étaient spécifiés dans le corps <task body> soit maintenu.

Cette abréviation est développée avant toute extension des abréviations qui se trouvent dans les expressions contenues.

2.7.2 Création

Grammaire abstraite

Create-request-node :: *Process-identifier*
[*Expression*]*

La longueur d'[*Expression*]* doit être la même que le nombre de paramètres *Process-formal-parameter* dans la définition *Process-definition* de l'identificateur *Process-identifier*.

Chaque *expression* qui correspond positionnellement à un paramètre *Process-formal-parameter* doit avoir la même sorte que ce paramètre dans la définition *Process-definition* désignée par l'identificateur *Process-identifier*.

Grammaire textuelle concrète

<create request> ::=
create <create body>

<create body> ::=
{ <process identifier> | **this** } [<actual parameters>]

<actual parameters> ::=
([<expression>] {,[<expression>]}*)

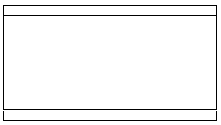
Les virgules après la dernière <expression> dans <actual parameters> peuvent être omises.

this ne peut être spécifié que dans une définition <process type definition> et dans des portées englobées par une définition <process type definition>.

Grammaire graphique concrète

<create request area> ::=
<create request symbol> ***contains*** <create body>

<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 de processus dans le même bloc. Le processus **parent** créé a la même valeur PId que le processus créateur **self**. Les expressions des processus créé **self** et créateur **offspring** ont toutes deux la même nouvelle et unique valeur PId (voir D.10.1).

Lorsqu'une instance de processus 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 5.4.3) aux paramètres formels correspondants. Le processus débute alors par l'interprétation du nœud de départ dans le graphe de processus.

Le processus créé se déroule ensuite de manière asynchrone et en parallèle avec les autres processus.

Si l'on tente de créer un nombre d'instances de processus supérieur à celui qui est spécifié par le nombre maximal d'instances dans la définition de processus, aucune nouvelle instance n'est créée, l'expression **offspring** du processus de création a la valeur Null et l'interprétation se poursuit.

Si une <expression> dans <actual parameter> est omise, le paramètre formel correspondant n'a pas de valeur associée, c'est-à-dire qu'il est non défini.

Modèle

L'énoncé **this** est une syntaxe dérivée dénotant l'identificateur implicite <process identifier> pour l'ensemble d'instances auquel appartient le processus qui exécute la création.

2.7.3 Appel de procédure

Grammaire abstraite

Call-node	::	*Procedure-identifier* [*Expression*]*
Procedure-identifier	=	*Identifier*

La longueur de l'[*Expression*]* doit être la même que le nombre de paramètres *Procedure-formal-parameters* dans la définition *Procedure-definition* de l'identificateur *Procedure-identifier.*

Chaque *Expression* correspondant par sa position à un paramètre **in** *Procedure-formal-parameter* doit avoir la même sorte que le paramètre *Procedure-formal-parameter*.

Chaque *Expression* correspondant par sa position à un paramètre **in/out** *Procedure-formal-parameter* doit avoir un identificateur *Variable-identifier* avec le même identificateur *Sort-reference-identifier* que le paramètre *Procedure-formal-parameter*.

Grammaire textuelle concrète

<procedure call> ::=
call <procedure call body>

<procedure call body> ::=
[**this**] <procedure identifier> [<actual parameters>]

Un exemple d'appel <procedure call> est donné à la Figure 2.10.13.

Une <expression> dans <actual parameters> correspondant à un paramètre formel **in/out** ne peut être omise et doit être un identificateur <variable identifier>.

Une fois le *modèle* de **this** appliqué, l'identificateur <procedure identifier> doit désigner une procédure contenant une transition de départ.

Si **this** est utilisé, l'identificateur <procedure identifier> doit dénoter une procédure englobante.

Grammaire graphique concrète

<procedure call area> ::=
<procedure call symbol> ***contains*** <procedure call body>

<procedure call symbol> ::=

La zone <procedure call area> représente le nœud *Call-node*.

Un exemple de zone <procedure call area> est donné à la Figure 2.10.14.

Sémantique

L'interprétation d'un nœud d'appel de procédure transfère l'interprétation vers la définition de procédure indiquée dans le nœud d'appel et le graphe de procédure est interprété.

L'interprétation de la transition contenant l'appel de procédure continue lorsque l'interprétation de la procédure appelante est terminée.

Les expressions de paramètres réels sont interprétées dans l'ordre donné.

Une sémantique spéciale est nécessaire en ce qui concerne l'interprétation des données et des paramètres (l'explication est donnée en 2.4.6).

Si une <expression> dans <actual parameter> est omise, le paramètre formel correspondant n'a pas de valeur associée, c'est-à-dire qu'il est «non défini».

Modèle

Si l'identificateur <procedure identifier> n'est pas défini à l'intérieur du service ou processus englobant, l'appel de procédure est transformé en un appel d'un sous-type local créé implicitement de la procédure.

this implique que lorsque la procédure est spécialisée, l'identificateur <procedure identifier> est remplacé par l'identificateur de la procédure spécialisée.

2.7.4 Sortie

Grammaire abstraite

Output-node	::	*Signal-identifier*
		[*Expression*]*
		[*Signal-destination*]
		Direct-via
Signal-destination	=	*Expression* | *Process-identifier*
Direct-via	=	(*Signal-route-identifier* | *Channel-identifier*)-**set**

La longueur de l'[*Expression*]* doit être la même que le nombre d'identificateurs *Sort-reference-identifiers* qui figure dans la définition *Signal-definition* indiquée par l'identificateur *Signal-identifier*.

Chaque *Expression* doit avoir la même sorte que l'identificateur *Sorte-identifier-reference* correspondant (par position) qui figure dans la définition *Signal-definition*.

Pour chaque sous-ensemble homogène pouvant exister (voir 3), il doit exister au moins un trajet de communication (soit implicite vers son type de processus propre, soit explicite utilisant des trajets de signal et éventuellement des canaux) vers l'environnement ou vers un ensemble d'instances de processus ou de service ayant un identificateur *Signal-identifier* dans son ensemble de signaux d'entrée valides et provenant de l'ensemble d'instances de processus ou de service où le nœud *Output-node* est utilisé.

Pour chaque identificateur *Signal-route-identifier* dans le trajet *Direct-via*, il est nécessaire que l'*Origine* dans (l'un des) trajets *Signal-route-path(s)* du trajet de signal désigne:

a) soit un ensemble d'instances de processus qui contient le processus qui interprète le nœud *Output-node* ou qui contient un service qui interprète le nœud *Output-node*; ou bien

b) un service à l'intérieur du processus d'origine qui interprète le nœud *Output-node.*

De plus, le trajet *Signal-route-path* doit inclure l'identificateur *Signal-identifier* dans son ensemble d'identificateurs *Signal-identifiers.* Si l'*Origine* désigne un ensemble d'instances de processus et le signal est envoyé à partir d'un service, il doit alors exister un trajet de signal à l'intérieur du processus capable d'acheminer le signal du service émetteur au trajet de signal mentionné dans le trajet *Direct-via*.

Pour chaque identificateur *Channel-identifier* dans le trajet *Direct-via,* il doit exister un ou deux trajets de signal et zéro ou plusieurs canaux de sorte que le canal soit atteignable via ce trajet avec l'identificateur *Signal-identifier* du processus ou du service et le trajet *Channel-path* dans la direction à partir du processus doit inclure l'identificateur *Signal-identifier* dans son ensemble d'identificateurs *Signal-identifiers*.

Grammaire textuelle concrète

<output> ::=
output <output body>

<output body> ::=
<signal identifier>
[<actual parameters>]{**,** <signal identifier> [<actual parameters>]}*
[**to** <destination>] [**via** [**all**] <via path>]

<destination> ::=
<PId expression> | <process identifier> | **this**

<via path> ::=
<via path element> {, <via path element>}*

<via path element> ::=
<signal route identifier>
| <channel identifier>
| <gate identifier>

L'expression <PId expression> de l'identificateur <process identifier> représente la destination *Signal-destination*. Il existe une ambiguïté syntaxique entre <PId expression> et <process identifier>. 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 <process identifier>. Ce dernier doit désigner un processus atteignable à partir du processus d'origine.

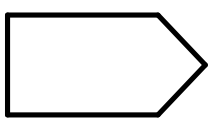
La construction **via** représente le trajet *Direct-via*. Le trajet <via path> est considéré comme une liste d'éléments <via path element>.

via all ne peut être spécifié que lorsqu'il n'existe pas de *Signal-destination.* **via all** représente une multidestination comme l'explique le *Modèle*.

this ne peut être spécifié que dans une définition <process type definition> et dans des parties englobées par une définition <process type definition>.

Grammaire graphique concrète

<output area> ::=
<output symbol> ***contains*** <output body>
<output symbol> ::=
<plain output symbol>
| <internal output symbol>
<plain output symbol> ::=



Sémantique

L'énoncé d'un identificateur *Process-identifier* dans une destination *Signal-destination* désigne la destination *Signal-destination* comme toute instance existante de l'ensemble des instances de processus désigné par l'identificateur *Process-identifier*. S'il n'existe pas d'instances, le signal est écarté.

Si aucun identificateur *Signal-route-identifier* n'est spécifié dans le trajet *Direct-via* et qu'aucune destination *Signal-destination* n'est spécifiée, tout processus pour lequel il existe un trajet de communication peut recevoir le signal.

Les valeurs acheminées par l'instance du signal sont les valeurs des paramètres réels dans la sortie. Si une <expression> dans <actual parameters> est omise, aucune valeur n'est acheminée avec la place correspondante de l'instance du signal, c'est-à-dire que la place correspondante est «non définie».

La valeur du PId du processus d'origine est également acheminée 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'intervalles définie en 5.3.1.9.1 s'applique à l'expression.

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** à l'ensemble des trajets mentionnés dans le trajet *Direct-via*.

Si la destination *Signal-destination* est *Expression*, l'instance de signal est délivrée à l'instance de processus désignée par *Expression*. Si cette instance n'existe pas, ou si elle est non atteignable à partir du processus d'origine, l'instance du signal est écartée.

Si la destination *Signal-expression* est *Process-identifier*, l'instance du signal est délivrée à une instance arbitraire de l'ensemble d'instances de processus désignés par l'identificateur *Process-identifier*. S'il n'existe pas de telle instance, l'instance du signal est écartée.

A titre d'exemple, si la destination *Signal-destination* est spécifiée comme Null dans un nœud *Output-node,* l'instance du signal sera écartée 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 de processus 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 de processus. S'il n'est pas possible de sélectionner une instance, l'instance du signal est écartée.

Il faut noter que si l'on spécifie le même identificateur *Channel-identifier* ou le même identificateur *Signal-route-identifier* dans le trajet *Direct-via* de deux nœuds *Output-nodes,* cela ne signifie pas automatiquement que les signaux sont mis dans une file d'attente à l'accès d'entrée dans le même ordre que celui où les nœuds *Output-nodes* 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 ou encore si le processus ou service d'origine et le processus ou service de destination sont définis dans le même bloc.

Modèle

Si plusieurs paires (identificateur <signal identifier> 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 de (identificateur <signal identifier> paramètres <actual parameters>). La clause **to** et la clause **via** sont répétées dans chacune des sorties <output> ou zones <output area>.

L'énoncé **via all** constitue une syntaxe dérivée pour l'acheminement multidestinataire du signal via les trajets de communication mentionnés dans le trajet <via path>; les signaux sont envoyés dans le même ordre d'apparition des éléments <via path element>, et à raison d'un signal par élément <via path element>. La multidestination désigne une séquence de sorties du même signal. Les valeurs acheminées avec chaque instance de signal résultant ne sont évaluées qu'une seule fois avant l'interprétation de la première sortie. Des variables implicites sont ensuite utilisées pour mémoriser les valeurs à utiliser avec chaque sortie. Si un même élément <via path element> apparaît plusieurs fois dans un trajet <via path>, un signal est émis pour chaque apparition.

L'énoncé **this** constitue une syntaxe dérivée pour désigner en tant qu'identificateur <process identifier> l'identificateur implicite <process identifier> pour l'ensemble des instances auquel appartient le processus exécutant la sortie.

2.7.5 Décision

Grammaire abstraite

Decision-node	::	*Decision-question* *Decision-answer*-**set** [*Else-answer*]
Decision-question	=	*Expression* | *Informal-text*
Decision-answer	::	(*Range-condition* | *Informal-text*) *Transition*
Else-answer	::	*Transition*

Les conditions *Range-conditions* des réponses *Decision-answers* doivent s'exclure mutuellement et les expressions *Ground-Expressions* des conditions *Range-conditions* doivent être de la même sorte.

Si la question *Decision-question* est une *Expression*, la condition *Range-condition* des réponses *Decision-answers* doit être de la même sorte que la question *Decision-question*.

Grammaire textuelle concrète

<decision> ::=
decision <question> <end> <decision body> **enddecision**

<decision body> ::=
{<answer part> <else part>}
| {<answer part> {<answer part>}+ [<else part>] }

<answer part> ::=
([<answer>]) : [<transition>]

<answer> ::=
<range condition> | <informal text>

<else part> ::=
else : [<transition>]

<question> ::=
<question expression> | <informal text> | **any**

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> contient une <transition> où 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.

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>.

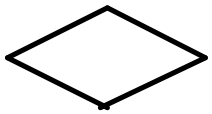
Le contexte de la décision (c'est-à-dire la sorte) est déterminé sans considération pour les réponses <answer> qui sont des chaînes <character string>.

Si une chaîne <character string> de la <question> ou de toute réponse <answer> contient des caractères de commande, la chaîne est interprétée comme un texte <informal text>.

Grammaire graphique concrète

<decision area> ::=
<decision symbol> ***contains*** <question>
is followed by
{ {<graphical answer part> <graphical else part>} ***set***
| {<graphical answer part> {<graphical answer part>}+
[<graphical else part>] } ***set*** }

<decision symbol> ::=



<graphical answer> ::=
[<answer>] | ([<answer>])

<graphical answer part> ::=
<flow line symbol> ***is associated with*** <graphical answer>
is followed by <transition area>

<graphical else part> ::=
<flow line symbol> ***is associated with*** **else**
is followed by <transition area>

Les termes réponse <graphical answer> et **else** peuvent être placés le long du symbole <flow line symbol> associé, ou dans le symbole <flow line symbol> interrompu.

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 la réponse <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 la valeur donnée 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 la valeur de l'expression sur laquelle la question est posée, n'est pas couverte par les valeurs ou l'ensemble des valeurs spécifiées dans les autres réponses.

Lorsque la réponse *Else-answer* n'est pas spécifiée, la valeur obtenue à partir de l'évaluation de l'expression de la question doit correspondre à l'une des réponses.

Modèle

Si une décision <decision> n'est pas une décision terminale, il s'agit donc d'une syntaxe dérivée pour une décision <decision> lorsque toutes les parties <answer part> et la partie <else part> ont inséré dans 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 mot clé **any** dans une décision <decision> est une abréviation de l'utilisation de l'expression <anyvalue 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 une 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>.

2.8 Temporisateur

Grammaire abstraite

Timer-definition	::	*Timer-name* *Sort-reference-identifier**
Timer-name	=	*Name*
Set-node	::	*Time-expression* *Timer-identifier* *Expression**
Reset-node	::	*Timer-identifier* *Expression**
Timer-identifier	=	*Identifier*
Time-expression	=	*Expression*

Les sortes de l'expression *Expression** dans le nœud *Set-node* et le nœud *Reset-node* doivent correspondre par leur position à l'identificateur *Sort-reference-identifier** suivant directement le nom *Timer-name* identifié par l'identificateur *Timer-identifier*.

Grammaire textuelle concrète

<timer definition> ::=
timer
<timer definition item> {,<timer definition item>}* <end>

<timer definition item> ::=
<timer name> [<sort list>] [:= <Duration ground expression>]

<reset> ::=
reset (<reset statement> { , <reset statement> }*)

<reset statement> ::=
<timer identifier> [(<expression list>)]

<set> ::=
set <set statement> { , <set statement> }*

<set statement> ::=
([<Time expression> ,] <timer identifier> [(<expression list>)])

Une instruction <set statement> peut omettre l'expression <Time expression> si l'identificateur <timer identifier> désigne un temporisateur qui possède dans sa définition une expression <Duration ground expression>.

Une instruction <reset statement> représente un nœud *Reset-node*; une instruction <set statement> représente un nœud *Set-node*.

Grammaire graphique concrète

<set area> ::=
<task symbol> ***contains*** <set>

<reset area> ::=

<task symbol> ***contains*** <reset>

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'opérateur «=» appliqué à toutes les expressions correspondantes dans les listes donne vrai (True) (c'est-à-dire si les deux listes d'expressions ont les mêmes valeurs).

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 du processus. 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 la même valeur que l'expression **self**. Si une liste d'expressions est donnée lors de l'initialisation du temporisateur, les valeurs de ces expressions sont contenues 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 en 5.3.19.1 appliquée à l'expression correspondante dans une initialisation ou réinitialisation, doit être True (vraie), sinon, le système présente une erreur et le comportement ultérieur du système est indéterminé.

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 instruction <set statement> qui ne contient pas d'expression <Time expression> est une syntaxe dérivée pour une instruction <set statement> dans laquelle l'expression <Time expression> est «**now** + <Duration ground expression>» où <Duration ground expression> est dérivée de la définition du temporisateur.

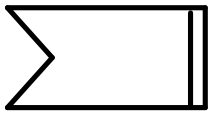
Une réinitialisation <reset> ou une initialisation <set> peut contenir plusieurs instructions <reset statement> ou <set statement> respectivement. Cela constitue une syntaxe dérivée pour spécifier une séquence de <reset> ou <set>, une pour chaque instruction <reset statement> ou <set statement> de manière à conserver l'ordre original dans lequel elles ont été spécifiées dans <reset> ou <set>. Cette abréviation est étendue avant l'extension des abréviations dans les expressions contenues.

2.9 Entrée et sortie internes

Les symboles définis dans le présent paragraphe sont spécifiés de manière à être compatibles avec les digrammes existants. Ils ne sont pas recommandés pour les nouvelles descriptions en SDL et possèdent la même sémantique que les symboles <plain input symbol> <plain output symbol> respectivement.

Grammaire graphique concrète

<internal input symbol> ::=



<internal output symbol> ::=

2.10 Exemples

```
task t3 := 0 /*example*/;
```

FIGURE 2.10.1/Z.100

Exemple de note (SDL/PR)

```
---------
task 'task1' comment 'example';
task 'task2';
---------
```

FIGURE 2.10.2/Z.100

Exemple de commentaire (SDL/PR)

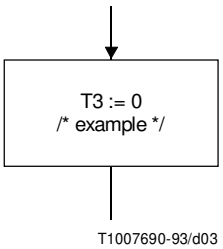


FIGURE 2.10.3/Z.100

Même exemple que celui de la Figure 2.10.1 en SDL/GR

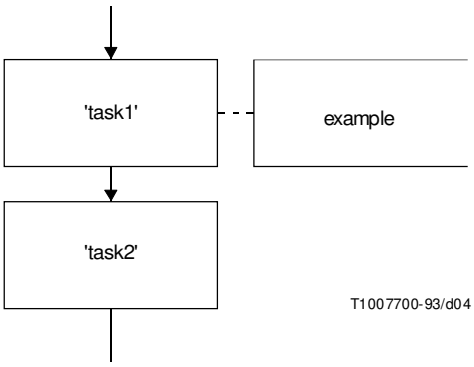


FIGURE 2.10.4/Z.100

Même exemple que celui de la Figure 2.10.2 en SDL/GR

```
system Daemongame;

   signal Newgame, Probe, Result, Endgame, Gameid, Win, Lose,

      Score (Integer);

   channel Gameserver.in

      from env to Game

      with Newgame, Probe, Result, Endgame;

   endchannel Gameserver.in;

   channel Gameserver.out

      from Game to env

      with Gameid, Win, Lose, Score;

   endchannel Gameserver.out;

   block Game referenced;

endsystem Daemongame;
```

FIGURE 2.10.5/Z.100

Exemple de spécification de système (SDL/PR)

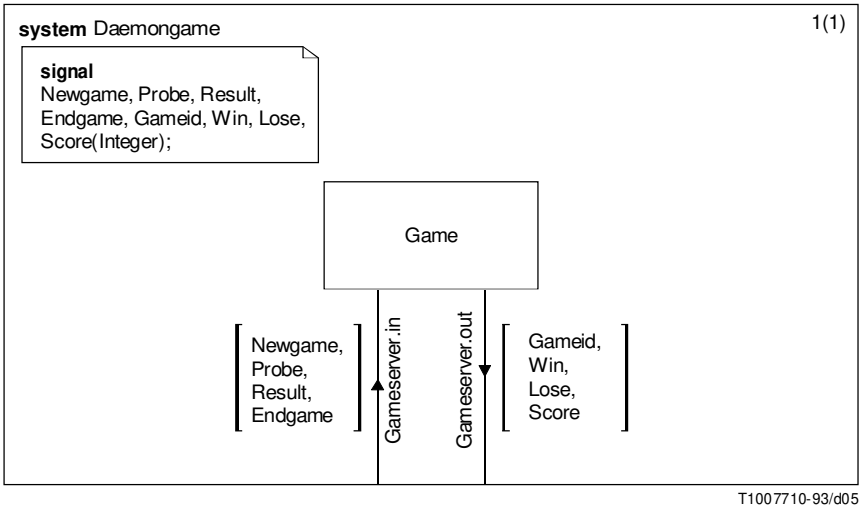


FIGURE 2.10.6/Z.100

Exemple de spécification de système (SDL/GR)

```
block Game;
   signal Gameover(PId);
   connect Gameserver.in and R1,R2;
   connect Gameserver.out and R3;
   signalroute R1 from env to Monitor with Newgame;
   signalroute R2 from env to Game with Probe, Result, Endgame;
   signalroute R3 from Game to env
      with Gameid, Win, Lose, Score;
   signalroute R4  from Game to Monitor with Gameover;

   process Monitor (1,1) referenced;

   process Game (0,) referenced;
endblock Game;
```

FIGURE 2.10.7/Z.100

Exemple de spécification de bloc (SDL/PR)

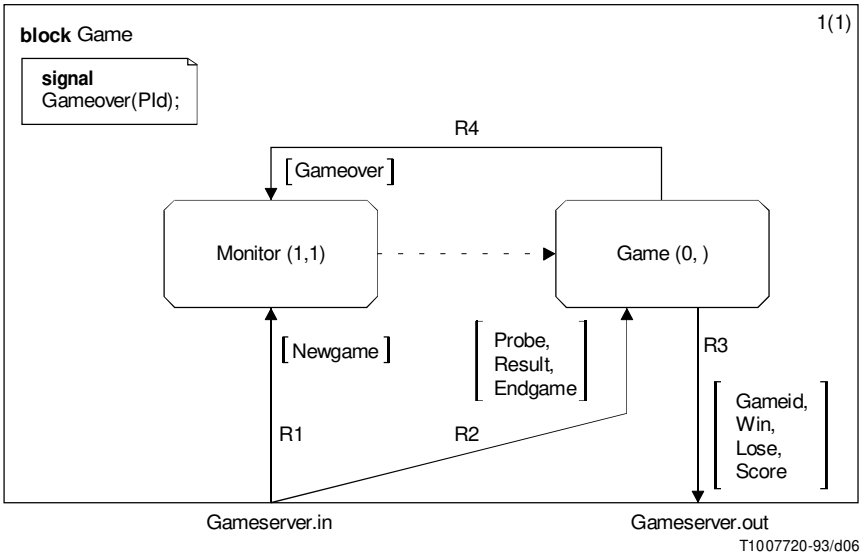


FIGURE 2.10.8/Z.100

Exemple de spécification de bloc (SDL/GR)

```
process Game (0, ); fpar player Pid;

dcl count Integer := 0 ; /*counter to keep track of score */

start;
     output Gameid to player;
     nextstate even;

state even;
   input none;
     nextstate odd;
  input Probe;
     output Lose to player;
     task count:= count-1;
     nextstate -;

state odd;
  input Probe;
     output Win to player;
     task count:= count+1;
     nextstate -;
  input none;
     nextstate even;

state even, odd;
  input Result;
     output Score(count) to player;
     nextstate -;
  input Endgame;
     output Gameover(player);
     stop;

endprocess Game;
```

FIGURE 2.10.9/Z.100

Exemple de spécification de processus (SDL/PR)

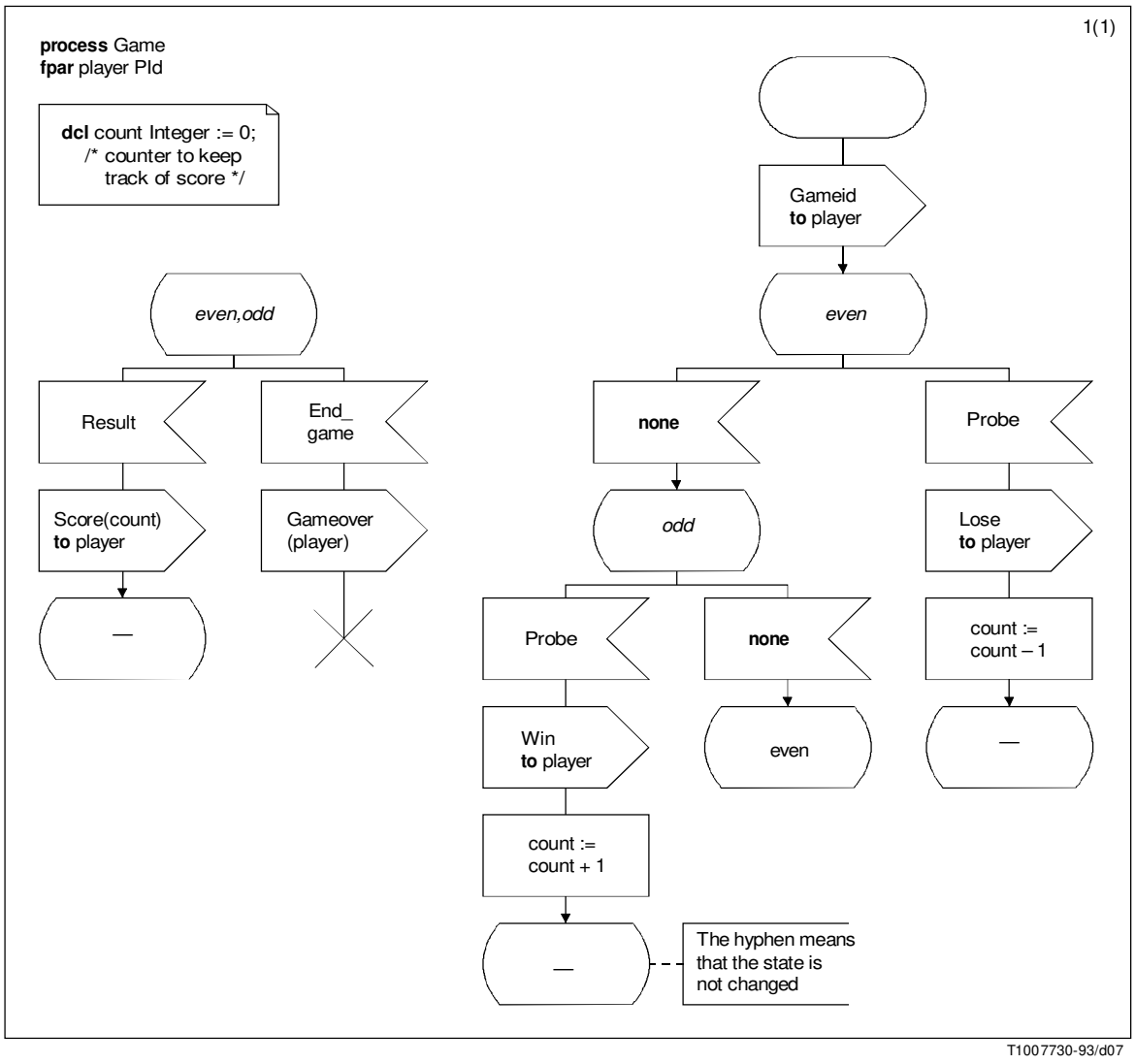


FIGURE 2.10.10/Z.100

Exemple de spécification de processus (SDL/GR)

```
procedure Check;
   fpar in/out x, y Integer;
   /* The following signal definitions are assumed:
   signal sig1(Boolean), sig2, sig3(Integer,PId);    */
   dcl   sum, index Integer,
         nice Boolean;
   start;
         task sum := 0, index := 1;
      nextstate idle;
   state idle;
      input sig1(nice);
      decision nice;
         (True): task 'Calculate sum';
           output sig3(sum, sender);
           return;
         (False): nextstate problems;
      enddecision;
      input sig2;
      join 1;
   .........
endprocedure Check;
```

FIGURE 2.10.11/Z.100

Exemple de fragment de spécification de procédure (SDL/PR)

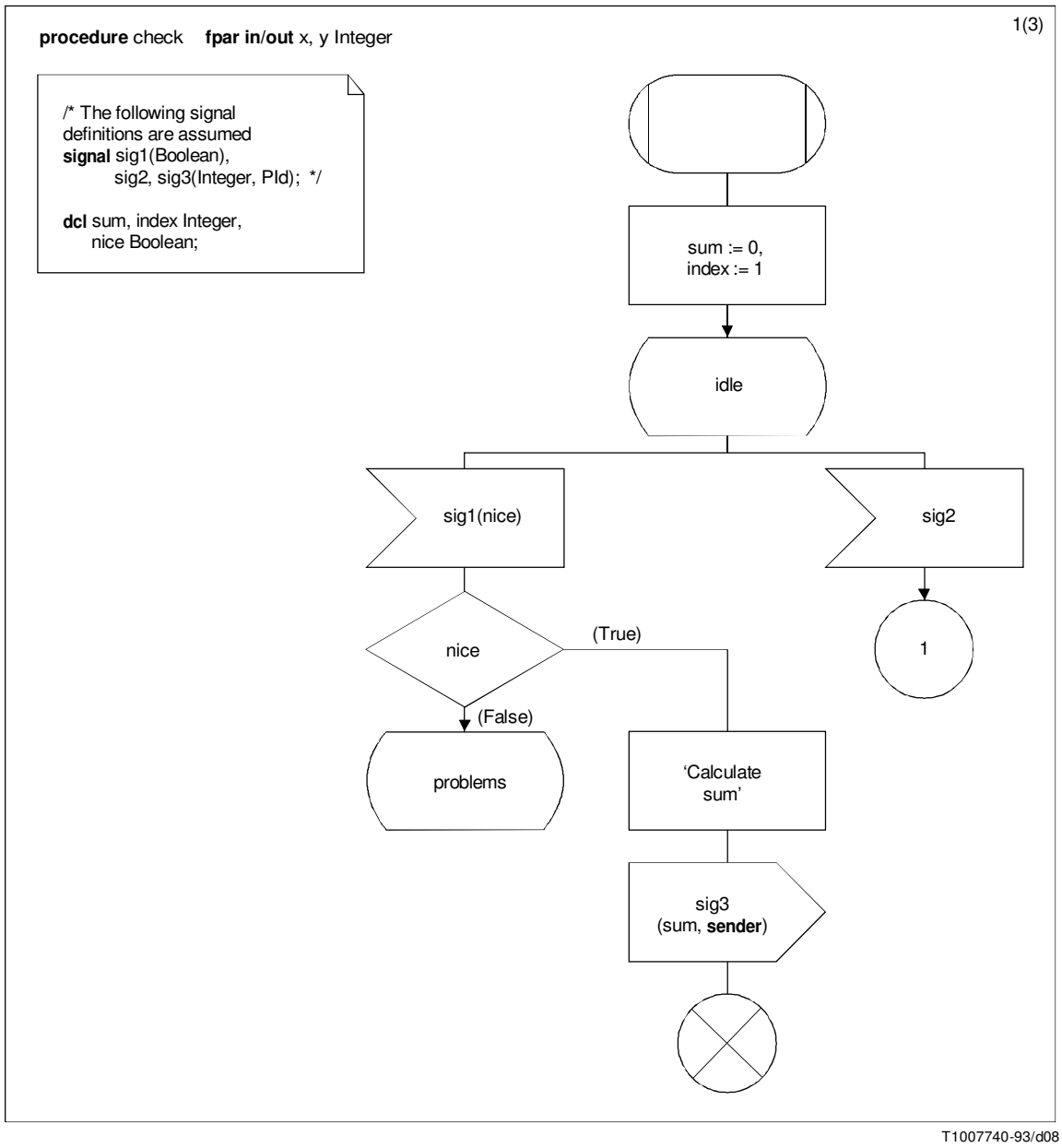


FIGURE 2.10.12/Z.100

Exemple de fragment de spécification de procédure (SDL/GR)

```
/* The following signal definition is assumed:
signal inquire(Integer,Integer,Integer);  */
process alfa;
   dcl a,b,c Integer;
       .....
       ......
   input inquire (a,b,c);
      call check (a,b);
   ......
endprocess;
```

FIGURE 2.10.13/Z.100

Exemple d'appel de procédure dans un fragment de définition de processus (SDL/PR)

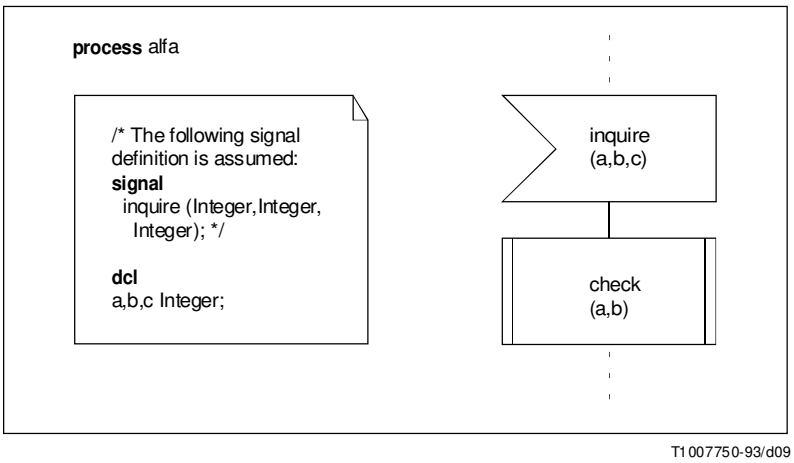


FIGURE 2.10.14/Z.100

Exemple d'appel de procédure dans un fragment de définition de processus (SDL/GR)

Les Figures 2.10.15 à 2.10.20 ci-dessous donnent un exemple d'une définition <process definition> contenant des définitions <service definition> avec les définitions <service definition> correspondantes. Ce processus a le même comportement que le processus donné aux Figures 2.10.9 et 2.10.10.

```
process Game;
   fpar player Pid;
   signal Proberes (Integer);

   signalroute IR1 from Game_handler to env with Score,Gameid;
   signalroute IR2 from Game_handler to env with Gameover;
   signalroute IR3 from env to Game_handler with Result,Endgame;
   signalroute IR4 from env to Probe_handler with Probe;
   signalroute IR6 from Probe_handler to env with Lose,Win;
   signalroute IR7 from Probe_handler to Game_handler with Proberes;

   connect R2 and IR3,IR4;
   connect R3 and IR1,IR6;
   connect R4 and IR2;

   service Game_handler referenced;
   service Probe_handler referenced;

endprocess Game;
```

FIGURE 2.10.15/Z.100

Exemple de processus (le même que celui de la Figure 2.10.9) décomposé en services (SDL/PR)

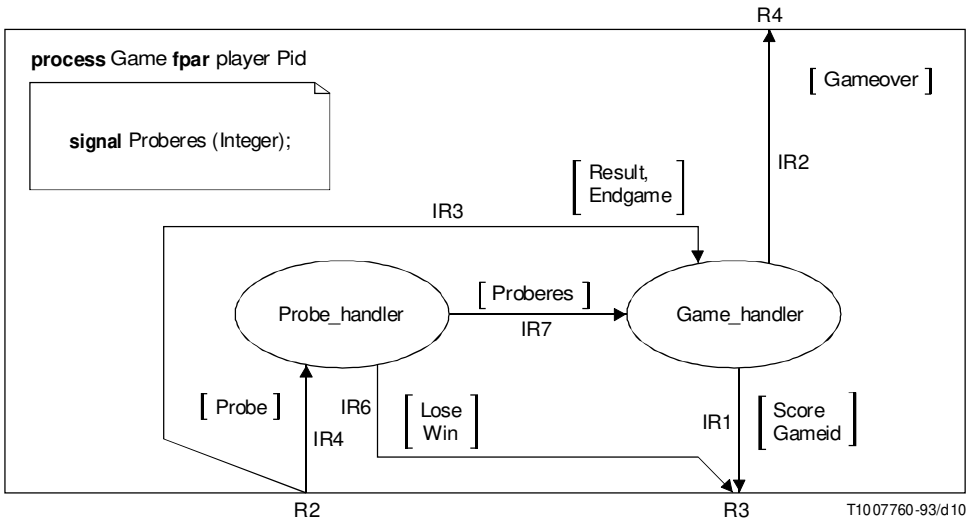


FIGURE 2.10.16/Z.100

Exemple de processus (le même que celui de la Figure 2.10.10) décomposé en services (SDL/GR)

```
service Game_handler;

  /* The service handles a game with actions to start a game, to end
  a game, to keep track of the score and to communicate the score */

  dcl   count Integer,     /*Counter to keep track of the score*/
        a Integer;         /* counter increment */

  start;
     output Subscr;
     output Gameid to player;
     task count:=0;
     nextstate started;
  state started;
     priority input Proberes(a);
          task count:=count+a;
          nextstate -;
     input Result;
          output Score(count) to player;
          nextstate -;
     input Endgame;
          output Gameover;
          stop;
  endstate started;
endservice Game_handler;
```

FIGURE 2.10.17/Z.100

Exemple de service (SDL/PR)

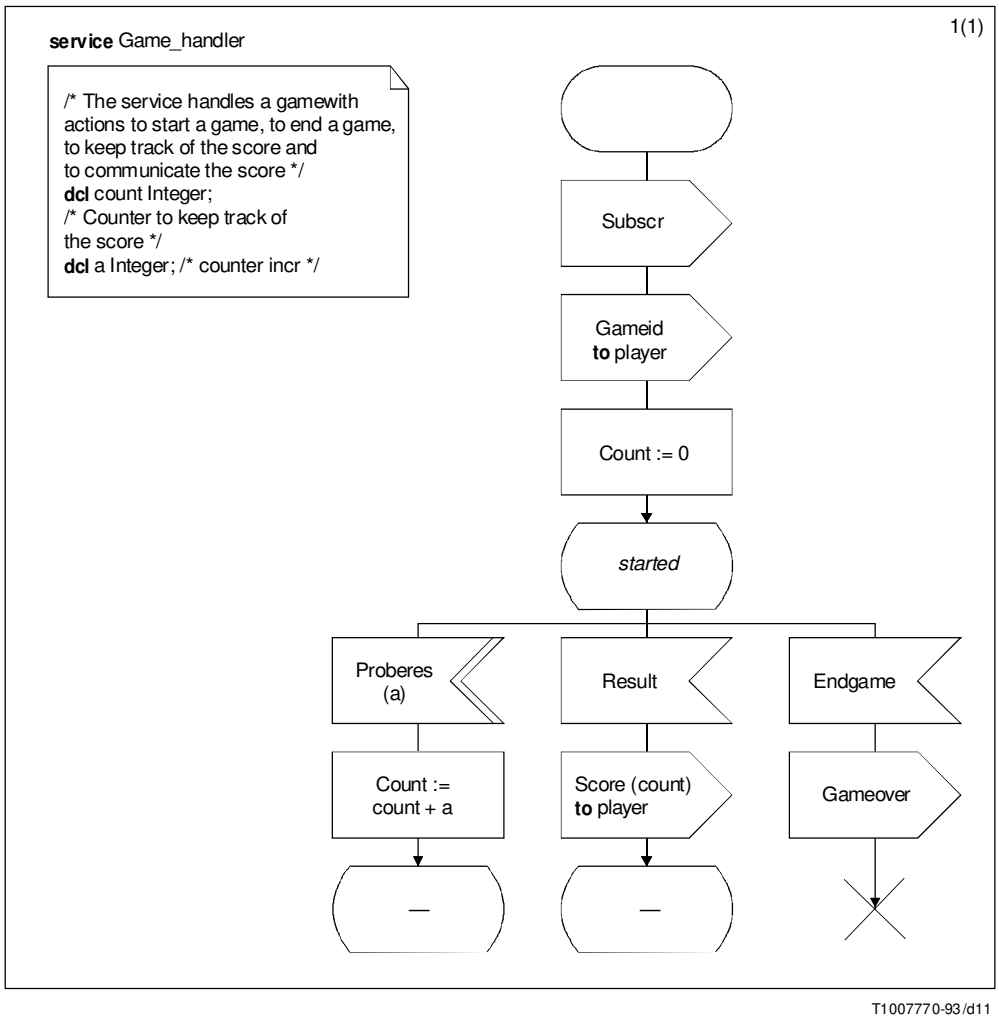


FIGURE 2.10.18/Z.100

Exemple de service (le même que celui de la Figure 2.10.17) (SDL/PR)

```
service Probe_handler;

/* The service has actions to register the bumps and to handle probes from the player.
   The probe result is sent to the player but also to the service Game_handler */
   start;
      nextstate even;
   state even;
      input Probe;
         output Lose to player;
         output Proberes(-1) to self;
         nextstate -;
      input none;
         nextstate Odd;
   endstate even;
   state odd;
      input none;
         nextstate even;
      input Probe;
         output Win to player;
         output Proberes(1) to self;
         nextstate -;
   endstate Odd;
endservice Bump_handler;
```

FIGURE 2.10.19/Z.100

Exemple de service (SDL/PR)

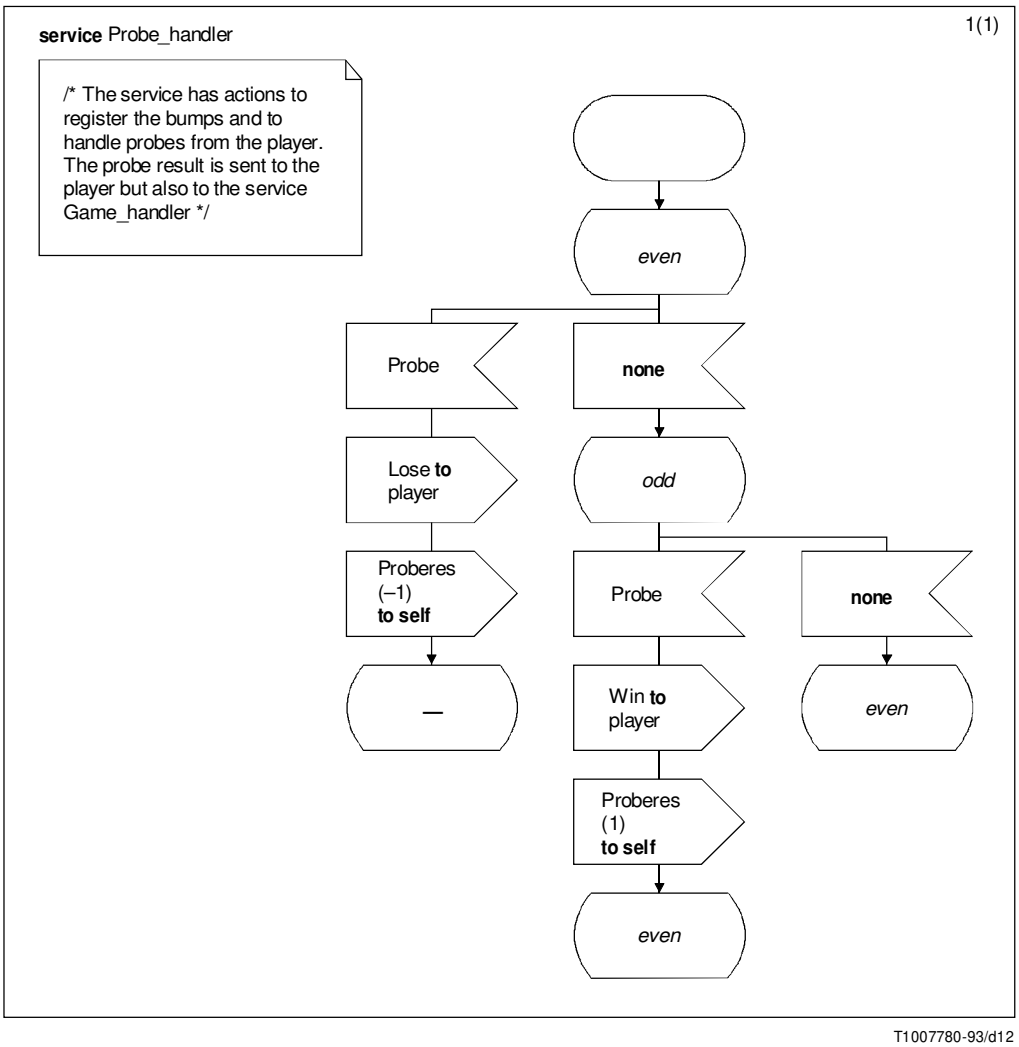


FIGURE 2.10.20/Z.100

Exemple de service (le même que celui de la Figure 2.10.19) (SDL/GR)

3 Concepts structurels dans le SDL

3.1 Introduction

Le présent article a pour objet de définir un certain nombre de concepts dont on a besoin pour traiter des structures hiérarchiques dans le SDL. La base de ces concepts est définie en 2. Les concepts qui sont définis ci-après sont de strictes additions à ceux qui sont définis en 2.

Les concepts présentés ci-après ont pour objet de fournir aux utilisateurs du SDL le moyen de spécifier des systèmes vastes et/ou complexes. Les concepts définis en 2 conviennent pour spécifier des systèmes relativement petits, que l'on peut comprendre et traiter au seul niveau du bloc. Lorsqu'on est amené à spécifier un système plus vaste ou un système complexe, on doit subdiviser la spécification du système en unités de gestion que l'on puisse traiter et comprendre indépendamment. Il est souvent indiqué de procéder à cette subdivision en un certain nombre d'étapes, d'où résulte une structure hiérarchique des éléments qui représentent le système. En plus des concepts introduits dans le présent article, le concept de progiciel défini en 2 et les concepts de définition de type structurels définis en 6 sont particulièrement utiles pour la description de grands systèmes.

Le terme subdiviser signifie scinder une unité en plusieurs sous-unités qui sont des composantes de l'unité. Le processus de subdivision n'affecte pas l'interface statique d'une unité. Parallèlement à l'opération de subdivision, il est nécessaire d'ajouter de nouveaux détails au comportement d'un système lorsque l'on descend vers les niveaux les plus bas de la structure hiérarchique de la spécification d'un système. Cette opération s'appelle affinage.

3.2 Subdivision

3.2.1 Considérations générales

On peut subdiviser un bloc en un ensemble de sous-blocs, de canaux et de sous-canaux. On peut aussi subdiviser un canal en un ensemble de blocs, de canaux et de sous-canaux. Dans les syntaxes concrètes, chaque définition de bloc (ou de type de bloc) et chaque définition de canal peut exister en deux versions: une version non subdivisée et une version subdivisée. Toutefois, les sous-structures de canal subissent des transformations lorsqu'on établit la correspondance avec la syntaxe abstraite. Ces deux versions ont la même interface statique, mais leur comportement peut différer dans une certaine mesure, car l'ordre des signaux de sortie peut ne pas être le même. Une définition de sous-bloc est une définition de bloc, et une définition de sous-canal est une définition de canal.

Dans une définition concrète de système et dans une définition abstraite de système, la version non subdivisée et la version subdivisée d'une définition de bloc peuvent toutes deux apparaître. Dans ce cas, une définition concrète de système contient plusieurs sous-ensembles de subdivisions cohérents, chaque sous-ensemble correspondant à une instance de système. Un sous-ensemble de subdivision cohérent est une sélection de définitions *block definitions* dans une définition *system definition* telle que:

a) si elle contient une définition *Block-definition* elle doit contenir la définition de l'unité de portée qui l'englobe s'il y en a une;

b) elle doit contenir toutes les définitions *Block-definitions* définies au niveau du système et si elle contient une définition *Sub-block-definition* d'une définition *Block-definition*, elle doit aussi contenir toutes les autres définitions *Sub-block-definitions* de cette définition *Block-definition*;

c) toutes les définitions *Block-definitions* «feuilles» de la structure résultante contiennent des définitions *Process-definitions*.

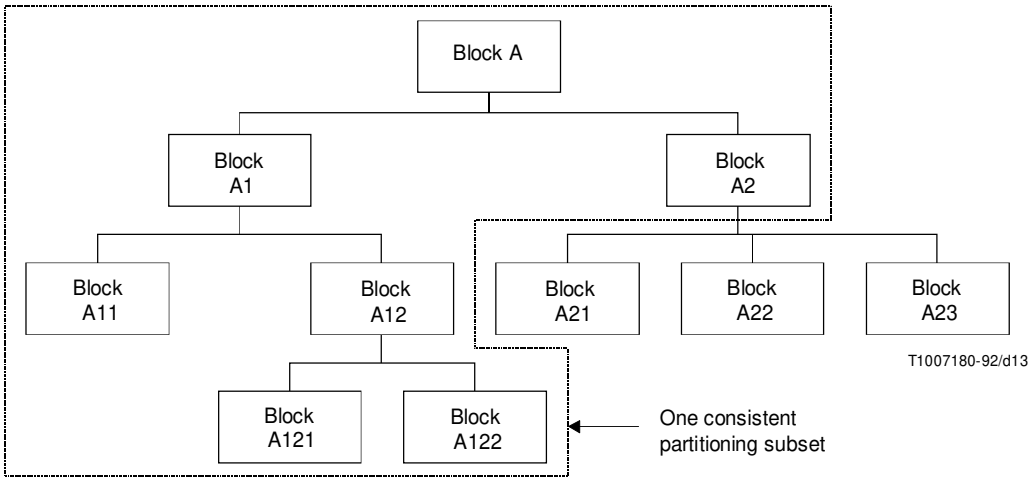


FIGURE 3.2.1/Z.100

Sous-ensemble de subdivision cohérent illustré par un diagramme auxiliaire

Au moment de l'interprétation d'un système, un sous-ensemble de subdivision cohérent est interprété. Les processus de chacun des blocs feuilles dans le sous-ensemble de subdivision cohérent sont interprétés. Si ces blocs feuilles contiennent également des sous-structures, celles-ci n'ont pas d'effet. Les sous-structures des blocs non-feuilles ont un effet sur la visibilité, et les processus à l'intérieur de ces blocs ne sont pas interprétés. Voir la Figure 3.2.1.

3.2.2 Subdivision des blocs

Grammaire abstraite

Block-substructure-definition	::	*Block-substructure-name* *Sub-block-definition*-**set** *Channel-connection*-**set** *Channel-definition*-**set** *Signal-definition*-**set** *Data-type-definition* *Syntype-definition*-**set**
Block-substructure-name	=	*Name*
Sub-block-definition	=	*Block-definition*
Channel-connection	::	*Channel-identifier*-**set** *Sub-channel-identifier*-**set**
Sub-channel-identifier	=	*Channel-identifier*
Channel-identifier	=	*Identifier*

La définition *Block-substructure-definition* doit contenir au moins une définition *Sub-block-definition*. Sauf indication contraire, on suppose dans la suite que la définition *Block-substructure-definition* contient un terme de syntaxe abstraite.

Un identificateur *Block-identifier* contenu dans une définition *Channel-definition* doit indiquer une définition *Sub-block-definition*. Une définition *Channel-definition* reliant une définition *Sub-block-definition* à la frontière de la définition *Block-substructure-definition* est appelée définition de sous-canal.

Chaque définition *Channel-definition* dans le bloc englobant rattachée à la définition *Block-substructure-definition* doit être membre d'exactement une connexion *Channel-connection*. Les identificateurs *Channel-identifiers* dans la connexion

Channel-connection doivent identifier les définitions *Channel-definition* dans la définition du bloc englobant. Chaque identificateur *Sub-channel-identifier* doit apparaître dans une connexion *Channel-connection* exactement.

Pour les signaux qui sortent de la définition *Block-substructure-definition*, l'union des ensembles *Signal-identifier*-**set**s associés à l'ensemble *Sub-channel-identifier*-**set** contenu dans une connexion *Channel-connection* doit être identique à l'union des ensembles *Signal-identifier*-**set**s associés à l'ensemble *Channel-identifier*-**set** contenu dans la connexion *Channel-connection.* La même règle s'applique aux signaux qui pénètrent dans la définition *Block-substructure-definition*. Toutefois, cette règle est modifiée en cas d'affinement du signal, voir 3.3.

Etant donné qu'une définition *Sub-block-definition* est une définition *Block-definition*, elle peut être subdivisée; cette opération peut être répétée un nombre quelconque de fois, et donne une structure hiérarchique en arbre de définitions *Block-definitions* et leurs définitions *Sub-block-definitions*. On dit que les définitions *Sub-block-definitions* d'une définition *Block-definition* existent au niveau inférieur suivant de l'arbre de blocs, voir aussi la Figure 3.2.2.

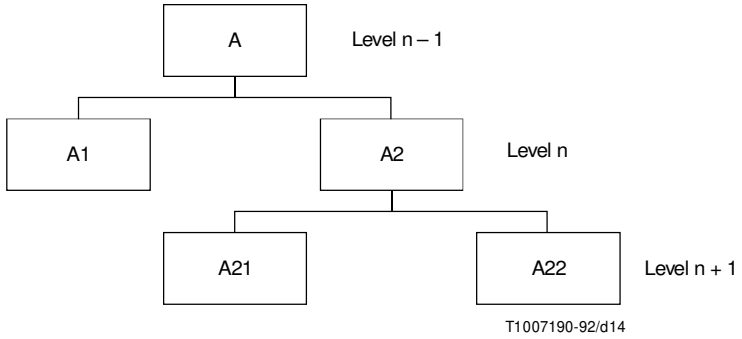


FIGURE 3.2.2/Z.100

Diagramme d'arbre de blocs

Le diagramme d'arbre de blocs est un diagramme auxiliaire.

Grammaire textuelle concrète

<block substructure definition> ::=
substructure
{[<block substructure name>]
| <block substructure identifier> } <end>
{ <entity in system> | <channel connection> }+
endsubstructure
[{ <block substructure name> | <block substructure identifier>}] <end>

Si le nom <block substructure name> après le mot clé **substructure** est omis, il est le même que le nom de la définition <block definition> ou <block type definition> englobante.

<textual block substructure reference> ::=
substructure <block substructure name> **referenced** <end>

<channel connection> ::=
connect <channel identifiers>
and <subchannel identifiers> <end>

<subchannel identifiers> ::=
<channel identifiers>

Si une définition <block substructure definition> contient des définitions <channel definition> et <textual typebased block definition>, alors chaque accès des définitions <block type definition> des définitions <textual typebased block definition> doit être connecté à un canal.

S'agissant d'une définition <block substructure definition> dans une définition <block type definition>, les connexions <channel connection> ne peuvent pas être données. Elles sont dérivées pour les définitions <textual typebased block definition> résultantes selon la définition donnée en 6.1.4.

Grammaire graphique concrète

<block substructure diagram> ::=
<frame symbol>
contains {<block substructure heading>
{ {<block substructure text area>}*
{<macro diagram>}*
<block interaction area>
{<type in system area> }*}***set*** }
is associated with {<channel identifiers>}*

Les identificateurs <channel identifiers> identifient des canaux reliés à des sous-canaux dans le diagramme <block substructure diagram>. Ils sont placés à l'extérieur du symbole <frame symbol>, à proximité du point d'extrémité des sous-canaux, sur le symbole <frame symbol>.

Un symbole <channel symbol> à l'intérieur du symbole <frame symbol> et relié à ce dernier, désigne un sous-canal.

<block substructure heading> ::=
substructure
{<block substructure name> | <block substructure identifier>}

<block substructure text area> ::=
<system text area>

<block substructure area> ::=
<graphical block substructure reference>
| <block substructure diagram>
| <open block substructure diagram>

<graphical block substructure reference> ::=
<block substructure symbol> ***contains*** <block substructure name>

<block substructure symbol> ::=
<block symbol>

<open block substructure diagram> ::=
{ {<block substructure text area>}*
{<macro diagram>}*
<block interaction area>} ***set***

Lorsqu'une zone <block substructure area> est un diagramme <open block substructure diagram>, le diagramme <block diagram> ou <block type diagram> englobant ne doit pas contenir d'autres définitions que celles des paramètres de contexte formel, des accès et de la sous-structure.

Sémantique

Voir 3.2.1.

Modèle

Un diagramme <open block substructure diagram> est transformé en un diagramme <block substructure diagram> de manière telle que dans l'en-tête <block substructure heading>, le nom <block substructure name> soit le même que nom <name> du diagramme <block diagram> ou <block type diagram> englobant.

Exemple

La Figure 3.3.1 illustre un exemple de diagramme <open block substructure diagram>.

Un exemple de définition <block substructure definition> est donné ci-après.

```
substructure A ;
   signal s5(nat), s6, s8, s9(min);
   block a1 referenced;
   block a2  referenced;
   block a3  referenced;
   channel c1    from a2 to env with s1, s2; endchannel c1;
   channel c2    from env to a1 with s3;
                 from a1 to env with s1; endchannel c2;
   channel d1    from a2 to env with s7; endchannel d1;
   channel d2    from a3 to env with s10; endchannel d2;
   channel e1    from a1 to a2 with s5, s6; endchannel e1;
   channel e2    from a3 to a1 with s8; endchannel e2;
   channel e3    from a2 to a3 with s9; endchannel e3;
   connect c and c1, c2 ;
   connect d and d1, d2 ;
endsubstructure  A;
```

Le diagramme <block substructure diagram> du même exemple est représenté par la Figure 3.2.3.

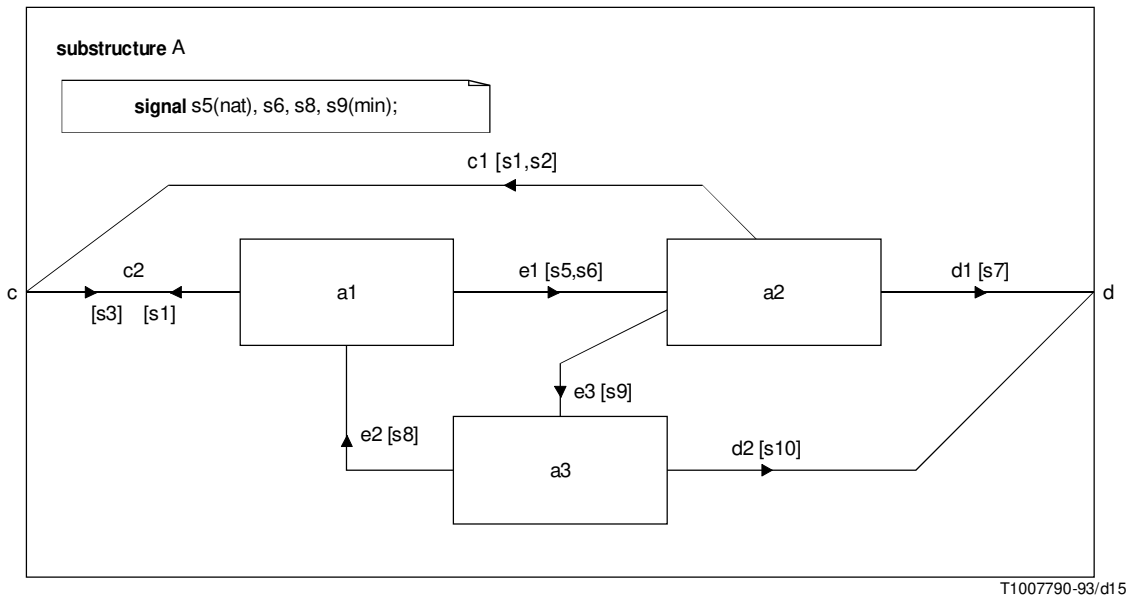


FIGURE 3.2.3/Z.100

Diagramme de sous-structure de bloc pour le bloc A

3.2.3 Subdivision des canaux

Toutes les conditions statiques sont énoncées en utilisant la grammaire textuelle concrète. Des conditions analogues sont également valables pour la grammaire graphique concrète.

Grammaire textuelle concrète

<channel substructure definition> ::=
substructure {[<channel substructure name>]
| <channel substructure identifier> } <end>
{ <entity in system>
| <channel endpoint connection> }+
endsubstructure [{ <channel substructure name>
| <channel substructure identifier>}] <end>

Si le nom <channel substructure name> après le mot clé **substructure** est omis, il est le même nom que le nom <channel name> dans la définition <channel definition> englobante.

<textual channel substructure reference> ::=
substructure <channel substructure name> **referenced** <end>

<channel endpoint connection> ::=
connect {<block identifier> | **env**}
and <subchannel identifiers> <end>

Les deux points d'extrémité de la définition <channel definition> subdivisée doivent être distincts et pour chaque point d'extrémité on doit avoir exactement une connexion <channel endpoint connection>. L'identificateur <block identifier> ou **env** dans une connexion <channel endpoint connection> doit identifier un des points d'extrémité de la définition <channel definition> subdivisée.

Des conditions statiques supplémentaires d'une définition <channel substructure definition> sont définies au moyen de la définition <block definition> qui résulte de la transformation décrite dans *Modèle*.

Grammaire graphique concrète

<channel substructure diagram> ::=
<frame symbol>
contains {<channel substructure heading>
{ {<channel substructure text area>}*
{<macro diagram>}*
{<type in system area>}*
<block interaction area> } ***set*** }
is associated with {<block identifier> | **env**}+

L'identificateur <block identifier> ou **env** désigne un point d'extrémité du canal subdivisé. L'identificateur <block identifier> est placé à l'extérieur du symbole <frame symbol> au voisinage du point d'extrémité du sous-canal associé au symbole <frame symbol>. Le symbole <channel symbol> à l'intérieur du symbole <frame symbol> et qui est relié à ce dernier indique un sous-canal.

<channel substructure heading> ::=
substructure
{ <channel substructure name> | <channel substructure identifier>}

<channel substructure text area> ::=
<system text area>

<channel substructure association area> ::=
<dashed association symbol>
is connected to <channel substructure area>

<channel substructure area> ::=
<graphical channel substructure reference>
| <channel substructure diagram>

<graphical channel substructure reference> ::=
<channel substructure symbol> ***contains*** <channel substructure name>

<channel substructure symbol> ::=

<block symbol>

Modèle

Une définition <channel definition> qui contient une définition <channel substructure definition> est transformée en une définition <block definition> et deux définitions <channel definition> telles que:

a) Les deux définitions <channel definition> sont chacune connectées au bloc et à un point d'extrémité du canal d'origine. Les définitions <channel definition> ont des nouveaux noms distincts et chaque référence au canal d'origine dans les constructions **via** est remplacée par une référence au nouveau canal approprié. Les deux canaux implicites sont des canaux à retard si et seulement si le canal subdivisé est un canal à retard.

b) La définition <block definition> a un nouveau nom distinct et contient uniquement une définition <block substructure definition> portant le même nom et contenant les mêmes définitions que la définition <channel substructure definition> d'origine. Les qualificatifs de la nouvelle définition <block definition> sont modifiés afin d'inclure le nom de bloc. Les deux connexions <channel endpoint connection> de la définition <channel substructure definition> d'origine sont représentées par deux connexions <channel connection> dans lesquelles l'identificateur <block identifier> ou **env** est remplacé par le nouveau canal approprié.

c) Les sorties <output> à l'intérieur de la sous-structure du canal qui mentionne le canal dans un trajet <via path> ont leur clause via qui mentionne l'identificateur <channel identifier> remplacée par une clause via contenant un ou deux des canaux implicites du trajet <via path> de sorte qu'un canal qui a l'identificateur <signal identifier> dans sa liste <signal list> pour une direction à partir de la définition <block definition> se trouve dans le trajet <via path>. Dans le cas où l'identificateur <channel identifier> est remplacé par deux canaux implicites dans le trajet <via path>, cela se produit dans un ordre arbitraire.

Exemple

Un exemple de définition <channel substructure definition> est donné ci-après.

```
channel C from A to B  with s1;
       from B to A with s2;
       substructure C;
        signal s3(hel), s4(boo), s5;
        block b1 referenced;
        block b2 referenced;
        channel c1 from env to b1 with s1;
                   from b1 to env with s2; endchannel c1;
        channel c2   from b2 to env with s1;
                   from env to b2 with s2; endchannel c2;
        channel e1   from b1 to b2 with s3; endchannel e1;
        channel e2   from b2 to b1 with s4, s5; endchannel e2;
        connect A and c1;
        connect B and c2;
       endsubstructure C;
 endchannel C;
```

Le diagramme <channel substructure diagram> pour cet exemple est représenté par la Figure 3.2.4.

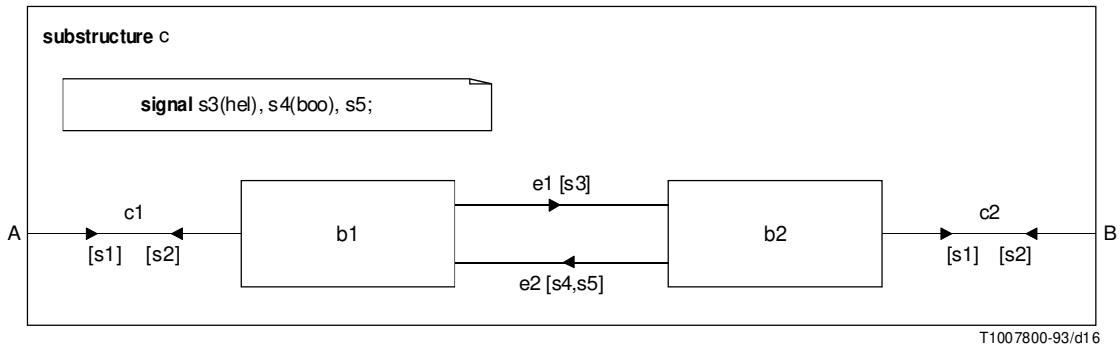


FIGURE 3.2.4/Z.100

Diagramme de sous-structure de canal pour le canal C

3.3 Affinage

L'affinage s'applique sur la définition de signal dans un ensemble de définitions de sous-signal. Une définition de sous-signal est une définition de signal et peut être affinée. Cet affinage peut être répété un nombre quelconque de fois, donnant une structure hiérarchique des définitions de signal et de leurs définitions de sous-signal (directes ou indirectes). Une définition de sous-signal d'une définition de signal n'est pas une composante de la définition de signal.

Grammaire abstraite

Signal-refinement :: *Subsignal-definition*-**set**

Subsignal-definition :: [**REVERSE**] *Signal-definition*

Pour chaque connexion *Channel-connection* il faut que pour chaque identificateur *Signal-identifier* associé à chaque identificateur *Channel-identifier*, soit que l'identificateur *Signal-identifier* soit associé à au moins un des identificateurs *Sub-channel-identifiers*, soit que chacun de ses identificateurs de sous-signal soit associé à au moins un des identificateurs *Sub-channel-identifiers*. Il s'agit là d'une modification des règles correspondantes de subdivision.

On ne peut pas avoir sur différents niveaux d'affinage du même signal, deux signaux de l'ensemble complet des signaux d'entrée valides d'un ensemble d'instances de processus désigné par une définition de processus ou des nœuds *Output-nodes* d'une définition de processus.

Grammaire textuelle concrète

<signal refinement> ::=
refinement
{<subsignal definition>}+
endrefinement

<subsignal definition> ::=
<reverse> <signal definition>

<reverse> ::=
[**reverse**]

Une définition <signal definition> dans une définition <subsignal definition> ne peut pas avoir de paramètres de contexte formel.

Sémantique

Lorsqu'un signal est défini pour être transporté par un canal, le canal sera automatiquement le moyen de transport utilisé pour tous les sous-signaux du signal. L'affinage peut avoir lieu lorsque le canal est subdivisé ou connecté à des sous-canaux d'une sous-structure. Dans ce cas, les sous-canaux transporteront les sous-signaux à la place du signal affiné. Le sens d'un sous-signal est donné par le sous-canal qui le transporte, un sous-signal peut avoir une direction opposée au signal affiné, ce qui est indiqué par le mot clé **reverse**. Les signaux ne peuvent être affinés qu'aux connexions <channel connection> et <channel endpoint connection>.

Lorsqu'une définition de système contient un affinage de signal, le concept de sous-ensemble de subdivision cohérent se trouve restreint. On dit alors que cette définition de système contient plusieurs sous-ensembles d'affinage cohérent.

Un sous-ensemble d'affinage cohérent est un sous-ensemble de subdivision cohérent (comme défini en 3.2.1) auquel on apporte des restrictions au moyen de la règle suivante:

d) lors du choix du sous-ensemble de subdivisions cohérent, les signaux doivent être utilisés au même niveau d'affinage dans les processus qui communiquent. Cela signifie que pour chaque ensemble d'instances de processus qui peut acheminer un signal à un autre ensemble d'instances de processus via des trajets de communication, le signal doit être utilisé au même niveau d'affinage dans les deux ensembles.

Exemple

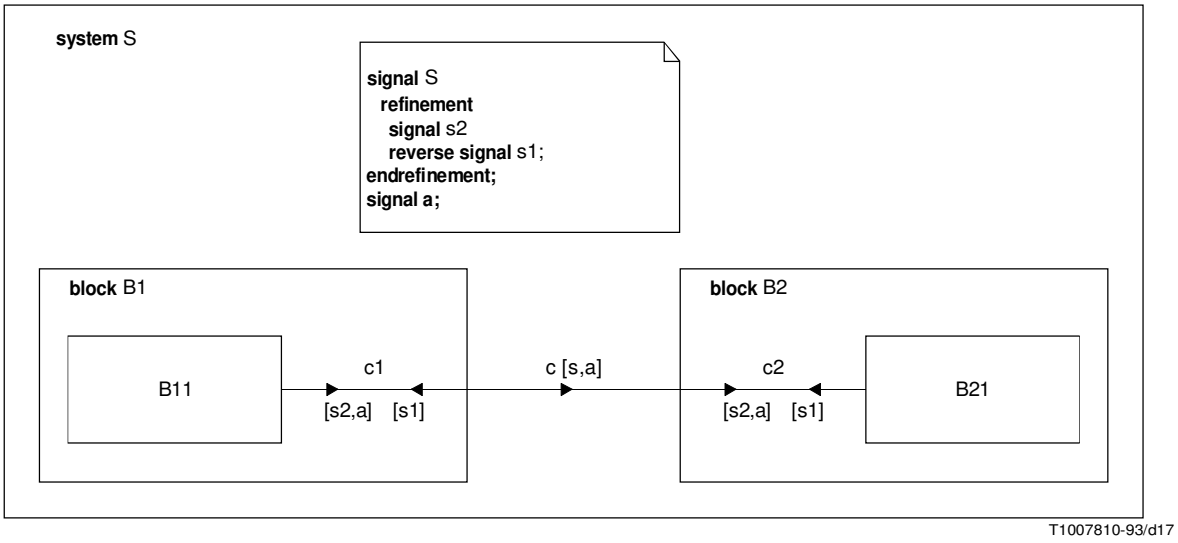


FIGURE 3.3.1/Z.100

Diagramme de système contenant un affinage de signal

Dans l'exemple de la Figure 3.3.1 ci-dessus, le signal s est affiné dans la définition de bloc B1 et B2, mais pas le signal a. Au niveau d'affinage le plus élevé, les processus dans B1 et B2 communiquent en utilisant les signaux s et a. Sur le niveau inférieur suivant, les processus en B11 et B21 communiquent en utilisant les signaux s1, s2 et a.

L'affinage dans une seule des définitions de bloc B1 et B2 n'est pas permis, étant donné qu'il n'y a pas de transformation dynamique entre un signal et ses sous-signaux, mais seulement relation statique.

4 Concepts supplémentaires dans le SDL de base

4.1 Introduction

Cet article définit un certain nombre de concepts supplémentaires. Ils sont introduits pour améliorer la facilité d'utilisation, en sus des abréviations des autres articles de la Recommandation.

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 en 7. L'ordre de transformation est également suivi dans les définitions des concepts du présent article.

4.2 Macro

Dans le texte qui suit, les termes définition de macro et appel de macro sont utilisés dans un sens général, intéressant à la fois le SDL/GR et le SDL/PR. Une définition de macro contient un ensemble de symboles graphiques ou d'unités lexicales, qui peuvent apparaître à un ou plusieurs endroits dans la 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.

4.2.1 Règles lexicales

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

4.2.2 Définition de macro

Grammaire textuelle concrète

```
<macro definition> ::=
                    macrodefinition <macro name>
                        [<macro formal parameters>] <end>
                        <macro body>
                    endmacro [<macro name>] <end>

<macro formal parameters> ::=
                    fpar <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 les mots clé **endmacro** et **macrodefinition**.

Grammaire graphique concrète

```
<macro diagram> ::=
                    <frame symbol> contains {<macro heading> <macro body area>}
```

<macro heading> ::=

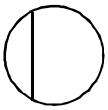
macrodefinition <macro name> [<macro formal parameters>]

<macro body area> ::=

{ {<any area> }*
<any area> [***is connected to*** <macro body port1>] }***set***
| { <any area> ***is connected to*** <macro body port2>
<any area> ***is connected to*** <macro body port2>
{ <any area> [***is connected to*** <macro body port2>]}*}***set***

<macro inlet symbol>::=

<macro outlet symbol> ::=



<macro body port1> ::=

<outlet symbol> ***is connected to*** {<frame symbol>
[***is associated with*** <macro label>]
| { <macro inlet symbol> | <macro outlet symbol> }
[{ ***contains*** <macro label>
| ***is associated with*** <macro label> }] }

<macro body port2> ::=

<outlet symbol> ***is connected to*** {<frame symbol>
is associated with <macro label>
| { <macro inlet symbol> | <macro outlet symbol> }
{ ***contains*** <macro label>
| ***is associated with*** <macro label> }}

<macro label> ::=

<name>

<outlet symbol> ::=

<dummy outlet symbol>
| <flow line symbol>
| <channel symbol>
| <signal route symbol>
| <solid association symbol>
| <dashed association symbol>
| <create line symbol>

<dummy outlet symbol> ::=

<solid association symbol>

<any area> ::=

<block area>
| <block interaction area>
| <block substructure area>
| <block substructure text area>
| <block text area>
| <block type reference>
| <channel definition area>
| <channel substructure area>

| <channel substructure association area>
| <channel substructure text area>
| <comment area>
| <continuous signal area>
| <continuous signal association area>
| <create line area>
| <create request area>
| <decision area>
| <enabling condition area>
| <existing typebased block definition>
| <export area>
| <graphical block reference>
| <graphical typebased block definition>
| <graphical procedure reference>
| <graphical process reference>
| <in-connector area>
| <input area>
| <input association area>
| <macro call area>
| <merge area>
| <nextstate area>
| <operator heading>
| <operator text area>
| <option area>
| <out-connector area>
| <output area>
| <package reference area>
| <package text area>
| <priority input area>
| <priority input association area>
| <procedure area>
| <procedure call area>
| <procedure graph area>
| <procedure start area>
| <procedure text area>
| <process area>
| <process graph area>
| <process interaction area>
| <process text area>
| <process type graph area>
| <process type reference>
| <remote procedure call area>
| <remote procedure input area>
| <remote procedure save area>
| <reset area>
| <return area>
| <save area>
| <save association area>
| <service area>
| <service interaction area>
| <service graph area>
| <service text area>
| <service type reference>
| <set area>
| <signal list area>
| <signal route definition area>
| <spontaneous transition association area>
| <spontaneous transition area>
| <start area>
| <state area>
| <stop symbol>
| <system text area>
| <task area>

| <text extension area>
| <transition area>
| <transition option area>
| <transition string area>
| <type in system area>
| <type in block area>
| <type in process area>

Aucun élément ne doit être associé à un symbole <dummy outlet symbol> sauf pour ce qui est de l'étiquette <macro label>.

Pour un symbole <outlet symbol> qui n'est pas un symbole <dummy outlet symbol>, le symbole <inlet symbol> correspondant dans l'appel de macro doit être un symbole <dummy inlet symbol>.

Un corps <macro body> peut apparaître dans un texte quelconque dont il est fait référence dans une zone <any area>.

Sémantique

Une définition <macro definition> contient des unités lexicales, un diagramme <macro diagram> contient des unités syntaxiques. Ainsi, la mise en correspondance des constructions de macro dans une syntaxe textuelle et dans une syntaxe graphique n'est généralement pas possible. Pour la même raison, des règles détaillées distinctes s'appliquent à la syntaxe textuelle et à la syntaxe graphique, bien qu'il y ait certaines règles communes.

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.

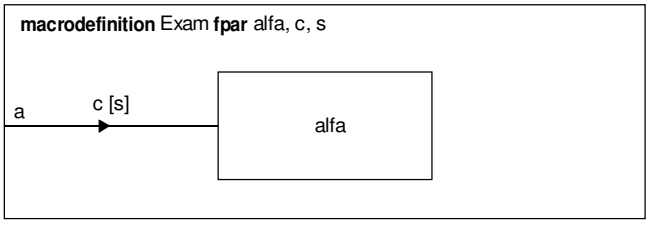
Le mot clé **macroid** peut être utilisé comme un paramètre formel de pseudo-macro à l'intérieur de chaque définition de macro. Aucun paramètre <macro actual parameter> ne peut lui être attribué, et il est remplacé par un nom <name> unique pour chaque développement de la définition de macro (à l'intérieur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

Exemple

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

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

Le diagramme <macro diagram> pour cet exemple est donné ci-après. Dans ce cas, on n'a pas besoin du paramètre <macro formal parameter> a.



T1007820-93/d18

4.2.3 Appel de macro

Grammaire textuelle concrète

<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>, alors 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.

Grammaire graphique concrète

<macro call area> ::=
<macro call symbol> ***contains*** {<macro name> [<macro call body>]}
[***is connected to***
{<macro call port1> | <macro call port2> {<macro call port2>}+}]

<macro call symbol> ::=

<macro call port1> ::=
<inlet symbol> [***is associated with*** <macro label>]
is connected to <any area>

<macro call port2> ::=
<inlet symbol> ***is associated with*** <macro label>
is connected to <any area>

<inlet symbol> ::=
<dummy inlet symbol>
| <flow line symbol>
| <channel symbol>
| <signal route symbol>
| <solid association symbol>
| <dashed association symbol>
| <create line symbol>

<dummy inlet symbol> ::=
<solid association symbol>

Aucun élément ne peut être associé avec symbole <dummy inlet symbol> sauf en ce qui concerne l'étiquette <macro label>. Pour chaque symbole <inlet symbol> on doit avoir un symbole <outlet symbol> dans le diagramme <macro diagram> correspondant, associé avec la même étiquette <macro label>. Pour un symbole <inlet symbol> qui n'est pas un symbole <dummy inlet symbol>, le symbole <outlet symbol> doit être un symbole <dummy outlet symbol>.

A l'exception du cas des symboles <dummy inlet symbol> et des symboles <dummy outlet symbol>, il est possible d'avoir plusieurs unités <lexical unit> (textuelles) associées avec un symbole <inlet symbol> ou un symbole <outlet symbol>. Dans ce cas, l'unité <lexical unit> la plus proche du symbole <macro call symbol> ou du symbole <frame

symbol> du diagramme <macro diagram> est prise pour être l'étiquette <macro label> associée au symbole <inlet symbol> ou au symbole <outlet symbol>.

La zone <macro call area> peut apparaître à un endroit quelconque où une zone est autorisée. Toutefois, un certain espace est requis entre le symbole <macro call symbol> et tout autre symbole graphique clos. Si un tel espace ne doit pas être vide conformément aux règles syntaxiques, le symbole <macro call symbol> est relié au symbole graphique clos avec un symbole <dummy inlet symbol>.

Sémantique

Une définition de système peut contenir des définitions de macro et des appels de macro. Avant de pouvoir analyser cette définition de système, 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, remplace l'appel de macro.

Lorsqu'une définition de macro est appelée, elle est développée. C'est-à-dire qu'une copie de la définition 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>.

Modèle

La zone <macro call area> est remplacée par une copie du diagramme <macro diagram> de la manière suivante: tous les symboles <macro inlet symbol> et tous les symboles <macro outlet symbol> sont supprimés. Un symbole <dummy outlet symbol> est remplacé par le symbole <inlet symbol> ayant la même étiquette <macro label>. Un symbole <dummy inlet symbol> est remplacé par le symbole <outlet symbol> ayant la même étiquette <macro label>. Ensuite, les étiquettes <macro label> attachées aux symboles <inlet symbol> et aux symboles <outlet symbol> sont supprimées. L'accès <macro body port1> et l'accès <macro body port2> qui n'ont pas d'accès <macro call port1> ou d'accès<macro call port2> sont supprimés.

Exemple

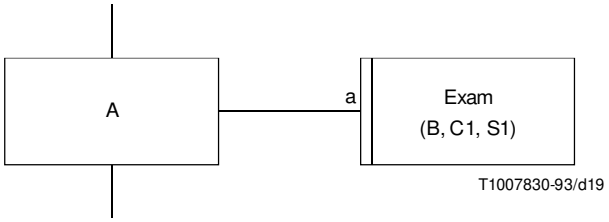
Un exemple d'appel <macro call>, dans une partie d'une définition <block definition> est donné ci-après.

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

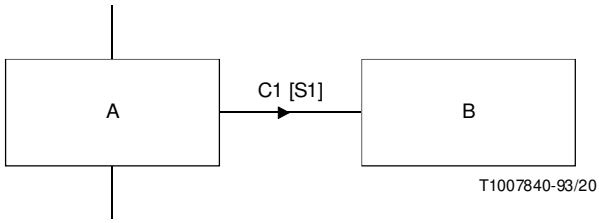
Le développement de cet appel de macro, utilisant l'exemple du § 4.2.2 donne le résultat suivant:

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

La zone <macro call area> pour le même exemple, dans une partie de la zone <block interaction area> est donnée ci-après.



Le développement de cet appel de macro donne le résultat suivant:



4.3 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 valeurs ne sont pas spécifiées. 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 une valeur 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.

4.3.1 Synonyme externe

Grammaire textuelle concrète

<external synonym definition>::=
synonym <external synonym definition item>
{, <external synonym definition item> }*

<external synonym definition item> ::=
<external synonym name> <predefined sort> = **external**

<external synonym> ::=
<external synonym identifier>

Une définition <external synonym definition> peut apparaître à tout endroit où une définition <synonym definition> est permise (voir 5.3.1.13). Un synonyme <external synonym> peut être utilisé à tout endroit où un synonyme <synonym> est autorisé (voir 5.3.3.3). Les sortes prédéfinies sont: Boolean, (booléen), Character (caractère), Charstring (chaîne de caractères), Integer (entier), Natural (entier naturel), Real (réel), PId, Duration (durée) ou Time (temps).

Sémantique

Un synonyme <external synonym> est un synonyme <synonym> dont les valeurs ne sont pas spécifiées dans la définition du système. Cela est indiqué par le mot clé **external** qui est utilisé à la place d'une expression <simple expression>.

Une définition générique de système est une définition de système qui contient des synonymes <external synonym> ou du texte <informal text> dans une option de transition (voir 4.3.4). Une définition particulière de système est créée à partir d'une définition générique de système en donnant des valeurs aux synonymes <external synonym>, 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.

4.3.2 Expression simple

Grammaire textuelle concrète

<simple expression> ::=
<ground expression>

L'expression *Ground-expression* représentée par une expression <simple expression> ne doit contenir que des opérateurs et des littéraux définis dans le progiciel Predefined, comme défini dans l'Annexe D.

Sémantique

Une expression simple est une expression *Ground-expression*.

4.3.3 Définition optionnelle

Grammaire textuelle concrète

```
<select definition> ::=
                    select if (<Boolean simple expression> ) <end>
                    {       <system type definition>
                        |   <textual system type reference>
                        |   <block type definition>
                        |   <textual block type reference>
                        |   <block definition>
                        |   <textual block reference>
                        |   <textual typebased block definition>
                        |   <channel definition>
                        |   <signal definition>
                        |   <signal list definition>
                        |   <remote variable definition>
                        |   <remote procedure definition>
                        |   <data definition>
                        |   <process type definition>
                        |   <textual process type reference>
                        |   <process definition>
                        |   <textual process reference>
                        |   <textual typebased process definition>
                        |   <service type definition>
                        |   <textual service type reference>
                        |   <service definition>
                        |   <textual service reference>
                        |   <textual typebased service definition>
                        |   <timer definition>
                        |   <channel connection>
                        |   <channel endpoint connection>
                        |   <variable definition>
                        |   <view definition>
                        |   <imported variable specification>
                        |   <procedure definition>
                        |   <textual procedure reference>
                        |   <imported procedure specification>
                        |   <signal route definition>
                        |   <channel to route connection>
                        |   <signal route to route connection>
                        |   <select definition>
                        |   <macro 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 opérateurs des sortes définies à l'intérieur du progiciel Predefined, comme défini dans l'Annexe D.

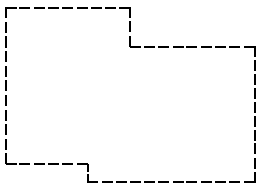
Une définition <select definition> ne peut contenir que les définitions syntaxiquement autorisées à cet endroit.

Grammaire graphique concrète

<option area> ::=

<option symbol> ***contains***
{ **select if** (<Boolean simple expression>)
{ <system type diagram>
| <system type reference>
| <block type diagram>
| <block type reference>
| <block area>
| <channel definition area>
| <system text area>
| <block text area>
| <process text area>
| <procedure text area>
| <block substructure text area>
| <channel substructure text area>
| <service text area>
| <macro diagram>
| <process type diagram>
| <process type reference>
| <process area>
| <service type diagram>
| <service type reference>
| <service area>
| <procedure area>
| <signal route definition area>
| <create line area>
| <option area> } + }

Le symbole <option symbol> est un polygone en pointillés ayant des angles droits par exemple:



T1007680-93/d21

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 <process graph area>, de zone <process type graph area>. Une zone <option area> ne peut contenir que les zones et les diagrammes qui sont syntaxiquement autorisés à cet endroit.

Sémantique

Si la valeur de l'expression <Boolean simple expression> est fausse (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

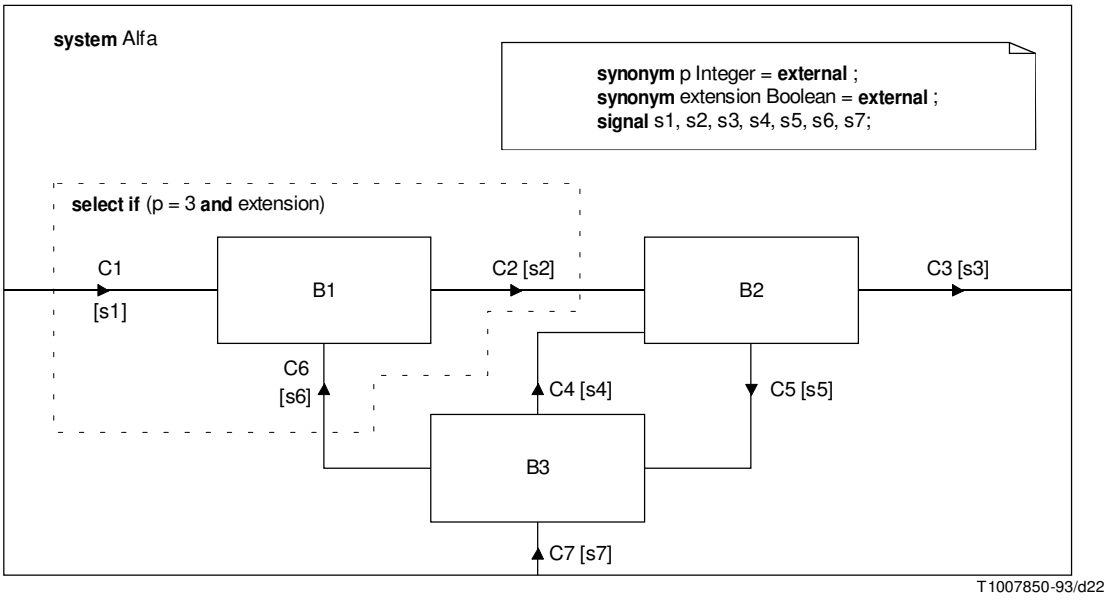
La définition <select definition> et la zone <option area> sont supprimées à la transformation et remplacées par les constructions sélectionnées qu'elles contiennent, s'il y en a. Tous les connecteurs connectés à une zone dans des zones <option area> non sélectionnées sont également supprimés.

Exemple

Dans le système Alfa il y a trois blocs: B1, B2 et B3. Le Bloc B1 et les canaux qui sont connectés sont optionnels, selon les valeurs des synonymes externes p et extension. Dans le SDL/PR, on représente cet exemple comme suit:

```
system Alfa;
     synonym p Integer = external;
     synonym extension Boolean = external;
     signal s1,s2,s3,s4,s5,s6,s7;
     select  if (p = 3 and extension);
        block B1 referenced;
        channel C1 from env to B1 with s1 ; endchannel C1;
        channel C2 from B1 to B2 with s2 ; endchannel C2;
        channel C6 from B3 to B1 with s6; endchannel C6;
     endselect;
     channel C3 from B2 to env with s3 ; endchannel C3;
     channel C4 from B3 to B2 with s4 ; endchannel C4;
     channel C5 from B2 to B3 with s5; endchannel C5;
     channel C7 from env to B3 with s7 ; endchannel C7;
     block B2 referenced;
     block B3 referenced;
endsystem Alfa;
```

Le même exemple en SDL/GR est représenté ci-dessous.



T1007850-93/d22

4.3.4 Chaîne de transition optionnelle

Grammaire textuelle concrète

<transition option> ::=
alternative <alternative question> <end>
{ <answer part> <else part>
| <answer part> { <answer part> }+ [<else part>] }
endalternative

<alternative question>::=
<simple expression>
| <informal text>

Chaque expression <ground expression> dans une réponse <answer> doit être une expression <simple expression>. Les réponses <answer> dans une option <transition option> doivent s'exclure mutuellement. Si la question <alternative question> est une <expression>, la condition *Range-condition* des réponses <answer> doit être de la même sorte que la question <alternative question>.

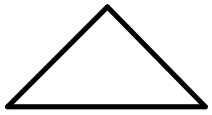
Aucune réponse <answer> dans les parties <answer part> d'une option <transition option> ne peut être omise.

Grammaire graphique concrète

<transition option area> ::=

<transition option symbol> ***contains*** {<alternative question>}
is followed by {<option outlet1>
{<option outlet1> | <option outlet2> }
{ <option outlet1> }* } ***set***

<transition option symbol> ::=



<option outlet1> ::=

<flow line symbol> ***is associated with*** <graphical answer>
is followed by <transition area>

<option outlet2> ::=

<flow line symbol> ***is associated with*** **else**
is followed by <transition area>

Le symbole <flow line symbol> dans <option outlet1> et <option outlet2> est relié 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. La réponse <graphical answer> et le terme **else** peuvent être placés le long du symbole <flow line symbol> associé, ou dans le symbole <flow line symbol> discontinu.

Les réponses <graphical answer> dans une zone <transition option area> doivent s'exclure mutuellement.

Sémantique

Les constructions dans <option outlet1> sont sélectionnées si la réponse <answer> contient la valeur de la question <alternative question>. Si aucune des réponses <answer> ne contient la valeur de la question <alternative question> ce sont les constructions dans <option outlet2> qui sont sélectionnées.

Si aucun <option outlet2> n'est fourni 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>; ou

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 2.7.5.

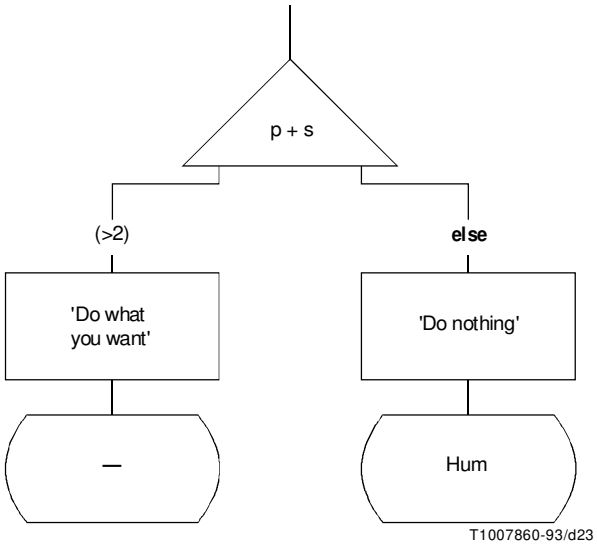
L'option <transition option> et la zone <transition option area> sont effacées à la transformation et remplacées par les constructions sélectionnées contenues.

Exemple

Une partie d'une définition <process definition> contenant une option <transition option> est montrée ci-après. *p* et *s* sont des synonymes.

```
.............
alternative p + s;
   (>2) : task 'Do what you want';
      nextstate -;
   else: task 'Do nothing';
      nextstate Hum;
endalternative;
.............
```

Le même exemple en syntaxe graphique concrète est montré ci-après:



4.4 Etat astérisque

Grammaire textuelle concrète

<asterisk state list> ::=

<asterisk> [(<state name> {, <state name>}*)]

<asterisk> ::=

*

Les noms <state name> dans une liste <asterisk state list> doivent être distincts et doivent être contenus dans d'autres listes <state list> dans le corps <body> englobant ou dans le corps <body> d'un supertype.

Modèle

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 <body> en question, sauf pour les noms <state name> qui sont contenus dans la liste <asterisk state list>.

4.5 Apparition multiple d'état

Grammaire textuelle concrète

Un nom <state name> peut apparaître dans plusieurs états <state> d'un corps <body>.

Modèle

Lorsque plusieurs états <state> contiennent le même nom <state name>, ces états <state> sont concaténés en un état <state> ayant ce nom <state name>.

4.6 Entrée astérisque

Grammaire textuelle concrète

<asterisk input list> ::=

<asterisk>

Un état <state> peut contenir au plus une liste <asterisk input list>. Un état <state> ne doit pas contenir à la fois une liste <asterisk input list> et une liste <asterisk save list>.

Modèle

Une liste <asterisk input list> est transformée en une liste de parties <input part>, une pour chaque membre de l'ensemble des signaux d'entrée valides de la définition <process definition> englobante ou de la définition <service definition> sauf pour les identificateurs <signal identifier> de signaux d'entrée implicites introduits par les concepts supplémentaires des paragraphes 4.10-4.14 et pour les identificateurs <signal identifier> contenus dans les autres listes <input list> et listes <save list> de l'état <state>.

4.7 Sauvegarde astérisque

Grammaire textuelle concrète

<asterisk save list> ::=

<asterisk>

Un état <state> peut contenir au plus une liste <asterisk save list>. Un état <state> ne peut pas contenir à la fois une liste <asterisk input list> et une liste <asterisk save list>.

Modèle

Une liste <asterisk input list> est transformée en une liste de <stimulus> contenant la totalité de l'ensemble des signaux d'entrée valides de la définition <process definition> englobante ou de la définition <service definition> sauf pour les identificateurs <signal identifier> de signaux d'entrée implicites introduits par les concepts supplémentaires des § 4.10-4.14 et pour les identificateurs <signal identifier> contenus dans les autres listes <input list> et listes <save list> de l'état <state>.

4.8 Transition implicite

Grammaire textuelle concrète

Un identificateur <signal identifier> contenu dans l'ensemble complet des signaux d'entrée valides d'une définition <process definition> ou d'une définition <service definition> peut être omis dans l'ensemble des identificateurs <signal identifier> contenus dans les listes <input list>, les listes <priority input list> et la liste <save list> d'un état <state>.

Modèle

Pour chaque état <state>, il y a une partie <input part> implicite contenant une <transition> qui ne contient qu'un état <nextstate> ramenant au même état <state>.

4.9 Etat suivant pointillé

Grammaire textuelle concrète

<dash nextstate> ::=
<hyphen>

<hyphen>::=
-

La <transition> contenue dans un départ <start> ne doit pas conduire, directement ou indirectement, dans un état <dash nextstate>.

Modèle

Dans chaque état <nextstate> d'un état <state>, l'état <dash nextstate> est remplacé par le nom <state name> de l'état <state>.

4.10 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> <transition>

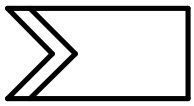
<priority input list> ::=
<stimulus> {, <stimulus>}*

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 followed by <transition area>

<priority input symbol> ::=



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 6.3.3.

L'entrée prioritaire est transformée de la manière suivante:

Un état dont le nom state_name contient des entrées <priority input> est divisé en deux états. La transformation nécessite deux variables implicites n et newn. La variable n a la valeur initiale 0. De plus, un signal implicite X_cont acheminant une valeur entière est requis. Un signal X_cont unique est requis pour chaque service si le processus contient des services. Les entrées prioritaires à l'état d'origine sont connectées au premier état, tandis que toutes les autres entrées sont connectées au second état (State2) et sauvegardées dans le premier. Les transitions spontanées, les entrées de procédure distante et les mises en réserve de procédure distante de l'état d'origine sont connectées aux deux états. La chaîne de transition conduisant à l'état d'origine conduit à présent au premier état (State1). Les chaînes d'action suivantes sont ajoutées à la chaîne de transition:

1) Tous les états <nextstate> qui mentionnent le nom state_name sont remplacés par **join 1**;

2) La transition suivante est insérée:

l: **task** n := n+1;
output X_cont(n) **to self**;
nextstate State1;

3) La partie suivante est ajoutée au premier état:

input X_cont(newn);
decision (newn = n);
(True): **nextstate** State2;
(False): **nextstate** - ;
enddecision;

4.11 Signal continu

Lorsqu'on décrit des systèmes, on peut se trouver dans un cas où un usager aimerait décrire une transition causée directement par la valeur vraie (True) d'une expression booléenne. Pour y parvenir, il faut calculer l'expression lorsqu'on est dans l'état et lancer la transition si l'évaluation de l'expression est vraie (True). Cette opération est désignée sous forme abrégée par signal continu, qui permet de lancer une transition directement lorsqu'une certaine condition est remplie.

Grammaire textuelle concrète

<continuous signal> ::=
provided [<virtuality>]
<Boolean expression> <end>
[**priority** <Integer literal name> <end>] <transition>

Grammaire graphique concrète

<continuous signal association area> ::=
<solid association symbol> ***is connected to*** <continuous signal area>

<continuous signal area> ::=
<enabling condition symbol>
contains{ [<virtuality>] <Boolean expression>
[[<end>]***priority***<Integer literal name>]}
is followed by <transition area>

Sémantique

L'expression <Boolean expression> dans le signal <continuous signal> est évaluée avant d'entrer dans l'état auquel elle est associée, et tant que l'on attend dans l'état, chaque fois que l'on trouve dans l'accès d'entrée aucun <stimulus> d'une liste <input list> adjointe. Si la valeur de l'expression <Boolean expression> est Vrai (True), la transition a lieu. Si la valeur de l'expression <Boolean expression> est Vrai (True) dans plus d'un signal <continuous signal>, la transition qui doit être engagée est déterminée par le signal <continuous signal> présentant la priorité la plus élevée, c'est-à-dire la valeur la plus faible pour le nom <Integer literal name>. Si plus de signaux <continuous signal> ont la même priorité, un choix non déterministe est effectué entre eux. Si la valeur est Faux (False) pour tous les signaux

<continuous signal> avec une priorité explicite, les expressions <Boolean expression> des signaux <continuous signal> sans priorité sont prises en compte dans un ordre arbitraire.

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 6.3.3.

L'état avec le nom state_name contenant des signaux <continuous signal> est transformé en ce qui suit. Cette transformation nécessite deux variables implicites n et newn. La variable n est initialisée à 0. De plus, un signal implicite sx.emptyQ acheminant une valeur entière est nécessaire; sx désigne ici un service (s'il y en a) contenant l'état.

1) Tous les états <nextstate> qui mentionnent le nom state_name sont remplacés par **join 1**;

2) La transition suivante est insérée:

```
1: task  n:= n+1;
   output  sx.emptyQ(n) to self;
   nextstate  state_name;
```

3) La partie <input part> suivante est ajoutée au nom state_name de l'état <state>:

```
input  sx.emptyQ (newn);
```

et une décision <decision> contenant la <question>

```
(newn=n)
```

4a) La partie <answer part> correspondant à faux contient

```
nextstate state_name;
```

4b) La partie <answer part> correspondant à vrai contient une séquence de décisions <decision> correspondant aux signaux <continuous signal> dans l'ordre de priorité. La priorité la plus élevée est indiquée par la valeur la plus faible du nom <Integer literal name>. Si plusieurs signaux <continuous signal> ont la même priorité, ils sont tous évalués dans un ordre arbitraire avant le niveau de priorité suivant. Lorsque tous les signaux <continuous signal> avec priorité explicite ont été évalués, les signaux <continuous signal> sans priorité sont évalués dans un ordre arbitraire.

La partie <answer part> correspondant à faux contient la décision <decision> suivante, sauf pour ce qui est de la dernière décision <decision> pour laquelle cette partie <answer part> contient: **join 1**.

Chaque partie <answer part> vrai de ces décisions <decision> conduit à la <transition> du signal <continuous signal> correspondant.

L'ordre arbitraire d'évaluation d'un nombre de signaux <continuous signal> peut être obtenu en utilisant l'abréviation de décision non déterministe, c'est-à-dire qu'un choix non déterministe entre les évaluations des signaux <continuous signal> possibles est effectué, et si l'évaluation d'un signal <continuous signal> donne faux, il est marqué pour ne pas être évalué en cas d'un nouveau choix non déterministe. A chaque tour, l'existence de signaux <continuous signal> qui restent encore à évaluer est enregistrée.

Exemple

Voir 4.12

4.12 Condition de validation

Dans le SDL, la réception d'un signal ou d'une transition spontanée dans un état, provoque immédiatement une transition. Le concept de condition de validation permet d'imposer une condition supplémentaire pour le déclenchement de la transition.

Grammaire textuelle concrète

<enabling condition> ::=

provided <Boolean expression> <end>

Grammaire graphique concrète

<enabling condition area> ::=

<enabling condition symbol> ***contains*** <Boolean expression>

<enabling condition symbol> ::=

Sémantique

L'expression <Boolean expression> dans la condition <enabling condition> est évaluée avant d'entrer dans l'état en question 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 évaluées séquentiellement dans un ordre arbitraire avant d'entrer dans l'état. Le modèle de transformation garantit la réévaluation répétée de l'expression en renvoyant d'autres <stimulus> par l'accès d'entrée. Un signal décrit dans la liste <input list> qui précède la condition <enabling condition> peut déclencher la transition seulement si la valeur de l'expression <Boolean expression> correspondante est Vrai. Si cette valeur est Faux, le signal est alors sauvegardé. Une transition <spontaneous transition> précédée par une condition <enabling condition> peut être activée seulement si la valeur de l'expression <Boolean expression> correspondante est Vrai.

Modèle

L'état nom state_name contenant les conditions <enabling condition> est transformé comme suit. Cette transformation nécessite la présence de deux variables implicites n et newn. La variable n est initialisée à 0. De plus, un signal implicite sx.emptyQ acheminant une valeur entière est requis; sx désigne ici un service (s'il y en a) contenant l'état.

1) Tous les état <nextstate> qui mentionnent state_name sont remplacés par **join 1**;

2) La transition suivante est insérée:

   1: **task** n:= n+1;

   output sx.emptyQ (n) **to self**;

   Un certain nombre de décisions, dont chacune contient une seule expression <Boolean expression> correspondant à une condition <enabling condition> associée à l'état, sont ajoutées hiérarchiquement dans un ordre arbitraire tel que toutes les combinaisons de valeurs Vrai puissent être évaluées pour toutes les conditions de validation associées à l'état. Chacune de ces combinaisons aboutit à un nouvel état distinct.

3) Chacun de ces nouveaux états a un ensemble de parties <input part> consistant en une copie des parties <input part> de l'état sans les conditions de validation plus les parties <input part> pour lesquelles l'évaluation des expressions <Boolean expression> des conditions <enabling condition> donne Vrai pour cet état et un ensemble des transitions <spontaneous transition> pour lesquelles l'évaluation des expressions <Boolean expression> des conditions <enabling condition> donne Vrai pour cet état.

   Les <stimulus> et les listes <remote procedure identifier list> pour les parties <input part> restantes constituent la liste <save list> et la sauvegarde <remote procedure save> pour une nouvelle partie <save part> adjointe à cet état. Les parties <save part> et transitions <spontaneous transition> de l'état d'origine sont également copiées dans ce nouvel état.

4) Ajouter à chacun des nouveaux états:

input sx.emptyQ (newn);

Une décision <decision> contenant la <question>

(newn=n);

5a) La partie <answer part> faux contient un état <nextstate> ramenant à ce même nouvel état.

5b) La partie <answer part> vrai contient un **join 1**;

6) Si les signaux <continuous signal> et les conditions <enabling condition> sont utilisés dans le même état <state>, les évaluations des expressions <Boolean expression> à partir des signaux <continuous signal> sont effectuées en remplaçant l'étape 5b du modèle pour la condition <enabling condition> par l'étape 4b du modèle pour le signal <continuous signal>.

Exemple

L'exemple illustrant la transformation d'un signal continu et d'une condition de validation apparaissant dans un état est donné par la Figure 4.12.1.

Il faut remarquer que dans cet exemple, le connecteur ec a été introduit pour des raisons de commodité. Il ne fait pas partie du modèle de transformation.

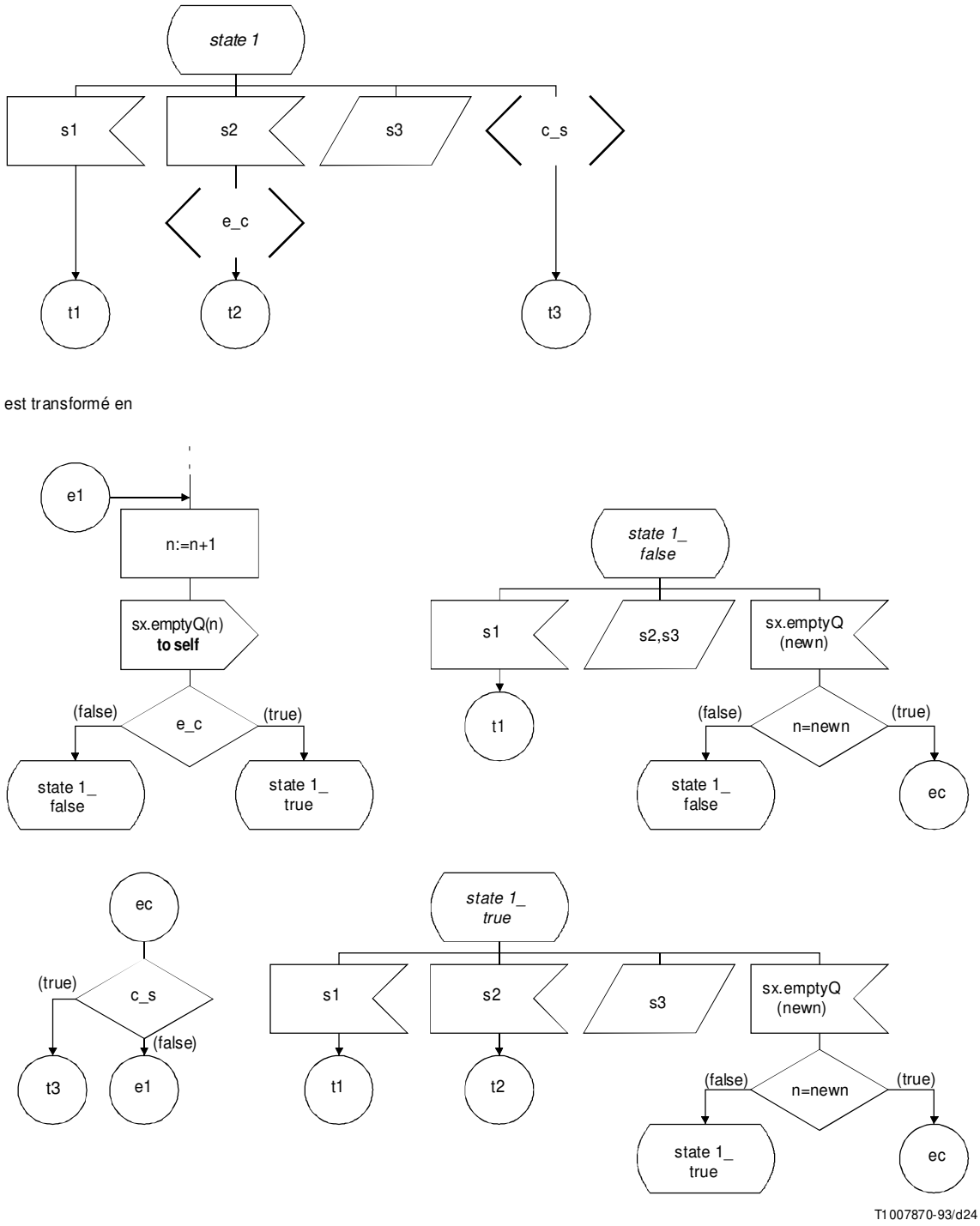


FIGURE 4.12.1/Z.100

Transformation d'un signal continu el d'une condition de validation dans le même état

4.13 Valeur importée et valeur exportée

Dans le SDL, une variable appartient toujours à une instance de processus dont elle est une variable locale. Normalement, elle n'est visible que de l'instance de processus à laquelle elle appartient; elle peut cependant être déclarée comme valeur révélée (voir 2.6.1.1) ce qui permet à d'autres instances de processus du même bloc d'avoir accès à la valeur de la variable. Si une instance de processus d'un autre bloc doit accéder à la valeur d'une variable, on a besoin pour cela d'un échange de signaux avec l'instance de processus à laquelle cette variable appartient.

Cette opération peut être réalisée en utilisant l'abréviation valeur importée et exportée. On peut également utiliser une abréviation pour exporter des valeurs en direction d'instances de processus dans le même bloc, auquel cas elle offre une alternative à l'utilisation de valeurs révélées.

Grammaire textuelle concrète

<remote variable definition>::=
remote
<remote variable name> {,<remote variable name>}* <sort> [**nodelay**]
{, <remote variable name> {,<remote variable name>}* <sort> [**nodelay**]}*
<end>

<imported variable specification> ::=
imported
<remote variable identifier> {, <remote variable identifier> }* <sort>
{, <remote variable identifier> {, <remote variable identifier> }* <sort>}* <end>

<import expression> ::=
import (<remote variable identifier> [, <destination>])

<export> ::=
export (<variable identifier> {, <variable identifier> }*)

nodelay désigne des canaux implicites sans retard pour l'échange de signaux résultant de la transformation décrite dans *Modèle*.

L'identificateur d'une spécification <imported variable specification> doit désigner une définition <remote variable definition> de la même sorte qui peut être implicite (voir ci-dessous).

L'identificateur <remote variable identifier> suivant **as** dans une définition de variable exportée doit désigner une définition <remote variable definition> de la même sorte que la définition de la variable exportée. Dans le cas où il n'y a pas de clause **as**, la définition de la variable distante dans l'unité de portée englobante la plus proche ayant le même nom et la même sorte que la définition de la variable exportée est désignée.

Pour chaque expression <import expression>, il doit exister une spécification <imported variable spécification> de l'identificateur de la variable distante dans un type de processus, une définition de processus, un type de service ou une définition de service englobant(e).

Pour chaque variable importée dans un processus, il doit exister au moins un processus d'exportation.

Les définitions de variables distantes peuvent être omises et, dans ce cas, il existe une définition de variable distante implicite pour chaque variable exportée. La définition de variable distante a le même nom et la même sorte que la variable exportée et elle est insérée dans l'unité de portée dans laquelle la sorte est définie. La définition de variable distante implicite n'a pas de propriété **nodelay**.

L'identificateur <variable identifier> dans <export> doit désigner une variable définie avec **exported**.

Grammaire graphique concrète

<export area>::=
<task symbol> ***contains*** <export>

Sémantique

Une définition <remote variable definition> définit le nom et la sorte des variables importées et exportées.

Une définition de variable exportée est une définition de variable avec le mot clé **exported**.

L'association entre une spécification <imported variable specification> et une définition <exported variable definition> est établie par la référence de cette spécification et de cette définition à la même définition <remote variable definition>.

L'instance de processus à laquelle appartient une variable dont les valeurs sont exportées vers d'autres instances de processus est appelée exportateur de la variable. Les autres instances de processus sont les importateurs de la variable. La variable est appelée variable exportée.

Une instance de processus peut être à la fois importatrice et exportatrice, mais elle ne peut importer à partir de l'environnement ou exporter vers l'environnement.

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'exécution d'un <export> par lequel un exportateur divulgue la valeur courante d'une variable exportée. Une opération d'export provoque le stockage de la valeur courante de la variable exportée dans sa copie implicite.

b) *Opération d'import*

Pour chaque spécification <imported variable specification> dans un importateur, il y a un ensemble de variables implicites ayant un nom unique et la sorte donnés dans la spécification <imported variable specification>. Ces variables implicites sont utilisées pour le stockage des valeurs importées.

Une opération d'import est l'exécution d'une expression <import expression> dans laquelle un importateur accède à la valeur d'une variable exportée. La valeur est stockée dans une variable implicite indiquée par l'identificateur <import 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 de processus 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 la spécification <imported variable specification>.

Modèle

Une opération d'import est modélisée par un échange de signaux. Ces signaux sont implicites et sont acheminés sur des canaux et des acheminements de signaux implicites. 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 la même valeur qui se trouve contenue dans la copie implicite de la variable exportée.

Si une initialisation par défaut est attachée à la variable d'export ou si la variable d'export est initialisée lorsqu'elle est définie, la copie implicite est aussi initialisée, et avec la même valeur 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> La copie implicite de la variable exportée est appelée *imcx*.

a) *Importateur*

L'expression <import expression> '**import** (x, destination)' est transformée en ce qui suit, où la clause **to** est omise, si la destination n'est pas donnée:

output xQUERY **to** destination;
Attendre dans l'état xWAIT, en sauvegardant tous les autres signaux;
input xREPLY (x);

Dans tous les autres états, xREPLY est sauvegardé.

Remplacer l'expression <import expression> par x, (le nom <name> de la variable implicite);

b) *Exportateur*

A tous les états <state> de l'exportateur, exception faite des états implicites, dérivés d'une importation, la partie <input part> suivante est ajoutée:

input xQUERY;
output xREPLY (imcx) **to sender**/* état suivant identique*/

L'<export> '**export** (x)' est transformé en ce qui suit:

task imcx := x,

NOTES

1 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 processus dont la spécification garantit l'existence au moment de la réception du signal xQUERY.

2 Le caractère facultatif des définitions de variables distantes est introduit pour assurer la compatibilité avec le SDL-88 et n'est pas recommandé pour les nouvelles descriptions SDL.

4.14 Procédures distantes

Un processus client peut appeler une procédure définie dans un autre processus, au moyen d'une demande au processus serveur à travers un appel de procédure distante d'une procédure dans le processus serveur.

Grammaire textuelle concrète

<remote procedure definition> ::=
remote procedure <remote procedure name> [**nodelay**] <end>
[<procedure signature> <end>]

<imported procedure specification> ::=
imported procedure <remote procedure identifier> <end>
[<procedure signature> <end>]

<remote procedure call> ::=
call <remote procedure call body>

<remote procedure call body> ::=
<remote procedure identifier> [<actual parameters>]
[**to** <destination>]

<remote procedure input transition>::=
input [<virtuality>] **procedure**
<remote procedure identifier list> <end>
[<enabling condition>]
<transition>

<remote procedure save> ::=
save [<virtuality>] **procedure**
<remote procedure identifier list> <end>

<remote procedure identifier list> ::=
<remote procedure identifier> {,<remote procedure identifier>}*

nodelay désigne des canaux implicites sans retard pour l'échange de signaux résultant de la transformation décrite dans *Modèle*.

La signature de spécification <imported procedure specification> doit être la même signature que celle de la définition <remote procedure definition> identifiée par l'identificateur <remote procedure identifier>.

L'identificateur <remote procedure identifier> suivant **as** dans une définition de procédure exportée doit désigner une définition <remote procedure definition> ayant la même signature que 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 environnante la plus proche ayant le même nom est implicite.

Pour chaque appel <remote procedure call>, il doit exister une spécification <imported procedure specification> de l'identificateur de la procédure distante dans un type de processus, une définition de processus, un type de service ou une définition service englobant(e) ayant le même nom <remote procedure name>.

Pour chaque spécification de procédure importée dans un processus, il doit exister au moins une procédure exportée dans un certain processus.

Grammaire graphique concrète

<remote procedure call area> ::=

<procedure call symbol> ***contains*** <remote procedure call body>

<remote procedure input area>::=

<input symbol> ***contains***

{ [<virtuality>] **procedure** <remote procedure identifier list> }

is followed by { [<enabling condition area>] <transition area> }

<remote procedure save area> ::=

<save symbol> ***contains***

{[<virtuality>] **procedure** <remote procedure identifier list> <end> }

Sémantique

Une définition <remote procedure definition> introduit le nom et la signature des procédures importées et exportées.

Une procédure exportée est une procédure contenant le mot clé **exported**.

L'association entre une spécification <imported procedure specification> 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 processus *demandeur* conduit ce dernier à attendre que le processus *serveur* exécute la procédure. Les signaux envoyés au processus demandeur pendant l'attente sont sauvegardés. Le processus serveur exécutera 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 sauvegarde <remote procedure save> ni aucune transition <remote procedure input transition> n'est spécifiée pour un état, une transition implicite qui consiste en un appel de procédure ramenant au même état est ajoutée. Si une transition <remote procedure input transition> est spécifiée pour un état, une transition implicite qui consiste en l'appel de procédure suivi par <transition> est ajoutée. Si une sauvegarde <remote procedure save> est spécifiée pour un état, une sauvegarde implicite du signal pour la procédure demandée est ajoutée.

Modèle

L'appel de procédure distante est modélisé par un échange de signaux. Ces signaux sont implicites et acheminés sur des canaux et des acheminements de signaux implicites. Le processus *demandeur* envoie un signal contenant les paramètres réels de l'appel de procédure au processus *serveur* et attend la réponse. En réponse à ce signal, le processus serveur exécute la procédure distante correspondante, envoie un signal en retour au processus demandeur avec la valeur de tous les paramètres **in/out** et exécute alors 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>.

a) Processus demandeur

L'appel <remote procedure call> '**call** Proc(apar) **to** destination', où apar est la liste des paramètres réels, est transformé de la manière suivante, où la clause **to** est omise, si la destination n'est pas donnée:

output pCALL(apar) **to** destination;

Wait in state pWAITe, saving all other signals;

input pREPLY (aINOUTpar);

où aINOUTpar est la liste modifiée des paramètres réels **in/out**.

Dans tous les autres états, pREPLY est sauvegardé.

b) Processus serveur

Il existe une déclaration de variable implicite, appelée ivar, de la sorte PId.

On ajoute la partie <input part> suivante à tous les états <state> comportant une transition d'entrée de procédure distante:

input pCALL(fpar);

ivar := **sender**

call Proc(fpar);

output pREPLY (fINOUTpar) **to** ivar;

<transition>

où fpar et fINOUT par sont identiques à apar et aINOUT ci-dessus.

La partie <save part> suivante est ajoutée à tous les états <state> ayant une sauvegarde de procédure distante:

save pCALL;

On ajoute la partie <input part> a tous les autres états <state> exception faite des états implicites dérivés d'une importation:

input pCALL(fpar);

ivar := **sender**

call Proc(fpar);

output pREPLY (fINOUTpar) **to** ivar;

/* next state the same */

Une transition <remote procedure input transition> ou une zone <remote procedure input area> qui contient <virtuality> est appelée transition virtuelle d'entrée de procédure distante. Une sauvegarde <remote procedure save> ou une zone <remote procedure save area> qui contient <virtuality> est appelée sauvegarde virtuelle de procédure distante. La transition virtuelle d'entrée et la sauvegarde virtuelle de procédure distante sont décrites en détail au § 6.3.3.

NOTE – 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 processus dont la spécification garantit l'existence au moment de la réception du signal pCALL.

5 Données dans le SDL

5.1 Introduction

L'introduction a pour objet de donner un aperçu du modèle formel utilisé pour définir les types de données et de renseigner sur la façon dont la suite du 5 est structurée.

Dans un langage de spécification, il faut pouvoir décrire formellement les types de données du point de vue de leur comportement, plutôt que de les composer à partir de primitives fournies, comme dans un langage de programmation. Cette dernière méthode implique invariablement une mise en oeuvre particulière du type de données et restreint donc la liberté du réalisateur au niveau du choix des représentations appropriées du type de données. La méthode de type de données abstraites permet toute mise en oeuvre à condition qu'elle soit possible et correcte du point de vue de la spécification.

5.1.1 Abstraction dans les types de données

Dans le SDL, tous les types de données sont des types abstraits qui sont définis essentiellement en termes de propriétés abstraites plutôt qu'en termes de mise en oeuvre concrète. On trouvera dans l'Annexe D des exemples de définitions de type de données abstraites qui définissent les facilités prédéfinies en matière de données du langage.

Bien que tous les types de données soient abstraits, et que les facilités prédéfinies en matière de données puissent être transgressées par l'utilisateur, le SDL essaie d'offrir un ensemble de facilités prédéfinies en matière de données qui soient familières à la fois dans leur comportement et dans leur syntaxe. Les noms des types de données prédéfinies sont les suivants:

a) Booléen (Boolean)
b) Caractère (Character)
c) Chaîne (String)
d) Chaîne de caractères (Charstring)
e) Entier (Integer)
f) Entier naturel (Natural)
g) Réel (Real)
h) Tableau (Array)
i) Mode ensembliste (Powerset)
j) Pid (PId)
k) Durée (Duration)
l) Temps (Time).

Des objets composites peuvent être formés en utilisant le concept de sorte structurée (**struct**).

5.1.2 Aperçu des formalismes utilisés pour modéliser les données

Les données sont modélisées par une algèbre initiale. L'algèbre a des sortes désignées et un ensemble d'opérateurs établissant des fonctions entre les sortes. Chaque sorte définit toutes les valeurs possibles qui peuvent être produites par l'ensemble associé d'opérateurs. Chaque valeur peut être décrite par au moins un terme dans le langage contenant seulement des littéraux et des opérateurs. Les littéraux constituent un cas particulier d'opérateurs sans argument.

Les sortes et les opérateurs, ainsi que le comportement du type de données, constituent les propriétés du type de données. Un type de données est introduit dans un certain nombre de définitions partielles de type, dont chacune définit une sorte, des opérateurs et les règles algébriques associées à cette sorte.

Le mot clé **newtype** introduit une définition partielle de type qui définit une nouvelle sorte distincte. Une sorte peut être créée avec des propriétés héritées d'une autre sorte, mais avec différents identificateurs pour la sorte et les opérateurs.

Un syntype introduit un sous-ensemble des valeurs d'une sorte.

Un générateur est une description **newtype** incomplète: avant de décrire l'état d'une sorte, elle doit être transformée en définition de type partielle en fournissant l'information manquante.

Certains des opérateurs introduits par la définition de type partielle ont comme domaine d'arrivée la sorte elle-même, et ainsi produisent les valeurs (éventuellement nouvelles) de la sorte. D'autres opérateurs donnent une signification à la sorte en ayant comme domaine d'arrivée des sortes définies. De nombreux opérateurs ont comme domaine d'arrivée la sorte booléenne à partir d'autres sortes, mais il est strictement interdit pour ces opérateurs d'étendre la sorte booléenne.

5.1.3 Terminologie

La terminologie utilisée en 5 ou le modèle de données est choisi(e) pour être en harmonie avec les travaux qui ont été publiés sur les algèbres initiales. En particulier le terme «type de données» est utilisé pour désigner un ensemble de sortes plus un ensemble d'opérateurs associés à ces sortes et la définition des propriétés de ces sortes et opérateurs par des équations algébriques. Une «sorte» est un ensemble de valeurs présentant des caractéristiques communes. Un «opérateur» est une relation entre sortes. Une «Equation» est la définition d'équivalence entre les termes d'une sorte. Une valeur est un ensemble de termes équivalents. Un «axiome» est une équation qui exprime qu'un «terme» booléen est équivalent à Vrai. Toutefois, le terme «axiomes» est utilisé pour désigner des «axiomes» ou des «équations».

5.1.4 Structure du texte sur les données

Le modèle d'algèbre initiale utilisé dans le cas des données pour le SDL, est décrit de manière à permettre la définition de la plupart des concepts relatifs aux données en termes de noyau de données du langage de données abstrait du SDL.

Le texte de l'article 5 est divisé en plusieurs parties: l'introduction (5.1), le langage de noyau de données (5.2), utilisation passive des données (5.3), utilisation active des données (5.4).

Le langage du noyau de données définit la partie des données dans le SDL qui correspond directement avec l'approche d'algèbre initiale sous-jacente. Le texte concernant l'algèbre initiale dans l'Annexe C est une introduction plus détaillée aux bases mathématiques de cette méthode. On trouvera dans l'Appendice I à l'Annexe C une formulation mathématique plus précise.

L'utilisation passive du SDL comprend les caractéristiques implicites et abrégées des données du SDL qui permettent son utilisation pour la définition de type abstrait de données. Elle comprend également l'interprétation des expressions qui ne font pas intervenir des valeurs assignées aux variables. Ces expressions «passives» correspondent à l'utilisation fonctionnelle du langage.

L'utilisation active des données étend le langage de manière à inclure l'affectation. Ceci comprend l'affectation, l'utilisation et l'initialisation des variables. Lorsque le SDL est utilisé pour effectuer des affectations aux variables ou pour accéder aux valeurs des variables, on dit qu'il est utilisé de façon active. La différence entre les expressions actives et passives est que la valeur d'une expression passive est indépendante de son instant d'interprétation, tandis qu'une expression active peut être interprétée de façon différente selon les valeurs actuelles associées avec les variables ou l'état actuel du système.

Les données prédéfinies du SDL sont définies dans l'Annexe D qui définit les sortes énumérées en 5.1.1.

5.2 Le langage de noyau de données

Le noyau de données peut être utilisé pour définir des types abstraits de données.

5.2.1 Définitions des types de données

A un point quelconque d'une spécification SDL, il y a une définition applicable de type de données. La définition de type de données définit la validité des expressions et les relations entre les expressions. La définition introduit des opérateurs et des ensembles de valeurs (sorte).

Il n'y a pas de correspondance simple entre la syntaxe concrète et la syntaxe abstraite pour les définitions des types de données car la syntaxe concrète introduit la définition du type de données de façon progressive en insistant sur les sortes (voir également l'Annexe C).

Les définitions dans la syntaxe concrète sont souvent interdépendantes et ne peuvent être séparées dans différentes unités de portée. Par exemple:

```
newtype even literals 0;
   operators
     plusee: even, even   -> even;
     plusoo: odd, odd  -> even;
   axioms
        plusee(a,0)  == a;
        plusee(a,b)  == plusee(b,a);
        plusoo(a,b)  == plusoo(b,a);
endnewtype even; /* even "numbers" with plus-depends on odd */

newtype odd literals 1;
   operators
     plusoe: odd, even -> odd;
     pluseo: even, odd -> odd;
   axioms
        plusoe(a,0)  == a;
        pluseo(a,b)  == plusoe(b,a);
endnewtype odd; /*odd "numbers" with plus - depends on even*/
```

Chaque définition de type de données est complète, il n'y a pas de référence aux sortes ou aux opérateurs qui ne sont pas inclus dans la définition du type de données qui s'applique à un point donné. Une définition de type de données doit également ne pas invalider la sémantique de la définition de type de données dans l'unité de portée immédiatement englobante. Un type de données dans une unité de portée englobée enrichit seulement les opérateurs des sortes définies dans l'unité de portée extérieure. Une valeur d'une sorte définie dans une unité de portée peut être librement utilisée et transférée entre ou à partir d'unités de portée hiérarchiquement inférieures. Puisque les données prédéfinies sont définies dans un progiciel prédéfini et implicitement utilisé, les sortes prédéfinies (par exemple booléennes ou entières) peuvent être librement utilisées dans le système.

Grammaire abstraite

Data-type-definition	::	*Sorts* *Signature*-**set** *Equations*
Sorts	=	*Sort-name*-**set**
Sort-name	=	*Name*
Equations	=	*Equation*-**set**

Une définition *data type definition* ne doit pas ajouter de nouvelles valeurs à une *sorte* du *type de données* du genre <scope unit kind> englobant.

Si un *terme* (voir 5.2.3) est non équivalent à un autre *terme* selon le type de données qui s'applique dans le genre <scope unit kind> environnant, ces *termes* ne doivent pas être définis comme étant équivalents par la définition *data type definition*.

Grammaire textuelle concrète

<partial type definition> ::=
newtype <sort name>
[<formal context parameters>]
[<extended properties>] <properties expression>
endnewtype [<sort name>]

<properties expression> ::=
<operators>
{ <internal properties>
| <external properties> }
[<default initialization>]

<internal properties> ::=
[<operator definitions>]
[**axioms** <axioms>] [<literal mapping>]

Un paramètre <formal context parameter> des paramètres <formal context parameters> doit être soit un paramètre <sort context parameter> ou un paramètre <synonym context parameter>.

Les paramètres <formal context parameters>, les propriétés <extended properties>, les définitions <operator definitions>, les propriétés <external properties>, la correspondance <literal mapping> et l'initialisation <default initialization> font partie du noyau des données.

La définition *data type definition* est représentée par l'ensemble de toutes les définitions <partial type definition> dans la catégorie <scope unit kind> courante associée avec la définition *data type definition* de la catégorie <scope unit kind> environnante.

Chacune des catégories <scope unit kind> suivantes représente un point dans la syntaxe abstraite qui contient une définition *data type definition*: définition <system definition>, définition <block definition>, définition <process definition>, définition <service definition>, définition <procedure definition>, définition <channel substructure definition> ou définition <block substructure definition> ou les diagrammes correspondants dans la syntaxe graphique. La définition <partial type definition> dans une définition <system type definition>, <block type definition>, <process type definition> ou <service type definition> est transformée en définition <partial type definition> dans l'instance résultante (ou l'ensemble d'instances résultant) lorsque le type est instancié.

Les *sortes* pour une catégorie <scope unit kind> sont représentées par l'ensemble des noms <sort name> introduits par l'ensemble des définitions <partial type definition> de la catégorie <scope unit kind>.

L'ensemble des signatures *signature* et les équations *equations* pour une catégorie <scope unit kind> sont représentées par les expressions <properties expression> des définitions <partial type definition> de la catégorie <scope unit kind>.

Les opérateurs <operator> d'une expression <properties expression> représentent une partie de l'ensemble *signature* dans la syntaxe abstraite. L'ensemble complet *signature* est l'union des ensembles *signature* définis par les définitions <partial type definition> dans la catégorie <scope unit kind>.

Les axiomes <axioms> d'une expression <properties expression> représentent la partie de l'ensemble *equation* dans la syntaxe abstraite. Les équations *equations* sont l'union des ensembles d'*équations* définis par la définition <partial type definition> dans la catégorie <scope unit kind>.

Les sortes de données prédéfinies ont leurs définitions <partial type definition> implicites dans le progiciel Predefined dans l'Annexe D.

Sémantique

La définition de type de données définit un type de données. Un type de données a un ensemble de propriétés de type, c'est-à-dire: un ensemble de sortes, un ensemble d'opérateurs et un ensemble d'équations.

Les propriétés des types de données sont définies dans la syntaxe concrète par des définitions partielles de type. Une définition partielle de type ne produit pas toutes les propriétés du type de données mais définit partiellement certaines des propriétés. On peut obtenir les propriétés complètes d'un type de données par combinaison de toutes les définitions partielles de type qui s'appliquent à l'intérieur de l'unité de portée contenant la définition du type de données.

Une sorte est un ensemble de valeurs de données. Deux sortes différentes n'ont aucune valeur en commun.

La définition de type de données est formée à partir de la définition de type de données de l'unité de portée définissant l'unité de portée courante prise avec les sortes, les opérateurs et les équations définis dans l'unité de portée courante.

Sauf à l'intérieur d'une définition <partial type definition> ou d'un affinage <signal refinement>, la définition de type de données qui s'applique en un point quelconque est le type de données défini pour l'unité de portée qui englobe immédiatement ce point. A l'intérieur d'une définition <partial type definition> ou d'un affinage <signal refinement>, la définition de type de données qui s'applique est la définition de type de données de l'unité de portée englobant respectivement la définition <partial type definition> ou <signal refinement>.

L'ensemble des sortes d'un type de données est l'union de l'ensemble des sortes introduites dans l'unité de portée courante et de l'ensemble des sortes de type de données qui s'appliquent dans le genre <scope unit kind> environnant. L'ensemble des opérateurs d'un type de données est l'union de l'ensemble des opérateurs introduits dans l'unité de portée courante et de l'ensemble des opérateurs du type de données qui s'appliquent dans le genre <scope unit kind> environnant. L'ensemble des équations d'un type de données est l'union de l'ensemble des équations introduites dans l'unité de portée courante et de l'ensemble des équations du type de données qui s'appliquent dans le genre <scope unit kind> environnant.

Chaque sorte introduite dans une définition de type de données a un identificateur qui est le nom introduit par une définition partielle de type dans l'unité de portée désignée par l'identificateur de l'unité de portée.

5.2.2 Littéraux et opérateurs paramétrisés

Grammaire abstraite

Signature	=	*Literal-signature* | *Operator-signature*
Literal-signature	::	*Literal-operator-name* *Result*
Operator-signature	::	*Operator-name* *Argument-list* *Result*
Argument-list	=	*Sort-reference-identifier* +
Result	=	*Sort-reference-identifier*
Sort-reference-identifier	=	*Sort-identifier* | *Syntype-identifier*
Literal-operator-name	=	*Name*
Operator-name	=	*Name*
Sort-identifier	=	*Identifier*

Les syntypes et les identificateurs *syntype identifiers* ne font pas partie du noyau (voir 5.3.1.9).

Grammaire textuelle concrète

<operators> ::=
[<literal list>] [<operator list>]

<literal list> ::=
literals <literal signature> {, <literal signature> }* [<end>]

<literal signature> ::=
<literal operator name>
| <extended literal name>

<operator list> ::=
operators
<operator signature> { <end> <operator signature> }* [<end>]

<operator signature> ::=
<operator name> : <argument list> -> <result>
| <ordering>
| <noequality>

<operator name> ::=
<operator name>
| <extended operator name>

<argument list> ::=
<argument sort> { , <argument sort> }*

<argument sort> ::=
<extended sort>

<result> ::=
<extended sort>

<extended sort> ::=
<sort>
| <generator sort>

<sort> ::=
<sort identifier>
| <syntype>

Les alternatives nom <extended operator name>, nom <extended literal name>, relation <ordering>, sorte <generator sort>, syntype <syntype> et <noequality> ne font pas partie du noyau des données.

Les littéraux sont introduits par les signatures <literal signature> figurant après le mot clé **literals**. Le résultat *result* d'une signature *literal signature* est la sorte introduite par la définition <partial type definition> définissant le littéral.

Chaque signature <operator signature> dans la liste des signatures <operator signature> après le mot clé **operators** représente une signature *operator signature* avec un nom *operator name*, une liste *argument list* et un résultat *result*.

Le nom <operator name> correspond à un nom *operator name* dans la syntaxe abstraite qui est unique à l'intérieur de la définition de l'unité de portée même si le nom peut ne pas être unique dans la syntaxe concrète.

L'unique nom *Operator-name* ou *Literal-operator-name*, dans la syntaxe abstraite est obtenu à partir:

a) du nom <operator name> (ou du nom <literal operator name>); plus

b) la liste des identificateurs d'argument de sorte; plus

c) l'identificateur de sorte résultant; plus

d) l'identificateur de sorte de la définition partielle de type dans laquelle le nom <operator name> (ou <literal operator name>) est défini.

Chaque fois qu'un identificateur <operator identifier>, se trouve spécifié, on obtient l'unique nom *Operator-name* dans l'identificateur *Operator-identifier* de la même façon avec la liste des sortes d'arguments et de la sorte résultante obtenue à partir du contexte. Deux opérateurs avec le même nom <name> qui diffèrent par une ou plusieurs sortes d'arguments ou de résultats ont des *noms* différents.

Chaque sorte <argument sort> dans une liste <argument list> représente un identificateur *Sort-reference-identifier* dans une liste *Argument-list*. Un résultat <result> représente l'identificateur *Sort-reference-identifier* d'un résultat *Result*.

Lorsqu'un qualificateur <qualifier> d'un identificateur <operator identifier> (ou d'un identificateur <literal operator identifier>) contient un élément <path item> avec le mot clé **type**, le nom <sort name> après ce mot clé ne fait pas partie du qualificateur *Qualifier* de l'identificateur *Operator-identifier* (ou *Literal-operator-identifier*) mais est utilisé pour obtenir le *Nom* unique de l'identificateur *Identifier*. Dans ce cas, le qualificateur *Qualifier* est formé à partir de la liste des éléments <path item> précédant le mot clé **type**.

Sémantique

Un opérateur est dit «total» lorsque l'application de cet opérateur à toute liste de valeurs des sortes arguments désigne une valeur de la sorte résultante.

Une signature d'opérateur définit la façon dont l'opérateur peut être utilisé dans les expressions. La signature d'opérateur est l'identité d'opérateur plus la liste des sortes arguments et la sorte du résultat. C'est la signature de l'opérateur qui détermine si l'expression est une expression correcte dans le langage conformément aux règles spécifiées pour assurer la correspondance des sortes des expressions argument.

Un opérateur sans argument est appelé littéral.

Un littéral représente une valeur fixe appartenant à la sorte résultante de l'opérateur.

Un opérateur a une sorte résultante qui est la sorte identifiée par le résultat.

NOTE – A titre de directive: une signature <operator signature> doit mentionner la sorte introduite par la définition <partial type definition> englobante comme étant soit un argument <argument sort>, soit un résultat <result>.

Exemple 1

```
literals   free, busy ;
```

Exemple 2

```
operators
     findstate:   Telephone -> Availability;
```

Exemple 3

```
literals
     empty_list
operators
     add_to_list: list_of_telephones, Telephone -> list_of_telephones;
     sub_list:    list_of_telephones, Telephone -> list_of_telephones
```

5.2.3 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 par là même 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.

Grammaire abstraite

Equation	=	*Unquantified-equation* |
		Quantified-equations |
		Conditional-equation |
		Informal-text
Unquantified-equation	::	*Term Term*
Quantified-equations	::	*Value-name*-**set**
		Sort-identifier
		Equations
Value-name	=	*Name*
Term	=	*Ground-term* |
		Composite-term |
		Error-term
Composite-term	::	*Value-identifier* |
		Operator-identifier Term^{+} |
		Conditional-composite-term
Value-identifier	=	*Identifier*
Operator-identifier	=	*Identifier*
Ground-term	::	*Literal-operator-identifier* |
		Operator-identifier Ground-term^{+} |
		Conditional-ground-term
Literal-operator-identifier	=	*Identifier*

Les possibilités *Conditional-composite-term* et *Conditional-ground-term* dans les règles *Composite-term* et *Ground-term* respectivement ne font pas partie du noyau de données, bien que les équations contenant ces termes puissent être

remplacées par des équations sémantiquement équivalentes écrites dans le langage du noyau (voir 5.3.1.6). La possibilité *Error-term* dans la règle *Term* ne fait pas partie du noyau de données.

Les définitions de texte *informal text* et *conditional equations* sont respectivement données en 2.2.3 et 5.2.4.

Chaque *term* (ou *ground term*) dans la liste des termes après un identificateur *operator identifier* doit avoir la même sorte que la sorte correspondante (position) dans la liste *argument list* de la signature *operator signature*.

Les deux termes *terms* dans une équation *unquantified equation* doivent être de la même sorte.

Grammaire textuelle concrète

```
<axioms> ::=
                        <equation> { <end> <equation>}* [ <end> ]

<equation> ::=
                        <unquantified equation>
                |       <quantified equations>
                |       <conditional equation>
                |       <informal text>

<quantified equations> ::=
                        <quantification> ( <axioms> )

<quantification> ::=
                        for all <value name> { , <value name> }*
                        in <extended sort>

<unquantified equation> ::=
                        <term> == <term>
                |       <Boolean axiom>

<term> ::=
                        <ground term>
                |       <composite term>
                |       <error term>
                |       <spelling term>

<composite term> ::=
                        <value identifier>
                |       <operator identifier> ( <composite term list> )
                |       ( <composite term> )
                |       <extended composite term>

<composite term list> ::=
                        <composite term> { , <term> }*
                |       <term> , <composite term list>

<ground term> ::=
                        <literal identifier>
                |       <operator identifier> ( <ground term> { , <ground term> }* )
                |       ( <ground term> )
                |       <extended ground term>

<literal identifier> ::=
                        <literal operator identifier>
                |       <extended literal identifier>
```

Les possibilités axiome <Boolean axiom> de la règle équation <unquantified equation>, terme <error term> et terme <spelling term> de la règle terme <term>, terme <extended composite term> de la règle terme <composite term>, terme <extended ground term> de la règle terme <ground term> et identificateur <extended literal identifier> de la règle identificateur <literal identifier> ne font pas partie du noyau de données.

La sorte <sort> dans une <quantification> représente l'identificateur *sort identifier* dans les équations *quantified equations*. Le nom <value name> dans une <quantification> représente l'ensemble des noms *value name* dans les équations *quantified equations*.

Une liste <composite term list> représente une liste de *terme*. Un identificateur *operator identifier* suivi par une liste de *terme* est un terme *composite term* seulement si la liste de *terme* contient au moins un identificateur *value identifier*.

Un identificateur <identifier> qui est un nom qualifié apparaissant dans un <term> représente:

a) un identificateur *operator identifier* s'il précède une parenthèse ouverte (ou si c'est un nom <operator name> qui est un nom <extended operator name>) (voir 5.3.1); sinon

b) un identificateur *value identifier* s'il y a une définition de ce nom dans une <quantification> d'équation <quantified equation> englobant le terme <term> d'une sorte adaptée à ce contexte; sinon

c) un identificateur *literal operator identifier* s'il y a un littéral visible portant ce nom d'une sorte adaptée au contexte; sinon

d) un identificateur *value identifier* qui a une équation *quantified equation* implicite dans la syntaxe abstraite pour l'équation <unquantified equation>.

Deux occurrences au moins du même identificateur non lié value identifier dans la même équation <unquantified equation>, ou si cette équation est contenue dans une équation <conditional equation>, alors dans cette équation <conditional equation>, impliquent une *quantification*.

Un identificateur *operator identifier* est obtenu à partir du contexte de manière à ce que si le nom <operator name> est surchargé (c'est-à-dire que le même nom <name> est utilisé pour plusieurs opérateurs), il sera alors le nom *operator name* qui identifie un opérateur visible avec le même nom et les sortes d'argument et sorte résultante cohérente avec l'application de l'opérateur. Si le nom <operator name> est surchargé, il peut être nécessaire d'obtenir les sortes d'argument à partir des arguments et la sorte résultante à partir du contexte afin de déterminer le nom *operator name*.

Dans une équation <unquantified equation>, ou si cette équation est contenue dans une équation <conditional equation>, alors dans 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.

Il doit être possible de lier chaque identificateur <operator identifier> ou identificateur <literal operator identifier> non qualifié à exactement un identificateur *operator identifier* ou *literal operator identifier* défini qui satisfait la condition dans la construction dans laquelle l'identificateur <identifier> est utilisé. C'est-à-dire que la liaison doit être unique.

NOTE – A titre de directive: un axiome doit appartenir à la sorte de la définition partielle de type englobante en mentionnant un opérateur ou un littéral avec un résultat de cette sorte ou un opérateur qui a un argument de cette sorte; un axiome ne doit être défini qu'une seule fois.

Sémantique

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.

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.

Si un axiome quelconque contient un texte informel, l'interprétation des expressions n'est pas forcément définie par le LDS, 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 dans la syntaxe abstraite, et la valeur correspondante a 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,z in X (for all z in X _ )
```

introduit un seul identificateur de valeur appelé z de la sorte X.

Dans la syntaxe concrète, 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 qui est la sorte identifiée dans les équations quantifiées par l'identificateur *sort reference identifier*. 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.

Exemple 1

for all b **in** logical (eq(b,b)==T)

Exemple 2

neq(T, F)== T; neq(T, T) == F;
neq(F, T)== T; neq(F, F) == F;

Exemple 3

eq(b, b) == T;
eq(F, eq(T, F)) == T;
eq(eq(b, a),eq(a, b)) == T;

5.2.4 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 abstraite

Conditional-equation	::	*Restriction*-**set** *Restricted-equation*
Restriction	=	*Unquantified-equation*
Restricted-equation	=	*Unquantified-equation*

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 la même valeur seulement quand tout identificateur de valeur dans les équations restreintes prend une valeur dont on peut prouver à partir des autres équations qu'elle 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.

Exemple

```
z /= 0 == True ==> (x/z)*z==x
```

5.3 Utilisation passive des données du LDS

Le paragraphe 5.3.1 traite des extensions aux constructions de définitions de données du 5.2. Dans le 5.3.3 on étudie la manière d'interpréter l'utilisation des types abstraits de données dans les expressions si l'expression est passive (c'est-à-dire qu'elle ne dépend pas de variables ou de l'état du système). La façon d'interpréter les expressions, qui ne sont pas des passives (c'est-à-dire qui sont des expressions «actives») est définie en 5.4.

5.3.1 Constructions de définitions de données étendues

Les constructions définies en 5.2 constituent la base des formes plus concises expliquées ci-après.

Grammaire abstraite

Il n'y a pas de syntaxe abstraite supplémentaire pour la plupart de ces constructions. En 5.3.1 et en tous ses paragraphes, la syntaxe abstraite applicable se trouve pratiquement en 5.2.

Grammaire textuelle concrète

<extended properties> ::=
<inheritance rule>
| <generator transformations>
| <structure definition>

<extended composite term> ::=
<extended operator identifier> (<composite term list>)
| <composite term> <infix operator> <term>
| <term> <infix operator> <composite term>
| <monadic operator> <composite term>
| <conditional composite term>

<extended ground term> ::=
<extended operator identifier>
(<ground term> {, <ground term> }*)
| <ground term> <infix operator> <ground term>
| <monadic operator> <ground term>
| <conditional ground term>

<extended operator identifier> ::=
<operator identifier> <exclamation>
| <generator formal name>
| [<qualifier>] <quoted operator>

<extended operator name> ::=
<operator name> <exclamation>
| <generator formal name>
| <quoted operator>

<exclamation> ::=
!

<extended literal name> ::=
<character string literal>
| <generator formal name>
| <name class literal>

<extended literal identifier> ::=
<character string literal identifier>
| <generator formal name>

Le choix du nom <generator formal name> est uniquement valable dans une expression <properties expression> dans un texte <generator text> (voir 5.3.1.12) qui porte le même nom défini comme étant un paramètre formel.

Les possibilités avec <exclamation> ne sont valables que dans les définitions <operator definitions>.

Le choix de terme <extended composite term> et de terme <extended ground term> avec un nom <generator formal name> précédent un «(» ne sont valables que si le nom <generator formal name> est défini comme appartenant à la catégorie **operator** (voir 5.3.1.12).

Le choix du nom <extended literal name> avec un nom <generator formal name> n'est valable que si le nom <generator formal name> est défini comme appartenant à la catégorie **literal** (voir 5.3.1.12).

Le choix d'identificateur <extended literal identifier> avec un nom <generator formal name> n'est valable que si le nom <generator formal name> est défini comme appartenant à la catégorie **literal** ou à la catégorie **constant** (voir 5.3.1.12).

Lorsqu'un nom d'opérateur est défini avec une <exclamation>, cette <exclamation> fait sémantiquement partie du nom *name*.

Les formes nom <operator name> <exclamation> ou identificateur <operator identifier> <exclamation> représentent respectivement le nom *operator name* (5.2.2) et l'identificateur *operator identifier* (5.2.3).

Sémantique

Un nom d'opérateur défini avec une <exclamation> a la sémantique normale d'un opérateur, mais le nom d'opérateur n'est visible que dans les axiomes et dans les listes <inheritance list>.

5.3.1.1 Opérateurs spéciaux

Ce sont des noms d'opérateur qui ont une forme syntaxique particulière. La syntaxe particulière est telle que les opérateurs arithmétiques et les opérateurs booléens peuvent avoir leur forme syntaxique habituelle. C'est-à-dire que l'utilisateur peut écrire «(1 + 1) = 2» au lieu d'être obligé d'employer pour l'exemple equal(add(1,1),2). Les sortes qui sont valables pour chaque opérateur dépendront de la définition du type de données.

Grammaire textuelle concrète

<quoted operator> ::=
<quote> <infix operator> <quote>
| <quote> <monadic operator> <quote>

<quote> ::=
"

<infix operator> ::=
=> | **or** | **xor** | **and** | **in** | /= | = | > | < | <=
| >= | + | / | * | // | **mod** | **rem** | -

<monadic operator> ::=
- | **not**

Sémantique

Un opérateur infixe dans un terme a la sémantique normale d'un opérateur mais avec la syntaxe infixe ou préfixe entre quotes ci-dessus.

Un opérateur monadique est un terme qui a la sémantique normale d'un opérateur mais avec la syntaxe préfixe ou préfixe entre quotes ci-dessus.

Les formes entre quotes des opérateurs infixes ou unaires sont des noms valables pour les opérateurs.

Les opérateurs infixes auront un ordre de priorité qui détermine les liens entre les opérateurs. Le lien est le même que celui dans <expression> tel qu'il se trouve spécifié en 5.3.3.1.

Lorsque le lien est ambigü comme par exemple dans

a **or** b **xor** c ;

le lien va de gauche à droite si bien que le terme ci-dessus est équivalent à:

(a **or** b) **xor** c ;

Modèle

Un terme de la forme terme <term1> opérateur <infix operator> terme <term2> est une syntaxe dérivée pour opérateur «<infix operator>» (terme <terme1>, terme <term2>) avec opérateur «<infix operator>» comme nom légal. Opérateur «<infix operator>» représente un nom *operator name*.

De manière analogue, opérateur <monadic operator> terme <term> est une syntaxe dérivée pour opérateur «<monadic operator>» (terme <term>) avec opérateur «<monadic operator>» comme nom légal et représentant un nom *operator name*.

NOTE – L'opérateur d'égalité (=) ne doit pas être confondu avec le symbole d'équivalence de terme (==).

5.3.1.2 Littéral de chaîne de caractères

Grammaire textuelle concrète

<character string literal identifier> ::=
[<qualifier>] <character string literal>

<character string literal> ::=
<character string>

Une chaîne <character string> est une unité lexicale.

Un identificateur <character string literal identifier> représente un identificateur *Literal-operator-identifier* dans la syntaxe abstraite.

Un littéral <character string literal> représente un unique nom *Literal-operator-name* (5.2.2) dans la syntaxe abstraite obtenue à partir de la chaîne <character string>.

Sémantique

Les identificateurs de littéraux de chaîne de caractères sont des identificateurs formés à partir de littéraux de chaîne de caractères dans les termes et les expressions.

Les littéraux de chaîne de caractères sont utilisés pour les sortes de données prédéfinies Charstring et Character (voir l'Annexe D). Ils ont également un lien particulier avec les littéraux de classe de nom (voir 5.3.1.14) et les mises en correspondance de littéraux (voir 5.3.1.15). Ces littéraux peuvent être également définis pour d'autres utilisations.

Un littéral <character string literal> a une longueur qui est la longueur des alphanumériques <alphanumeric> plus les caractères <other character> plus les spéciaux <special> plus point <full stop> plus souligné <underline> plus espace <space> plus les paires <apostrophe> <apostrophe> dans la chaîne <character string> (voir 2.2.1).

Si littéral <character string literal>

a) a une longueur supérieure à un; et

b) a une sous-chaîne formée en supprimant le dernier caractère (alphanumérique <alphanumeric> ou caractère <other character> ou spécial <special> ou point <full stop> ou souligné <underline> ou espace <space> ou paire <apostrophe> <apostrophe>) de la chaîne <character string>; et

c) a une sous-chaîne définie comme un littéral et telle que

substring // deleted_character_in_apostrophes

est un terme valide avec la même sorte que le littéral <character string literal>, l'opérateur de concatenation et ses arguments étant qualifiés avec la sorte englobante

alors, il y a une équation implicite donnée par la syntaxe concrète telle que le littéral <character string literal> est équivalent à la sous-chaîne suivie par l'opérateur infixe «//» suivi par le caractère supprimé avec les apostrophes pour former une chaîne <character string>.

Par exemple, les littéraux 'ABC', 'AB'' et 'AB' dans

```
newtype s
literals 'ABC', 'AB''', 'AB', 'A', 'B', 'C', '''';
operators "//": s, s -> s;
```

ont les équations implicites suivantes

'ABC' == 'AB' // 'C' ;
'AB'' == 'AB' // '' ;
'AB' == 'A' // 'B';

5.3.1.3 Données prédéfinies

Les données prédéfinies, y compris la sorte Boolean qui définissent les propriétés des deux littéraux True and False, sont définies à l'Annexe D. La sémantique pour l'égalité (5.3.1.4), les axiomes booléens (5.3.1.5), les termes conditionnels (5.3.1.6), la relation ordre (5.3.1.8), et les syntypes (5.3.1.9) reposent sur la définition de la sorte booléenne (D.1). La sémantique de littéraux de classe de nom (si des intervalles <regular interval> sont utilisés – 5.3.1.14) et la mise en correspondance des littéraux (5.3.1.15) reposent également respectivement sur la définition de character (D.2) et charstring (D.4).

De plus, les deux termes *terms* Booléen True (Vrai) et False (Faux) ne doivent pas être définis (directement ou indirectement) comme équivalents. Chaque expression close booléenne 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.

Il n'est pas permis de réduire le nombre de valeurs de la sorte prédéfinie PId.

Les données prédéfinies sont définies dans un progiciel Predefined utilisé implicitement (voir 2.4.1.2). Ce progiciel est défini dans l'Annexe D.

5.3.1.4 Egalité et inégalité

Grammaire textuelle concrète

<noequality> ::=

noequality

Sémantique

Les symboles = et /= dans la syntaxe concrète représentent les noms des opérateurs qui sont appelés les opérateurs égal ou non égal.

Modèle

Toute définition <partial type definition> introduisant une certaine sorte appelée S sans règle <inheritance rule> a une paire de signature *operator signature* impliquée équivalente à

"=" : S, S -> << **package** Predefined >> Boolean;

"/=": S, S -> << **package** Predefined >> Boolean;

où Boolean est la sorte booléenne prédéfinie.

Toute définition <partial type definition> introduisant une sorte appelée S, à moins

a) qu'elle ne possède le mot clé **noequality** dans sa liste <operator list>; ou

b) qu'elle ne soit fondée sur l'héritage sans le mot clé **noequality** dans sa liste <inheritance list>,

a un ensemble d'équation impliqué

```
for all a,b,c in S (
   a = a == True;
   a = b == b = a;
   ((a = b) and (b = c)) ==> a = c == True;
   a /= b == not(a = b);
   a = b  == True ==> a == b;)
```

et une équation <literal equation> impliquée

```
for all L1,L2 in S literals (
   Spelling(L1) /= Spelling(L2) ==> L1 = L2 == False;)
```

5.3.1.5 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, et qui peut par conséquent être utilisé pour spécifier le comportement du type de données.

Modèle

Un axiome de la forme

terme <Boolean term>;

est une syntaxe dérivée pour l'équation en syntaxe concrète

<Boolean term> == << **package** Predefined/**type** Boolean >> True;

qui a la relation normale d'une équation avec la syntaxe abstraite.

5.3.1.6 Termes conditionnels

Dans ce qui suit, l'équation contenant le terme conditionnel est appelée équation de terme conditionnel.

Grammaire abstraite

Conditional-composite-term	=	*Conditional-term*
Conditional-ground-term	=	*Conditional-term*
Conditional-term	::	*Condition* *Consequence* *Alternative*
Condition	=	*Term*
Consequence	=	*Term*
Alternative	=	*Term*

La sorte de la *Condition* doit être la sorte booléenne prédéfinie et la *Condition* ne doit pas être le terme *Error-term*. La conséquence *Consequence* et l'alternative *Alternative* doivent avoir la même sorte.

Un terme *Conditional term* est un terme *Conditional composite term* si et seulement si un ou plusieurs des termes *terms* dans la *condition*, la conséquence *Consequence* ou l'alternative *Alternative* est un terme *Composite term*.

Un terme *Conditional term* est un terme conditional ground term si et seulement si tous les *terms* dans la *condition*, la conséquence *consequence* ou l'alternative *alternative* sont des termes *ground terms*.

Grammaire textuelle concrète

<conditional composite term> ::=
<conditional term>

<conditional ground term> ::=
<conditional term>

<conditional term> ::=
if <condition> **then** <consequence> **else** <alternative> **fi**

<condition> ::=
<Boolean term>

<consequence> ::=
<term>

<alternative> ::=
<term>

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 la *condition* par chaque terme *ground term* de la sorte appropriée. Dans cet ensemble d'équations, la *condition* aura toujours été remplacée par un terme *ground term* 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'*equations* qui contient:

a) pour chaque *equation* dans l'ensemble clos développé pour laquelle la *condition* est équivalente à True, cette *equation* obtenue à partir de l'ensemble clos développé avec le terme *conditional term* remplacée par la *consequence* (close), et

b) pour chaque *equation* dans l'ensemble clos développé pour laquelle la *condition* est équivalente à False, cette *equation* obtenue à partir de l'ensemble clos développé avec le terme *conditional term* remplacée par l'*alternative* (close).

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.

Exemple

```
if i = j * j then posroot(i) else abs(j) fi ==
if positive(j) then j else –j fi;
```

NOTE – Il y a de meilleures façons de spécifier ces propriétés, il ne s'agit que d'un exemple.

5.3.1.7 Erreurs

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 abstraite

Error-term :: ( )

Un terme *error term* ne doit pas être utilisé comme terme argument pour un identificateur *operator identifier* dans un terme *composite term*.

Un terme *error term* ne doit pas être utilisé dans une *restriction*.

Il ne doit pas être possible de déduire à partir d'équations *Equations* qu'un identificateur *literal operator identifier* est égal à un terme *error term*.

Grammaire textuelle concrète

<error term> ::=

error <exclamation>

Sémantique

Un terme peut être une erreur de telle sorte qu'il est possible de spécifier les circonstances dans lesquelles un opérateur produit une erreur. Si ces circonstances se produisent durant l'interprétation, le comportement ultérieur du système est indéfini.

5.3.1.8 Relation d'ordre

Grammaire textuelle concrète

<ordering> ::=
ordering

Sémantique

Le mot clé **ordering** est une abréviation pour spécifier explicitement les opérateurs de relation d'ordre et un ensemble d'équations d'ordre pour une définition partielle de type.

Modèle

Une définition <partial type definition> introduisant une sorte nommée S avec le mot clé **ordering** implique un ensemble de signature *operator signature* équivalant aux définitions explicites:

```
"<" : S,S -> package Predefined Boolean;
">" : S,S -> package Predefined Boolean;
"<=": S,S -> package Predefined Boolean;
">=": S,S -> package Predefined Boolean;
```

où Boolean est la sorte prédéfinie booléenne, et implique les axiomes *axioms* booléens:

```
for all a, b, c, d in S
(  a = b        ==> a < b  == False;
a < b              == b > a;
a <= b             == a < b or a = b;
a >= b             == a > b or a = b;
a < b == True   ==> b < a  == False;
a < b and b = c and c < d == True ==> a < d == True; );
```

Lorsqu'une définition <partial type definition> inclut à la fois une liste <literal list> et le mot clé **ordering**, les signatures <literal signature> sont nommées selon l'ordre ascendant, c'est-à-dire

```
literals A,B,C;
operators ordering;
```

implique A<B, B<C.

5.3.1.9 Syntypes

Un syntype spécifie un ensemble de valeurs 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 ces valeurs appartiennent à l'ensemble des valeurs de la sorte.

Grammaire abstraite

Syntype-identifier	=	*Identifier*
Syntype-definition	::	*Syntype-name* *Parent-sort-identifier* *Range-condition*
Syntype-name	=	*Name*
Parent-sort-identifier	=	*Sort-identifier*

Grammaire textuelle concrète

<syntype> ::=
<syntype identifier>

<syntype definition> ::=

syntype

<syntype name> = <parent sort identifier>

[<default initialization>] [**constants** <range condition>]

endsyntype [<syntype name>]

| **newtype** <syntype name> [<extended properties>]

<properties expression> **constants** <range condition>

endnewtype [<syntype name>]

<parent sort identifier> ::=

<sort>

Un syntype <syntype> est une alternative pour une sorte <sort>.

Une définition <syntype definition> avec le mot clé **syntype** et le terme «=identificateur <syntype identifier>» 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 identifier> est utilisé comme sorte d'argument <argument sort> dans une liste <argument list> définissant l'opérateur, la sorte pour l'argument dans une liste *argument list* est l'identificateur *parent sort identifier* du syntype.

Lorsqu'un identificateur <syntype identifier> est utilisé comme résultat pour un opérateur, la sorte du résultat *result* est l'identificateur *parent sort identifier* du syntype.

Lorsqu'un identificateur <syntype identifier> est utilisé comme qualificateur pour un nom, le qualificateur *qualifier* est l'identificateur *parent sort identifier* du syntype.

Si le mot **syntype** est utilisé et que la condition <range condition> est omise, toutes les valeurs de la sorte sont situées dans la condition d'intervalle.

Sémantique

Une définition de syntype définit un syntype qui se réfère à un identificateur de sorte et à une condition d'intervalle. Spécifier un identificateur de syntype est identique 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 5.4.3);

b) sortie d'un signal si une des sortes spécifiées pour le signal est un syntype (voir 2.5.4 et 2.7.4);

c) appel d'une procédure où une des sortes spécifiées pour les variables du paramètre **in** de la procédure est un syntype (voir 2.7.3 et 2.4.6);

d) création d'un processus lorsque l'une des sortes spécifiées pour les paramètres de processus est un syntype (voir 2.7.2 et 2.4.4);

e) entrée d'un signal et une des variables qui est associée avec l'entrée, a une sorte qui est un syntype (voir 2.6.4);

f) utilisation dans une expression d'un opérateur qui a un syntype défini comme étant soit une sorte d'argument, soit une sorte de résultat (voir 5.3.3.2 et 5.4.2.4);

g) instruction d'initialisation ou de réinitialisation ou une expression active sur un temporisateur, une des sortes dans la définition de temporisateur étant un syntype (voir 2.8);

h) définition de variable distante ou de procédure distante si l'une des sortes pour la dérivation de signaux implicites est un syntype (voir 4.13 et 4.14);

i) paramètre de contexte formel de procédure avec un paramètre **in/out** dans la signature <procedure signature> correspondant à un paramètre de contexte réel où le paramètre formel correspondant ou le paramètre **in/out** dans la signature <procedure signature> est un syntype;

j) toute expression de valeur pour laquelle la valeur retournée sera dans l'intervalle (voir 5.4.4.6).

Par exemple, une définition <syntype definition> avec le mot clé **syntype** et «=identificateur <syntype identifier>» est équivalente à la substitution de l'identificateur <parent sort identifier> par l'identificateur <parent sort identifier> de la définition <syntype definition> de l'identificateur <syntype identifier>. C'est-à-dire

```
syntype s2 = n1 constants a1:a3 endsyntype s2;
syntype s3 = s2 constants a1:a2 endsyntype s3;
```

est équivalent à

```
syntype s2 = n1 constants a1:a3 endsyntype s2;
syntype s3 = n1 constants a1:a2 endsyntype s3;
```

Lorsqu'un syntype est spécifié en termes d'identificateur <syntype identifier>, 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 un domaine qui est l'ensemble des valeurs spécifiées par les constantes de la définition de syntype.

Modèle

Une définition <syntype definition> avec le mot clé **newtype** peut être distinguée d'une définition <partial type definition> par l'inclusion de conditions **constants** <range condition>. Une telle définition <syntype definition> est une abréviation d'introduction de définition <partial type definition> avec un nom anonyme, suivie par une définition <syntype definition> avec le mot clé **syntype**, fondée sur ce nom anonyme. C'est-à-dire

```
newtype X /* details */
 constants /* constant list */
endnewtype X;
```

est équivalent à

```
newtype anon /* details */
endnewtype anon;
```

suivi par

```
syntype X = anon
 constants /* constant list */
endsyntype X;
```

5.3.1.9.1 Condition d'intervalle

Grammaire abstraite

Range-condition	::	*Or-operator-identifier* *Condition-item*-**set**
Condition-item	=	*Open-range* | *Closed-range*
Open-range	::	*Operator-identifier* *Ground-expression*
Closed-range	::	*And-operator-identifier* *Open-range* *Open-range*
Or-operator-identifier	=	*Identifier*
And-operator-identifier	=	*Identifier*

Grammaire textuelle concrète

<range condition> ::=
{ <closed range> | <open range> }
{ , { <closed range> | <open range> } }*

<closed range> ::=
<constant> : <constant>

<open range> ::=
<constant>
| { = | /= | < | > | <= | >= } <constant>

<constant> ::=
<ground expression>

Le symbole «<» («<=», «>», «>=» respectivement) ne doit être utilisé que dans la syntaxe concrète de la condition <range condition> si les symboles ont été définis avec une signature <operator signature>.

```
P, P -> package Predefined Boolean;
```

où P est la sorte du syntype. Ces symboles représentent un identificateur *operator identifier.*

Un intervalle <closed range> doit seulement être utilisé si le symbole «<=» est défini avec une signature <operator signature>.

```
P, P -> package Predefined Boolean;
```

où P est la sorte du syntype.

Une constante <constant> dans une condition <range condition> doit avoir la même sorte que la sorte du syntype.

Sémantique

Une condition d'intervalle définit une vérification d'intervalle. Une vérification d'intervalle est utilisée quand un syntype a une sémantique additionnelle à la sorte du syntype (voir 2.6.1.1, 5.3.1.9 et les cas où les syntypes ont des sémantiques différentes – voir les sections auxquelles on fait référence aux points a) à j) du 5.3.1.9, *Sémantique*). Une vérification d'intervalle est également utilisée pour déterminer l'interprétation d'une décision (voir 2.7.5).

La vérification d'intervalle est l'application de l'opérateur formé à partir de la condition d'intervalle. Pour les vérifications d'intervalle de syntype, l'application de cet opérateur doit être équivalente à Vrai, dans le cas contraire, le comportement ultérieur du système est indéfini. La vérification d'intervalle est obtenue comme suit:

a) Chaque élément (intervalle <open range> ou intervalle <closed range>) dans la condition <range condition> a un intervalle *open range* ou un intervalle *closed range* 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 un terme donné, A,

   i) un intervalle <open range> de la forme = constante <constant>, /= constante <constant>, < constante <constant>, <= constante <constant>, > constante <constant>, et >= constante <constant>, a des sous-termes dans la vérification d'intervalle de la forme A = constante <constant>, A/= constante <constant>, A < constante <constant>, A <= constante <constant>, A > constante <constant>, et A >= constante <constant> respectivement;

   ii) l'intervalle <closed range> de la forme constante <first constant> : constante <second constant> a un sous-terme dans la vérification d'intervalle de la forme constante <first constant> <= A **and** A <= constante <second constant> où **and** correspond à l'opérateur **and** booléen prédéfini et correspond à l'identificateur *And-operator-identifier* dans la syntaxe abstraite.

d) Il y a un identificateur *Or-operator-identifier* pour l'opérateur distribué sur tous les éléments dans l'ensemble *condition-item*-**set** qui est une union booléenne **or** de tous les éléments. La vérification d'intervalle est le terme formé à partir de l'union booléenne **or** prédéfinie de tous les sous-termes obtenus à partir de la condition <range condition>.

Si un syntype est spécifié sans une condition <range condition>, la vérification d'intervalle est vraie.

5.3.1.10 Sortes de structure

Grammaire textuelle concrète

<structure definition> ::=
struct <field list> [<end>] [**adding**]

<field list> ::=
<fields> { <end> <fields> }*

<fields> ::=
<field name> { , <field name> }* <field sort>

<field sort> ::=
<sort>

Chaque nom <field name> d'une sorte structure doit être différent de chaque autre nom <field name> de la même définition <structure definition>.

Sémantique

Une définition de structure définit une sorte structure dont les valeurs sont composées à partir d'une liste de valeurs de champ des sortes.

La longueur de la liste des valeurs est déterminée par la définition de structure et la sorte d'une valeur est déterminée par sa position dans la liste des valeurs.

Modèle

Une définition de structure est une syntaxe dérivée pour la définition de:

a) un opérateur, Make!, pour créer des valeurs de structure; et

b) des opérateurs à la fois pour modifier les valeurs de structure et pour extraire les valeurs de champ à partir des valeurs de structure.

Le nom de l'opérateur implicite pour modifier un champ est le nom de champ concaténé avec «Modify!».

Le nom de l'opérateur implicite permettant d'extraire un champ est le nom de champ concaténé avec «Extract!».

La liste <argument list> pour l'opérateur Make! est la liste des sortes <field sort> apparaissant dans la liste des champs dans l'ordre avec lesquelles elles apparaissent.

Le résultat <result> pour l'opérateur Make! est l'identificateur de sorte de la structure.

La liste <argument list> pour l'opérateur de modification de champ est l'identificateur de sorte de la structure suivie par la sorte <field sort> de ce champ. Le résultat <result> pour un opérateur de modification de champ est l'identificateur de sorte de la structure.

La liste <argument list> pour un opérateur d'extraction de champ est l'identificateur de sorte de la structure. Le résultat <result> pour un opérateur d'extraction de champ est la sorte <field sort> de ce champ.

Il y a une équation implicite pour chaque champ qui montre que l'affectation d'un champ d'une structure à une valeur donne le même résultat que la construction d'une valeur de structure avec cette valeur pour le champ.

Il y a une équation implicite pour chaque champ qui montre que l'extraction d'un champ d'une valeur de structure retournera la valeur associée avec ce champ quand la valeur de la structure a été construite.

A moins que **noequality** ne soit spécifié dans la liste <operator list>, il existe une équation implicite qui définit l'égalité des valeurs de structure par l'égalité portant sur les éléments.

Par exemple,

```
newtype s struct
   b  Boolean;
   i  Integer;
   c  Character;
endnewtype   s;
```

implique

```
newtype s
 operators
  Make! : Boolean, Integer, Character  -> s;
  bModify!   : s, Boolean              -> s;
  iModify!   : s, Integer              -> s;
  cModify!   : s, Character            -> s;
  bExtract!  : s                       -> Boolean;
  iExtract!  : s                       -> Integer;
  cExtract!  : s                       -> Character;
 axioms
  bModify!   (Make!(x,y,z),b)     == Make!(b,y,z);
  iModify!   (Make!(x,y,z),i)     == Make!(x,i,z);
  cModify!   (Make!(x,y,z),c)     == Make!(x,y,c);
  bExtract!  (Make!(x,y,z))       == x;
  iExtract!  (Make!(x,y,z))       == y;
  cExtract!  (Make!(x,y,z))       == z;
  Make!(x1,y1,z1) = Make!(x2,y2,z2) == (x1=x2) and (y1=y2) and (z1=z2);
endnewtype s;
```

5.3.1.11 Héritage

Grammaire textuelle concrète

<inheritance rule> ::=
inherits <sort type expression>
[<literal renaming>]
[[**operators**] { **all** | (<inheritance list>) }
[<end>]] [**adding**]

<inheritance list> ::=
<inherited operator> { , <inherited operator> }*
[, **noequality**]

<inherited operator> ::=
[<operator name> =] <inherited operator name>

<inherited operator name> ::=
<base type operator name>

<literal renaming> ::=
literals <literal rename list> <end>

<literal rename list> ::=
<literal rename pair> { , <literal rename pair> }*

<literal rename pair> ::=
<literal rename signature> = <base type literal rename signature>

<literal rename signature> ::=
<literal operator name>
| <character string literal>

Si l'expression <sort type expression> contient des paramètres <actual context parameters>, la règle <inheritance rule> ne peut pas contenir <literal renaming> ou <inheritance list>, et le mot clé **all** est implicite s'il est omis. Dans ce cas, tous

les opérateurs littéraux et les axiomes du type <base type> sont hérités, y compris les opérateurs égaux et non égaux et leurs axiomes associés (s'il y en a).

Toutes les signatures <literal rename signature> dans une liste <literal rename list> doivent être distinctes. Toutes les signatures <base type literal rename signature> dans une liste <literal rename list> doivent être différentes.

Tous les noms <inherited operator name> dans une liste <inheritance list> doivent être distincts. Tous les noms <operator name> dans une liste <inheritance list> doivent être distincts.

Un nom <inherited operator name> spécifié dans une liste <inheritance list> doit être un opérateur du type <base type> défini dans la définition <partial type definition> définissant le type <base type>.

Les noms d'opérateurs «=» et «/=» sont implicitement contenus dans la liste <inheritance list> sans aucun renommage.

Lorsque plusieurs opérateurs du type <base type> ont le même nom, comme le nom <inherited operator name>, alors tous ces opérateurs sont hérités.

Modèle

Si l'expression <sort type expression> contient des paramètres <actual context parameters>, on utilise le modèle de spécialisation du 6.3, sinon, le modèle suivant est utilisé. Lorsque le modèle ci-dessous est appliqué, les définitions <operator definition>, les diagrammes <operator diagram> et l'initialisation <default intialization> ne sont pas hérités.

Une sorte (S) peut être fondée sur une autre sorte (BS) (de base) en utilisant **newtype** en association avec une règle d'héritage. La sorte définie au moyen d'une règle d'héritage est disjointe du type de base.

Si le type de base a des littéraux définis, les noms de littéraux sont hérités comme des noms pour des littéraux de la sorte qui est définie à moins qu'un renommage de littéraux n'ait eu lieu pour ces littéraux. Le renommage de littéral a eu lieu pour un littéral si le nom de littéral du type de base apparaît comme deuxième nom dans une paire de renommage de littéral auquel cas, le littéral est renommé comme premier nom dans cette paire.

Il existe un opérateur de conversion de type implicite qui prend un argument de la sorte du type de base et a un résultat de la nouvelle sorte. Le nom de cet opérateur est le nom de la sorte qui est définie, concaténée avec «!».

Il existe des opérateurs hérités comme cela est spécifié par **all** ou par la liste d'héritage. Le nom d'un opérateur hérité est défini comme suit:

a) le même que le nom d'opérateur du type de base si **all** est spécifié et si le nom est explicitement ou implicitement défini comme étant un nom d'opérateur dans la définition partielle de type ou dans la définition de syntype définissant le type de base; sinon

b) si le nom d'opérateur du type de base est donné dans la liste d'héritage et un nom d'opérateur suivi par «=» est donné pour l'opérateur hérité, alors il est renommé avec ce nom; sinon

c) si le nom d'opérateur du type de base est donné dans la liste d'héritage et un nom d'opérateur suivi par «=» n'est pas donné pour l'opérateur hérité, alors il a le même nom que le nom d'opérateur du type de base.

Si ni **all** ni une liste d'héritage n'est spécifié(e), les opérateurs hérités sont les opérateurs égaux et non égaux uniquement (voir 5.3.1.4).

Les sortes d'argument et le résultat d'un opérateur d'héritage sont les mêmes que ceux de l'opérateur correspondant du type de base, à l'exception du fait que chaque occurrence du type de base dans les opérateurs hérités est modifiée en la nouvelle sorte.

Pour chaque opérateur hérité, (à l'exception des opérateurs «=» et «/=» si **noequality** est spécifiée dans la liste <inheritance list>), il y a une équation implicite

$IO(t_1,...,t_n) == S!(BTO(v_1,..., v_n));$ si le résultat est de la nouvelle sorte

$IO(t_1,...,t_n) == BTO(v_1,...,v_n);$ autrement

où IO est l'opérateur hérité, BTO est l'opérateur correspondant du type de base, S! est l'opérateur de conversion de type implicite. Les v_i sont des valeurs d'identificateurs disjointes. Si la sorte de v_i est le type de base alors $t_i = S!(v_i)$, sinon, $t_i = v_i$.

Pour chaque paire de renommage de littéral $il_i = bl_i$, il existe une équation implicite

il_i == S!(bl_i);

Il existe une équation littérale implicite

```
for all ilv in S literals(
for all blv in BS literals(
Spelling(ilv) /= Spelling(il1),.., Spelling(ilv) /= Spelling(iln),
Spelling(ilv) == Spelling(blv) ==>      ilv == S!(blv);))
```

où il_i sont les littéraux mentionnés dans les paires de renommage de littéraux $il_i = bl_i$.

NOTE – Du point de vue de l'utilisateur, il n'y a pas de différence sémantique dans les modèles, car toutes les propriétés des sortes avec paramètres de contexte sont décrites localement. Toutefois, les différences suivantes existent pour les sortes de base avec et sans paramètres de contexte:

	avec paramètre de contexte	sans paramètre de contexte
Renommage d'opérateur/de littéral autorisé	non	oui
Définition d'opérateur héritée	oui	non
Correspondance dans **atleast** (voir 6.2.9)	non	oui, mais uniquement sans renommage d'opérateur/de littéral

Exemple

```
newtype bit
   inherits  Boolean
      literals 1 = True, 0 = False;
      operators  ("not", "and", "or")
   adding
   operators
      Exor: bit,bit -> bit;
   axioms
      Exor(a,b) == (a and (not b)) or ((not a) and b));
endnewtype bit;
```

5.3.1.12 Générateurs

Un générateur permet de définir un squelette de texte paramétré qui est développé par transformation avant que la sémantique des types de données ne soit considérée.

5.3.1.12.1 Définition de générateur

Grammaire textuelle concrète

<generator definition> ::=
generator <generator name> (<generator parameter list>)
<generator text>
endgenerator [<generator name>]

<generator text> ::=
[<generator transformations>] <properties expression>

<generator parameter list> ::=
<generator parameter> { , <generator parameter> }*

<generator parameter> ::=
{ **type** | **literal** | **operator** | **constant** }
<generator formal name> {, <generator formal name> }*

<generator formal name> ::=
<generator formal name>

<generator sort> ::=

<generator formal name>

| <generator name>

Un nom <generator name> ou nom <generator formal name> doit uniquement être employé dans une expression <properties expression> si l'expression <properties expression> figure dans un texte <generator text>.

Dans une définition <generator definition>, tous les noms <generator formal name> de la même catégorie (**type**, **literal**, **operator** ou **constant**) doivent être distincts. Un nom de la catégorie **literal** doit être distinct de chaque nom de la catégorie **constant** appartenant à la même définition <generator definition>.

Le nom <generator name> placé après le mot clé **generator** doit être distinct de tous les noms de sorte appartenant à la définition <generator definition> mais aussi de tous les paramètres <generator parameter> **type** de cette définition <generator definition>.

Une sorte <generator sort> n'est valable que si elle apparaît sous la forme d'une sorte <extended sort> dans un texte <generator text> et si le nom est soit le nom <generator name> de cette définition <generator definition>, soit un nom <generator formal name> décrit dans cette définition.

Si une sorte <generator sort> est un nom <generator formal name>, il faut qu'il s'agisse d'un nom défini comme appartenant à la catégorie **type**.

Le nom <generator name> facultatif figurant après **endgenerator** doit être le même que le nom <generator name> indiqué après **generator**.

Un nom <generator formal name> ne doit pas être employé dans un qualificateur <qualifier>. Un nom <generator name> ou un nom <generator formal name> ne doit pas:

a) être qualifié; ou

b) être suivi d'une <exclamation>; ou

c) être employé dans une initialisation <default initialization>.

Sémantique

Un générateur nomme une partie de texte qui peut être employée dans les transformations de générateur.

On considère que les textes de transformations de générateur à l'intérieur d'un texte de générateur sont développés au point où le texte du générateur est défini.

Chaque paramètre de générateur comprend une catégorie (**type**, **literal**, **operator**, ou **constant**) spécifiée par le mot clé **type**, **literal**, **operator**, ou **constant**, respectivement.

Modèle

Le texte correspondant à une définition du générateur ne se rapporte à la syntaxe abstraite que si le générateur est transformé. Il n'existe pas de syntaxe abstraite correspondante pour la définition du générateur là où la définition est donnée.

Exemple

```
generator bag(type item)
literals empty;
operators
   put   :  item, bag -> bag;
   count :  item, bag -> Integer;
   take  :  item, bag -> bag;
axioms
   take  (i,put(i,b)) == b;
   take  (i,empty)    == error!;
   count(i,empty)     == 0;
   count(i,put(j,b))  == count(i,b) + if i=j then 1 else 0 fi;
   put(i,put(j,b))    == put(j,put(i,b));
endgenerator bag;
```

5.3.1.12.2 Transformation de générateur

Grammaire textuelle concrète

<generator transformations> ::=
{ <generator transformation> [<end>] [**adding**] }+

<generator transformation> ::=
<generator identifier> (<generator actual list>)

<generator actual list> ::=
<generator actual> { , <generator actual> }*

<generator actual> ::=
<extended sort>
| <literal signature>
| <operator name>
| <ground term>

Si la classe d'un paramètre <generator parameter> est **type**, le générateur <generator actual> correspondant doit alors être une sorte <extended sort>.

Si la classe d'un paramètre <generator parameter> est **literal**, le générateur <generator actual> correspondant doit alors être une signature <literal signature>.

Une signature <literal signature> qui est un littéral <name class literal> peut être utilisée comme un générateur <generator actual> à la seule condition que le nom <generator formal name> correspondant n'apparaisse pas dans les axiomes <axioms>, ou dans la correspondance <literal mapping> de l'expression <properties expression> dans le texte <generator text>.

Si la classe d'un paramètre <generator parameter> est **operator**, le générateur <generator actual> correspondant doit alors être un nom <operator name>.

Si la classe d'un paramètre <generator parameter> est **constant**, le générateur <generator actual> correspondant doit alors être un terme <ground term>.

Si le générateur <generator actual> est un nom <generator formal name>, la classe du nom <generator formal name> doit alors être la même que la classe correspondant au générateur <generator actual>.

Sémantique

L'utilisation d'une transformation de générateur dans des propriétés étendues ou dans un texte de générateur signifie la transformation du texte identifié par l'identificateur de générateur. Un texte transformé applicable aux littéraux, aux opérateurs et aux axiomes est formé à partir du texte du générateur de la forme suivante:

a) les paramètres réels du générateur sont substitués aux paramètres du générateur; et

b) le nom du générateur est remplacé

   i) si la transformation de générateur se trouve dans une définition partielle de type ou une définition de syntype, par l'identité de la sorte définie par la définition partielle de type ou par la définition du syntype; autrement

   ii) dans le cas de transformation de générateur à l'intérieur d'un générateur, par le nom de ce générateur.

Le texte transformé applicable aux littéraux est le texte transformé à partir des littéraux figurant dans l'expression de propriétés du texte du générateur, le mot clé **literals** étant omis.

Le texte transformé applicable aux opérateurs est le texte transformé à partir de la liste d'opérateurs figurant dans l'expression de propriétés du texte du générateur, le mot clé **operators** étant omis.

Le texte transformé applicable aux axiomes est le texte transformé instancié à partir des axiomes figurant dans l'expression de propriétés du texte du générateur, le mot clé **axioms** étant omis.

Lorsqu'il existe plus d'une transformation de générateur dans la liste des transformations de générateur, les textes transformés applicables aux littéraux (opérateurs et axiomes) sont formés par concaténation du texte instancié applicables aux littéraux (opérateurs, axiomes, respectivement) de tous les générateurs dans l'ordre de leur apparition dans la liste.

Le texte transformé applicable aux littéraux est une liste de littéraux destinée à l'expression de propriétés de la définition partielle de type englobante, de la définition de syntype ou de la définition de générateur qui apparaît avant une liste quelconque de littéraux explicitement mentionnée dans l'expression de propriétés. Autrement dit, si un classement a été spécifié, les littéraux définis par les transformations de générateur seront placés dans l'ordre dans lequel ils ont été transformés et avant tout autre littéral.

Le texte transformé applicable aux opérateurs et aux axiomes est ajouté à la liste d'opérateurs et aux axiomes, respectivement, de la définition partielle de type englobante, de la définition de syntype ou de la définition de générateur.

Lorsqu'un texte transformé est ajouté à une expression de propriétés, il y a lieu d'ajouter, au besoin, les mots clés **literals**, **operators** et **axioms** pour créer la syntaxe concrète correcte.

Modèle

La syntaxe abstraite correspondant à une transformation de générateur est déterminée après transformation. La relation est déterminée à partir du texte transformé, dans le contexte où la transformation <generator transformation> apparaît.

Exemple

```
newtype boolbag bag(Boolean)
   adding
   operators
      yesvote : boolbag -> Boolean;
   axioms
      yesvote(b) == count(True,b) > count(False,b);
endnewtype boolbag;
```

5.3.1.13 Synonymes

Un synonyme donne un nom à une expression close qui représente une des valeurs d'une sorte.

Grammaire textuelle concrète

<synonym definition> ::=
<internal synonym definition>
| <external synonym definition>

<internal synonym definition> ::=
synonym <synonym definition item>
{, <synonym definition item> }*

<synonym definition item> ::=
<synonym name> [<sort>] = <ground expression>

Dans la syntaxe concrète, l'expression <ground expression> désigne un terme *ground term* de la syntaxe abstraite, conformément à la définition donnée en 5.3.3.2.

Si une sorte <sort> est spécifiée, le résultat de l'expression <ground expression> est lié à cette sorte *sort*. L'expression <ground expression> représente le terme *ground term* correspondant dans la syntaxe abstraite.

Si la sorte de l'expression <ground expression> ne peut être déterminée de manière unique, il faut alors spécifier une sorte dans la définition <synonym definition>.

La sorte identifiée par la sorte <sort> doit être l'une des sortes auxquelles l'expression <ground expression> peut être liée.

L'expression <ground expression> ne doit pas se référer directement ou indirectement (par l'intermédiaire d'un autre synonyme) au synonyme décrit dans la définition <synonym definition>.

Sémantique

La valeur que représente le synonyme est déterminée par le contexte dans lequel la définition de synonyme apparaît.

Si la sorte de l'expression close ne peut être déterminée uniquement dans le contexte du synonyme, la sorte est alors indiquée par la sorte <sort>.

Un synonyme a une valeur qui est la valeur du terme clos dans la définition de synonyme.

Un synonyme a une sorte qui est la sorte du terme clos dans la définition de synonyme.

5.3.1.14 Littéraux de classe de nom

Un littéral de classe de nom est une notation abrégée permettant d'écrire un ensemble (éventuellement infini) de noms de littéral défini par une expression régulière.

Grammaire textuelle concrète

<name class literal> ::=
nameclass <regular expression>

<regular expression> ::=
<partial regular expression>
{ [**or**] <partial regular expression> }*

<partial regular expression>::=
<regular element> [<Natural literal name> | + | *]

<regular element> ::=
(<regular expression>)
| <character string literal>
| <regular interval>

<regular interval> ::=
<character string literal> : <character string literal>

Les noms formés par le littéral <name class literal> doivent satisfaire aux conditions statiques normales applicables aux littéraux (voir 5.2.2) et, soit aux règles lexicales relatives aux noms (voir 2.2.1), soit à la syntaxe concrète relative au littéral <character string literal>.

Dans un intervalle <regular interval>, il faut que les littéraux <character string literal> soient l'un et l'autre de longueur un et soient des littéraux définis par la sorte caractère (voir D.2).

Sémantique

Un littéral de classe de nom est une autre façon de spécifier les signatures de littéraux.

Modèle

L'ensemble de noms d'un littéral de classe de nom se définit par l'ensemble de noms conforme à la syntaxe spécifiée par l'expression <regular expression>. Le littéral de classe de nom est l'équivalent de cet ensemble de noms dans la syntaxe abstraite.

Une expression <regular expression> qui est une liste d'expression <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> suivie 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 seconde de ces expressions <partial regular expression>. A noter que **or** constitue un lien plus étroit qu'une simple mise en séquence, de sorte que

```
nameclass 'A' '0' or '1' '2';
```

équivaut à

```
nameclass 'A' ('0' or '1') '2';
```

et définit les littéraux A02, A12.

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>.

Par exemple,

```
nameclass 'A' ('A' or 'B') 2
```

définit les noms AAA, AAB, ABA et ABB.

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.

Par exemple,

```
nameclass 'A' ('A' or 'B')*
```

définit les noms A, AA, AB, AAA, AAB, ABA, ABB, AAAA, ... etc.

Si un élément <regular element> est suivi d'un '+' l'expression <partial regular expression> équivaut à l'élément <regular element> répété une ou plusieurs fois.

Par exemple,

```
nameclass 'A' ('A' or 'B') +
```

définit les noms AA, AB, AAA, AAB, ABA, ABB, AAAA, ..., etc.

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 un littéral <character string literal> définit la séquence de caractères indiquée dans le littéral 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 caractère (voir D.2). Par exemple,

'a':'f'

définit les alternatives 'a' ou 'b' ou 'c' ou 'd' ou 'e' ou 'f'.

Si l'ordre de définition des noms est important (par exemple, si **ordering** est spécifié), on considère alors que les noms sont définis dans l'ordre alphabétique d'après la relation d'ordre de la sorte character. Les caractères sont considérés comme des majuscules, et un préfixe réel d'un mot est considéré inférieur au mot complet.

5.3.1.15 Mise en correspondance de littéral

Les mises en correspondance de littéral sont des notations abrégées utilisées pour définir la correspondance entre des littéraux et des valeurs.

Grammaire textuelle concrète

<literal mapping> ::=
map <literal equation> { <end> <literal equation> }* [<end>]

<literal equation> ::=
<literal quantification>
(<literal axioms> { <end> <literal axioms> }* [<end>])

<literal axioms> ::=

<equation>
| <literal equation>

<literal quantification> ::=

for all <value name> { , <value name> }* **in** <extended sort> **literals**

<spelling term> ::=

spelling (<value identifier>)

Les règles de correspondance <literal mapping> et terme <spelling term> ne font pas partie du noyau de données mais apparaissent dans les règles expression <properties expression> et terme <ground term> respectivement.

Sémantique

La mise en correspondance 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. Grâce à cette mise en correspondance, les littéraux d'une sorte peuvent être mis en correspondance avec les valeurs de la sorte.

Le mécanisme terme orthographique est employé dans les mises en correspondance 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 mises en correspondance de littéral.

Modèle

Une correspondance <literal mapping> est une notation abrégée d'un ensemble d'axiomes <axioms>. Cet ensemble d'axiomes <axioms> est obtenu à partir des équations <literal equation> dans la correspondance <literal mapping>. Les équations <equation> qui sont utilisées pour les obtenir sont toutes les équations <equation> contenues dans les axiomes <axioms> des règles axiomes <literal axioms>. Dans chacune de ces équations <equation>, on remplace les identificateurs <value identifier> définis par le nom <value name> dans la quantification <literal quantification>. Dans chaque équation <equation> obtenue, le même identificateur <value identifier> est remplacé, chaque fois qu'il apparaît, par le même identificateur <literal operator identifier> de la sorte <sort> de la quantification <literal quantification>. 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** et avant le mot clé **map** dans la même définition <partial type definition>.

Par exemple,

```
newtype abc literals 'A',b,'c';
 operators
   "<" : abc,abc -> Boolean;
   "+" : abc,abc -> Boolean;
 map for all x,y in abc literals
   (x < y => y + x);
endnewtype abc;
```

est de la syntaxe concrète dérivée de

```
newtype abc literals 'A',b,'c';
 operators
   "<" : abc,abc -> Boolean;
   "+" : abc,abc -> Boolean;
 axioms
   'A'  < 'A' => 'A' + 'A';
   'A'  <  b  =>  b  + 'A';
   'A'  < 'c' => 'c' + 'A';
    b   < 'A' => 'A' +  b ;
    b   <  b  =>  b  +  b ;
    b   < 'c' => 'c' +  b ;
   'c'  < 'A' => 'A' + 'c';
   'c'  <  b  =>  b  + 'c';
   'c'  < 'c' => 'c' + 'c';
endnewtype abc;
```

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.4).

Si la signature <literal signature> de l'identificateur <literal operator identifier> d'un terme <spelling term> est un nom <literal operator name>, le terme <spelling term> est alors la notation abrégée d'une chaîne de caractères en lettres majuscules obtenue à partir de l'identificateur <literal operator identifier>. La charstring contient l'orthographe en lettres majuscules du nom <literal operator name> de l'identificateur <literal operator identifier>.

Si la signature <literal signature> de l'identificateur <literal operator identifier> d'un terme <spelling term> est un littéral <character string literal>, le terme <spelling term> est alors la notation abrégée d'une charstring obtenue à partir du littéral <character string literal>. La charstring contient l'orthographe du littéral <character string literal>.

La 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 ci-dessus.

Par exemple,

```
newtype abc literals 'A',Bb,'c';
 operators
  "<" : abc,abc -> Boolean;
 map for all x,y in abc literals
  spelling(x) < spelling(y) => x < y;
endnewtype abc;
```

est la syntaxe concrète dérivée pour

```
newtype abc literals 'A',Bb,'c';
 operators
  "<" : abc,abc -> Boolean;
 axioms
  /* note that 'A', Bb, 'c' are bound to the local sort abc */
  /* '''A''', 'BB' and '''c''' should be qualified by the
 Charstring identifier if these literals are ambiguous - to be
 concise this is omitted below*/
'''A'''      < '''A'''    => 'A' < 'A';
'''A'''      < 'BB'       => 'A' <  Bb;
'''A'''      < '''c'''    => 'A' < 'c';
'BB'         < '''A'''    =>  Bb < 'A';
'BB'         < 'BB'       =>  Bb <  Bb;
'BB'         < '''c'''    =>  Bb < 'c';
'''c'''      < '''A'''    => 'c' < 'A';
'''c'''      < 'BB'       => 'c' <  Bb;
'''c'''      < '''c'''    => 'c' < 'c';
endnewtype abc;
```

Un terme <spelling term> doit faire partie d'une correspondance <literal mapping>.

L'identificateur <value identifier> faisant partie d'un terme <spelling term> doit être un identificateur <value identifier> défini par une quantification <literal quantification>.

5.3.2 Définitions d'opérateurs

Les définitions d'opérateurs permettent de définir les opérateurs de manière semblable aux procédures renvoyant une valeur. Toutefois, les opérateurs ne doivent pas avoir accès à l'état global ni le modifier. Ils ne contiennent donc qu'une seule transition. La sémantique des définitions d'opérateurs est exprimée en transformant la transition en une transition de départ de procédure.

Grammaire textuelle concrète

<operator definitions> ::=

{ <operator definition> | <textual operator reference>}+

<operator definition> ::=

operator
{ <operator identifier> | <operator name>} <end>
<formal parameters> <end> <operator result> <end>
{ <data definition>
| <variable definition>
| <macro definition>
| <select definition> }*
<start> { <free action> }*
endoperator
[{<operator identifier> | <operator name> }] <end>

<operator result> ::=

returns [<variable name>] <extended sort>

<textual operator reference> ::=

operator <operator name>
[<formal parameters> <operator result>]
referenced <end>

Une définition <operator definition> ou un diagramme <operator diagram> ne doit pas être utilisé(e) pour définir les opérateurs d'égalité implicites «=» et «/=».

Le départ <start> d'une définition <operator definition> doit contenir <virtuality>.

Pour chaque définition <operator definition> ou diagramme <operator diagram>, il doit exister une signature <operator signature> dans la même unité de portée ayant le même nom <operator name>, ayant par position les mêmes sortes <argument sort> que celles spécifiées dans les paramètres <formal parameters> et le même résultat <result> que celui spécifié dans le résultat <operator result>.

Pour chaque signature <operator signature>, on peut donner au plus une définition <operator definition> ou un diagramme <operator diagram> correspondant(e).

Les paramètres <formal parameters> et le résultat <operator result> dans une référence <textual operator reference> peuvent être omis s'il n'y a pas d'autre référence <textual operator reference> dans la même sorte, ayant le même nom. Dans ce cas, les paramètres <formal parameters> et le résultat <operator result> sont dérivés de la signature <operator signature>.

Une <transition> ou une zone <transition area> ne peut ni se référer à un opérateur <imperative operator> quelconque ni à un identificateur <identifier> quelconque défini à l'extérieur de la définition <operator definition> englobante ou du diagramme <operator diagram> englobant respectivement, à l'exception des identificateurs <synonym identifier>, <operator identifier>, <literal identifier> et des sortes <sort>.

Un opérateur défini par une définition <operator definition> ou par un diagramme <operator diagram> ne doit pas apparaître dans un axiome, dans un générateur ou dans une expression <ground expression>. Une définition <operator definition> ne doit pas apparaître dans un générateur.

Grammaire graphique concrète

<operator diagram> ::=

<frame symbol> ***contains***
{ <operator heading>
{ { <operator text area>
| <macro diagram> }*
<procedure start symbol> ***is followed by*** <transition area>
{<in-connector area> } * } ***set*** }

<operator heading> ::=

operator
{<operator identifier> | <operator name> }
<formal parameters>
<operator result>

<operator text area> ::=

<text symbol> ***contains***

{ <data definition>

| <variable definition>

| <select definition> }*

Comme il n'y a pas de grammaire graphique pour les définitions de sortes, un diagramme <operator diagram> ne peut être utilisé qu'à travers une référence <textual operator reference>.

Sémantique

Une définition d'opérateur est une unité de portée, définissant ses propres données et variables qui peuvent être manipulées à l'intérieur d'une transition.

Les variables introduites dans les paramètres <formal parameters> sont aussi modifiables.

Modèle

Une définition <operator definition> ou un diagramme <operator diagram> est transformé(e) respectivement en définition <procedure definition> ou diagramme <procedure diagram> comme défini en 7.

L'application dans une expression d'un opérateur défini par une définition <operator definition> est transformée en appel <value returning procedure call> comme défini en 7.

5.3.3 Utilisation des données

La façon dont les types de données, les sortes, les littéraux, les opérateurs et les synonymes sont interprétés dans les expressions est définie dans les paragraphes suivants.

5.3.3.1 Expressions

Les expressions sont des littéraux, des opérateurs, des accès de variables, des expressions conditionnelles et des opérateurs impératifs.

Grammaire abstraite

Expression = *Ground-expression* |
Active-expression

Une *expression* est une expression *active expression* si elle contient un primaire *active primary* (voir 5.4).

Une *expression* qui ne contient pas de primaire *active primary* est une expression *ground expression.*

Grammaire textuelle concrète

Pour simplifier la description, aucune distinction n'est établie entre la syntaxe concrète de l'expression *ground expression* et de l'expression *active expression*. La syntaxe concrète de l'<expression> est indiquée en 5.3.3.2.

Sémantique

Une expression est interprétée en tant que valeur de l'expression close ou de l'expression active. Si la valeur est **erreur**, le comportement ultérieur du système n'est pas défini.

L'expression a la sorte de l'expression close ou de l'expression active.

5.3.3.2 Expressions closes

Grammaire abstraite

Ground-expression :: *Ground-term*

Les conditions statiques applicables au terme *ground term* sont également valables pour l'expression *ground expression*.

Grammaire textuelle concrète

```
<ground expression> ::=
                        <ground expression>

<expression> ::=
                        <sub expression>
                    |   <value returning procedure call>

<sub expression> ::=
                        <operand0>
                    |   <sub expression> => <operand0>

<operand0> ::=
                        <operand1>
                    |   <operand0> { or | xor } <operand1>

<operand1> ::=
                        <operand2>
                    |   <operand1> and <operand2>

<operand2> ::=
                        <operand3>
                    |   <operand2> { = | /= | > | >= | < | <= | in } <operand3>

<operand3> ::=
                        <operand4>
                    |   <operand3> { + | - | // } <operand4>

<operand4> ::=
                        <operand5>
                    |   <operand4> { * | / | mod | rem } <operand5>

<operand5> ::=
                        [ - | not ] <primary>

<primary> ::=
                        <ground primary>
                    |   <active primary>
                    |   <extended primary>

<ground primary> ::=
                        <literal identifier>
                    |   <operator identifier> ( <ground expression list> )
                    |   ( <ground expression> )
                    |   <conditional ground expression>

<extended primary> ::=
                        <synonym>
                    |   <indexed primary>
                    |   <field primary>
                    |   <structure primary>
```

<ground expression list> ::=

<ground expression> { , <ground expression> }*

<operator identifier> ::=

<operator identifier>

| [<qualifier>] <quoted operator>

Une <expression> qui ne contient aucun primaire <active primary> représente une expression *ground expression* dans la syntaxe abstraite. Une expression <ground expression> ne doit pas contenir un primaire <active primary>.

Si une <expression> est un primaire <ground primary> avec un identificateur <operator identifier> et si une sorte <argument sort> de la signature <operator signature> est un syntype <syntype>, le contrôle de l'intervalle de ce syntype défini en 5.3.1.9.1 est alors appliqué à la valeur d'argument correspondante. La valeur du contrôle de l'intervalle doit être Vrai (True).

Si une <expression> est un primaire <ground primary> avec un identificateur <operator identifier> et si le résultat <result> de la signature <operator signature> est un syntype <syntype>, le contrôle de l'intervalle de ce syntype défini en 5.3.1.9.1 est alors appliqué à la valeur du résultat. La valeur du contrôle de l'intervalle doit être Vrai.

Si une <expression> contient un primaire <extended primary> (c'est-à-dire, un synonyme <synonym>, un primaire <indexed primary>, un primaire <field primary> ou un primaire <structure primary>), un remplacement est opéré au niveau de la syntaxe concrète selon la définition donnée en 5.3.3.3, 5.3.3.4, 5.3.3.5 et 5.3.3.6, respectivement, avant que la relation avec la syntaxe abstraite ne soit considérée.

Le qualificateur <qualifier> facultatif placé avant un opérateur <quoted operator> possède la même relation avec la syntaxe abstraite qu'un qualificateur <qualifier> d'un identificateur <operator identifier> (voir 5.2.2).

Sémantique

Une expression close est interprétée en tant que valeur représentée par le terme clos syntaxiquement équivalent, à l'expression close.

En général, il n'est ni nécessaire ni justifié d'établir une distinction entre le terme clos et la valeur de ce terme. Par exemple, le terme clos de la valeur du nombre entier représentant l'unité peut s'écrire «1». Il existe normalement plusieurs termes clos pour désigner la même valeur, 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 la valeur.

Une expression close a une sorte qui est la sorte du terme clos équivalent.

Une expression close a une valeur qui est la valeur du terme clos équivalent.

5.3.3.3 Synonyme

Grammaire textuelle concrète

<synonym> ::=

<synonym identifier>

| <external synonym>

La variante synonyme <external synonym> est décrite en 4.3.1.

Sémantique

Un synonyme est une notation abrégée permettant de désigner une expression définie ailleurs.

Modèle

Un synonyme <synonym> représente l'expression <ground expression> décrite dans la définition <synonym definition> identifiée par l'identificateur <synonym identifier>. Un identificateur <identifier> utilisé dans l'expression <ground expression> représente un identificateur *identifier* en syntaxe abstraite, conformément au contexte de la définition <synonym definition>.

5.3.3.4 Primaire indexé

Un primaire indexé est une notation syntaxique abrégée qui peut être utilisée pour désigner l'«indexation» d'une valeur d'un «tableau». Toutefois, à l'exception de sa forme syntaxique spéciale, un primaire indexé n'a pas de propriétés spéciales et désigne un opérateur avec le primaire comme paramètre.

Grammaire textuelle concrète

<indexed primary> ::=

<primary> (<expression list>)

Sémantique

Une expression indexée représente l'application d'un opérateur Extract!

Modèle

Un primaire <primary> suivi d'une liste <expression list> entre crochets est une syntaxe concrète représentant la syntaxe concrète

Extract!(<primary>, <expression list>)

et il faut alors considérer qu'il s'agit d'une expression correcte même si Extract! n'est pas admis, dans la syntaxe concrète, comme nom d'opérateur pour les expressions. La syntaxe abstraite est déterminée à partir de cette expression concrète conformément au 5.3.3.2.

5.3.3.5 Primaire de champ

Un primaire de champ est une notation syntaxique abrégée qui peut être utilisée pour désigner la «sélection de champ» des «structures». Toutefois, à l'exception de sa forme syntaxique spéciale, un primaire de champ n'a pas de propriétés spéciales et désigne un opérateur avec le primaire comme paramètre.

Grammaire textuelle concrète

<field primary> ::=

<primary> <field selection>

<field selection> ::=

! <field name>

| (<field name> { , <field name> }*)

Le nom de champ doit être un nom de champ défini pour la sorte du primaire.

Sémantique

Un primaire de champ représente l'application de l'un des opérateurs d'extraction de champ d'une sorte structurée.

Modèle

La forme

<primary> (<field name>)

est la syntaxe dérivée de

<primary> ! (<field name>)

La forme

<primary> (<first field name> { , <field name> }*)

est la syntaxe dérivée de

<primary> ! <first field name> { ! <field name> }*

où l'ordre des noms de champ est préservé.

La forme

<primary> ! <field name>

est la syntaxe dérivée représentant

<field extract operator name> (<primary>)

où le nom d'opérateur d'extraction de champ est formé de la concaténation du nom de champ et de «Extract!» dans cet ordre. Par exemple,

s ! f1

est la syntaxe dérivée de

f1Extract!(s)

et il faut alors considérer cela comme une expression correcte même si f1Extract! n'est pas admis, dans la syntaxe concrète, comme nom d'opérateur pour les expressions. La syntaxe abstraite est déterminée à partir de cette expression concrète conformément au 5.3.3.2.

Au cas où il existe un opérateur défini pour une sorte, de telle façon que

Extract!(s,name)

est un terme valable lorsque le «nom» est le même qu'un nom de champ valable de la sorte de s, il s'ensuit qu'un primaire

s(name)

est la syntaxe concrète dérivée de

Extract!(s,name)

et la sélection de champ doit s'écrire

s ! name

5.3.3.6 Primaire de structure

Grammaire textuelle concrète

<structure primary> ::=
[<qualifier>] (. <expression list> .)

Sémantique

Un primaire de structure représente une valeur d'une sorte structurée qui est construite à partir d'expressions correspondant à chaque champ de la structure.

La forme

(. <expression list> .)

est la syntaxe concrète dérivée de

Make!(<expression list>)

qui est considérée comme une expression close correcte même si Make! n'est pas admis, dans la syntaxe concrète, comme un nom d'opérateur pour les expressions closes. La syntaxe abstraite est déterminée à partir de cette expression close concrète conformément au 5.3.3.1.

5.3.3.7 Expression close conditionnelle

Grammaire textuelle concrète

<conditional ground expression> ::=
if <Boolean ground expression>
then <consequence ground expression>
else <alternative ground expression>
fi

<consequence ground expression> ::=
<ground expression>

<alternative ground expression> ::=
<ground expression>

L'expression <conditional ground expression> représente une expression *ground expression* dans la syntaxe abstraite. Si l'expression <Boolean ground expression> représente la valeur Vrai, l'expression *ground expression* est représentée par l'expression <consequence ground expression>, autrement elle est représentée par l'expression <alternative ground expression>.

La sorte de l'expression <consequence ground expression> doit être la même que la sorte de l'expression <alternative ground expression>.

Sémantique

Une expression close conditionnelle est un primaire clos qui est interprété comme étant l'expression close de conséquence ou l'expression close d'alternative.

Si l'expression <Boolean ground expression> a la valeur Vrai, il s'ensuit que l'expression <alternative ground expression> n'est pas interprétée. Si l'expression <Boolean ground expression> a la valeur Faux, il s'ensuit que l'expression <consequence ground expression> n'est pas interprétée. Le comportement ultérieur du système n'est pas défini si l'expression <ground expression> qui est interprétée a la valeur d'une erreur.

Une expression close conditionnelle a une sorte qui est la sorte de l'expression close de conséquence (et également la sorte de l'expression close d'alternative).

5.4 Utilisation de données comportant des variables

Le présent paragraphe définit l'utilisation de données et de variables déclarées dans les processus et procédures, ainsi que les opérateurs impératifs qui obtiennent des valeurs de l'état du système sous-jacent.

Une variable a une sorte et une valeur associée de cette sorte. La valeur associée à une variable peut être modifiée si une nouvelle valeur est affectée à la variable. On peut utiliser la valeur associée à la variable dans une expression en accédant à la variable.

Une expression quelconque contenant une variable est considérée comme «active» étant donné que la valeur obtenue en interprétant l'expression peut varier selon la dernière valeur affectée à la variable.

5.4.1 Variables et définition de données

Grammaire textuelle concrète

<data definition> ::=
{ <partial type definition>
| <syntype definition>
| <generator definition>
| <synonym definition> } <end>

Une définition de données fait partie d'une définition *data type definition* s'il s'agit d'une définition <partial type definition> ou d'une définition <syntype definition>.

La syntaxe qui régit l'introduction des variables de processus ainsi que les variables des paramètres de procédure est indiquée en 2.5.1.1 et 2.3.4, respectivement. Une variable définie dans une procédure ne doit pas être révélée.

Sémantique

Une définition de données est utilisée pour la définition d'une partie d'un type de données ou pour la définition d'un synonyme d'une expression conformément à la définition plus complète donnée en 5.2.1, 5.3.1.9 ou 5.3.1.13.

5.4.2 Accès aux variables

L'interprétation d'une expression comportant des variables est décrite ci-après.

5.4.2.1 Expressions actives

Grammaire abstraite

Active-expression	=	*Variable-access* |
		Conditional-expression |
		Operator-application |
		Imperative-operator |
		Error-term

Grammaire textuelle concrète

<active expression> ::=
<active expression>

<active primary> ::=
<variable access>
| <operator application>
| <conditional expression>
| <imperative operator>
| (<active expression>)
| <active extended primary>
| **error**

<active extended primary> ::=
<active extended primary>

<expression list> ::=
<expression> { , <expression> }*

Une <expression> est une expression <active expression> si elle contient un primaire <active primary>.

Un primaire <extended primary> est un primaire <active extended primary> s'il contient un primaire <active primary>. Dans le cas d'un primaire <extended primary>, le remplacement au niveau de la syntaxe concrète intervient conformément à la définition donnée en 5.3.3.3, 5.3.3.4, 5.3.3.5 et 5.3.3.6, avant que la relation avec la syntaxe abstraite ne soit envisagée.

Sémantique

Une expression active est une expression dont la valeur dépendra de l'état actuel du système.

Une expression active a une sorte qui est la sorte du terme clos équivalent.

Une expression active a une valeur qui est le terme clos équivalent de l'expression active au moment de l'interprétation.

Si une expression contient **error** est interprétée, le comportement ultérieur du système n'est pas défini.

Dans une expression active, chaque opérateur est interprété dans l'ordre déterminé en parcourant la syntaxe concrète donnée en 5.3.3.2 de gauche à droite. Dans une liste d'expression active, ou une liste d'expression, chaque élément de la liste est interprété dans l'ordre de gauche à droite.

Modèle

Chaque fois que l'expression active est interprétée, la valeur correspondante est déterminée en recherchant le terme clos équivalant à cette expression active. Ce terme clos est déterminé à partir d'une expression close formée en remplaçant chaque primaire actif figurant dans l'expression active par le terme clos équivalant à la valeur de ce primaire actif. La valeur d'une expression active est la même que celle de l'expression close.

5.4.2.2 Accès de variable

Grammaire abstraite

Variable-access = *Variable-identifier*

Grammaire textuelle concrète

<variable access> ::=
<variable identifier>

Sémantique

L'interprétation d'un accès de variable donne la valeur associée à la variable identifiée.

Un accès de variable a une sorte qui est celle de la variable identifiée par l'accès de variable.

Un accès de variable a une valeur qui est la dernière valeur associée à la variable ou qui est **erreur** si la variable est «non définie». Si la valeur d'un accès de variable est **erreur**, il s'ensuit que le comportement futur du système n'est pas défini.

5.4.2.3 Expression conditionnelle

Une expression conditionnelle est une expression qui est interprétée comme conséquence ou comme alternative.

Grammaire abstraite

Conditional-expression :: *Boolean-expression*
Consequence-expression
Alternative-expression

Boolean-expression = *Expression*

Consequence-expression = *Expression*

Alternative-expression = *Expression*

La sorte de l'expression *consequence expression* doit être la même que celle de l'expression *alternative expression*.

Grammaire textuelle concrète

<conditional expression> ::=
if <Boolean active expression>
then <consequence expression>
else <alternative expression>
fi
| **if** <Boolean expression>
then <active consequence expression>
else <alternative expression>
fi
| **if** <Boolean expression>
then <consequence expression>
else <active alternative expression>
fi

<consequence expression> ::=
<expression>

<alternative expression> ::=
<expression>

Une expression <conditional expression> diffère d'une expression <conditional ground expression> par l'occurrence d'une expression <active expression> dans l'expression <conditional expression>.

Sémantique

L'expression conditionnelle est interprétée comme l'interprétation de la condition suivie de l'interprétation de l'expression de conséquence ou de l'interprétation de l'expression d'alternative. La conséquence n'est interprétée que si la condition comporte la valeur Vrai, de sorte que si la condition comporte la valeur Faux, il s'ensuit que le comportement futur du système n'est pas défini uniquement si l'expression d'alternative est une erreur. De même, l'alternative n'est interprétée que si la condition comporte la valeur Faux, de sorte que si la condition comporte la valeur Vrai, il s'ensuit que le comportement futur du système n'est pas défini uniquement si l'expression de conséquence est une erreur.

L'expression conditionnelle a une sorte qui est la même que celle de la conséquence et de l'alternative. L'expression conditionnelle a une valeur qui est celle de la conséquence si la condition est Vrai ou celle de l'alternative si la condition est Faux.

5.4.2.4 Application d'opérateur

Une application d'opérateur est l'application d'un opérateur dans laquelle un ou plusieurs des arguments réels est une expression active.

Grammaire abstraite

Operator-application :: *Operator-identifier*
Expression^{+}

Si une *sorte* d'argument de la signature *operator signature* est un *syntype* et si l'*expression* correspondante de la liste d'*expression* est une expression *ground expression*, le contrôle de l'intervalle défini en 5.3.1.9.1 qui est appliqué à la valeur de l'*expression* doit être Vrai.

Grammaire textuelle concrète

<operator application> ::=
<operator identifier> (<active expression list>)

<active expression list> ::=

<active expression> [, <expression list>]

| <ground expression> , <active expression list>

Une application <operator application> diffère de l'expression <ground expression> analogue sur le plan syntaxique en ce sens que l'une des <expression> de la liste d'<expression> figurant entre crochets est une expression <active expression>. Si toutes les <expression> entourées de crochets sont des expressions <ground expression>, la construction représente une expression *ground expression* selon la définition du 5.3.3.2.

Sémantique

Une application d'opérateur est une expression active qui a la valeur du terme clos équivalent de l'application d'opérateur. Le terme clos équivalent est déterminé de la même façon qu'en 5.4.2.1.

La liste d'expressions concernant l'application d'opérateur est interprétée dans l'ordre indiqué avant l'interprétation de l'opérateur.

Si une sorte d'argument de la signature d'opérateur est un syntype et si l'expression correspondante contenue dans la liste d'expressions actives est une expression active, le contrôle de l'intervalle défini en 5.3.1.9.1 est appliqué à la valeur de l'expression. Si le contrôle de l'intervalle est Faux au moment de l'interprétation, le système est alors en erreur et le comportement ultérieur du système n'est pas défini.

Si la sorte de résultat de la signature de l'opérateur est un syntype, il s'ensuit que le contrôle de l'intervalle défini en 5.3.1.9.1 est appliqué à la valeur de l'application d'opérateur. Si le contrôle de l'intervalle est Faux au moment de l'interprétation, le système est alors en erreur et le comportement ultérieur du système n'est pas défini.

5.4.3 Instruction d'affectation

Grammaire abstraite

Assignment-statement :: *Variable-identifier*
Expression

La sorte de l'identificateur *variable identifier* doit être la même que celle de l'*expression*.

Si la *variable* est déclarée avec un *syntype* et si l'*expression* est une expression *ground expression*, il s'ensuit que le contrôle de l'intervalle tel que défini en 5.3.1.9.1 et appliqué à l'*expression* doit être Vrai.

Grammaire textuelle concrète

<assignment statement> ::=

<variable> := <expression>

<variable> ::=

<variable identifier>

| <indexed variable>

| <field variable>

Si la <variable> est un identificateur <variable identifier> l'<expression> en syntaxe concrète représente alors l'<expression> en syntaxe abstraite. Les autres formes de <variable>, variable <indexed variable> et variable <field variable> correspondent à des syntaxes dérivées et l'<expression> en syntaxe abstraite est déduite de la syntaxe concrète équivalente définie en 5.4.3.1 et 5.4.3.2.

Sémantique

Une instruction d'affectation est interprétée comme créant une association entre la variable identifiée dans l'instruction d'affectation et la valeur de l'expression contenue dans l'instruction d'affectation. L'association antérieure de la variable est perdue.

Si la variable est déclarée avec un syntype et si l'expression est une expression active, il s'ensuit que le contrôle de l'intervalle défini en 5.3.1.9.1 est appliqué à l'expression. Si ce contrôle de l'intervalle est équivalent à Faux, l'affectation est alors dans l'erreur et le comportement ultérieur du système n'est pas défini.

5.4.3.1 Variable indexée

Une variable indexée est une notation syntaxique abrégée qui peut être utilisée pour désigner l'«indexation» des «tableaux». Toutefois, à l'exception de sa forme syntaxique spéciale, un primaire actif indexé n'a pas de propriétés spéciales et désigne un opérateur avec le primaire actif comme paramètre.

Grammaire textuelle concrète

<indexed variable> ::=
<variable> (<expression list>)

Il faut qu'il existe une définition appropriée d'un opérateur appelé Modify!.

Sémantique

Une variable indexée représente l'affectation d'une valeur formée par l'application de l'opérateur Modify! à un accès de la variable et à l'expression indiquée dans la variable indexée.

Modèle

La forme syntaxique concrète

<variable> (<expression list>) := <expression>

est la syntaxe concrète dérivée de

<variable> := Modify!(<variable>,<expression list>,<expression>)

où la même <variable> est répétée et le texte est considéré comme une affectation correcte même si Modify! n'est pas admis, dans la syntaxe concrète, comme nom d'opérateur pour les expressions. La syntaxe abstraite est déterminée, pour cette instruction <assignment statement> conformément au 5.4.3 ci-dessus.

NOTE – Ce modèle a pour conséquence qu'une valeur doit être affectée à un tableau complet, avant de pouvoir modifier un élément.

5.4.3.2 Variable de champ

Une variable de champ est une notation abrégée permettant d'affecter une valeur à une variable de telle manière que la valeur existant dans un champ de cette variable est la seule à être modifiée.

Grammaire textuelle concrète

<field variable> ::=
<variable> <field selection>

Il faut qu'il existe une définition appropriée d'un opérateur appelé Modify! Normalement, cette définition sera déduite d'une définition structurée de la sorte.

Sémantique

Une variable de champ représente l'affectation d'une valeur formée par l'application d'un opérateur de modification de champ.

Modèle

La sélection de champ entre crochets est la syntaxe dérivée de !<field name> sélection de champ conformément à la définition donnée en 5.3.3.5.

La forme syntaxique concrète

<variable> ! <field name> := <expression>

est la syntaxe concrète dérivée de

<variable> := <field modify operator name> (<variable>, <expression>)

où

a) la même <variable> est répétée; et

b) le nom <field modify operator name> est formé à partir de la concaténation du nom de champ et de «Modify!»; puis

c) le texte est considéré comme affectation correcte même si le nom <field modify operator name> n'est pas admis, dans la syntaxe concrète, comme nom d'opérateur pour les expressions.

S'il existe plus d'un nom <field name> dans la sélection de champ, ils sont alors représentés de la façon indiquée ci-dessus, chaque !<field name>, étant développé tour à tour de droite à gauche et la partie restante de la variable <field variable> étant considérée comme une <variable>. Par exemple,

var ! fielda ! fieldb := expression;

est représenté tout d'abord par

var ! fielda := fieldbModify!(var ! fielda, expression);

puis par

var := fieldbModify!(var, fieldbModify!(var ! fielda, expression));

La syntaxe abstraite est déterminée pour l'instruction <assignment statement> formée à partir du modèle conformément en 5.4.3 ci-dessus.

NOTE – Ce modèle a pour conséquence qu'une valeur doit être affectée à une structure complète, avant de pouvoir modifier un élément.

5.4.3.3 Initialisation par défaut

Une initialisation par défaut permet l'initialisation de toutes les variables d'une sorte spécifiée avec la même valeur, lorsque les variables sont créées.

Grammaire textuelle concrète

<default initialization> ::=
default <ground expression> [<end>]

Une définition <partial type definition> ou une définition <syntype definition> ne doit pas contenir plus d'une initialisation <default initialization>.

Sémantique

Une initialisation par défaut est ajoutée, à titre facultatif, à une expression de propriétés d'une sorte. Une initialisation par défaut spécifie que n'importe quelle variable déclarée avec la sorte introduite par la définition partielle de type ou la définition de syntype prend au départ la valeur de l'expression close associée.

Modèle

L'initialisation par défaut est une abréviation pour spécifier une initialisation explicite pour les variables de la sorte <sort> déclarées sans expression <ground 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 sa valeur soit dans l'intervalle.

Une variable déclarée à l'intérieur d'un type paramétré dont la sorte est un paramètre de contexte formel ne prend pas l'initialisation par défaut de la sorte.

5.4.4 Opérateurs impératifs

Les opérateurs impératifs obtiennent des valeurs de l'état du système sous-jacent.

Les transformations décrites dans les *Modèles* de ce paragraphe sont effectuées au même moment que le développement de l'import. Une étiquette attachée à une action dans laquelle apparaît un opérateur impératif est déplacée à la première tâche insérée pendant la transformation décrite. Si plusieurs opérateurs impératifs apparaissent dans une expression, les tâches sont insérées dans le même ordre que l'ordre d'apparition des opérateurs impératifs dans l'expression.

Grammaire abstraite

Imperative-operator = *Now-expression* |
Pid-expression |
View-expression |
Timer-active-expression |
Anyvalue-expression

Grammaire textuelle concrète

<imperative operator> ::=
<now expression>
| <import expression>
| <PId expression>
| <view expression>
| <timer active expression>
| <anyvalue expression>

Les opérateurs impératifs sont des expressions permettant l'accès à l'horloge du système, aux valeurs des variables importées, aux valeurs de PId associées à un processus, aux valeurs des variables visibles, aux états des temporisateurs ou permettant de fournir des valeurs non spécifiées.

5.4.4.1 Expression maintenant (now)

Grammaire abstraite

Now-expression :: ( )

Grammaire textuelle concrète

<now expression> ::=
now

Sémantique

L'expression maintenant 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.

Modèle

L'utilisation de l'expression <now expression> dans une expression est une abréviation pour insérer une tâche juste avant une action, où se produit l'expression qui affecte à une variable implicite la valeur de l'expression <now

expression> et utilise ensuite cette variable implicite dans l'expression. Si l'expression <now expression> se produit plusieurs fois dans une expression, une variable doit être utilisée pour chaque occurrence.

5.4.4.2 Expression d'import

Grammaire textuelle concrète

La syntaxe concrète d'une expression d'import est définie en 4.13.

Sémantique

En plus de la sémantique définie en 4.13, une expression d'import est interprétée comme un accès de variable (voir 5.4.2.2) à la variable implicite pour l'expression d'import.

Modèle

L'expression d'import a une syntaxe implicite pour l'importation de la valeur définie en 4.13 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 une action, où se produit l'expression qui affecte à une variable implicite la valeur 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.

5.4.4.3 Expression PId

Grammaire abstraite

Pid-expression	=	*Self-expression* |
		Parent-expression |
		Offspring-expression |
		Sender-expression
Self-expression	::	( )
Parent-expression	::	( )
Offspring-expression	::	( )
Sender-expression	::	( )

Grammaire textuelle concrète

<PId expression> ::=
self
| **parent**
| **offspring**
| **sender**

Sémantique

Une expression PId permet d'accéder à une des variables implicites de processus définies en 2.4.4. L'expression variable de processus est interprétée comme étant la dernière valeur associée à la variable implicite correspondante.

Une expression PId a une sorte qui est PId.

Une expression PId a une valeur qui est la dernière valeur associée à la variable correspondante comme définie en 2.4.4.

5.4.4.4 Expression de vue

Une expression de vue permet à un processus d'obtenir la valeur attribuée à une variable d'un autre processus du même bloc comme si la variable était définie localement. Le processus de visualisation ne peut modifier la valeur associée à la variable.

Grammaire abstraite

View-expression :: *View-identifier*
[*Expression*]

L'*expression* doit être une expression PId.

Grammaire textuelle concrète

<view expression> ::=
view (<view identifier> [, <PId expression>])

Sémantique

Une expression de vue est interprétée de la même façon qu'un accès de variable (voir 5.4.2.2).

Une expression de vue a une valeur qui est la valeur d'accès de variable et une sorte qui est la sorte de la définition *View-definition*.

Si une *Expression* est donnée, la variable d'accès est la variable dans l'instance de processus à l'intérieur du même bloc identifié par l'*Expression*. Si l'*Expression* désigne une instance non existante ou si le processus désigné par *Expression* ne contient pas de variable du même nom et de la même sorte, aucun accès de variable ne peut être effectué.

Si aucune expression *Expression* n'est donnée, la variable d'accès est la variable dans une instance arbitraire de processus à l'intérieur du bloc, contenant une variable révélée avec le même nom et la même sorte. S'il n'existe pas de telles instances, aucun accès de variable ne peut être effectué.

Si aucun accès de variable ne peut être effectué sur la base de l'expression <view expression>, le comportement ultérieur du système n'est pas défini.

Modèle

L'utilisation de l'expression <view 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 la valeur de l'expression <view expression> et utilise ensuite cette variable implicite dans l'expression. Si l'expression <view expression> se produit plusieurs fois dans une expression, une variable doit être utilisée pour chaque occurrence.

5.4.4.5 Expression active de temporisateur

Grammaire abstraite

Timer-active-expression :: *Timer-identifier*
*Expression**

Les sortes de l'*Expression** contenues dans l'expression *Timer-active-expression* doivent correspondre par leur position à l'identificateur *Sort-reference-identifier** suivant directement le nom *Timer-name* (2.8) 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 la valeur Vrai si le temporisateur identifié par l'identificateur de temporisateur, et positionné avec les mêmes valeurs que celles indiquées par la liste d'expressions (le cas échéant), est actif (voir 2.8). Dans le cas contraire, l'expression active de temporisateur a la valeur Faux. Les expressions sont interprétées dans l'ordre indiqué.

Si une sorte spécifiée dans un temporisateur est un syntype, la vérification de l'intervalle définie en 5.3.19.1 appliquée à l'expression correspondante dans la liste <expression list> doit donner Vrai, sinon le système est dans un état d'erreur et son comportement ultérieur n'est pas défini.

Modèle

L'utilisation de l'expression <timer active 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 la valeur de l'expression <timer active expression> et utilise ensuite cette variable implicite dans l'expression. Si l'expression <timer active expression> se produit plusieurs fois dans une expression, une variable doit être utilisée pour chaque occurrence.

5.4.4.6 Expression valeur quelconque (Anyvalue)

Grammaire abstraite

Anyvalue-expression :: *Sort-reference-identifier*

Grammaire textuelle concrète

<anyvalue expression> ::=
any(<sort>)

Sémantique

Une expression *Anyvalue-expression* est une valeur non spécifiée de la sorte ou du syntype désigné par l'identificateur *Sort-reference-identifier*. S'il n'existe pas de valeur, le comportement ultérieur du système est non défini. Si l'identificateur *Sort-reference-identifier* désigne un identificateur *Syntype-identifier*, la valeur résultante sera dans l'intervalle de ce syntype. L'expression *Anyvalue-expression* est utile pour modéliser le comportement lorsque la donnée d'une valeur spécifique constituera une surspécification. On ne peut pas tirer des hypothèses concernant les autres valeurs retournées par une expression *Anyvalue-expression* à partir d'une valeur retournée par une expression *Anyvalue-expression*.

Modèle

L'utilisation de l'expression <anyvalue 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 la valeur de l'expression <anyvalue expression> et utilise ensuite cette variable implicite dans l'expression. Si l'expression <anyvalue expression> se produit plusieurs fois dans une expression, une variable doit être utilisée pour chaque occurrence.

5.4.5 Appel de procédure renvoyant une valeur

Grammaire textuelle concrète

<value returning procedure call> ::=
<procedure call>
| <remote procedure call>

Un appel <value returning procedure call> ne doit pas se produire dans une 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 identifier> dans un appel <value returning procedure call> doit identifier une procédure (ou une procédure distante) ayant au moins un paramètre dans ses paramètres <procedure formal parameters> et le paramètre formel final doit avoir l'attribut **in/out**. Cette règle s'applique après transformation du résultat <procedure result>.

NOTE – Normalement, l'identificateur <procedure identifier> identifie une procédure ayant un résultat <procedure result>.

Modèle

L'utilisation de l'appel <value returning procedure call> dans une expression est une abréviation pour insérer une action d'appel de procédure, juste avant l'action où l'expression se produit, contenant l'appel <value returning procedure call> où une expression supplémentaire a été ajoutée aux paramètres <actual parameters>, et utiliser ensuite cette <expression> à la place de l'appel <value returning procedure call> dans l'action suivante.

L'<expression> construite se compose de l'identificateur d'une nouvelle variable implicite distincte dont la sorte <sort> est la sorte du paramètre formel final de la procédure.

La transformation est effectuée lorsque d'autres opérateurs impératifs sont supprimés des expressions (voir 5.4.4).

NOTE – La transformation de l'appel de procédure renvoyant une valeur après le résultat <procedure result> implique qu'une procédure avec <procedure result> peut être utilisée dans un appel <procedure call> et qu'une procédure sans <procedure result> peut être utilisée dans un appel <value returning procedure call> si elle répond aux conditions énoncées dans ce paragraphe.

5.4.6 Données externes

Grammaire textuelle concrète

<external properties> ::=

alternative
<external formalism name> [, <word>] <end>
<external data description>
[**endalternative**] [<end>]

<external formalism name> ::=

<text>

<external data description> ::=

<text>

Le nom <external formalism name> ne doit pas contenir le caractère «;» ou «,». Si le mot <word> est présent, il désigne une séquence qui termine la description <external data description>. Si cette description ne contient pas le mot clé **endalternative**, le mot <word> peut être omis. Sinon, le mot <word> termine la définition de données de rechange et **endalternative** peut être omis. Le mot <word> terminal ne fait pas formellement partie de la description SDL.

Sémantique

Le mot clé **alternative** indique que la description <external data description> ne fait pas formellement partie de la description SDL. L'utilisation du nom <external formalism name> permet la liaison de la description <external data description> avec certains formalismes externes. Les relations avec les formalismes externes n'entrent pas dans le cadre de la présente Recommandation.

Les littéraux et opérateurs utilisés à l'extérieur d'une définition <partial type definition> sont uniquement ceux qui sont définis dans les opérateurs <operators>.

Si une Recommandation séparée définit la correspondance d'un formalisme désigné par le nom <external formalism name> avec des valeurs fondées sur le progiciel Predefined (voir l'Annexe D), la correspondance peut impliquer des littéraux et des opérateurs implicites. Dans ce cas, ces derniers sont considérés en plus de ceux qui sont déclarés dans les opérateurs <operators> d'une définition <partial type definition>.

La sémantique d'une définition <partial type definition> avec des propriétés <external properties> est conceptuellement supposée donnée dans un ensemble d'axiomes non disponibles pour la description SDL.

Exemple

```
newtype application_data

     literals empty;

     operators

        anyuseExtract!  : application_data         -> Boolean;

        idExtract!      : application_data         -> Integer;

        anyuseModify! : Boolean, application_data     -> application_data;

        idModify!       : Integer, application_data    -> application_data;

        Make!           : Boolean, Integer         -> application_data;

     alternative ASN.1;

        SET {   anyuse BOOLEAN

                id    INTEGER }

     endalternative;

endnewtype application_data;
```

6 Concepts de définition de type dans le SDL

Le présent article 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/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 progiciel;

b) des définitions d'instances fondées sur les types qui définissent 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.

Les concepts définis dans le présent article sont des concepts supplémentaires. Les propriétés d'une notation abrégée sont dérivées de la manière dont elle est modélisée en termes de concepts primitifs (ou transformée en ces concepts). Afin d'assurer une utilisation facile et non ambiguë des notations abrégées, et pour réduire les effets indésirables lors de la combinaison de plusieurs notations abrégées, ces concepts sont transformés dans un ordre spécifié, comme défini en 7.

6.1 Types, instances et accès

On distingue dans les descriptions du SDL entre les définitions d'instances (ou d'ensemble d'instances) et les définitions de types. Le présent paragraphe spécifie (en 6.1.1) les définitions de types pour les systèmes, les blocs, les processus et les services ainsi que les spécifications des instances correspondantes (en 6.1.3). Les articles 2 et 5 définissent les signaux, les procédures, les temporisateurs et les sortes en tant que types. Une définition de type n'est connectée (par des canaux ou des acheminements de signaux) à aucune instance; par contre, les définitions de types introduisent des accès (6.1.4). Il s'agit des points de connexion sur les instances fondées sur les types pour les canaux et les acheminements des signaux.

6.1.1 Définitions de types

6.1.1.1 Type de système

Grammaire textuelle concrète

<system type definition> ::=
system type {<system type name> | <system type identifier>}
[<formal context parameters>]
[<specialization>] <end>
{<entity in system>}*
endsystem type [<system type name> | <system type identifier>] <end>

<textual system type reference>::=
system type <system type name> **referenced** <end>

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

Grammaire graphique concrète

<system type diagram> ::=
is associated with
<frame symbol> ***contains***
{<system type heading>
{ {<system text area>}*
{<macro diagram>}*
<block interaction area>
{<type in system area>}*} ***set*** }

<system type heading> ::=

system type {<system type name> | <system type identifier>}
[<formal context parameters>]
[<specialization>]

<type in system area> ::=

<block type diagram>
| <block type reference>
| <process type diagram>
| <process type reference>
| <service type diagram>
| <service type reference>
| <procedure diagram>
| <graphical procedure reference>

<system type reference>::=

<system type symbol> ***contains*** { **system** <system type name> }

<system type symbol>::=

<block type symbol>

Sémantique

Une définition <system type definition> définit un type de système. Tous les systèmes d'un type de système donné ont les mêmes propriétés que celles qui sont définies pour ce type de système.

6.1.1.2 Type de bloc

Grammaire textuelle concrète

<block type definition> ::=

[<virtuality>]
block type {<block type name> | <block type identifier>}
[<formal context parameters>]
[<virtuality constraint>]
[<specialization>] <end>
{<gate definition>}*
{<entity in block>}*
[<block substructure definition>
| <textual block substructure reference>]
endblock type [<block type name> | <block type identifier>] <end>

<textual block type reference>::=

[<virtuality>] **block type** <block type name>
referenced <end>

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

Grammaire graphique concrète

<block type diagram> ::=
<frame symbol>
contains {<block type heading>
{ {<block text area>}* {<macro diagram>}*
{<type in block area>}*
[<process interaction area>]
[<block substructure area>]} ***set***}
is associated with
{{{ <gate> }* { <graphical gate constraint> }* } ***set*** }

<block type heading> ::=
[<virtuality>]
block type {<block type name> | <block type identifier>}
[<formal context parameters>] [<virtuality constraint>]
[<specialization>]

<type in block area>::=
<block type diagram>
| <block type reference>
| <process type diagram>
| <process type reference>
| <service type diagram>
| <service type reference>
| <procedure diagram>
| <graphical procedure reference>

<block type reference>::=
<block type symbol> ***contains***
{[<virtuality>] <block name> }

<block type symbol>::=

Sémantique

Une définition <bloc type definition> définit un type de bloc. Tous les blocs d'un type de bloc donné ont les mêmes propriétés que celles qui sont définies pour ce type de bloc.

6.1.1.3 Type de processus

Grammaire textuelle concrète

<process type definition> ::=
[<virtuality>]
process type { <process type name> | <process type identifier> }
[<formal context parameters>]
[<virtuality constraint>]
[<specialization>] <end>
[<formal parameters> <end>] [<valid input signal set>]
{<gate definition> }*
{<entity in process>}*
[<process type body>]
endprocess type [<process type name> | <process type identifier>] <end>

<process type body> ::=
<procedure body>

<textual process type reference> ::=
[<virtuality>] **process type** <process type name>
referenced <end>

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

Grammaire graphique concrète

<process type diagram> ::=
<frame symbol>
contains {<process type heading>
{ {<process text area>}*
{<type in process area>}*
{<macro diagram> }*
{<process type graph area> | <service interaction area> }}***set*** }
is associated with
{{ {<gate>}* {<graphical gate constraint>}*} ***set*** }

<process type heading> ::=
[<virtuality>]
process type {<process type name> | <process type identifier>}
[<formal context parameters>]
[<virtuality constraint>]
[<specialization>] [<end>]
[<formal parameters>]

<process type graph area> ::=
[<start area>] { <state area> | <in-connector area> }*

<type in process area>::=
<service type diagram>
| <service type reference>
| <procedure diagram>
| <graphical procedure reference>

<process type reference>::=
<process type symbol> ***contains***
{[<virtuality>] <process type name> }

<process type symbol>::=

Sémantique

Une définition <process type definition> définit un type de processus. Tous les processus d'un type de processus donné ont les mêmes propriétés que celles qui sont définies pour ce type de processus.

L'ensemble complet des signaux d'entrées valides d'un type de processus est l'union de l'ensemble complet des signaux d'entrée valides de son supertype, des listes <signal list> au niveau de tous les accès en direction du type de processus, l'ensemble <valid input signal set>, les signaux d'entrée implicites introduits par les concepts supplémentaires des paragraphes 4.10-4.14 et leurs signaux de temporisation.

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

6.1.1.4 Type de service

Grammaire textuelle concrète

<service type definition> ::=
[<virtuality>]
service type{<service type name> | <service type identifier>}
[<formal context parameters>]
[<virtuality constraint>]
[<specialization>] <end>
[<valid input signal set>]
{<gate definition>}*
{<entity in service>}*
[<service type body>]
endservice type [{<service type name> | <service type identifier>}] <end>

<service type body>::=
<process type body>

<textual service type reference>::=
[<virtuality>] **service type** <service type name>
referenced <end>

Un paramètre <formal context parameter> des paramètres <formal context parameters> ne doit pas être un paramètre <timer context parameter>.

Une définition <variable definition> dans une définition <service type definition> ne doit pas contenir le mot clé **revealed**.

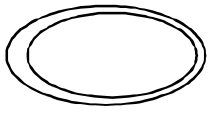
Grammaire graphique concrète

<service type diagram> ::=
<frame symbol> ***contains***
{<service type heading>
{ {<service text area> }*
{<graphical procedure reference>}*
{<procedure diagram>}*
{<macro diagram>}*
<service graph area> } ***set*** }
is associated with
{ { {<gate>}* {<graphical gate constraint>}*} ***set*** }

<service type heading> ::=
[<virtuality>]
service type {<service type name> | <service type identifier>}
[<formal context parameters>]
[<virtuality constraint>]
[<specialization>]

<service type reference>::=
<service type symbol> ***contains***
{[<virtuality>] <service type name>}

<service type symbol> ::=



Sémantique

Une définition <service type definition> définit un type de service. Tous les services d'un type de service donné ont les mêmes propriétés que celles qui sont définies pour ce type de service.

L'ensemble complet des signaux d'entrées valides d'un type de service est l'union de l'ensemble complet des signaux d'entrée valides de son supertype, des listes <signal list> au niveau de tous les accès en direction du type de service, l'ensemble <valid input signal set>, les signaux d'entrée implicites introduits par les concepts supplémentaires des paragraphes 4.10-4.14 et leurs signaux de temporisation.

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

6.1.2 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 6.3.

Grammaire textuelle concrète

<type expression>::=
<base type> [<actual context parameters>]

<base type>::=
<identifier>

Les paramètres <actual context parameters> ne peuvent être spécifiés que si et seulement si le type <base type> désigne un type paramétré. Les paramètres de contexte sont définis en 6.2.

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

Une expression <type expression> ne représente pas la spécialisation si le type <base type> est un type paramétré (voir 6.3).

NOTE – En dépit du fait que la définition désignée par le type <base type> satisfait à toutes conditions statiques, l'utilisation de l'expression <type expression> peut violer les propriétés associées au type <base type>. Cela peut se produire dans l'un des cas suivants:

1) 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.

2) 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.

3) 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 processus ou d'un service conduit à une spécification erronée si l'utilisation de tels identificateurs à l'intérieur de la procédure viole l'utilisation valide du processus ou du service.

4) Lorsqu'une action de création, une action de sortie ou une définition de vue à l'intérieur d'un type de processus ou de service défini dans un bloc fait référence à un ensemble d'instances de processus, la spécialisation et/ou l'instanciation du type de processus dans une sous-structure de bloc conduit à une spécification erronée.

5) Lorsque des types de services sont instanciés, le processus résultant est erroné si deux services ou plus ont le même signal dans l'ensemble complet des signaux d'entrée valides.

6) Lorsqu'une unité de portée a un paramètre de contexte de processus utilisé dans une action de sortie, l'existence d'un trajet de communication possible dépend du paramètre de contexte réel qui sera utilisé.

7) Lorsqu'une unité de portée a un paramètre de contexte de sorte et qu'un opérateur de la signature de sorte est utilisé dans les axiomes, l'application du paramètre de contexte de sorte réel pour lequel l'opérateur est défini en utilisant une définition d'opérateur conduit à une spécification erronée.

8) Si un paramètre formel d'une procédure ajouté dans une spécialisation a un genre <parameter kind> **in/out**, un appel à un sous-type dans un supertype (en utilisant **this**) conduira à un paramètre **in/out** réel omis, c'est-à-dire à une spécification erronée.

9) 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**, un appel au paramètre de contexte de procédure réel dans le type paramétré peut conduire à un paramètre **in/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 inséré 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> omis, les noms des paramètres <formal context parameter> sont insérés. Ces paramètres <actual context parameter> possèdent ainsi le contexte de définition de l'unité de portée courante.

6.1.3 Définitions fondées sur les types

Une définition de système, de bloc, de processus ou de service fondée sur le type définit respectivement un système, un bloc, un ensemble d'instances de processus ou un service conformément au type désigné par l'expression <type expression>. Les entités définies possèdent les propriétés des types sur lesquelles elles sont fondées.

6.1.3.1 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> : <system type expression>

Grammaire graphique concrète

<graphical typebased system definition>::=

<frame symbol> ***contains*** <typebased system heading>

Sémantique

Une définition de système fondée sur le type définit une définition *System-definition* dérivée d'un type de système par transformation.

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>.

6.1.3.2 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 block instances>] : <block type expression>

<number of block instances>::=
(<Natural simple expression>)

Si le nombre <number of block instances> est omis, le nombre de blocs est 1. Le nombre <number of block instances> doit être supérieur ou égal à 1.

Grammaire graphique concrète

<graphical typebased block definition>::=
<block symbol> ***contains***
{ <typebased block heading>
{ <gate>* }***set*** }

<existing typebased block definition>::=
<dashed block symbol> ***contains*** { <block identifier> { <gate>* }***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 une définition *Block-definitions* dérivée d'un type de bloc par transformation.

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 block telles qu'elles sont définies par l'expression <block type expression>. Le nombre des définitions <block definition> dérivées est spécifié par le nombre <number of block instances>.

Une définition <existing typebased 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 d'un bloc existant.

6.1.3.3 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 process instances>] : <process type expression>

Le type de processus désigné par le type <base type> dans l'expression <process type expression> doit contenir une transition de départ.

Grammaire graphique concrète

<graphical typebased process definition>::=
<process symbol> ***contains***
{ <typebased process heading>
{ <gate>* }***set*** }

<existing typebased process definition>::=
<dashed process symbol> ***contains*** { <process identifier> { <gate>* }***set*** }

<dashed process symbol>::=

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

Sémantique

Une définition de processus fondée sur le type définit une définition *Process-definition* dérivée d'un type de processus par transformation.

La création d'instances de processus individuelles est décrite en 2.4.4 (initialisation du système) et en 2.7.2 (demande de création dynamique).

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 <existing typebased 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 acheminements de signaux supplémentaires connectés aux accès d'un processus existant.

6.1.3.4 Définition de service fondée sur le type de service

Grammaire textuelle concrète

<textual typebased service definition>::=
service <typebased service heading> <end>

<typebased service heading> ::=
<service name> : <service type expression>

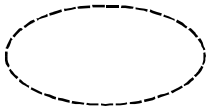
Le type de service désigné par le type <base type> dans l'expression <service type expression> doit contenir une transition de départ.

Grammaire graphique concrète

<graphical typebased service definition>::=
<service symbol> ***contains***
{ <typebased service heading>
{ <gate>* }***set*** }

<existing typebased service definition>::=
<dashed service symbol> ***contains*** { <service identifier> { <gate>* }***set*** }

<dashed service symbol> ::=



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

Sémantique

Une définition de service fondée sur le type définit une définition *Service-definition* dérivée d'un type de service par transformation.

Modèle

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

Une définition <existing typebased service definition> peut seulement apparaître dans une définition de sous-type. Elle représente le service défini dans le supertype de la définition du sous-type. Il est possible de spécifier des acheminements de signaux supplémentaires connectés aux accès du service existant.

6.1.4 Accès

Les accès sont définis dans les types de bloc, de processus ou de service (comme définis en 6.1.1) et représentent des points de connexion pour les canaux et les acheminements de signaux qui relient les instances de ces types (comme définies en 6.1.3) aux autres instances du même genre d'entité ou au symbole de cadre englobant.

Grammaire textuelle concrète

<gate definition>::=
gate <gate>
[**adding**] <gate constraint> <end>
[<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**]<identifier>

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

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

Un canal/acheminement de signal connecté à l'accès doit être compatible avec la contrainte de l'accès. Un canal/acheminement de signal est compatible avec la contrainte de l'accès si l'autre point extrémité du canal/acheminement de signal est un bloc/processus/service du type indiqué par l'identificateur <identifier> dans la contrainte du point extrémité ou un sous-type de ce type (dans le cas de **atleast**<identifier>) et si l'ensemble des signaux sur le canal/acheminement de signal est égal ou est inclus dans l'ensemble des signaux spécifié pour l'accès dans la direction respective.

Le type <base type> dans la définition <textual typebased block definition> ou <textual typebased process definition> doit contenir pour chaque combinaison (accès, signal, direction) définie par le type au moins un acheminement de signal mentionnant **env**, l'accès et le signal pour la direction donnée si le type de base contient des acheminements de signaux.

Une définition <block substructure definition> d'un type <base type> dans une définition <textual typebased block definition> doit contenir pour chaque combinaison (accès, signal, direction) définie par le type de bloc au moins un canal mentionnant **env**, l'accès et le signal pour la direction donnée.

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 <identifier> 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.

Grammaire graphique concrète

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

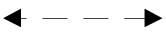
<endpoint constraint> ::=
{ <block symbol> | <process symbol> | <service symbol> }
contains <textual endpoint constraint>

<gate symbol> ::=
<signal route symbol>

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

<gate symbol 1> ::=

<gate symbol 2> ::=



La contrainte <graphical gate constraint> se trouve à l'extérieur du cadre de diagramme.

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

Le symbole figurant dans la contrainte <endpoint constraint> doit être le symbole d'une définition d'instance correspondante à la définition du type dans laquelle l'accès est défini, c'est-à-dire, <block symbol>, <process symbol> ou <service symbol>.

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

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

Il peut exister une zone <signal list area> pour chaque flèche sur le symbole <gate symbol>. Une zone <signal list area> doit être suffisamment et sans ambiguïté proche de la flèche qui lui est associée. 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

Pour chaque instance d'un type définissant un accès <gate>, une connexion <signal route to route connection> ou <channel to route connection> ou <channel connection> est dérivée:

a) pour chaque instanciation d'un type de processus, une connexion <signal route to route connection> est dérivée dans la définition <process definition> résultante où:

   – les identificateurs <external signal route identifiers> sont les acheminements de signaux définis dans le bloc englobant qui mentionne le processus et l'accès dans un trajet <signal route path>;

   – les identificateurs <signal route identifiers> sont les acheminements de signaux définis à l'intérieur de la définition <process definition> qui mentionne le mot clé **env** et l'accès dans le trajet <signal route path>.

b) pour chaque instanciation d'un type de bloc, une connexion <channel to route connection> est dérivée dans la définition <block definition> résultante où:

   – les identificateurs <channel identifiers> sont les canaux définis dans l'unité de portée englobant le bloc qui mentionne le bloc et l'accès dans un trajet <channel path>;

   – les identificateurs <signal route identifiers> sont les acheminements de signaux définis à l'intérieur de la définition <block definition> qui mentionne le mot clé **env** et l'accès dans le trajet <channel path>.

c) pour chaque instanciation d'un type de bloc contenant une définition <block substructure definition>, une connexion <channel connection> est dérivée dans l'unité de portée du bloc résultante où:

   – les identificateurs <channel identifiers> sont les canaux définis dans l'unité de portée englobant le bloc environnant qui mentionne le bloc et l'accès dans un trajet <channel path>;

   – les identificateurs <subchannel identifiers> sont les canaux définis à l'intérieur de la définition <block substructure definition> qui mentionne le mot clé **env** et l'accès dans un trajet <channel path>. Toute règle concernant l'utilisation de l'affinage du signal dans les instances de bloc fondées sur le type est définie à travers les règles de la connexion <channel connection> résultante (voir 3.3).

6.2 Paramètre 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 identificateurs réels, les paramètres de contexte réels, conformément en 6.1.2.

Les définitions de types suivantes peuvent avoir des paramètres de contexte formels: type de système, type de bloc, type de processus, type de service, procédure, signal et sorte.

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 possèderont ces propriétés.

Grammaire textuelle concrète

<formal context parameters>::=

<context parameters start> <formal context parameter>

{<end> <formal context parameter> }* <context parameters end>

<actual context parameters>::=

<context parameters start>

[<actual context parameter>] {, [<actual context parameter>] }* <context parameters end>

<actual context parameter> ::=
<identifier>

<context parameters start> ::=
<

<context parameters end> ::=
>

<formal context parameter>::=
<process 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>

NOTE – Les caractères «<» et «>» délimitent les paramètres de contexte et ne sont pas uniquement utilisés ci-dessus en tant que métasymboles.

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.

Les paramètres de contexte formels ne peuvent être utilisés ni comme type <base type> dans l'expression <type expression> ni dans des 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 6.1.2.

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

La liaison d'un paramètre de contexte synonyme ou variable réel avec sa définition n'est pas résolue par le contexte.

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> ou d'une règle <inheritance rule> 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 qualificateurs, voir 2.2.2), 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 formel omis dans la spécialisation 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.

6.2.1 Paramètre de contexte de processus

Grammaire textuelle concrète

<process context parameter>::=
process <process name> <process constraint>

<process constraint>::=
[**atleast**] <process identifier> | <process signature>

<process signature> ::=
[[<end>] <formal parameters signature>]

<formal parameters signature>::=
fpar <sort>{**,** <sort> }*

Sémantique

Un paramètre de processus réel doit identifier une définition de processus. Son type doit être un sous-type du type de processus de contrainte (**atleast** <process identifier>) sans adjonction de paramètres formels aux paramètres du type de contrainte, ou il doit être le type désigné par l'identificateur <process identifier> ou il doit être compatible avec la signature de processus formel. Une définition de processus est compatible avec la signature de processus formel si les paramètres formels de la définition du processus ont les mêmes sortes que les sortes <sort> correspondantes de la signature <formal parameters signature>.

6.2.2 Paramètre de contexte de procédure

Grammaire textuelle concrète

<procedure context parameter>::=
procedure <procedure name> <procedure constraint>

<procedure constraint>::=
atleast <procedure identifier>
| <procedure signature>

<procedure signature>::=
[[<end>] **fpar** <procedure formal parameter constraint>
{**,** <procedure formal parameter constraint> }*
[<end> **returns** <sort>]]
| [<end>] **returns** <sort>

<procedure formal parameter constraint> ::=
<parameter kind> <sort>

Sémantique

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 est compatible avec la signature de procédure formelle.

a) si 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 renvoient tous une valeur de la même sorte <sort> ou ne renvoient aucune valeur; ou

b) si cette règle est satisfaite après substitution d'une spécification de résultat dans un paramètre **in/out** supplémentaire; et

c) si chaque paramètre **in/out** de la définition de procédure a le même identificateur <sort identifier> ou <syntype identifier> que le paramètre correspondant de la signature.

6.2.3 Paramètre de contexte de procédure distante

Grammaire textuelle concrète

<remote procedure context parameter>::=
remote procedure <procedure name> <procedure signature>

Sémantique

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

6.2.4 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>]
[<signal refinement>]

Sémantique

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.

Une définition est compatible avec la signature de signal formel si les sortes du signal sont les mêmes que les sortes de la liste de contraintes de sorte et si, dans le cas où l'affinage <signal refinement> du paramètre <signal context parameter> est énoncé, les définitions du signal de l'affinage <signal refinement> sont compatibles avec l'affinage <signal refinement> du paramètre <signal context parameter>.

Deux affinages <signal refinement> sont compatibles s'ils définissent le même ensemble de noms de signaux et si chaque définition <subsignal definition> correspondante a le même attribut <reverse>.

6.2.5 Paramètre de contexte de variable

Grammaire textuelle concrète

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

Sémantique

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

6.2.6 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> }*

Sémantique

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

6.2.7 Paramètre de contexte de temporisateur

Grammaire textuelle concrète

<timer context parameter>::=
timer <timer name> <timer constraint>
{, <timer name> <timer constraint> }*

<timer constraint> ::=
[<sort list>]

Sémantique

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.

6.2.8 Paramètre de contexte de synonyme

Grammaire textuelle concrète

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

<synonym constraint> ::=
<sort>

Sémantique

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

6.2.9 Paramètre de contexte de sorte

Grammaire textuelle concrète

<sort context parameter>::=
newtype <sort name> [<sort constraint>]

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

<sort signature>::=
<operators> **endnewtype** [<sort name>]

Sémantique

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 <literal renaming> ou une liste <inheritance list> 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 opérateurs de la sorte comprennent les opérateurs dans la signature de sorte formelle et les opérateurs ont la même signature.

Une signature <sort signature> contient implicitement les opérateurs égal et différent (voir 5.3.1.4). **noequality** n'est pas autorisé dans la signature <sort signature>.

ordering dans la signature <sort signature> donne la signature pour les opérateurs d'ordre comme définis au § 5.3.1.8.

6.3 Spécialisation

Afin d'exprimer la spécialisation de concept, un type peut être défini comme une spécialisation d'un autre type (le supertype) conduisant ainsi à un nouveau type. Un sous-type peut avoir des propriétés supplémentaires aux propriétés du supertype et peut redéfinir des types et transitions locaux virtuels.

La spécialisation dans une définition de type spécialisé est spécifiée par «**inherits** <type expression>», où <type expression> désigne le type général. Ce dernier 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.

Il convient de noter que l'expression <type expression> dans sa totalité représente le supertype. Seulement si l'expression <type expression> contient uniquement <base type>, le supertype est appelé.

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 6.3.2.

6.3.1 Adjonction de propriétés

Grammaire textuelle concrète

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

Si un type subT est dérivé 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 dérivé de subT;

c) les définitions englobées par T ne doivent pas être dérivées de subT.

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 2.2.2. Il existe toutefois trois exceptions à ces règles; il s'agit des cas suivants:

1) une redéfinition d'un type virtuel est une définition avec le même nom que celui du type virtuel;

2) 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;

3) 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 supertype.

La définition <block substructure definition> donnée dans une définition de type de bloc spécialisé est ajoutée à la définition de sous-structure du supertype de bloc. Le nom de la sous-structure du sous-type s'il est présent doit être le même que le nom de la sous-structure du supertype.

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 6.2).

Les paramètres de contexte formels d'un type de processus spécialisé ou d'une procédure spécialisée sont les paramètres de contexte formels du supertype de processus ou de la procédure, suivis par les paramètres formels ajoutés dans la spécialisation.

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

Le graphe résultant d'un type de processus, d'un type de service ou d'une définition de procédure après spécialisation se compose du graphe de la définition de son supertype suivi par le graphe du type de processus, du type de service ou de la définition de processus spécialisé(e).

Le graphe de processus d'un type de processus, d'un type de service ou d'une définition de procédure donné(e) doit avoir au plus une transition de départ.

Tous les noms <connector name> définis dans le corps <body> combiné doivent être distincts. Il est permis d'avoir un branchement à partir du corps <body> du processus/de la procédure/du service spécialisé(e) vers un connecteur défini dans le supertype.

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

Une définition de type spécialisé partielle peut ajouter des propriétés en termes d'opérateurs, de littéraux, d'axiomes, de définitions d'opérateurs et d'assignation par défaut.

NOTE – Lorsqu'un accès dans un sous-type est une extension d'un accès existant dans un supertype, le symbole <existing gate symbol> est utilisé dans le SDL/GR.

6.3.2 Type virtuel

Un type défini localement dans une définition de type (type *enclosing*) peut être spécifié comme étant un type virtuel. Les types de blocs, les types de processus, les types de services et les procédures peuvent être spécifiés en tant que types virtuels. Un type virtuel peut être contraint par un autre type du même genre d'entité et peut être redéfini dans des sous-types de son type englobant.

Grammaire textuelle concrète

<virtuality>::=

virtual | redefined | finalized

<virtuality constraint>::=

atleast <identifier>

Les règles syntaxiques des types virtuels sont introduites aux 6.1.1.2 (type de bloc), 6.1.1.3 (type de processus), 6.1.1.4 (type de service) et 2.4.6 (procédure).

Un type virtuel et sa contrainte ne peuvent pas avoir de paramètres de contexte.

Un type virtuel peut être contraint par le type identifié par l'identificateur <identifier> suivant le mot clé **atleast**. Cet identificateur doit identifier une définition de type du même genre d'entité que le type virtuel. Si la contrainte <virtuality constraint> est omise, cela correspond à spécifier le type virtuel lui-même en tant que contrainte.

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 de procédure référencée, les deux doivent être égaux.

Un type virtuel doit avoir <virtuality> dans sa définition.

Un type virtuel doit avoir les mêmes paramètres formels, les mêmes accès et signaux sur les accès que sa 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 uniquement, désignant le supertype particulier, ou dans le cas d'accès à un type caché à partir d'une sous-structure du supertype, de deux éléments de trajet, le premier désignant le type de bloc et le second la sous-structure.

Aussi bien dans la définition du type englobant que dans tout sous-type du type englobant, la définition du type virtuel doit être une spécialisation de sa contrainte. Dans une spécialisation du type englobant, une nouvelle définition du type virtuel est donnée par une définition de type avec le même nom et avec le mot clé **redefined**. Un type virtuel redéfini reste un type virtuel. Le mot clé **finalized** au lieu de **redefined** indique que le type n'est pas virtuel et peut alors avoir de nouvelles définitions dans les redéfinitions de sous-type supplémentaires.

Un type virtuel avec une contrainte explicite mais sans héritage explicite hérite implicitement du type de contrainte.

Un type redéfini ou finalisé sans contrainte explicite et sans héritage explicite hérite implicitement de la contrainte du type virtuel correspondant.

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

6.3.3 Transition/sauvegarde virtuelle

Le présent paragraphe décrit les départs virtuels, les entrées virtuelles et les sauvegardes virtuelles mentionnés dans d'autres paragraphes.

Les transitions ou les sauvegardes d'un type de processus, d'un type de service 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 concrète

Les syntaxes des transitions et sauvegardes virtuelles sont définies aux 2.4.6 (départ de procédure virtuelle), 2.6.2 (départ de processus virtuel), 2.6.4 (entrée virtuelle), 2.6.5 (sauvegarde virtuelle), 2.6.6 (transition spontanée virtuelle), 4.10 (entrée prioritaire virtuelle), 4.11 (signal continu virtuel) et 4.14 (entrée et sauvegarde de procédure distante virtuelle).

Les transitions ou sauvegardes virtuelles ne doivent pas apparaître dans les définitions (d'ensemble d'instances) de processus ou dans les définitions (d'instances) de service.

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 6.3.2).

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 continue virtuelle peut être redéfinie en une nouvelle transition continue. La redéfinition est indiquée par le même couple (état, [priorité]) 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'étape 14 du 7.1.

6.4 Exemples

La Figure 6.4.1 représente un progiciel «lib». La Figure 6.4.2 représente un type de bloc (englobé dans le progiciel) qui utilise ce progiciel «lib» et un autre progiciel qui «slib» qui est référencé. La Figure 6.4.3 représente un diagramme de système avec une clause **use**. Les Figures 6.4.4 et 6.4.5 représentent un type de bloc avec accès g1 et g2.

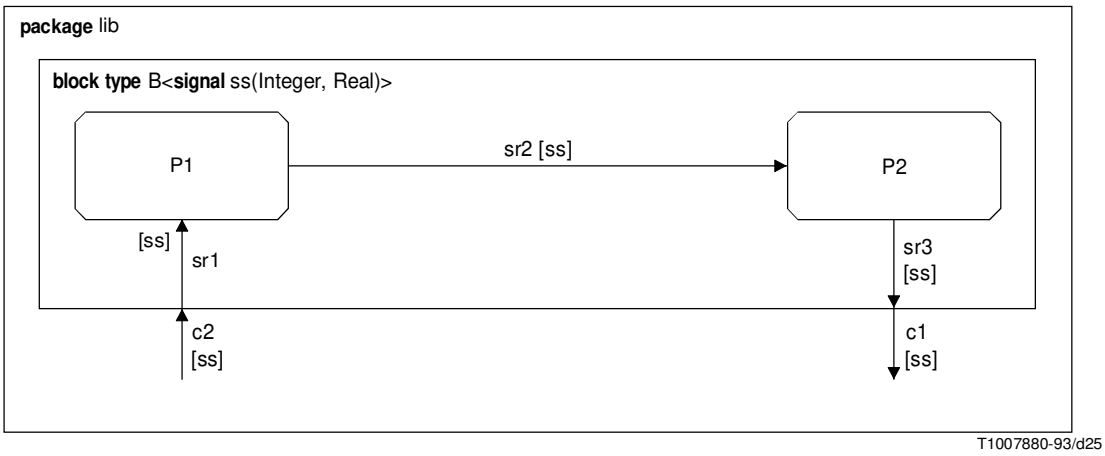


FIGURE 6.4.1/Z.100

Diagramme de progiciel (SDL/GR)

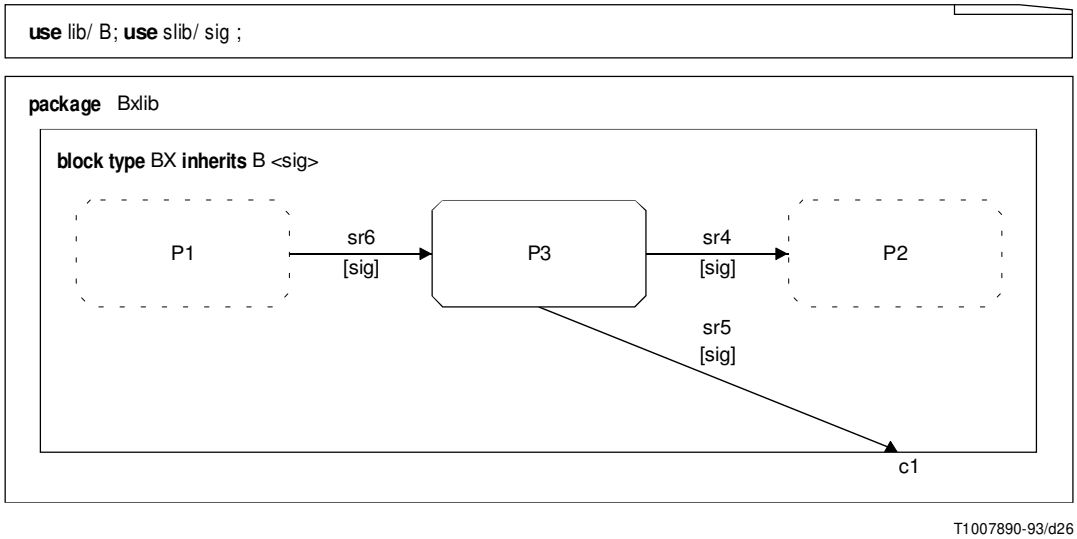


FIGURE 6.4.2/Z.100

Diagramme de progiciel avec une clause use (SDL/GR)

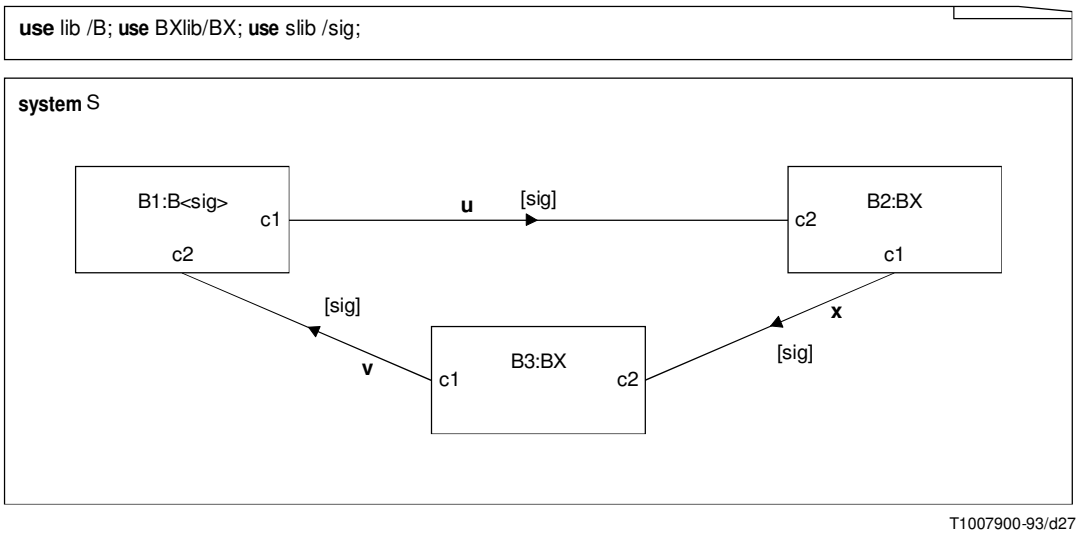


FIGURE 6.4.3/Z.100

Diagramme de système avec une clause use (SDL/GR)

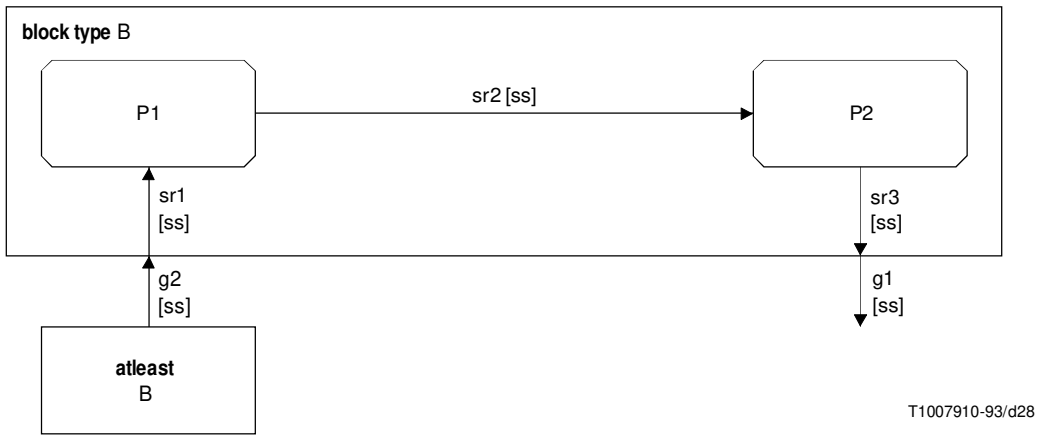


FIGURE 6.4.4/Z.100

Type de bloc avec accès g1 et g2 (SDL/GR)

```
block type B;
     gate  g1 out with ss;
     gate  g2 in from atleast B with  ss;
     signalroute sr2 from p1 to p2 with ss;
     signalroute sr1 from env via g2 to p1 with ss;
     signalroute sr3 from p2 to env via g1 with ss;
     process p1 referenced;
     process p2 referenced;
endblock type B;
```

FIGURE 6.4.5/Z.100

Type de bloc avec accès g1 et g2 (même type que celui de la Figure 6.4.4) (SDL/PR)

La Figure 6.4.6 représente en SDL/GR un type de bloc BX qui est un sous-type de B avec une définition de processus supplémentaire, et avec des références aux ensembles de processus existants (P1 et P3) afin de leur connecter le processus supplémentaire. La Figure 6.4.7 représente le même exemple en SDL/PR.

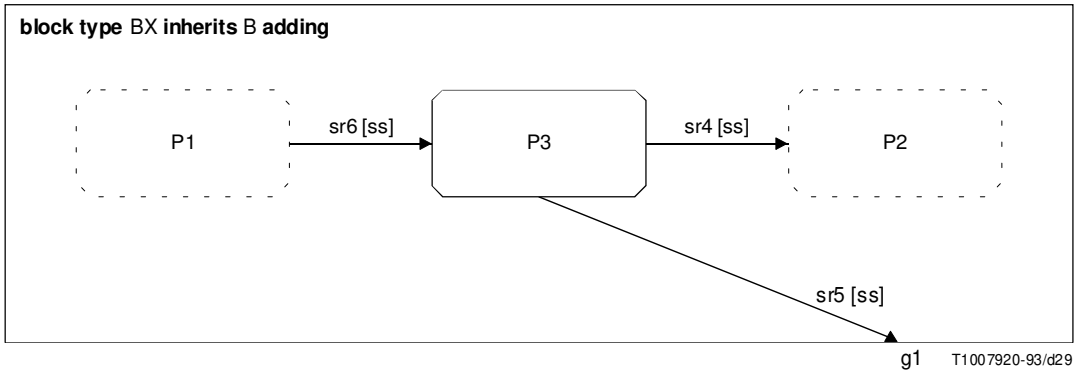


FIGURE 6.4.6/Z.100

Définition d'un sous-type (SDL/PR)

```
block type BX inherits B adding ;
     signalroute sr4 from p3 to p2 with ss;
     signalroute sr5 from p3 to env via g1 with ss;
     signalroute sr6 from p1 to p3 with ss;
process p3 referenced;
endblock type BX;
```

FIGURE 6.4.7/Z.100

Définition d'un sous-type en SDL/PR

Les Figures 6.4.8 et 6.4.9 montrent l'utilisation des types de bloc B et BX pour spécifier des instances de bloc et des connexions aux accès.

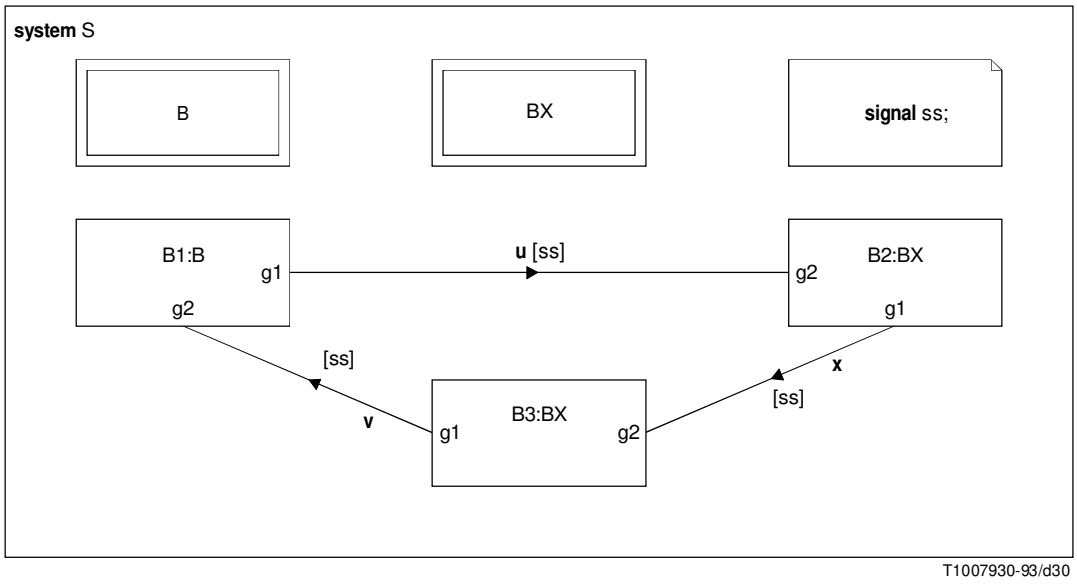


FIGURE 6.4.8/Z.100

Définition d'un système utilisant les types de bloc (SDL/GR)

```
system  S;
     signal ss;
     block type B referenced;
     block type BX referenced;

     block B1:B;
     block B2:BX;
     block B3:BX;

     channel u  from B1 via g1 to B2 via g2 with ss endchannel;
     channel v  from B3 via g1 to B1 via g2 with ss endchannel;
     channel x  from B2 via g1 to B3 via g2 with ss endchannel;

endsystem S;
```

FIGURE 6.4.9/Z.100

Définition d'un système utilisant les types de bloc (SDL/PR)

Des exemples de types virtuels sont donnés dans la Figure 6.4.10 ci-dessous:

```
procedure Proc; fpar i,j Integer ... endprocedure;

process type P
      virtual  procedure VProc1 atleast Proc
        inherits Proc ... endprocedure;

      virtual procedure VProc2 ... endprocedure;

     ...  call VProc1(1,2)  ;   call VProc2; ...
endprocess type ;

process type  P1 inherits P;

     redefined  procedure VProc1  atleast VProc1
        inherits << process type P >> VProc1;
          ...
     endprocedure;

     finalized procedure VProc2 ... endprocedure;
  ...
endprocess type ;

process type  P2 inherits P1;

     finalized procedure VProc1
        inherits << process type P1 >> VProc1;
          ...
     endprocedure;
  ...
endprocess type ;
```

FIGURE 6.4.10/Z.100

Exemples de types virtuels (SDL/PR)

Dans P, la procédure VProc1 est virtuelle et doit être une spécialisation de Proc, et la définition de VProc1 hérite de Proc. Si un processus P est généré, ou si un processus spécialisé ne fournit pas de définition de VProc1, cette définition s'applique. VProc2 est aussi virtuelle dans P avec une contrainte implicite qui est elle-même, c'est-à-dire que les redéfinitions de VProc2 dans les sous-types de P doivent être des sous-types de VProc2.

Dans P1, la procédure VProc1 reste virtuelle et contrainte par elle-même, alors que la procédure VProc2 n'est pas virtuelle. Les définitions s'appliquent dans le cas où un processus P1 est généré pour les appels effectués dans la définition de P. La définition de VProc1 est une spécialisation de VProc1 définie dans le processus P, et a par suite un qualificateur (**process type** P).

Dans P2, une nouvelle définition de VProc1 est donnée. Cela est possible étant donné que VProc1 est virtuelle dans P1. Une redéfinition de VProc2 n'est pas autorisée ici car elle a été finalisée dans P1.

Les Figures 6.4.11 à 6.4.16 représentent des exemples de types correspondant à l'exemple représenté par les Figures 2.10.5 à 2.10.10.

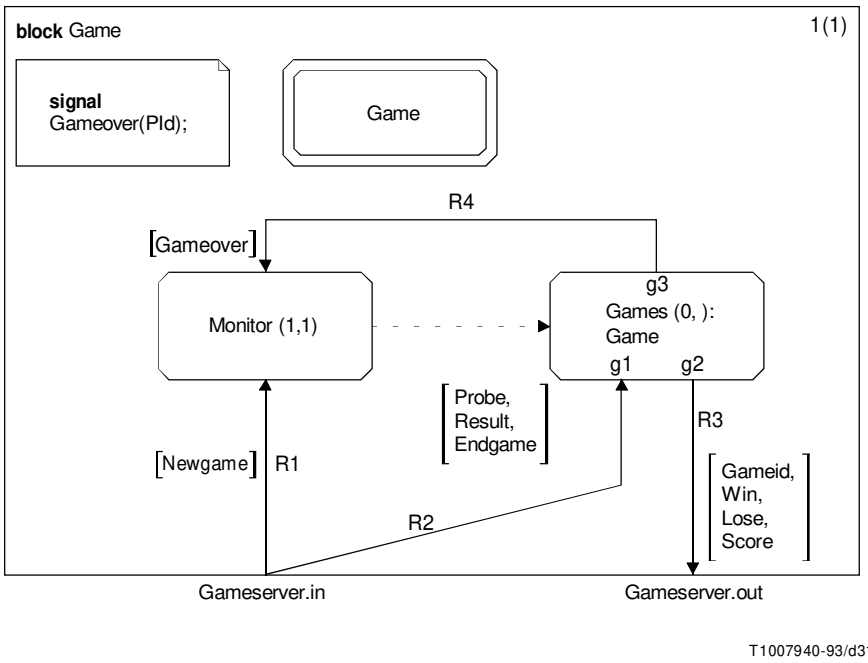


FIGURE 6.4.11/Z.100

Bloc avec type de processus (SDL/GR)

```
block Game;
      signal Gameover(PId);
      connect Gameserver.in and R1,R2;
      connect Gameserver.out and R3;
      signalroute R1 from env to Monitor with Newgame;
      signalroute R2 from env to Games via g1 with Probe, Result, Endgame;
      signalroute R3 from Games via g2 to env
         with Gameid, Win, Lose, Score;
      signalroute R4  from Games via g3 to Monitor with Gameover;

      process type Game referenced;

      process Monitor (1,1) referenced;

      process Games (0,) : Game referenced;

endblock Game;
```

FIGURE 6.4.12/Z.100

Bloc avec type de processus (SDL/PR)

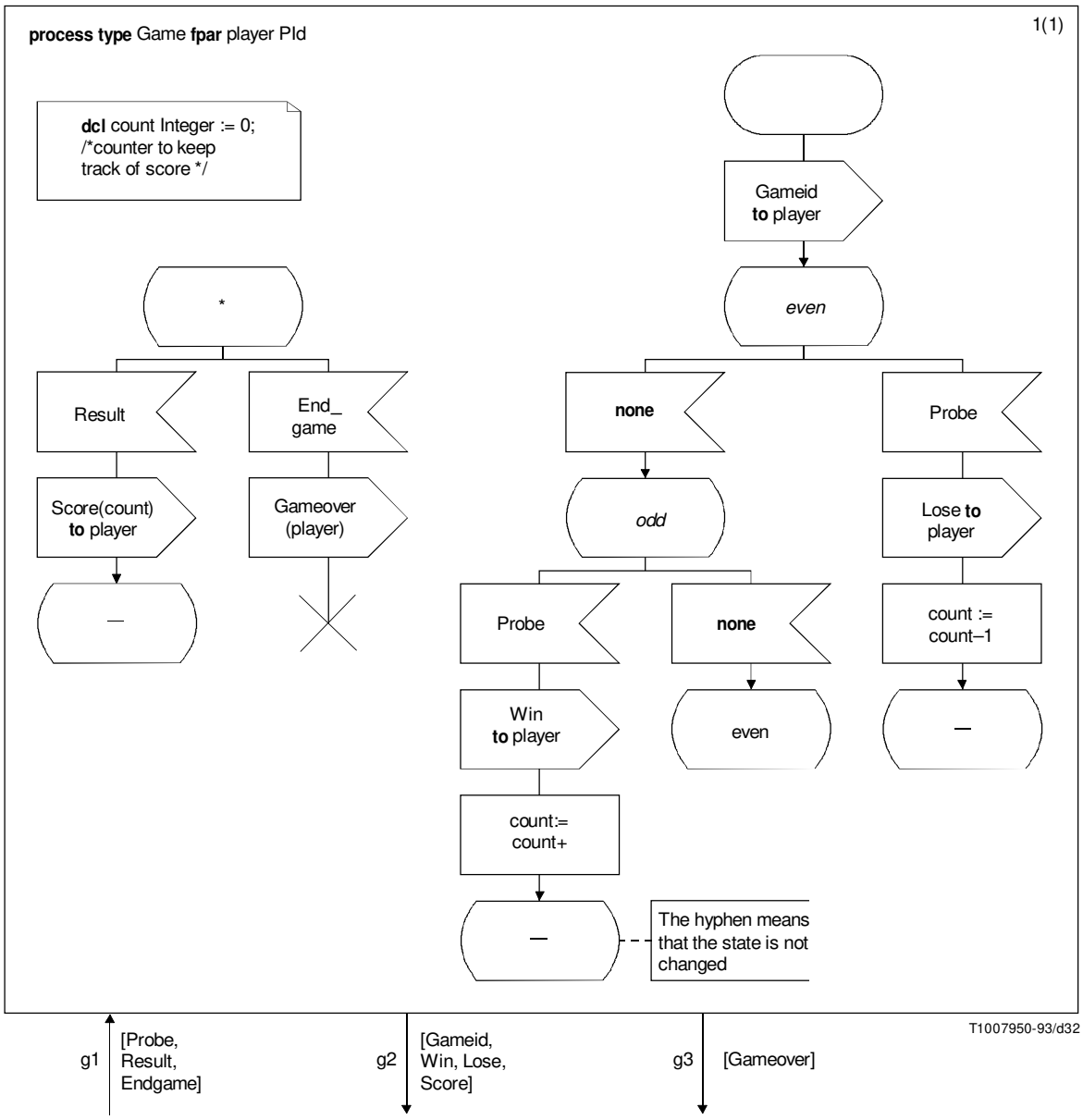


FIGURE 6.4.13/Z.100

Processus de jeu (Game) en tant que type de processus (SDL/GR)

```
process type  Game;
   fpar player Pid;

   gate   g1 in with Probe, Result, Endgame;
          g2 out with Gameid, Win, Lose, Score;
          g3 out with Gameover;
   dcl count Integer:=0;

   start;
output Gameid to player;
   nextstate Even;
   state Even;
    input none;
      nextstate Odd;
    input Probe;
      output Lose to player;
      task count:= count - 1;
      nextstate -;

   state Odd;
    input none;
      nextstate Even;
    input Probe;
      output Win to Player;
      task count:= count + 1;
      nextstate -;

   state *;
    input Result;
      output Score(count) to player;
       nextstate -;

    input Endgame;
      output Gameover(player);
      stop;

endprocess type Game;
```

FIGURE 6.4.14/Z.100

Processus de jeu (Game) en tant que type de processus (SDL/PR)

Les Figures 6.4.15 et 6.4.16 représentent une spécialisation SpecialGame du type de processus Game, en tenant compte d'un nouveau signal Evil d'un nouveau processus Devil. Dans tout état des états de Game, la réception de ce signal conduit à l'état Even, donnant au joueur une plus grande chance de perdre. Il convient de noter que la partie Game-part continue à se comporter exactement comme un processus Game. Si le processus Devil n'envoie aucun signal Evil, le comportement est le même que dans une instance du supertype Game.

Il convient de noter que l'extension du type de processus Game nécessite les extensions du bloc englobant les instances du type de processus spécialisé. Evil est supposé arriver sur le même accès que Probe, Result et Endgame.

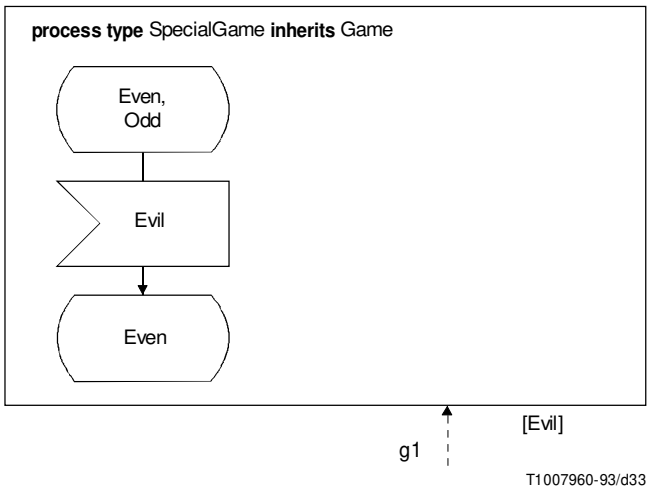


FIGURE 6.4.15/Z.100

Spécialisation du processus Game (SDL/GR)

```
process type SpecialGame inherits Game;

gate g1 adding in with Evil;

      state Even, Odd;
         input Evil;
         nextstate Even;

endprocess type SpecialGame;
```

FIGURE 6.4.16/Z.100

Spécialisation du processus Game (SDL/PR)

La Figure 6.4.17 représente des exemples de paramètres de contexte. Dans le cas a), la sorte réelle doit être une spécialisation de la sorte t (visible en ce point). Dans le cas b), le premier paramètre réel est un sous-type de t et le second paramètre réel est une sorte avec un opérateur acceptant deux valeurs du premier paramètre réel et retournant une valeur du second paramètre. Dans le cas c), le premier paramètre réel doit être un sous-type de SuperProc tandis que le second paramètre réel peut être une procédure qui admet deux paramètres (**in**) des sortes Integer et Boolean.

```
signal S<newtype t1 atleast t>  (t1);      /* case a) */

procedure P<newtype t1 atleast t;       /* case b) */
                    newtype t2 operators op: t1,t1 -> t2 endnewtype>;
      dcl x11,x12 t1, x2 t2;
           start;
             task x2:=op(x11,x12);
             return;
endprocedure;

procedure P                              /* case c) */
        < procedure Proc1  atleast SuperProc;
            procedure Proc2  fpar Integer, Boolean >;
          start;
            call Proc1;
            call Proc2(7, True);
            return;
endprocedure;
```

FIGURE 6.4.17/Z.100

Exemples de paramètres de contexte (SDL/PR)

7 Transformation des abréviations SDL

Le présent article explique en détail la transformation des constructions SDL dont la sémantique dynamique est donnée après une transformation du sous-ensemble SDL pour lequel il existe une *Grammaire abstraite*. Ces notations abrégées sont:

a) les constructions des articles 2, 3 et 5 pour lesquelles il existe une section *Modèle*; et

b) les constructions définies aux articles 4 et 6.

Les propriétés d'une notation abrégée sont dérivées de la manière dont elle est modélisée en termes de concepts primitifs (ou transformée en ces concepts). Afin d'assurer une utilisation facile et non ambiguë des notations abrégées, et pour réduire les effets indésirables lors de la combinaison de plusieurs notations abrégées, ces concepts sont transformés dans un ordre spécifié, comme défini dans le présent article.

L'ordre spécifié de transformation signifie que dans la transformation d'une notation abrégée d'ordre *n* on peut utiliser une autre notation abrégée d'ordre *m*, à condition que *m* soit supérieur à *n* (m>n).

Etant donné qu'il n'existe pas de syntaxe abstraite pour les abréviations, les termes de la syntaxe graphique aussi bien que de la syntaxe textuelle sont utilisés dans leurs définitions. Le choix entre les termes de syntaxe graphique et textuelle est fondé sur des considérations pratiques et ne limite pas l'utilisation des abréviations à une syntaxe concrète particulière.

Les transformations sont décrites comme un nombre d'étapes énumérées. Une étape peut décrire la transformation de plusieurs concepts et se composer donc de plusieurs sous-étapes, soit en raison du fait que ces concepts doivent être transformés comme un groupe, ou en raison du fait que l'ordre de transformation entre ces concepts n'a pas d'importance. Ce dernier cas est indiqué par un tiret (–) plutôt que par énumération.

7.1 Transformation de concepts supplémentaires

1. Transformations lexiques

   1) les définitions <macro definition> et les appels <macro call> (4.2) sont identifiés lexicalement et les appels <macro call> sont étendus;

   2) les diagrammes <macro diagram> sont remplacés par des occurrences de zones <macro call area>;

   3) le soulignement <underline> suivi par des caractères de séparation sont supprimés des noms et les séparateurs dans les noms sont remplacés par un soulignement <underline> (2.2.1). Après cette transformation, la spécification <sdl specification> est syntaxiquement considérée comme bien formée;

   4) les définitions <macro definition> sont supprimées (dans les définitions <package definition> également).

2. Les références de définition sont remplacées par des définitions <referenced definition> (2.4.1.3).

3. Les graphes sont normalisés, c'est-à-dire que:

   1) les décisions non terminales et les options de transition non terminale sont transformées en décisions terminales et options de transition terminale respectivement;

   2) les actions et/ou instructions terminales suivant les décisions et les options de transitions sont déplacées et apparaissent comme des actions <free action>. Ces actions <free action> générées qui n'ont pas d'étiquette attachée se voient attribuer des étiquettes anonymes;

   3) les listes d'action (y compris l'instruction terminale qui suit) dans lesquelles la première action (s'il y en a, et sinon l'instruction terminale suivante) a une étiquette attachée, sont remplacées par un branchement à l'étiquette et la liste d'actions apparaît comme une action <free action>.

4. Le progiciel Predefined est inclus dans la liste <package list>.

5. Transformation de système générique (4.3) et de données étendues (5.4.6):

   1) les identificateurs dans les expressions <simple expression> contenues dans la spécification <sdl specification> sont limités aux définitions. Pendant la liaison, seules les définitions <data definition> définies dans le progiciel Predefined et les définitions <external synonym definition> sont considérées (c'est-à-dire que toutes les autres définitions <data definition> sont ignorées);

2) les synonymes <external synonym> sont remplacés par des définitions <synonym definition> et le texte informel dans les options de transition est remplacé par une condition <range condition>. Le SDL ne définit pas la manière d'effectuer ces remplacements;

3) les expressions <simple expression> sont évaluées et les définitions <select definition>, les zones <option area>, les options <transition option> et les zones <transition option area> sont supprimées. Les transitions <transition> qui ne sont pas sélectionnées dans une option <transition option> ou dans une zone <transition option area> se voient attribuer des étiquettes anonymes et apparaissent alors comme des actions <free action> ou des zones <in-connector area> respectivement.

   Les valeurs du nombre <number of process instances> sont représentées par des identificateurs littéraux entiers totalement qualifiés;

4) les propriétés <external properties> sont remplacées par des axiomes et du texte informel. La manière d'effectuer ces remplacements n'entre pas dans le cadre du SDL.

6. Inclusion de progiciel (2.4.1.2)

Les progiciels <package> qui ne sont pas mentionnés dans une clause <package reference clause> mais qui ne font pas partie de la liste <package list> sont inclus dans la liste <package list>. Le SDL ne définit pas la manière d'effectuer cette inclusion.

Si la spécification <sdl specification> ne contient pas de définition <system definition> aucune étape supplémentaire n'est appliquée, mais les propriétés des progiciels garantissent que des étapes supplémentaires puissent être appliquées pratiquement.

7. Transformation

– des assignations multiples dans les corps <task body> (2.7.1);
– des décisions non déterministes (2.7.5);
– des opérateurs <infix operator> et transformation de leurs opérandes en forme de préfixe (5.4.1.1);
– des résultats <procedure result> (2.4.6);
– de la structure de sorte (5.3.1.10);
– de la liste d'états (2.6.3);
– de la liste de stimulus (2.6.4);
– du primaire de champ (5.3.2.5);
– du primaire de structure (5.3.2.6);
– des axiomes booléennes (5.3.1.5);
– des définitions <syntype definition> avec **newtype** (5.3.1.9);
– des signaux multiples dans les corps <output body> (2.7.4);
– des temporisateurs multiples dans <set> et <reset> (2.8).

8. Des qualificateurs complets sont insérés.

Conformément aux règles de visibilité et des règles de résolution par contexte (2.2.2), les qualificateurs sont étendus pour désigner des trajets complets.

NOTE – Comme certains concepts supplémentaires (par exemple, les transformations de générateur et la spécialisation) ont un impact sur l'ensemble de définitions lié à une unité de portée, les qualificateurs sont dérivés après suppression de tels concepts de l'unité de portée particulière.

Pour chaque unité de portée:

– Paramètres de contexte (6.2);
– Spécialisation (6.3);
– Liste de signaux (2.5.5);
– Ordre (5.3.1.8);
– Générateurs (5.3.1.12);
– Primaire indexé (5.3.3.4);
– Variable de champ (5.4.3.2);
– Variable indexée (5.4.3.1);
– Entrée de champs (2.6.4);
– Diagramme de sous-structure de bloc ouvert (3.2.2);
– Valeur de durée par défaut pour l'initialisation de temporisateur (2.8);
– Initialisation des variables de sortes avec initialisation par défaut (5.4.3.3);

sont transformés, après quoi les qualificateurs sont insérés. Ensuite, les transformations sont appliquées aux unités de portée englobées dans l'unité de portée.

Cette étape est définie individuellement au 7.2 ci-dessous.

9. Spécialisation implicite des procédures globales (2.7.3)

Si une occurrence d'un identificateur <procedure identifier> fait référence à une procédure qui n'est pas englobée par un processus ou par un service, elle entraîne la création d'une copie locale au processus et le changement de l'identificateur <procedure identifier> pour désigner la copie. Cette étape est répétée jusqu'à ce qu'il n'y ait plus de références à des procédures non locales à l'intérieur des processus et services.

Les définitions <operator definition> sont transformées en procédures ayant des noms anonymes et le résultat comme paramètre **in/out**. La définition <procedure définition> résultante est déplacée vers l'unité de portée englobante, les signatures des opérateurs pour chaque opérateur sont supprimées et les applications des opérateurs sont transformées en appels <value returning procedure call>.

10. Transformations:

– Les définitions <textual typebased system definition>, <graphical typebased system definition>, <textual typebased block definition>, <graphical typebased block definition>, <textual typebased process definition>, <graphical typebased process definition>, <textual typebased service definition> et <graphical typebased service definition> sont remplacées par <system definition>, <system diagram>, <block definition>, <block diagram>, <process definition>, <process diagram>, <service definition> et <service diagram> respectivement, en copiant les contenus du type désigné par le type <base type>. Dans cette copie, les paramètres <formal context parameter> sont remplacés par les identificateurs <identifier> dans les paramètres <actual context parameters> de la même manière que celle qui est décrite dans la sous-étape 2.2 de l'étape 8 (voir plus bas).

– Le mot clé **this** est remplacé par un identificateur de procédure.

– Les occurrences de noms de types à l'intérieur des qualificateurs sont remplacées par les noms d'instances respectifs, et les occurrences de **this** là où il désigne un identificateur <process identifier> sont remplacées par l'identificateur désignant la définition <process definition> englobante ou le diagramme <process diagram> englobant.

– Les connexions <channel connection>, <channel to route connection>, <signal route to route connection>, les ensembles complets de signaux et les acheminements de signaux omis (s'il y en a) sont dérivés des accès <gate> rencontrés dans les canaux et les acheminements de signaux connectés à l'instance.

  Ces transformations s'appliquent à la fois aux définitions <process definition> et <service definition>.

– Les accès <gate> dans les corps <output body> sont remplacés par les canaux et acheminements de signaux qui sont connectés à l'instance et qui acheminent le signal et mentionnent l'accès.

– Les accès <gate> des canaux et acheminements de signaux sont supprimés.

– Les ensembles de blocs sont supprimés et **via** est modifié:

  Chaque définition de canal ayant un ensemble de blocs comme point d'extrémité est remplacée par un nombre de canaux, un pour chaque bloc de l'ensemble. Chaque nouveau canal est connecté à une copie de la définition de bloc définie par l'ensemble de blocs. La copie a un nouveau nom distinct.

  Pour **via** et les connexions contenues à l'intérieur d'une copie de bloc, une occurrence de l'identificateur de canal d'origine est remplacée par l'identificateur de canal reliant le bloc membre particulier.

  Pour **via** et les connexions contenues à l'intérieur d'un bloc connecté à un ensemble de blocs, toutes les occurrences de l'identificateur de canal d'origine sont remplacées par la liste des identificateurs de canaux qui résulte de l'extension de l'ensemble de blocs.

  Si deux points d'extrémité du canal d'origine désignent le même identificateur <block identifier>, le nombre de canaux résultant est égal au carré du nombre des instances de blocs moins le nombre des instances de blocs.

11. Suppression

    1) Des types ayant des paramètres <formal context parameter>, des types de systèmes, des types de bloc, des types de processus, des types de service, des occurrences de <specialization> et de <inheritance rule>.

    2) Des procédures définies à l'extérieur des processus ou services et de <virtuality> pour les procédures restantes.

12. Les définitions à l'intérieur des progiciels (2.4.1.2) sont déplacées au niveau du système.

    Chaque nom défini à l'intérieur d'un progiciel sera changé en nom anonyme et les qualificateurs sont modifiés pour indiquer le niveau du système.

    La liste des progiciels est supprimée.

13. Transformation de la sous-structure de canal (3.2.3)

14. Suppression des états astérisques, des entrées astérisques et des sauvegardes astérisques:

    – Un corps émanant d'une définition de processus, d'une définition de service ou d'une définition de procédure sans spécialisation voit ses états astérisques étendus conformément au modèle défini au 4.4, ensuite ses entrées astérisques étendues conformément au modèle défini au 4.6 et enfin ses sauvegardes astérisques étendues conformément au modèle défini au 4.7.

    – Un corps qui est formé en combinant deux ou plusieurs corps de type (par spécialisation) conserve l'information concernant le corps de type duquel ses états sont issus. Cela signifie qu'il peut être considéré comme composé d'une liste de corps de type dans laquelle le premier corps de type est issu du type n'ayant pas de spécialisation et le corps de type N (N>1) de la liste est issu du corps de type de processus qui est une spécialisation du corps de type N-1.

    – Le corps combiné est formé en appliquant les étapes suivantes:

        1) inclure tous les états de tous les corps, mais exclure les transitions et sauvegardes associées qui sont redéfinies ou finalisées;

        2) étendre l'état astérisque de la même manière que pour un corps issu d'une définition sans spécialisation;

        3) étendre l'entrée astérisque et la sauvegarde astérisque de la même manière que pour un corps issu d'une définition sans spécialisation;

        4) remplacer les transitions et sauvegardes virtuelles et redéfinies par les transitions et sauvegardes redéfinies et finalisées pour les types, un par un, selon l'ordre ascendant de N. Si un état est un état astérisque, le remplacement s'applique à tous les états dans le corps combiné.

    – Les apparitions multiples d'un état sont fusionnées (4.5).

15. Les transitions implicites sont insérées (4.8).

16. **via all** est changé en une liste de sortie (2.7.4).

17. Les actions <free action> (2.6.7) sont supprimées:

    1) Tout branchement <join> ou zone <out-connector area> est remplacé(e) respectivement par l'action <free action> ou zone <in-connector area> contenant ce nom de connecteur, à moins que le branchement <join> ou la zone <out-connector area> ne soit contenu(e) dans la même action <free action> ou zone <in-connector area> que le nom de connecteur correspondant.

    2) Les actions <free action> et zones <in-connector area> sont écartées. Les graphes résultants ne sont plus acycliques, exception faite des boucles à l'intérieur d'une transition (les entrées ne partagent aucune transition).

18. L'état suivant pointillé est remplacé par un nom d'état (4.9).

19. Les virgules de poursuite dans <stimulus> (2.6.4), <create body> (2.7.2), <procedure body> (2.7.3) et <output body> sont insérées.

20. Les identificateurs de synonymes sont remplacés par les expressions qu'ils désignent (5.3.2.3).

21. Les entrées prioritaires sont transformées (4.10).

22. Les signaux continus sont transformés (4.11).

23. Les conditions de validation sont transformées (4.12).

24. Les tâches implicites pour les expressions contenant <now expression> (5.4.4.2), <import expression> (4.13), <view expression> (5.4.4.4), <timer active expression> (5.4.4.5), et les appels de procédures implicites pour les expressions contenant <value returning procedure call> (5.4.5) sont insérés.

25. Les valeurs importées et exportées (4.13) et les procédures distantes (4.14) sont transformées.

7.2 Insertion de qualificateurs complets

Le présent paragraphe décrit en détail le contenu de l'étape 8 (voir 7.1 ci-dessus); il s'agit des qualificateurs complets.

L'insertion des qualificateurs complets et la transformation des concepts transformés dans le présent paragraphe sont effectuées progiciel par progiciel (dans l'ordre donné par la liste <package list>) et se terminent avec la définition <system definition>. L'insertion des qualificateurs complets est effectuée en appliquant les règles de visibilité et les règles de résolution par contexte définies en 2.2.2.

Les étapes 1 à 9 ci-dessous sont répétées jusqu'à ce que toutes les unités de portée aient été transformées.

1. Sélection d'une unité de portée (a priori le <package> le plus à gauche dans la liste <package list> ou la définition <system definition>) qui satisfait aux conditions suivantes:

   1) l'unité de portée environnante (s'il y en a une) a été transformée;

   2) les paramètres <actual context parameter> omis sont insérés (6.1.2);

   3) si elle contient <specialization>, alors toute unité de portée contenue dans ce type donné par <base type> a été transformée;

   4) elle n'a pas encore été transformée.

2. Si l'unité de portée contient <specialization>, une copie du type <base type> est réalisée, et tous les paramètres <actual context parameter> contenus dans <specialization> sont liés. La copie du type désigné par <base type> est réalisée de sorte que:

   1) Les types virtuels qui ont aussi une définition dans l'unité de portée à transformer se voient attribuer des noms anonymes uniques, mais les unités de portée conservent les informations relatives à leur qualificateur complet d'origine de sorte que les identificateurs qui désignent les types virtuels d'origine (c'est-à-dire les identificateurs qualifiés avec le type <base type> à l'intérieur de l'unité de portée et à l'intérieur des spécialisations de l'unité de portée) puissent être remplacés par la nouvelle entité lorsque les qualificateurs seront insérés (étape 7 ci-dessous).

   2) Toutes les occurrences appliquées (à l'exception des générateurs) des paramètres <formal context parameter> définis pour ce type sont remplacées par les paramètres <actual context parameter> correspondants. Les sous-signaux, les littéraux et les opérateurs des paramètres <signal context parameter> et des paramètres <sort context parameter> respectivement voient leurs qualificateurs modifiés pour désigner l'identificateur du paramètre <actual context parameter>. Les paramètres <formal context parameter> correspondant aux paramètres <actual context parameters> omis ne sont pas remplacés mais deviennent (parties des) paramètres <formal context parameter> de l'unité de portée.

   3) Les transitions virtuelles ne sont pas transformées à cette étape, car il existe une interdépendance entre les entrées astérisques, les sauvegardes astérisques et les états astérisques. A la place de cette transformation, une information est insérée dans le corps combiné, concernant le corps de type duquel est issu tout état. Par conséquent, le corps combiné ne peut pas être considéré valide jusqu'à ce que les entrées astérisques, les sauvegardes astérisques et les états astérisques aient été transformés (voir l'étape 14 au 7.1).

   4) Les occurrences du nom de type dans les qualificateurs contenus dans le type sont remplacées par le nom de l'unité de portée (sauf dans les générateurs).

3. Transformation des transformations <generator transformation>

1) Les transformations <generator transformation> à l'intérieur des définitions <generator definition> rencontrées dans l'unité de portées sont transformées.

2) Les transformations <generator transformation> à l'intérieur des définitions <partial type definition> rencontrées dans l'unité de portées sont transformées.

4. <ordering> est transformé (5.3.1.8).

5. Les définitions <partial type definition> sont transformées:

1) Si une définition <partial type definition> contient une expression <type expression>, les littéraux et opérateurs sont copiés selon la liste d'héritage. Si le type <base type> d'une définition <partial type definition> a des paramètres formels <sort context parameter>, toute sorte d'argument ou sorte de résultat mentionnant un paramètre <sort context parameter> est modifiée en conséquence.

2) Les propriétés d'égalité sont dérivées (5.3.1.4).

3) Les qualificateurs des identificateurs dans les axiomes <axioms>, dans les conditions <range condition> et dans l'expression par défaut sont insérés et la quantification explicite est insérée. La résolution par contexte est effectuée en écartant toutes les définitions <partial type definition> ayant des paramètres <formal context parameter> (à l'exception de la définition englobante s'il en existe une).

4) Pour toute définition <partial type definition> contenant une expression <type expression> avec des paramètres <actual context parameter>, les axiomes <axioms>, l'initialisation <default initialization> et les conditions <range condition> contenues dans le type <base type> sont copiées dans la définition <partial type definition>, là où les paramètres <formal context parameter> ont été remplacés par les paramètres <actual context parameter> correspondants.

5) Pour toute définition <partial type definition> contenant une expression <type expression> sans paramètres <actual context parameter>, les axiomes <axioms> impliqués sont insérés (5.3.1.11).

6. Les définitions <signal definition> avec <specialization> sont supprimées:

Une définition de signal qui comporte <specialization> et dans laquelle le type <base type> désigne une définition <signal definition> sans <specialization> est choisie. Le modèle de <specialization> pour cette définition <signal definition> est appliqué de manière similaire à l'étape 2 ci-dessus. Cette étape est répétée jusqu'à ce qu'il n'y ait plus de définitions <signal definition> ayant <specialization> dans l'unité de portée.

7. Les qualificateurs dans les identificateurs directement contenus dans l'unité de portée et dans les identificateurs contenus dans les définitions de signal contenues sont insérés.

La résolution par contexte est effectuée en écartant toutes les définitions ayant des paramètres <formal context parameter>.

Les références directement contenues dans l'unité de portée à un type virtuel dans un supertype d'une unité de portée englobante sont transformées en références à la copie anonyme (voir l'étape 2.1 ci-dessus).

8. Les <stimulus> contenant des variables <indexed variable> et <field variable> sont transformés (2.6.4).

9. Transformation:

– des identificateurs de liste de signaux en listes d'identificateurs de signaux (2.5.5);

– des primaires indexés (5.3.2.4);

– des variables de champ (5.4.3.2);

– des variables indexées (5.4.3.1);

– de <return> avec <expression> (2.4.6).

Annexe A

Index
des articles 2 à 7 de la Recommandation Z.100 (parties normatives)

(Cette annexe fait partie intégrante de la présente Recommandation)

Les entrées sont:

a) les mots clés <keyword> de la grammaire concrète;

b) les non-terminaux de la grammaire concrète. Il s'agit des termes entre crochets (< et >). Leurs définitions sont les entrées en gras dans l'index;

c) les non-terminaux de la grammaire abstraite. Leurs définitions sont les entrées en gras dans l'index;

d) les éléments du glossaire. Il s'agit des mots marqués par un (G). La présente version ne contient que les entrées du glossaire relatives aux articles 2 à 7.

Si une entrée de définition et une entrée d'application d'un non-terminal apparaissent sur la même page, l'entrée en gras (pour la définition) est prioritaire. Les non-terminaux de la grammaire concrète qui n'ont pas d'entrée en gras

1) soit ne font pas partie des règles de syntaxe; ou

2) contiennent des parties soulignées (les parties soulignées ne sont pas indiquées dans l'index).

Les non-terminaux contenant des parties soulignées doivent être examinés en ignorant les parties soulignées pour les occurrences de définition.

<action statement>; 57; 58; 60; 68; 69; 101 <instruction d'action>
<action>; 55; 58 <action>
<active alternative expression>; 156 <expression active d'alternative>
<active consequence expression>; 156 <expression active de conséquence>
<active expression list>; 156; **157** <liste d'expressions actives>
<active expression>; **154**; 156; 157 <expression active>
<active extended primary>; 154 <primaire étendu actif>
<active primary>; 149; 150; **154** <primaire actif>
<actual context parameter>; 19; 172; 177; **178**; 198; 199 <paramètre réel de contexte>
<actual context parameters>; 137; 138; 171; **177**; 178; 183; 196; 198 <paramètres réels de contexte>
<actual parameters>; 63; 64; 65; 66; 67; 112; 164 <paramètres réels>
<additional heading>; 21 <en-tête supplémentaire>
<alphanumeric>; 14; 15; 128 <alphanumérique>
<alternative expression>; 156 <expression d'alternative>
<alternative ground expression>; 153 <expression close d'alternative>
<alternative question>; 100; 101 <question d'alternative>
<alternative>; 130 <alternative>
<answer part>; 68; 69; 100; 101; 106; 108 <partie réponse>
<answer>; 68; 69; 70; 101 <réponse>
<any area>; 92; 94; 95 <zone quelconque>
<anyvalue expression>; 69; 160; **163** <expression valeur quelconque>
<apostrophe>; 15; 22; 128 <apostrophe>
<argument list>; 119; 120; 133; 136 <liste d'arguments>
<argument sort>; 19; 119; 120; 121; 133; 147; 150 <sorte d'argument>
<assignment statement>; 55; 62; **157**; 158; 159 <instruction d'affectation>
<asterisk input list>; 54; **103** <liste d'entrées astérisque>
<asterisk save list>; 55; 103 <liste de sauvegardes astérisque>
<asterisk state list>; 52; **102** <liste d'états astérisque>
<asterisk>; 102; 103; 185 <astérisque>
<axioms>; 117; 118; **122**; 141; 145; 199 <axiomes>
<base type literal rename signature>; 137; 138 <signature de renommage de littéral de type de base>
<base type operator name>; 137 <nom d'opérateur de type de base>
<base type>; 49; 138; **171**; 174; 176; 178; 179; 182; 183; 196; 198; 199 <type de base>
<basic input area>; **54**; 55 <zone d'entrée de base>

<priority input>; 52; **104**; 105 <entrée prioritaire>
<procedure area>; **40**; 93; 99 <zone de procédure>
<procedure body>; 39; **40**; 42; 169; 197 <corps de procédure>
<procedure call area>; 59; **64**; 65; 93 <zone d'appel de procédure>
<procedure call body>; **64** <corps d'appel de procédure>
<procedure call symbol>; **64**; 113 <symbole d'appel de procédure>
<procedure call>; 41; 58; **64**; 163; 164 <appel de procédure>
<procedure constraint>; **179** <contrainte de procédure>
<procedure context parameter>; 178; **179** <paramètre de contexte de procédure>
<procedure definition>; 18; 24; 27; 28; 31; 33; 37; **39**; 40; 98; 118; 148; 196 <définition de procédure>
<procedure diagram>; 18; 27; 38; **40**; 41; 148; 167; 168; 169; 170 <diagramme de procédure>
<procedure formal parameter constraint>; **179** <contrainte de paramètre formel de procédure>
<procedure formal parameters>; **39**; 41;42; 164 <paramètre formel de procédure>
<procedure graph area>; 40; **41**; 42; 57; 93 <zone de graphe de procédure>
<procedure heading>; 40; **41** <en-tête de procédure>
<procedure identifier>; 39; 41; 64; 65; 164; 179; 180; 196 <identificateur de procédure>
<procedure name>; 33; 39; 40; 41; 179; 180 <nom de procédure>
<procedure preamble>; 33; **39**; 40; 41; 184 <préambule de procédure>
<procedure result>; 18; 39; **40**; 41; 42; 61; 164; 195 <résultat de procédure>
<procedure signature>; 112; 134; **179**; 180 <signature de procédure>
<procedure start area>; **41**; 42; 93 <zone de départ de procédure>
<procedure start symbol>; **41**; 147 <symbole de départ de procédure>
<procedure symbol>; 40; **41** <symbole de procédure>
<procedure text area>; **40**; 93; 99 <zone de texte de procédure>
<process area>; **31**; 32; 46; 93; 99 <zone de processus>
<process body>; **33**; 34; 37 <corps de processus>
<process constraint>; 17; **179** <contrainte de processus>
<process context parameter>; 17; 166; 167; 178; **179** <paramètre de contexte de processus>
<process definition>; 18; 27; 30; 32; **33**; 34; 37; 46; 47; 78; 98; 102; 103; 118; 174; 177; 196 <définition de processus>
<process diagram>; 18; 27; 31; **34**; 35; 196 <diagramme de processus>
<process graph area>; 34; **35**; 38; 57; 60; 93; 99 <zone de graphe de processus>
<process heading>; 34; **35** <en-tête de processus>
<process identifier>; 17; 33; 35; 45; 63; 64; 66;67; 174; 179; 196 <identificateur de processus>
<process interaction area>; **31**; 93; 168 <zone d'interaction de processus>
<process name>; 17; 31; 33; 35; 173; 179 <nom de processus>
<process signature>; **179** <signature de processus>
<process symbol>; **31**; 174; 176 <symbole de processus>
<process text area>; 34; **35**; 93; 99; 169 <zone de texte de processus>
<process type body>; 60; 168; **169**; 170 <corps de type de processus>
<process type definition>; 18; 24; 27; 28; 30; 32; 46; 47; 63; 66; 98; 118; **168**; 169 <définition de type de processus>
<process type diagram>; 18; 27; 46; 99; 167; 168; **169** <diagramme de type de processus>
<process type expression>; 173; 174 <expression de type de processus>
<process type graph area>; 60; 93; 99; **169** <zone de graphe de type de processus>
<process type heading>; **169** <en-tête de processus>
<process type identifier>; 168; 169 <identificateur de type de processus>
<process type name>; 168; 169 <nom de type de processus>
<process type reference>; 93; 99; 167; 168; **169** <référence de type de processus>
<process type symbol>; **169** <symbole de type de processus>
<properties expression>; **117**; 118; 126; 133;139; 140; 141; 145 <expression de propriétés>
<qualifier>; **17**; 18; 19; 27; 120; 126; 127; 140;150; 152 <qualificateur>
<quantification>; **122**; 123 <quantification>
<quantified equations>; **122**; 123 <équation quantifiée>
<question expression>; 68 <expression de question>
<question>; **68**; 69; 106; 108 <question>
<quote>; **127** <guillemets>
<quoted operator>; 17; 18; 19; 126; **127**; 150 <opérateur entre guillemets>
<range condition>; 68; 70; 133; 134; **135**; 195; 199 <condition d'intervalle>
<referenced definition>; 23; 24; **26**; 27; 194 <définition référencée>
<regular element>; **143**; 144 <élément régulier>
<regular expression>; **143**; 144 <expression régulière>
<regular interval>; 128; **143**; 144 <intervalle régulier>
<remote procedure call area>; 59; 93; **113** <zone d'appel de procédure distante>
<remote procedure call body>; **112**; 113 <corps d'appel de procédure distante>

<type in process area>; 34; 94; **169** <type dans une zone de processus>
<type in system area>; 25; 29; 85; 87; 94; 166; **167** <type dans une zone de système>
<typebased block heading>; **173** <en-tête de bloc fondé sur le type>
<typebased process heading>; **173**; 174 <en-tête de processus fondé sur le type>
<typebased service heading>; **174** <en-tête de service fondé sur le type>
<typebased system heading>; **172** <en-tête de système fondé sur le type>
<underline>; **14**; 15; 16; 17; 18; 128; 194 <soulignement>
<unquantified equation>; **122**; 123; 124 <équation non quantifiée>
<upward arrow head>; **14** <tête de flèche vers le haut>
<valid input signal set>; **33**; 34; 35; 36; 37; 38; 46; 47; 168; 169; 170; 171 <ensemble de signaux d'entrée valides>
<value identifier>; 19; 122; 145; 146 <identificateur de valeur>
<value name>; 122; 123; 145 <nom de valeur>
<value returning procedure call>; 42; 148; 149; **163**; 164; 196; 198 <appel de procédure retournant une valeur>
<variable access>; 154; **155** <accès aux variables>
<variable context parameter>; 166; 167; 169; 178; **181** <paramètre de contexte de variable>
<variable definition>; 17; 33; 35; 37; 38; 40; **50**; 51; 98; 111; 147; 148; 170 <définition de variable>
<variable identifier>; 55; 64; 110; 155; 157 <identificateur de variable>
<variable name>; 33; 40; 42; 50; 147; 181 <nom de variable>
<variables of sort>; **50** <variable de sorte>
<variable>; 54; 55; **157**; 158; 159 <variable>
<vertical line>; **14** <ligne verticale>
<via path element>; **66**; 67 <élément de trajet via>
<via path>; **66**; 67; 88 <trajet via>
<view definition>; 17; 33; 35; 37; 38; 50; **51**; 98 <définition de vue>
<view expression>; 160; **162**; 198 <expression de vue>
<view identifier>; 162 <identificateur de vue>
<view name>; 51 <nom de vue>
<virtuality constraint>; 39; 41; 167; 168; 169; 170; **184** <contrainte virtuelle>
<virtuality>; 21; 39; 41; 42; 51; 52; 54; 55; 56; 57; 104; 105; 106; 112; 113; 114; 147; 167; 168; 169; 170; **183**; 184; 185; 197 <virtualité>
<word>; **14**; 17; 164 <mot>
abstract data types (G); 115
action (G); 59
active; 15; 162
active expression (G); 148; 154
active timer (G); 71; 163
Active-expression; 148; **154**
actual context parameters (G); 177
actual parameter (G); 63; 64; 91
adding; 15; 136; 137; 139; 141; 175; 176; 182; 188; 193
all; 15; 66; 67; 122; 124; 129; 132; 137; 138; 139; 145; 146; 197
alternative; 15; 100; 102; 130; 164; 165
Alternative-expression; **155**
and; 15; 48; 74; 78; 84; 86; 87; 88; 100; 127; 129; 132; 135; 137; 139; 149; 190
And-operator-identifier; **134**
any; 15; 68; 69; 163
anyvalue expression (G); 163
Anyvalue-expression; 160; **163**
Argument-list; **119**
as; 15; 39; 40; 50; 110; 113
assignment statement (G); 157
Assignment-statement; 62; **157**
association area (G); 53; 87
atleast; 15; 139; 172; 175; 178; 179; 180; 182; 184; 187; 189; 193
axiom (G); 116; 121
axioms; 15; 117; 137; 139; 140; 141; 142; 145; 146
basic SDL (G); 13
block; 15; 17; 20; 24; 28; 30; 31; 73; 74; 86; 88; 96; 100; 167; 168; 173; 187; 188; 189; 190
block (G); 32
block substructure (G); 84
block tree diagram (G); 84
block type (G); 168
Block-definition; 28; **30**; 83

Annexe B

Glossaire du SDL

(Cette annexe fait partie intégrante de la présente Recommandation)

La présente Recommandation contient les définitions formelles de la terminologie du SDL. Le glossaire SDL est établi pour aider les nouveaux utilisateurs du SDL à lire la Recommandation et ses annexes; chaque terme y est assorti d'une brève définition, informelle, et d'un renvoi à l'article de la Recommandation où il est expliqué. Certaines définitions contenues dans le glossaire peuvent résumer ou paraphraser les définitions formelles, et de ce fait, être incomplètes.

Les termes en *italique* dans une définition peuvent aussi se trouver dans le glossaire. Si une expression en italique, par exemple, *identificateur de procédure*, ne figure pas dans le glossaire, alors cette expression peut être la concaténation de deux termes; dans le cas présent, le terme *procédure* précédé du terme *identificateur*. Quand un mot en italique ne se trouve pas dans le glossaire, il peut s'agir d'un mot dérivant d'un terme y figurant. Par exemple, *exported* est, en anglais, le temps passé du verbe *export*. Les mots clés du SDL sont en caractères **gras**.

Après la définition d'un terme, figure une référence à son utilisation principale dans la présente Recommandation, sauf si ce terme est le synonyme d'un autre terme. Ces références sont représentées entre crochets [ ] après les définitions. Par exemple, [3.2] indique que la référence principale figure au paragraphe 3.2.

type abstrait de données *(E: abstract data type)*

Les *types abstraits de données* définissent les données en termes de leurs propriétés abstraites plutôt qu'en termes d'implantation. Un *type abstrait de données* est une classe de type qui définit des ensembles de *valeurs (sortes)*, un ensemble d'*opérateurs* qui s'appliquent à ces *valeurs* et un ensemble de règles algébriques *(équations)* qui définissent le *comportement* des *opérateurs* appliqués à ces valeurs. [2.3.1, 5.1]

grammaire abstraite *(E: abstract grammar)*

La *grammaire abstraite* définit la *sémantique* du *SDL*. La *grammaire abstraite* est décrite par la *syntaxe abstraite* et les *règles de bonne formation*. [1.2, 1.4.1]

syntaxe abstraite *(E: abstract syntax)*

La *syntaxe abstraite* est un moyen permettant de décrire la structure conceptuelle d'une *spécification SDL*. Les *syntaxes concrètes* sont mises en correspondance avec la *syntaxe abstraite* pour assurer l'équivalence entre le *SDL/PR* et le *SDL/GR*. [1.2]

action *(E: action)*

Une *action* est une opération qui est exécutée à l'intérieur d'une *chaîne de transition*, par exemple, une *tâche,* une *sortie*, une *décision*, une *demande de création*, un *armement de temporisateur*, un *désarmement de temporisateur*, un *export* ou un *appel de procédure*. [2.6.8.1, 2.7]

expression active *(E: active expression)*

Une *expression active* est une *expression* dont la *valeur* dépend de l'état courant du système. Une *expression active* accède à une *variable* ou contient un *opérateur impératif*. [5.3.3.1, 5.4.2.1]

temporisateur actif *(E: active timer)*

Un *temporisateur actif* est un *temporisateur* qui a un *signal de temporisation* dans le *port d'entrée* du *processus* auquel il appartient ou qui produira un *signal de temporisation* dans le futur. [2.8, 5.4.4.5].

paramètre contextuel réel *(E: actual context parameter)*

Un *paramètre contextuel réel* est un *identificateur* qui donne la définition réelle d'un *paramètre contextuel formel* correspondant. L'*identificateur du paramètre réel* peut, soit être un *identificateur* d'un *paramètre contextuel formel* d'une définition de *sous-type*, soit correspondre à une définition qui est visible dans l'unité de portée englobante de l'*expression de type*, ou visible par l'intermédiaire d'un *composant référencé*. [6.2]

paramètre réel *(E: actual parameter)*

Un *paramètre réel* est une *expression* évaluée par un *processus* (ou une *procédure*) pour initialisation du *paramètre formel* correspondant, au moment de la *création* du *processus* (ou de l'*appel de la procédure*). A noter que dans certains cas, dans un *appel de procédure*, un *paramètre réel* doit être une *variable* (c'est-à-dire un type particulier d'*expression*; voir *IN/OUT*). [2.7.2, 2.7.3, 4.2.2]

expression à valeur aléatoire *(E: anyvalue expression)*

Une *expression à valeur aléatoire* est une *expression* qui rend une *valeur* non spécifiée du *syntype* ou de la *sorte* désignée. [5.4.4.6]

zone *(E: area)*

Une *zone* est une région bidimensionnelle de la *syntaxe graphique concrète*. Les *zones* correspondent, la plupart du temps, aux *nœuds* de la *syntaxe abstraite* et contiennent habituellement la *syntaxe textuelle commune*. Dans les *diagrammes d'interaction*, les *zones* peuvent être connectées au moyen de *canaux* ou d'*acheminements de signaux*. Dans les *diagrammes de flux de contrôle*, les *zones* peuvent être connectées au moyen de *lignes de flux*. [2.4.2.6]

array (tableau) *(E: array)*

Array est un *générateur de type prédéfini* utilisé pour introduire le concept de tableaux, ce qui simplifie la définition des tableaux. [Annexe D]

instruction d'affectation *(E: assignment statement)*

Une *instruction d'affectation* est une instruction dans une tâche, qui, lorsqu'elle est interprétée, associe une *valeur* à une *variable* et remplace la *valeur* précédente affectée à la *variable*. [5.4.3]

zone d'association *(E: association area)*

Une *zone d'association* est une connexion entre *zones* dans un *diagramme d'interaction* réalisée au moyen d'un *symbole* d'association. Il y a cinq zones d'association: la *zone d'association de sous-structure de canal,* la *zone d'association liée à une entrée,* la *zone d'association liée à une entrée prioritaire,* la *zone d'association liée à un signal continu,* la *zone d'association liée à une mise en réserve.* [1.5.3, 2.6.3, 3.2.3, 4.10.2, 4.11]

axiome *(E: axiom)*

Un *axiome* est une forme particulière d'*équation* implicitement équivalente au *littéral Booléen* True (Vrai). «*Axiomes*» est utilisé comme synonyme de «*axiomes* et *équations*». [5.1.3, 5.2.3]

SDL de base *(E: basic SDL)*

Le *SDL de base* est un sous-ensemble du *SDL* défini en 2. [2]

comportement *(E: behaviour)*

En SDL, le *comportement* est soit

1) le comportement externe observable, c'est-à-dire l'ensemble des séquences de réponses d'un *système* à des séquences de stimuli; soit
2) le comportement interne observable, c'est-à-dire l'ensemble des actions et des tâches engendrées par un stimulus et exécutées avant que la machine à états arrive dans un nouvel état. [1.1.3]

liaison *(E: binding)*

Une *liaison* associe soit, des *paramètres réels* avec des *paramètres formels* (lors de la création d'un processus ou lors d'un appel de procédure), soit, des *paramètres contextuels réels* avec des *paramètres contextuels* (formels). [1.3.1]

bloc *(E: block)*

Un *bloc* est un élément d'un *système* ou d'un *bloc* qui contient un ou plusieurs processus ou une *sous-structure de bloc*. Un *bloc* est une *unité de portée*; il fournit une interface statique. Quand il est utilisé en tant que tel, *bloc* est un synonyme d'*instance de bloc*. [2.4.3]

sous-structure de bloc *(E: block substructure)*

Une *sous-structure de bloc* est une *subdivision* du *bloc* en *sous-blocs* et en nouveaux *canaux* à un *niveau d'abstraction* inférieur. [3.2.2]

diagramme d'arbre de blocs *(E: block tree diagram)*

Un *diagramme d'arbre de blocs* est un document auxiliaire *SDL/GR* qui représente la *subdivision* d'un *système* en *blocs* à des *niveaux d'abstraction* inférieurs successifs, au moyen d'un diagramme d'arbre inversé (c'est-à-dire avec le *bloc* parent à la racine). [3.2.1]

type de bloc *(E: block type)*

Un *type de bloc* est l'association d'un *nom* et d'un ensemble de propriétés que toutes les *instances de bloc* du *type* posséderont. [6.1.1.2]

forme BNF (Backus-Naur form) [*(E: BNF (Backus-Naur Form)*]

La *forme BNF* (Backus-Naur form) est une notation formelle utilisée pour exprimer la *syntaxe textuelle concrète* d'un langage. Une forme étendue de la *BNF* est utilisée pour exprimer la *grammaire graphique concrète*. [1.5.2, 1.5.3]

Boolean (Booléen) *(E: Boolean)*

Boolean est une *sorte* définie dans une *définition partielle de type* prédéfinie et qui a les *valeurs* True (Vrai) et False (Faux). Pour la *sorte Booléenne,* les *opérateurs* prédéfinis sont **not, and, or, xor** et l'implication. [5.3.1.3, Annexe D]

canal *(E: channel)*

Un *canal* est une connexion qui achemine des *signaux* entre deux *blocs*. Les *canaux* acheminent aussi des *signaux* entre un *bloc* et l'*environnement*. Les *canaux* peuvent être unidirectionnels ou bidirectionnels. [2.5.1]

sous-structure de canal *(E: channel substructure)*

Une *sous-structure de canal* est la *subdivision* d'un *canal* en un ensemble de *canaux* et de *blocs* à un *niveau d'abstraction* inférieur. [3.2.3]

character (caractère) *(E: character)*

Character est une *sorte prédéfinie* dont les *valeurs* sont des éléments de l'alphabet CCITT nº 5, (c'est-à-dire 1, A, B, C, etc.). Pour la *sorte caractère*, les *opérateurs de relation d'ordre* sont prédéfinis. [Annexe D]

charstring (chaîne de caractères) *(E: charstring)*

Charstring est une *sorte prédéfinie* dont les *valeurs* sont des *chaînes* de *caractères* et les *opérateurs* sont ceux du *générateur* prédéfini *caractère* instancié pour les *caractères*. [Annexe D]

commentaire *(E: comment)*

Un *commentaire* est une information qui complète ou qui clarifie une *spécification SDL*. En *SDL/GR*, les *commentaires* peuvent être attachés à tout *symbole* au moyen d'une ligne pointillée. En *SDL/PR*, les *commentaires* sont intoduits au moyen du mot clé **comment**. Les *commentaires* n'ont pas de signification *SDL.* Voir aussi *Note*. [2.2.6]

grammaire textuelle commune *(E: common textual grammar)*

La *grammaire textuelle commune* est le sous-ensemble de la *grammaire textuelle concrète* qui s'applique à la fois au *SDL/GR* et au *SDL/PR.* [1.2]

trajet de communication *(E: communication path)*

Un *trajet de communication* est un moyen de transport qui véhicule des *instances de signaux* d'une *instance de processus* ou de l'*environnement* vers une autre *instance de processus* ou vers l'*environnement*. Un *trajet de communication* est composé, soit d'un ou plusieurs trajets de *canaux*, soit d'un ou plusieurs trajets d'*acheminements de signaux*, soit d'une combinaison des deux. [2.7.4]

ensemble complet de signaux d'entrée valides *(E: complete valid input signal set)*

L'*ensemble complet de signaux d'entrée valides* d'un *processus*, est l'union de l'*ensemble des signaux d'entrée valides*, des *signaux* locaux, des *signaux de temporisation* et des *signaux implicites* du *processus*. [2.4.4]

grammaire concrète *(E: concrete grammar)*

Une *grammaire concrète* représente la *syntaxe concrète* et les *règles de bonne formation* pour cette *syntaxe concrète*. Le *SDL/GR* et le *SDL/PR* sont les *grammaires concrètes* du *SDL.* Les *grammaires concrètes* sont mises en correspondance vers la *grammaire abstraite* pour déterminer leur *sémantique*. [1.2]

grammaire graphique concrète *(E: concrete graphical grammar)*

La *grammaire graphique concrète* est la *grammaire concrète* de la partie graphique du *SDL/GR*. [1.2, 1.5.3]

syntaxe graphique concrète *(E: concrete graphical syntax)*

La *syntaxe graphique concrète* est la *syntaxe concrète* pour la partie graphique du *SDL/GR*. La *syntaxe graphique concrète* est exprimée dans la présente Recommandation au moyen d'une forme étendue de la *BNF*. [1.2, 1.5.3]

syntaxe concrète *(E: concrete syntax)*

La *syntaxe concrète* pour les diverses représentations du *SDL* est composée des *symboles* réels utilisés pour représenter le *SDL* et des relations entre *symboles* requises par les règles syntaxiques du *SDL*. Les deux *syntaxes concrètes* utilisées dans la présente Recommandation sont la *syntaxe graphique concrète* et la *syntaxe textuelle concrète*. [1.2]

syntaxe textuelle concrète *(E: concrete textual syntax)*

La *syntaxe textuelle concrète* est composée de la *syntaxe concrète* pour le *SDL/PR* et des parties textuelles du *SDL/GR*. La *syntaxe textuelle concrète* est exprimée dans la présente Recommandation au moyen de la *BNF*. [1.2, 1.5.2]

expression conditionnelle *(E: conditional expression)*

Une *expression conditionnelle* est une *expression* qui contient une *expression booléenne* qui peut engendrer soit l'interprétation d'une *expression* de conséquence soit l'interprétation d'une *expression* d'alternative. [5.4.2.3]

connecter *(E: connect)*

Connecter représente la connexion d'un *canal* à un ou plusieurs *acheminements de signaux* ou l'interconnexion d'*acheminements de signaux*. [2.5.3]

connecteur *(E: connector)*

Un *connecteur* en *SDL/PR* est une *étiquette portant* sur une *action*. Un *connecteur* est un *symbole SDL/GR* qui est soit un *connecteur d'entrée* soit un *connecteur de sortie*. Une *ligne de flux* va implicitement des *connecteurs de sortie* au *connecteur d'entrée* associé, du même *processus* ou de la même *procédure*, par identification de l'*étiquette*. [2.6.7, 2.6.8, 2.2]

sous-ensemble de subdivision cohérent *(E: consistent partitioning subset)*

Un *sous-ensemble de subdivision cohérent* est un ensemble de *blocs* et de *sous-blocs* d'une *spécification de système* qui fournit une vue complète du *système* et des relations entre ses composants à un certain *niveau d'abstraction*. De ce fait, quand un *bloc* ou un *sous-bloc* fait partie d'un *sous-ensemble de subdivision cohérent*, ses ascendants et ses descendants en font également partie. [3.2.1]

sous-ensemble de raffinement cohérent *(E: consistent refinement subset)*

Un *sous-ensemble de raffinement cohérent* est un *sous-ensemble de subdivision cohérent* contenant tous les *blocs* et *sous-blocs* qui utilisent les *signaux* utilisés par l'un quelconque des *blocs* ou *sous-blocs*. [3.3]

contrainte *(E: constraint)*

La *contrainte* d'un *paramètre contextuel* contraint les *paramètres contextuels réels* et définit les propriétés du paramètre connues par le *type paramétré*. La *contrainte* d'un *type virtuel* contraint les redéfinitions et définit les propriétés du *type virtuel* connues par le *type* englobant. Voir aussi *contrainte de porte*. [6.2]

paramètre contextuel *(E: context parameter)*

Un *paramètre contextuel* (formel) d'une définition de *type paramétré* est l'association d'un *identificateur* et d'une *contrainte*. Un nouveau *type* est défini (au moyen d'une *expression de type*) en associant quelques-uns ou tous les *paramètres contextuels* avec les définitions réelles. [6.2]

signal continu *(E: continuous signal)*

Un *signal continu* est une *notation abrégée* qui permet de déclencher une transition lorsque la condition *booléenne* est vraie. [4.11]

diagramme de flux de contrôle *(E: control flow diagram)*

Un *diagramme de flux de contrôle* est soit un *diagramme de processus*, soit un *diagramme de procédure*, soit un *diagramme de service*.

create (créer) *(E: create)*

Create (créer) est synonyme de *demande de création*.

zone de ligne de création *(E: create line area)*

La *zone de ligne de création* dans un *diagramme de bloc* connecte la *zone de processus* du processus *créateur* (**parent**) avec la *zone de processus* du processus *créé* (**offspring**). [2.4.3]

demande de création *(E: create request)*

Une *demande de création* est l'*action* qui provoque la création et le démarrage d'une nouvelle *instance de processus* en utilisant un *type de processus* comme modèle. Les *paramètres réels* dans la *demande de création* remplacent les *paramètres formels* du *processus*. [2.7.2]

nouvel état «tiret» *(E: dash nextstate)*

Un *nouvel état «tiret»* est une *notation abrégée* indiquant que le *nouvel état* de l'*instance de processus* est l'*état* courant. [4.9]

type de données *(E: data type)*

Un *type de données* est un synonyme pour *type abstrait de données*.

définition de type de données *(E: data type definition)*

La *définition de type de données* à un endroit quelconque d'une *spécification* en *SDL* définit la validité des *opérateurs*, des *sortes* et des *expressions* et la relation entre les *expressions*. [5.2.1]

décision *(E: decision)*

Une *décision* est une *action* à l'intérieur d'une *transition* consistant à poser une question dont la réponse peut être obtenue à cet instant et à choisir en conséquence l'une des *transitions* possibles de sortie de la *décision* afin de continuer l'interprétation. [2.7.5]

initialisation par défaut *(E: default initialization)*

Une *initialisation par défaut* est une notation qui permet d'associer la même *valeur* à toutes les *variables* d'une *sorte* spécifiée avant que leur *processus* ou *procédure* associé ne soit interprété. [5.4.3.3]

définition *(E: definition)*

Une *définition* associe un nom et un ensemble de propriétés à un *type* ou à une *instance*. [1.3.1]

description *(E: description)*

Une *description* d'un *système* est la description de son *comportement* réel. [1.1]

diagramme *(E: diagram)*

Un *diagramme* est la représentation, en *SDL/GR*, d'une partie de *spécification*. [2.4.2]

durée (duration) *(E: duration)*

Duration (durée) est une *sorte prédéfinie* dont les *valeurs* sont représentées comme des *réels* et qui représente l'intervalle de temps entre deux instants. [Annexe D]

condition de validation *(E: enabling condition)*

Une *condition de validation* est un moyen d'accepter conditionnellement un *signal* en *entrée*. [4.12]

contrainte terminale *(E: endpoint constraint)*

Une *contrainte terminale* est la représentation en *SDL/GR* d'une *contrainte* portant sur une *porte* contraignant ceux des *blocs/processus/services* qui peuvent être à l'autre bout d'un *canal/acheminement de signal/acheminement de signal de service* connecté à la *porte*. [6.1.4]

classe d'entité *(E: entity kind)*

Une *classe d'entité* est une catégorisation des *types* du *SDL* basée sur la similitude d'usage. [2.2.2]

environnement *(E: environment)*

Le terme *environnement* est synonyme d'*environnement d'un système*. Lorsque le contexte le permet, ce peut être un synonyme de l'*environnement* d'un *bloc*, d'un *processus*, d'une *procédure*, ou d'un *service*.

environnement d'un système *(E: environment of a system)*

L'*environnement d'un système* est le monde extérieur du *système* en train d'être spécifié. L'environnement et le *système* interagissent par envoi et réception d'*instances de signaux*. [1.3.2]

équation *(E: equation)*

Une *équation* est une relation entre des *termes* de la même *sorte* qui se vérifie pour toutes les *valeurs* possibles substituées à chaque *identificateur de valeur* de l'*équation*. Une *équation* peut être un *axiome*. [5.1.3]

erreur *(E: error)*

Pendant l'interprétation d'une *spécification valide* d'un *système*, une *erreur* se produit lorsque l'une des conditions dynamiques du *SDL* est violée. Lorsqu'une *erreur* s'est produite, le *comportement* ultérieur du *système* n'est pas défini. [1.3.3]

export *(E: export)*

Le terme *export* est synonyme d'*opération d'exportation*.

variable exportée *(E: exported variable)*

Une *variable exportée* est une *variable* qui peut être utilisée dans une *opération d'exportation*. Sa *définition* contient le mot clé **exported**. [2.6.1.1]

exportateur *(E: exporter)*

Un *exportateur* d'une *variable* est l'*instance de processus* qui possède la *variable* et *exporte* ses *valeurs*. [4.13]

opération d'exportation *(E: export operation)*

Une *opération d'exportation* est l'exécution d'une *action d'exportation* par laquelle l'*exportateur* divulgue la *valeur* courante d'une *variable*. Voir aussi *opération d'importation*. [4.13]

expression *(E: expression)*

Une *expression* peut être un *littéral*, une application d'*opérateur*, un *synonyme*, un *accès à une variable*, une *expression conditionnelle* ou un *opérateur impératif* appliqué à une ou plusieurs *expressions*. Lorsqu'une *expression* est interprétée, une *valeur* est obtenue (ou le *système* se trouve en *erreur*). [2.3.4, 5.3.3.1]

synonyme externe *(E: external synonym)*

Un *synonyme* externe est une *sorte* prédéfinie dont la valeur n'est pas spécifiée dans la *spécification du système*. [4.3.1]

extract! *(E: extract!)*

Extract! est un *opérateur* implicite pour l'extraction d'un *champ* d'une *sorte structurée* et il est implicite dans une *expression* lorsqu'une *variable* est suivie immédiatement d'*expression*(s) entre parenthèses. [5.3.1.10, 5.3.3.4, 5.3.3.5]

champ *(E: field)*

Un *champ* est un élément d'une *sorte structurée*. [5.3.1.10]

ligne de flux *(E: flow line)*

Une *ligne de flux* est un *symbole* utilisé pour connecter des *zones* dans un *diagramme de flux de contrôle*. [2.2.4, 2.6.8.2.2]

paramètre contextuel formel *(E: formal context parameter)*

Voir *paramètre contextuel*.

paramètre formel *(E: formal parameter)*

Un *paramètre formel e*st un *nom* de *variable* auquel sont affectées des *valeurs réelles* ou qui sont remplacées par des *variables réelles*. [2.4.4, 2.4.6, 4.2]

porte *(E: gate)*

Une *porte* est définie dans un *type* de *bloc*/*processus*/*service* et représente un point de connexion pour les *canaux*/*acheminements de signaux* connectant des *instances* du *type* à d'autres instances ou au symbole de cadre englobant. [6.1.4]

contrainte de porte *(E: gate constraint)*

Une *contrainte de porte* contraint la façon dont les *canaux*/*acheminements de signaux* peuvent être connectés à la *porte* et fournit des restrictions sur les *signaux* entrants et sortants. [6.1.4]

définition de porte *(E: gate definition)*

Une *définition de porte* est la réprésentation d'une *porte* en *SDL/PR*. [6.1.4]

paramètres généraux *(E: general parameters)*

Dans la *spécification* et dans la *description* d'un *système*, les *paramètres généraux* se rapportent à des sujets qui ne concernent pas le *comportement* comme les limites de température, les caractéristiques de construction, la capacité des commutateurs, la qualité du service, etc., et ne sont pas définis en *SDL*. [1.1]

générateur *(E: generator)*

Un *générateur* permet de définir un modèle de texte paramétré qui est développé par transformation avant de considérer la sémantique des types de données. [5.3.1.12, 7.2]

graphe *(E: graph)*

Dans la *syntaxe abstraite*, un *graphe* est une partie d'une *spécification en SDL* telle qu'un *graphe de procédure*, un *graphe de service* ou un *graphe de processus*. [2.4.4, 2.4.5, 2.4.6]

contrainte de porte graphique *(E: graphical gate constraint)*

Une *contrainte de porte graphique* est la représentation d'une *contrainte de porte* en *SDL/GR*. [6.1.4]

expression close *(E: ground expression)*

Une *expression close* est une *expression* contenant uniquement des *opérateurs*, des *synonymes* et des *littéraux*. [5.3.3.2]

structure hiérarchique *(E: hierarchical structure)*

Une *structure hiérarchique* est une structure d'une *spécification de système* où la *subdivision* et le *raffinement* permettent différentes vues du *système* à différents *niveaux d'abstraction*. Voir aussi *diagramme d'arbre de bloc*. [3.1]

identificateur *(E: identifier)*

Un *identificateur* est l'unique identification d'un objet; un *identificateur* est formé d'un *qualificatif* et d'un *nom*. [2.2.2]

opérateur impératif *(E: imperative operator)*

Un *opérateur impératif* est une *expression de vue*, une *expression de temporisateur actif*, une *expression d'importation*, une *expression à valeur aléatoire*, une *expression «now»* ou l'une des *expressions de PId:* **self**, **parent**, **offspring** ou **sender**. [5.4.4]

transition implicite *(E: implicit transition)*

Dans la *syntaxe concrète,* une *transition implicite* est déclenchée par un *signal* appartenant à *l'ensemble complet des signaux d'entrée valides* mais non spécifié dans une *entrée* ou dans une *mise en réserve* pour *l'état* considéré. Une *transition implicite* ne contient aucune *action* et retourne directement au même *état*. [4.8]

import *(E: import)*

Le terme *import* est synonyme *d'opération d'importation*. [4.13]

variable importée *(E: imported variable)*

Une *variable importée* est une *variable* utilisée dans une *expression d'importation*. [4.13]

importateur *(E: importer)*

Un *importateur* d'une *variable importée* est *l'instance de processus* qui importe la *valeur*. [4.13]

expression d'importation *(E: import expression)*

Une *expression d'importation* spécifie le *mot clé* **import** et *l'identificateur* d'une *variable*. [4.13]

opération d'importation *(E: import operation)*

Une *opération d'importation* est l'exécution d'une *expression d'importation* par laquelle *l'importateur* accède à la *valeur* de la *variable exportée*. [4.13]

hériter *(E: inherit)*

Un *type spécialisé* ou une *sorte* possède ou *hérite* de toutes les propriétés de son *super-type* ou de la *sorte* référencée. [5.3.1.11, 6.3]

paramètre «in» *(E: in parameter)*

Un paramètre «**in**» est un attribut de *paramètre formel* qui indique qu'une *valeur* est passée à une *procédure* grâce à un *paramètre effectif*. [2.4.6]

paramètre «in/out» *(E: in/out parameter)*

Un paramètre «**in/out**» est un attribut de *paramètre formel* qui indique qu'un *nom de paramètre formel* est utilisé comme *synonyme* de la *variable* (c'est-à-dire que le *paramètre effectif* doit être une *variable*). [2.4.6]

connecteur d'entrée *(E: in-connector)*

Voir *connecteur*.

opérateur infixe *(E: infix operator)*

Un *opérateur infixe* est un des *opérateurs* dyadiques prédéfinis du *SDL* (=>, **or**, **xor**, **in**, /=, =, >, <, <=, >=, +, –, //, *, /, **mod**, **rem**) qui sont placés entre deux arguments. [5.3.1.1]

texte informel *(E: informal text)*

Un *texte informel* est du texte inclus dans une *spécification en SDL*, pour lequel la *sémantique* n'est pas définie par le *SDL*, mais par un autre modèle. Un *texte informel* est placé entre apostrophes. [2.2.3]

accès entrant *(E: inlet)*

Un *accès entrant* représente une ligne telle qu'un *canal* ou une *ligne de flux*, entrant dans un *appel de macro* en *SDL/GR*. [4.2.3]

entrée *(E: input)*

Une *entrée* est la consommation d'un *signal* depuis le *port d'entrée* qui commence une *transition*. Durant la consommation d'un *signal*, les *valeurs* qui lui sont associées deviennent disponibles pour l'*instance de processus*. [2.6.4, 4.10.2]

port d'entrée *(E: input port)*

Le *port d'entrée* d'un *processus* est une file d'attente qui reçoit et retient les *signaux* dans l'ordre de leur arrivée jusqu'à ce que les *signaux* soient consommés par une *entrée*. Le *port d'entrée* peut contenir un nombre quelconque de *signaux retenus*. [2.4.4]

instance *(E: instance)*

Une *instance* est une partie de système; elle possède des propriétés et un comportement. Une *instance* d'un type est un objet qui possède toutes les propriétés du type telles qu'elles sont données dans la définition. [1.3.1]

instanciation *(E: instantiation)*

L'*instanciation* est la création d'une *instance* d'un *type*. [1.3.1]

integer (entier) *(E: integer)*

Integer (entier) est une *sorte prédéfinie* pour laquelle les *valeurs* sont celles des entiers mathématiques (..., –2, –1, 0, +1, +2, ...). Pour la *sorte integer* les *opérateurs* prédéfinis sont +, –, *, /, et les *opérateurs de relation d'ordre*. [Annexe D]

diagramme d'interaction *(E: interaction diagram)*

Un *diagramme d'interaction* est un *diagramme de bloc*, un *diagramme de système*, un *diagramme de sous-structure de canal*, ou un *diagramme de sous-structure de bloc*.

branchement *(E: join)*

Un *branchement* indique un changement dans le flot d'exécution d'une *chaîne de transition* en indiquant le *connecteur* ou *l'étiquette* de *l'action* qui doit être exécutée ensuite. [2.6.8.2.2]

mot clé *(E: keyword)*

Un *mot clé* est une *unité lexicale* réservée dans la *syntaxe textuelle concrète*. [2.2.1]

étiquette *(E: label)*

Une *étiquette* est un *nom* suivi du caractère deux points et utilisé dans la *syntaxe textuelle concrète* pour indiquer la cible du transfert de contrôle à partir d'un *branchement* qui référence le même *identificateur* que l'*étiquette*. [2.6.7, 2.6.8.2.2]

niveau *(E: level)*

Le terme *niveau* est synonyme de *niveau d'abstraction*.

niveau d'abstraction *(E: level of abstraction)*

Un *niveau d'abstraction* est un des niveaux d'un *diagramme d'arbre de blocs*. Une description d'un *système* est un *bloc* au *niveau d'abstraction* le plus haut et se présente comme un *bloc* unique au sommet du *diagramme d'arbre de blocs*. [3.2.1]

règles lexicales *(E: lexical rules)*

Les *règles lexicales* sont des règles qui définissent la façon dont les *unités lexicales* sont construites à partir de caractères. [2.2.1, 4.2.1]

unité lexicale *(E: lexical unit)*

Les *unités lexicales* sont les *symboles* terminaux de la *syntaxe textuelle concrète.* [2.2.1]

littéral *(E: literal)*

Un *littéral* représente une *valeur*; c'est un *opérateur* sans argument. [2.3.3, 5.1.2, 5.2.2, 5.3.1.14]

macro *(E: macro)*

Une *macro* est une collection nommée d'éléments syntaxiques ou textuels, qui remplace l'*appel de macro* avant de considérer la signification de la représentation en *SDL* (une *macro* a une signification uniquement lorsqu'elle est remplacée dans un contexte particulier). [4.2]

appel de macro *(E: macro call)*

Un *appel de macro* est une indication de l'endroit où la *définition de macro* ayant le même *nom* doit être développée. [4.2.3]

make! *(E: make!)*

Make! est une opération utilisée uniquement dans les définitions de *types de données* pour constituer une *valeur* d'un type complexe (c'est-à-dire une *sorte structurée*). [5.3.1.10]

zone de fusion *(E: merge area)*

Une *zone de fusion* est l'endroit où une *ligne de flux* se connecte à une autre. [2.6.8.2.2]

Meta IV *(E: Meta IV)*

Meta IV est une notation formelle pour exprimer la *syntaxe abstraite* d'un langage. [1.5.1]

modèle *(E: model)*

Un *modèle* donne la correspondance entre des *notations abrégées* et leur expression en termes de *syntaxe concrète* précédemment définie. [1.4.1, 1.4.2]

modify! *(E: modify!)*

Modify! est un *opérateur* implicite des *sortes structurées* pour modifier un *champ* d'une *sorte structurée* et est implicite dans des *expressions* où une *variable* est immédiatement suivie d'expressions entre parenthèses puis de :=. Dans les axiomes, *modify!* est utilisé explicitement (voir aussi *extract!*). [5.3.1.10, 5.4.3.1, 5.4.3.2]

nom *(E: name)*

Un *nom* est une *unité lexicale* utilisée pour nommer des objets en *SDL*. [2.2.1, 2.2.2]

natural (naturel) *(E: natural)*

Natural est un *syntype prédéfini* pour lequel les *valeurs* sont les entiers non négatifs (c'est-à-dire, 0, 1, 2, ...). Les *opérateurs* sont les *opérateurs* de la *sorte integer*. [Annexe D]

newtype (nouveau type) *(E: newtype)*

Un *newtype* (nouveau type) introduit une *sorte*, un ensemble d'*opérateurs* et un ensemble d'*équations*. A noter que le terme *newtype* risque de prêter à confusion parce qu'en fait c'est une nouvelle *sorte* qui est introduite, mais le terme *newtype* est maintenu pour des raisons historiques. [5.2.1]

état suivant (nextstate) *(E: nextstate)*

Le *nouvel état (nextstate*) termine une *transition* et spécifie le *nouvel état* de *l'instance de processus*. [2.8.6.2.1]

nœud *(E: node)*

Dans la *syntaxe abstraite*, un *nœud* est une désignation d'un des concepts de base du *SDL*.

note *(E: note)*

Une *note* est un texte entouré par les signes /* et */ et qui n'a pas de *sémantique* définie en *SDL*. Voir aussi *commentaire*. [2.2.1]

expression «now» *(E: now expression)*

Une *expression «now»* (*maintenant*) est une *expression* de la *sorte time* qui fournit l'heure courante du système. [5.4.4.1]

null *(E: null)*

Null est un *littéral* de la *sorte prédéfinie PId*. [Annexe D]

offspring (descendant) *(E: offspring)*

offspring est une *expression* de la *sorte prédéfinie PId*. Lorsque **offspring** est évalué dans un *processus,* il fournit la *valeur PId* du dernier *processus* créé par ce *processus*. Si le *processus* n'a créé aucun *processus*, le résultat de l'évaluation de **offspring** est *null*. [2.4.4]

opérateur *(E: operator)*

Un *opérateur* est une désignation d'une opération. Les *opérateurs* sont définis dans une *définition partielle de type*. Par exemple +, –, *, /, sont les *noms des opérateurs* définis pour la *sorte integer*. [5.1.2, 5.1.3, 5.2.1, 5.3.2]

signature d'opérateur *(E: operator signature)*

Une *signature d'opérateur* définit la (les) *sorte*(s) des *valeurs* auxquelles s'applique l'*opérateur* et la *sorte* de la *valeur* résultante. [5.2.2]

option *(E: option)*

Une *option* est une construction de la *syntaxe concrète* dans une *spécification* générique *de système en SDL* autorisant le choix de différentes structures de *système* avant l'interprétation du *système*. [4.3.3, 4.3.4]

opérateurs de relation d'ordre *(E: ordering operators)*

Les *opérateurs de relation d'ordre* sont <, <=, > ou >=. [5.3.1.8]

connecteur de sortie *(E: out-connector)*

Voir *connecteur*.

accès sortant *(E: outlet)*

Un *accès sortant* représente une ligne telle qu'un *canal* ou une *ligne de flux*, sortant d'un *diagramme de macro*. [4.2.2]

sortie *(E: output)*

Une *sortie* est une *action* dans une *transition* qui engendre une *instance de signal*. [2.7.4]

composant *(E: package)*

Un *composant* est un ensemble de définitions. [2.4.1.2]

interface de composant *(E: package interface)*

Une *interface de composant* énumère les définitions visibles d'un *composant*. [2.4.1.2]

référence de composant *(E: package reference)*

Une *référence de composant* rend les définitions à l'intérieur d'un *composant* visibles d'un autre composant ou d'un *système*. Deviennent visibles les définitions énumérées dans l'*interface de composant* du composant soit individuellement, soit globalement. [2.4.1.2]

page *(E: page)*

Une *page* est un des composants d'une subdivision physique d'un *diagramme*. [2.2.5]

type paramétré *(E: parameterized type)*

Un *type paramétré* est un *type* dont certains aspects de sa dépendance vis-à-vis de l'unité de portée qui le contient sont exprimés par des *paramètres contextuels*. [1.3.1]

parent *(E: parent)*

parent est une *expression* de la *sorte prédéfinie PId*. Lorsqu'un *processus* évalue cette *expression*, le résultat est la *valeur PId* du *processus* parent. Si le *processus* a été créé à l'initialisation du *système*, le résultat est *null*. [2.4.4]

définition partielle de type *(E: partial type definition)*

La *définition partielle de type* pour une *sorte* définit certaines des propriétés liées à la *sorte*. Une *définition partielle de type* fait partie d'une *définition de type de données*. [5.2.1]

subdivision *(E: partitioning)*

La *subdivision* est le découpage d'une unité en composants plus petits qui, pris dans leur ensemble, ont le même *comportement* que l'unité d'origine. La *subdivision* ne modifie pas l'interface statique d'une unité. [3.1, 3.2]

PId *(E: PId)*

PId est une *sorte prédéfinie* pour laquelle il existe un *littéral*, *null*. *PId* est l'abréviation anglaise d'identificateur d'instance de processus, et les *valeurs* des *sortes* sont utilisées pour identifier des *instances de processus*. [Annexe D]

powerset (ensemble) *(E: powerset)*

Powerset est le *générateur prédéfini* utilisé pour introduire des ensembles mathématiques. Les *opérateurs* de *powerset* sont **in**, Incl, Del, union, intersection et les *opérateurs de relation d'ordre*. [Annexe D]

predefined (prédéfini) *(E: predefined)*

Predefined est un *composant* contenant les *définitions partielles de type* pour les *données prédéfinies*. [2.4.1.2, 5.2.1, 5.3.1.3, Annexe D]

données prédéfinies *(E: predefined data)*

Pour simplifier la description, le terme *données prédéfinies* s'applique à des *noms* prédéfinis de *sortes* introduites par des *définitions partielles de type* ainsi qu'à des *noms* prédéfinis de *générateurs de type de données*. Les *noms* de *sortes* prédéfinis sont : *Boolean*, *character*, *charstring*, *duration*, *integer*, *natural PId*, *real* et *time*. Les *noms de générateurs de type de données* prédéfinis sont *array, powerset* et *string*. Les *données prédéfinies* sont définies dans le *composant* implicitement utilisé *Predefined*. [2.4.1.2, 5.1.1, 5.3.1.3, Annexe D]

entrée prioritaire *(E: priority input)*

Une *entrée prioritaire* est une *notation abrégée* indiquant que la réception des *signaux énumérés* devrait avoir priorité sur les *signaux* énumérés dans les *entrées* de cet état. [4.10]

procédure *(E: procedure)*

Une *procédure* est un encapsulage d'une partie du *comportement* d'un *processus*. Une *procédure* est définie en un endroit mais peut être référencée plusieurs fois dans le même *processus*. [2.4.6]

appel de procédure *(E: procedure call)*

Un *appel de procédure* est l'invocation d'une *procédure* nommée pour interprétation de la *procédure* et passage des *paramètres réels* à la *procédure*. [2.7.3]

contrainte de procédure *(E: procedure constraint)*

Une *contrainte de procédure* est la condition portant sur les *paramètres réels* d'un *paramètre contextuel de procédure* formel sous la forme soit d'une *procédure,* soit d'une *contrainte de procédure*. [6.2.2]

paramètre contextuel de procédure *(E: procedure context parameter)*

Un *paramètre contextuel de procédure* est un *paramètre contextuel* pour lequel le *paramètre contextuel réel* doit être une *procédure* respectant la *contrainte de procédure*. [6.2.2]

signature de procédure *(E: procedure signature)*

Une *signature de procédure* est une condition portant sur un *paramètre contextuel de procédure* formel en termes de *contraintes* sur les *paramètres formels* de la *procédure* réelle. [6.2.2]

début de procédure *(E: procedure start)*

Dans une *procédure*, le *début de procédure* indique le point où commence l'exécution de la *procédure* lors d'un *appel de procédure*. [2.4.6]

processus *(E: process)*

Un *processus* est une machine à états finis étendue communicante. La communication peut avoir lieu par l'intermédiaire de *signaux* ou de *variables* partagées. Le *comportement* d'un *processus* dépend de l'ordre d'arrivée des *signaux* dans son *port d'entrée*. [2.4.4]

contrainte de processus *(E: process constraint)*

Une *contrainte de processus* est la condition portant sur les *paramètres réels* d'un *paramètre contextuel de processus* formel, soit en termes de *type de processus* (l'*instance de processus* réelle doit alors être un *processus* de ce *type* ou d'un *sous-type*), soit une *signature de processus* (l'*instance de processus* réelle doit alors être compatible avec la *signature du processus*). [6.2.1]

paramètre contextuel de processus *(E: process context parameter)*

Un *paramètre contextuel de processus* est un *paramètre contextuel* pour lequel le *paramètre réel* doit être une définition de processus ou une définition de processus typé respectant la *contrainte de processus*. [6.2.1]

instance de processus *(E: process instance)*

Une *instance de processus* est un élément de l'ensemble des *processus.* Une *instance de processus* est créée lors de la création du système ou dynamiquement à la suite d'une *demande de création*. Voir **self, sender, parent** et **offspring**. [2.4.4]

signature de processus *(E: process signature)*

Une *signature de processus* est une condition portant sur un *paramètre contextuel de processus* formel en termes de *contraintes* sur les *paramètres formels* des processus réels et sur l'*ensemble des signaux d'entrée valides* du *processus* réel. [6.2.1]

type de processus *(E: process type)*

Un *type de processus* est l'association d'un *nom* et d'un ensemble de propriétés que possèdent toutes les *instances de processus* de ce *type de processus*. [6.1.1.3]

qualificatif *(E: qualifier)*

Le *qualificatif* est la partie d'un *identificateur* qui constitue l'information supplémentaire qui doit être ajoutée à la partie *nom* pour assurer l'unicité. Les *qualificatifs* sont toujours présents dans la *syntaxe abstraite*, mais ne doivent être utilisées dans la *syntaxe concrète* que si cela est nécessaire, pour préserver l'unicité lorsque le *qualificatif* d'un *identificateur* ne peut être déduit du contexte d'utilisation de la partie *nom*. [2.2.2]

real (réel) *(E: real)*

Real est une *sorte prédéfinie* pour laquelle les *valeurs* sont les nombres qui peuvent être présentés par un *entier* divisé par un autre *entier*. Les *opérateurs* prédéfinis pour la *sorte real* ont les mêmes *noms* que les *opérateurs* de la sorte entier (integer). [Annexe D]

définition référencée *(E: referenced definition)*

Une *définition référencée* est un moyen syntaxique pour répartir une *définition de système* en plusieurs parties et pour établir des relations entre chaque partie. [2.4.1.3]

raffinement *(E: refinement)*

Le *raffinement* est l'adjonction de détails fonctionnels supplémentaires à un *niveau d'abstraction* donné. Le *raffinement* d'un *système* entraîne un enrichissement de son *comportement* ou de ses capacités de traiter davantage de types de *signaux* et d'information, y compris les *signaux* en provenance et à destination de l'*environnement*. Comparer avec *subdivision*. [3.3]

appel de procédure distante *(E: remote procedure call)*

Un *appel de procédure distante* est une demande envoyée par un *processus* client à un *processus* serveur pour exécuter une *procédure* définie dans le *processus* serveur. [4.14]

transition d'entrée de procédure distante *(E: remote procedure input transition)*

Une *transition d'entrée de procédure distante* indique un *état* dans lequel un *processus* va exécuter la procédure exportée spécifiée, suivi éventuellement par une *transition*. [4.14]

mise en réserve de procédure distante *(E: remote procedure save)*

Une *mise en réserve de procédure distante* indique que la *procédure* exportée spécifiée ne va pas être exécutée dans l'*état*. [4.14]

spécification de procédure distante *(E: remote procedure definition)*

Une *spécification de procédure distante* introduit le nom et la *signature de procédure* pour les *procédures importées* et *exportées*. [4.14]

spécification de variable distante *(E: remote variable definition)*

Une *spécification de variable distante* introduit le nom et la *signature de sorte* pour les *valeurs importées* et *exportées*. [4.13]

reset (réinitialisation) *(E: reset)*

Reset est une opération définie pour les *temporisateurs* qui permet, d'une part, de les désactiver et, d'autre part, de supprimer les *signaux de temporisation* appliqués au *port d'entrée du processus*. Voir *temporisateur actif*. [2.8]

signal retenu *(E: retained signal)*

Un *signal retenu* est un signal dans le *port d'entrée* d'un *processus*, c'est-à-dire un *signal* qui a été reçu mais pas consommé par le *processus*. [2.4.4]

retour *(E: return)*

Un *retour* (d'une *procédure*) est la passation du contrôle à la *procédure* ou au *processus* appelant. [2.6.8.2.4]

attribut révélé *(E: revealed attribut)*

Une *variable* qui appartient à un *processus* peut avoir l'*attribut révélé*, auquel cas un autre *processus* peut voir la *valeur* associée à la *variable*. Voir *définition de vue*. [2.6.1.1]

mise en réserve *(E: save)*

Une *mise en réserve* est la déclaration des *signaux* qui ne doivent pas être consommés dans un *état* donné. [2.6.5]

SDL (langage de description et de spécification du CCITT) *(E: SDL)*

Le langage *SDL* (langage de description et de spécification) du CCITT est un langage formel fournissant un ensemble de constructions pour la *spécification* du *comportement* d'un *système*.

SDL/GR *(E: SDL/GR)*

Le *SDL/GR* est la représentation graphique du langage *SDL* (ou *SDL* graphique). La *grammaire* du *SDL/GR* est définie par la *grammaire graphique concrète* et la *grammaire textuelle commune*. [1.2]

SDL/PR *(E: SDL/PR)*

Le *SDL/PR* est la représentation textuelle du langage *SDL* (ou *SDL* textuel). La *grammaire* du *SDL/PR* est définie par la *grammaire textuelle concrète*. [1.2]

unité de portée *(E: scope unit)*

Dans la *grammaire concrète*, une *unité de portée* définit l'intervalle de *visibilité* des *identificateurs*. Parmi les *unités de portée*, on peut citer le *système*, le *bloc*, le *processus*, la *procédure*, les *définitions partielles de types* et les *définitions de service*. [2.2.2]

sélection *(E: selection)*

La *sélection* consiste à fournir les *synonymes externes* nécessaires pour construire une *spécification de système* spécifique à partir d'une *spécification de système* générique. [4.3, 4.3.3, 4.3.4]

self *(E: self)*

self est une *expression* de la *sorte prédéfinie PId*. Lorsqu'un *processus* évalue cette *expression*, le résultat est la *valeur PId* de ce *processus*. **self** ne conduit jamais à la valeur *Null*. Voir également **parent**, **offspring**, *PId*. [2.4.4, 5.5.4.3]

sémantique *(E: semantics)*

La *sémantique* donne une signification à une entité: les propriétés qu'elle a, la manière dont son *comportement* est interprété, et toutes les conditions dynamiques qui doivent être remplies pour que le *comportement* de l'entité soit conforme aux règles du *SDL*. [1.4.1, 1.4.2]

sender (émetteur) *(E: sender)*

sender est une *expression PId*, qui lorsqu'elle est évaluée, donne la *valeur PId* du *processus* ayant émis le *signal* qui a activé la *transition* en cours. [2.4.4, 2.6.4, 5.5.4.3]

service *(E: service)*

Un *service* est un autre moyen de spécifier une partie du comportement d'un *processus*. Chaque *service* peut définir un *comportement* partiel d'un processus. [2.4.5]

type de service *(E: service type)*

Un *type de service* est l'association d'un *nom* et d'un ensemble de propriétés que possèdent tous les services de ce type de service. [6.1.1.4]

set (armement) *(E: set)*

Set est une opération définie pour les *temporisateurs* permettant de les rendre *actifs*. *Set* spécifie l'heure du système à laquelle le *temporisateur actif* doit renvoyer un *signal de temporisation*. [2.8]

notation abrégée *(E: shortand notation)*

Une *notation abrégée* est une notation de *syntaxe concrète* qui donne une représentation plus compacte renvoyant implicitement à des *concepts du SDL de base.* [1.4.2, 4.1]

signal *(E: signal)*

Un *signal* est une instance de *type* de signal communiquant des informations à une *instance de processus.* [2.5.4]

contrainte de signal *(E: signal constraint)*

Une *contrainte de signal* est la condition portant sur les *paramètres réels* d'un *paramètre contextuel de signal* formel, soit en termes de *type de signal* (spécifiant que la *définition de signal* doit être un *sous-type* du *type de la contrainte*) soit une *signature de signal* (l'*instance de signal* réelle doit être ensuite compatible avec la *signature du signal*). [6.2.4]

paramètre contextuel de signal *(E: signal context parameter)*

Un *paramètre contextuel de signal* est un *paramètre contextuel* pour lequel le *paramètre réel* doit être un *type de signal* respectant la *contrainte de signal*. [6.2.4]

définition de signal *(E: signal definition)*

Une *définition de signal* définit un *type de signal nommé* et associe au *nom de signal* une liste, éventuellement vide, *d'identificateurs* de *sorte*. Ceci permet aux *signaux* de transporter des *valeurs*. [2.5.4]

liste de signaux *(E: signal list)*

Une *liste de signaux* est une *liste d'identificateurs* utilisés dans des *définitions de canal* et d'*acheminement de signal* pour indiquer quels *signaux* peuvent être transportés par le *canal* ou l'*acheminement de signal* dans une direction. [2.5.5]

acheminement de signal *(E: signal route)*

Un *acheminement de signal* indique le flux des *signaux* entre un *type de processus* et soit un autre *type de processus* appartenant au même *bloc* ou soit les *canaux* liés à ce *bloc*. [2.5.2]

signature de signal *(E: signal signature)*

Une *signature de signal* est une condition portant sur un *paramètre contextuel de signal* formel en termes de *contraintes* sur les *sortes* des *paramètres* du *type de signal* réel. [6.2.4]

signature *(E: signature)*

La *signature* d'un *type* est l'ensemble des *sortes* et l'ensemble des *opérateurs* pour ce *type.* [5.2.1, 5.2.2]

expression simple *(E: simple expression)*

Une *expression simple* est une *expression* qui contient uniquement des *opérateurs*, des *synonymes* et des *littéraux* de *sortes* prédéfinies. [4.3.2]

sorte *(E: sort)*

Une *sorte* est un ensemble de *valeurs* ayant des caractéristiques communes. Les *sortes* sont toujours non vides et disjointes. [2.3.3, 5.1.2, 5.1.3, 5.2.1]

contrainte de sorte *(E: sort constraint)*

Une *contrainte de sorte* est une condition portant sur les *paramètres réels* d'un *paramètre contextuel de sorte* formel, soit en termes de *sortes* (la *sorte* réelle doit alors être un *sous-type* de ce *type*) soit une *signature de sorte* (la *sorte* réelle doit alors être compatible avec les *opérateurs* de cette *signature de sorte*). [6.2.9]

paramètre contextuel de sorte *(E: sort context parameter)*

Un *paramètre contextuel de sorte* est un *paramètre contextuel* dont le *paramètre réel* doit être une *sorte* respectant la *contrainte de sorte*. [6.2.9]

signature de sorte *(E: sort signature)*

Une *signature de sorte* est une condition portant sur un *paramètre contextuel de sorte* formel en termes de contraintes sur les *opérateurs* et les littéraux de la *sorte* réelle. [6.2.9]

spécialisation *(E: specialization)*

La *spécialisation* crée un nouveau *type*, appelé un *sous-type*, en ajoutant des propriétés et/ou en redéfinissant les propriétés d'un autre *type* appelé le *super-type*. [1.3.1]

spécification *(E: specification)*

Une *spécification* est une définition des besoins d'un *système*. Une *spécification* se compose des *paramètres généraux* du *système* et de la *spécification fonctionnelle* de son *comportement* souhaité. Le mot *spécification* peut également être utilisé comme une abréviation pour «*spécification* et/ou *description*», par exemple dans *spécification en SDL* ou dans *spécification de système*. [1.1]

transition spontanée *(E: spontaneous transition)*

Une *transition spontanée* permet l'interprétation d'une *transition* sans qu'il y ait consommation de *signal* par le *processus*. [2.6.6]

départ *(E: start)*

Le *départ* figurant dans un *processus* ou dans un *service* est une *chaîne de transition* interprétée à la création du *processus* avant tout *état* ou *action*. [2.6.2]

état *(E: state)*

Un *état* est une condition dans laquelle une *instance de processus* peut consommer un *signal*. [2.6.3]

arrêt *(E: stop)*

Un *arrêt* est une action qui termine une *instance de processus*. Lorsqu'un *arrêt* est interprété, toutes les *variables* qui appartiennent à l'*instance du processus* sont détruites et tous les *signaux retenus* dans le *port d'entrée* cessent d'être accessibles. [2.6.8.2.3]

string (chaîne) *(E: string)*

String est un *générateur prédéfini de données* utilisé pour introduire des listes. Parmi les *opérateurs* prédéfinis on peut citer: length (longueur), first (premier), last (dernier), substring (sous-chaîne) et la concaténation représentée par //. [Annexe D]

sorte structurée *(E: structured sort)*

Une *sorte structurée* est une *sorte* dont les valeurs se composent d'une liste de *valeurs* de *sortes* (*champs*). Une *sorte structurée* est pourvue d'*opérateurs* et d'*équations* implicites ainsi que d'une *syntaxe concrète* particulière pour représenter ces *opérateurs* implicites permettant d'accéder aux *valeurs* des *champs* et de modifier de façon indépendante les *valeurs* des *champs*. [5.3.1.10]

sous-bloc *(E: subblock)*

Un *sous-bloc* est un *bloc* contenu dans un autre *bloc*. Les *sous-blocs* résultent de la *subdivision* d'un *bloc*. [3.2.1, 3.2.2]

sous-canal *(E: subchannel)*

Un *sous-canal* est un *canal* résultant de la *subdivision* d'un *bloc*. Un *sous-canal* connecte un *sous-bloc* à la frontière du *bloc subdivisé* ou connecte un *bloc* à la frontière d'un *canal subdivisé*. [3.2.2, 3.2.3]

sous-signal *(E: subsignal)*

Un *sous-signal* est un *raffinement* d'un *signal* et peut être l'objet d'un *raffinement* ultérieur. [3.3]

sous-type *(E: subtype)*

Un *sous-type* est un *type* défini comme une *spécialisation* d'un autre *type* (le *super-type*). Les propriétés associées à un *sous-type* sont toutes les propriétés du *super-type*, sauf celles correspondant aux *virtuels* redéfinis, plus les propriétés supplémentaires particulières au *sous-type*. [1.3.1, 6.3]

super-type *(E: supertype)*

Un *super-type* est le *type* utilisé pour définir un *sous-type*. [1.3.1, 6.3]

symbole *(E: symbol)*

Un *symbole* est un terminal dans les *syntaxes concrètes*. Un *symbole* peut être l'une des formes dans la *syntaxe concrète graphique*, ou une suite de caractères dans la *syntaxe concrète textuelle*. [1.5.2, 1.5.3]

synonyme *(E: synonym)*

Un *synonyme* est un *nom* qui représente une *valeur* d'une *sorte*. [5.3.1.13]

paramètre contextuel de synonyme *(E: synonym context parameter)*

Un *paramètre contextuel de synonyme* est un *paramètre contextuel* dont le *paramètre réel* doit être un *synonyme* respectant les conditions portant sur les *paramètres réels* d'un *paramètre contextuel de synonyme* formel en termes d'un *type* de *sorte*. [6.2.8]

syntype *(E: syntype)*

Un *syntype* spécifie un ensemble de *valeurs* qui correspond à un sous-ensemble des *valeurs* de la *sorte*. Les *opérateurs* du *syntype* sont les mêmes que ceux de la *sorte* parente. [5.3.1.9]

système *(E: system)*

Un *système* est un ensemble de *blocs* reliés l'un à l'autre et à l'*environnement* par des *canaux*. [2.4.2]

type de système *(E: system type)*

Un *type de système* est l'association entre un *nom* et un ensemble de propriétés qu'auront toutes les *instances de système* de ce *type*. [6.1.1.1]

tâche *(E: task)*

Une *tâche* est une action à l'intérieur d'une *transition* contenant soit une suite d'*instructions d'affectation* soit du *texte informel*. [2.7.1]

terme *(E: term)*

Un *terme* est syntaxiquement équivalent à une *expression*. Les *termes* sont utilisés uniquement dans les *axiomes* et se distinguent des *expressions* par souci de clarté. [5.2.3, 5.3.3]

symbole d'extension de texte *(E: text extension symbol)*

Un *symbole d'extension de texte* contient du texte qui appartient au *symbole graphique* auquel le *symbole d'extension de texte* est associé. Le texte inclus dans le *symbole d'extension de texte* vient à la suite du texte inclus dans le symbole auquel il est associé. [2.2.7]

contrainte terminale textuelle *(E: textual endpoint constraint)*

Une *contrainte terminale textuelle* est la représentation *SDL/PR* d'une *contrainte terminale* d'une *porte* imposant des conditions aux *blocs/processus/services* pouvant être à l'autre bout d'un *canal/acheminement de signal/acheminement de signal de service* connecté à la porte. [6.1.4]

time (temps) *(E: time)*

Time est une *sorte prédéfinie de données* dont les *valeurs* sont représentées comme les *valeurs* des *nombres réels*. Les *opérateurs* prédéfinis pour les *sortes time* (temps) et *duration* (durée) sont + et –. [Annexe D]

temporisateur *(E: timer)*

Un *temporisateur* est un objet qui appartient à une *instance de processus* et qui peut être *actif* ou *inactif*. Un *temporisateur actif* renvoie un *signal de temporisation* à l'*instance de processus* à laquelle il appartient à un moment précis. Voir également *set* et *reset*. [2.8, 5.4.4.5]

expression active de temporisateur *(E: timer active expression)*

Une *expression active de temporisateur* est une *expression booléenne* qui indique, lorsqu'elle est exécutée, si le temporisateur identifié est actif. [5.4.4.5]

paramètre contextuel de temporisateur *(E: timer context parameter)*

Un *paramètre contextuel de temporisateur* est un *paramètre contextuel* pour lequel le *paramètre réel* doit être une définition de *temporisateur*. [6.2.7]

transition *(E: transition)*

Une *transition* est une séquence d'actions qui se produit lorsqu'une *instance de processus* passe d'un *état* à un autre. [2.6.8]

chaîne de transition *(E: transition string)*

Une *chaîne de transition* est une séquence d'*actions* dont le nombre est égal ou supérieur à zéro. [2.6.8.1]

type *(E: type)*

Un *type* est un ensemble de propriétés caractérisant les *instances*. Parmi les catégories de *types* en *SDL* on peut citer les *blocs*, les *processus*, les *services*, les *signaux* et les *systèmes*. [1.3.1]

définition de type *(E: type definition)*

Une *définition de type* définit les propriétés d'un *type*. [1.3.1]

expression de type *(E: type expression)*

Une *expression de type* représente un *type* qui est soit un *type* de base, soit un *type* anonyme défini en appliquant des *paramètres contextuels réels* au *type* de base paramétré. [6.1.2]

ensemble de signaux d'entrée valides *(E: valid input signal set)*

L'*ensemble des signaux d'entrée valides* d'un *processus* est la liste de tous les *signaux* externes traités pour toutes les *entrées* dans le *processus*. Il se compose des *signaux* présents dans les *acheminements de signaux* conduisant au *processus*. Comparer avec l'*ensemble complet des signaux d'entrée valides*. [2.4.4, 2.5.2]

spécification valide *(E: valid specification)*

Une *spécification valide* est une *spécification* qui respecte la *syntaxe concrète* et les *règles* statiques *de bonne formation*. [1.3.3]

valeur *(E: value)*

Une *valeur* d'une *sorte* est l'une des valeurs associées à une *variable* de cette *sorte*, et qui peut être utilisée avec un *opérateur* nécessitant une *valeur* de la sorte. Une *valeur* est le résultat de l'interprétation d'une *expression*. [2.3.3, 5.1.3]

procédure retournant une valeur *(E: value returning procedure)*

Une *procédure retournant une valeur* est une *procédure* qui peut renvoyer une *valeur*. [5.4.5]

variable *(E: variable)*

Une *variable* est une entité appartenant à une *instance de processus* ou à une *instance de procédure* et qui peut être associée à une *valeur* par l'intermédiaire d'une *instruction d'affectation*. Lorsqu'on y accède, une *variable* donne la dernière *valeur* qui lui a été affectée. [2.3.2]

paramètre contextuel de variable *(E: variable context parameter)*

Un *paramètre contextuel de variable* est un *paramètre contextuel* pour lequel le *paramètre réel* doit être une *variable* respectant les conditions portant sur les *paramètres réels* d'un *paramètre contextuel* de variable formel en termes d'un *type* de *sorte*; la *variable* réelle peut être une *variable* de ce *type*. [6.2.5]

définition de variable *(E: variable definition)*

Une *définition de variable* est la déclaration par laquelle les *noms de variables* énumérés deviennent visibles dans le *processus*, la *procédure* ou le *service* contenant la définition de la variable. [2.6.1.1]

définition de vue *(E: view definition)*

Une *définition de vue* définit la vue d'une *variable* dans un autre *processus* (dans lequel elle possède un *attribut révélé*). Cela permet au *processus* possédant la *définition de vue* d'accéder à la *valeur* de cette *variable*. [2.6.1.2]

expression de vue *(E: view expression)*

Une *expression de vue* est utilisée dans une *expression* pour obtenir la *valeur* courante d'une *variable vue*. [5.4.4.4]

signal continu virtuel *(E: virtual continuous signal)*

Un *signal continu virtuel* est un *signal continu* dont la *transition* peut être redéfinie dans un *sous-type*. [4.11, 6.3.3]

entrée virtuelle *(E: virtual input)*

Une *entrée virtuelle* est une *transition d'entrée* qui peut, dans un *sous-type*, être redéfinie en une *mise en réserve* ou en une *transition d'entrée*. [2.6.4, 6.3.3]

entrée prioritaire virtuelle *(E: virtual priority input)*

Une *entrée prioritaire virtuelle* est une transition qui peut être redéfinie en une *entrée prioritaire* ou en une *mise en réserve*. [6.3.3]

départ de procédure virtuelle *(E: virtual procedure start)*

Un *départ de procédure virtuelle* est une *transition* qui peut être redéfinie dans une *procédure* spécialisée. [2.4.6, 6.3.3]

entrée de procédure distante virtuelle *(E: virtual remote procedure input)*

Une *entrée de procédure distante virtuelle* est une *transition* qui peut être redéfinie en une nouvelle *transition d'entrée de procédure distante* ou en une *mise en réserve de procédure distante*. [6.3.3]

mise en réserve virtuelle *(E: virtual save)*

Une *mise en réserve virtuelle* est une *mise en réserve* qui peut, dans un *sous-type*, être redéfinie en une *transition d'entrée*. [2.6.5, 6.3.3]

départ virtuel *(E: virtual start)*

Un *départ virtuel* est une *transition de départ* qui peut, dans un *sous-type*, être redéfinie en une nouvelle *transition de départ*. [2.6.2, 6.3.3]

transition spontanée virtuelle *(E: virtual spontaneous transition)*

Une *transition spontanée virtuelle* est une *transition spontanée* dans un *type* qui peut être redéfinie dans un *sous-type*. [2.6.6, 6.3.3]

type virtuel *(E: virtual type)*

Un *type virtuel* est un *type* qui peut être redéfini en *sous-types* du *type* englobant. [6.3.2]

virtualité *(E: virtuality)*

La *virtualité* d'un *type* (resp. d'une *transition*) indique si le *type* (resp. la *transition*) est un *type virtuel*, redéfini dans un *sous-type* (et reste encore un *type virtuel*), ou finalisé (c'est-à-dire redéfini, mais n'est plus un *type virtuel*). [6.3.2]

contrainte de virtualité *(E: virtuality constraint)*

La *contrainte de virtualité* d'un *type virtuel* pose des conditions sur les redéfinitions du *type virtuel* grâce à une *contrainte de type*. A la fois la définition et les redéfinitions d'un *type virtuel* doivent être des définitions de *sous-types* de la *contrainte de type*. La *contrainte de type* d'un *type virtuel* détermine également les propriétés du *type virtuel* connu par le *type* englobant possédant le *type virtuel* local. [6.3.2]

visibilité *(E: visibility)*

La *visibilité* d'un *identificateur* est constituée par les *unités de portée* dans lesquelles il peut être utilisé. Dans la même *unité de portée*, deux définitions appartenant à la même *catégorie d'entités* ne peuvent avoir le même *nom*. [2.2.2]

règles de bonne formation *(E: well-formedness rules)*

Les *règles de bonne formation* sont des contraintes qui portent sur une *syntaxe abstraite* ou une *syntaxe concrète* imposant des conditions statiques non directement exprimées par les *règles syntaxiques*. [1.4.1, 1.4.2]