UNION INTERNATIONALE DES TÉLÉCOMMUNICATIONS

CCITT

COMITÉ CONSULTATIF
INTERNATIONAL
TÉLÉGRAPHIQUE ET TÉLÉPHONIQUE

Z.100

(11/1988)

SÉRIE Z: LANGAGES ET ASPECTS INFORMATIQUES GENERAUX DES SYSTEMES DE TELECOMMUNICATION

Langage de spécification et de description fonctionnelles (LDS)
Critères d'utilisation des techniques de description formelles (TDF)

LANGAGE DE DESCRIPTION ET DE SPÉCIFICATION (LDS)

Réédition de la Recommandation du CCITT Z.100, publiée dans le Livre Bleu, Fascicule X.1 (1988)

NOTES

1 La Recommandation Z.100 du CCITT a été publiée dans le fascicule X.1 du Livre Bleu. Ce fichier est un extrait du Livre Bleu. La présentation peut en être légèrement différente, mais le contenu est identique à celui du Livre Bleu et les conditions en matière de droits d'auteur restent inchangées (voir plus loin).

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

LANGAGE DE DESCRIPTION ET DE SPÉCIFICATION (LDS)

SOMMAIRE

Page

REMARQUE PRÉLIMINAIRE

La présente Recommandation remplace les Recommandations Z.100 à Z.104 et la Recommandation X.250 du Livre rouge du CCITT.

1 Introduction au LDS

1.1 *Introduction*

Le but poursuivi, en recommandant l'utilisation du LDS (langage de description et de spécification fonctionnelles), 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 LDS 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.

Remarque – Etant donné qu'il n'est pas fait de distinction entre l'utilisation du LDS 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.

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 LDS 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.

1.1.1 *Objectifs*

Les objectifs généraux qui ont été pris en compte lors de la définition du LDS 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 l'une quelconque de ces méthodologies.

1.1.2 *Domaine d'application*

Le domaine d'application principal du LDS 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.

Il va de soi que le LDS 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 LDS est un langage particulièrement riche qui peut êtte 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 LDS 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 LDS 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) concevoir un système (à la fois globalement et dans le détail),

g) effectuer des essais d'un système,

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

1.1.3 *Spécification d'un système*

Le LDS 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 LDS 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 LDS*

Le LDS offre le choix entre deux formes syntaxiques différentes pour représenter un système: une représentation graphique (LDS/GR) et une représentation textuelle (LDS/PR). Comme ces formes sont toutes les deux des représentations concrètes de la même sémantique du LDS, 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 LDS/PR est commun avec le LDS/GR. Ce sous-ensemble est appelé grammaire textuelle commune.

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

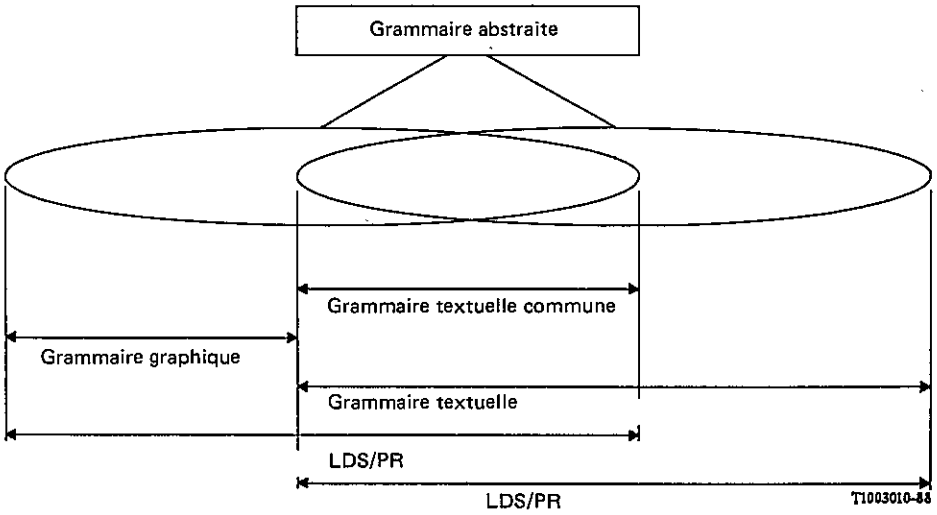


FIGURE 1.1

Grammaires du LDS

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 LDS 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 LDS/PR et le LDS/GR.

On dispose également d'une définition formelle du LDS qui définit la manière de transformer la spécification d'un système dans la syntaxe abstraite et comment interpréter une spécification, donnée en termes de syntaxe abstraite.

1.3 *Définitions fondamentales*

La présente Recommandation fait appel à des conventions et à des concepts généraux dont les définitions sont données ci-après:

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.

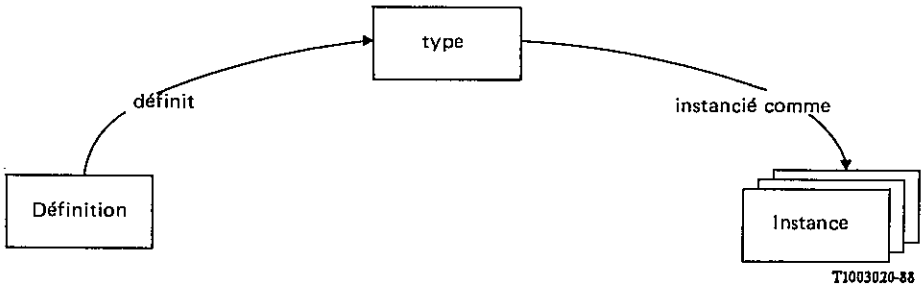


FIGURE 1.2

Concept de type

Un type se définit par des définitions, lesquelles définissent toutes les propriétés associées à ce type. Un type peut être décomposé en un nombre quelconque d'instances. Toute instance d'un type particulier possède toutes les propriétés définies pour ce type.

Ce schéma s'applique à plusieurs concepts du LDS, c'est-à-dire qu'il existe des définitions de système et des instances de système, des définitions de processus et des instances de processus.

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

Remarque – 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 LDS 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 le § 5.6.10).

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

1.3.3 *Erreurs*

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

Lorsqu'une spécification LDS 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*

Les § 2, 3, 4 et 5 de la présente Recommandation sont structurés par thèmes, ils 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 LDS/PR et le LDS/GR et la grammaire uniquement utilisée pour le LDS/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 au § 2.2.4).

d) *Sémantique* – donnant une signification à un type, confère les propriétés de ce type, la façon avec laquelle une instance de ce type est interprétée et toutes conditions dynamiques qui doivent être remplies par l'instance de ce type pour avoir un comportement correct au sens du LDS.

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 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 de la présente section 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 au § 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 LDS 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 au § 2.1 de la Recommandation Z.200 (voir également le § 1.5.2).

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

Dans de nombreux cas, il y a une 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 LDS) 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 LDS (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 au § 1.5.3.

La syntaxe graphique est suivie par des paragraphes définissant les conditions statiques qui doivent être satisfaites dans une LDS/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.

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 LDS (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, <diagramme de système> dans la grammaire concrète correspond à la *Définition-de-système* dans la grammaire abstraite.

L'expansion dans la syntaxe concrète provenant de constructions telles les définitions différées (§ 2.4.1), les macros (§ 4.2) et les mises en correspondance de littéraux (§ 5.4.1.15) etc., doit être examinée avant la mise en correspondance entre la syntaxe concrète et la syntaxe abstraite.

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 nœud-d'état, tous les *identificateurs de signaux d'entrée* (dans l'ensemble des signaux d'entrée valides) apparaissent soit dans un *ensemble de signaux mis en réserve* ou dans un *nœud d'entrée*».

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 la section générale traitant des identités au § 2. Par conséquent, cela n'est pas habituellement mentionné comme étant une propriété d'identité. Il n'est également pas nécessaire d'indiquer les sous-composantes d'une définition contenues par la définition étant donné que l'appartenance de telles sous-composantes est évidente à partir de la syntaxe abstraite. Par exemple, il est évident qu'une définition de bloc «a» englobé des 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 LDS 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 LDS».

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 le § 1.3.3).

Modèle

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

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 *exemples* contient des exemples.

1.5 *Métalangages*

Pour la définition des propriétés et des syntaxes du LDS, 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 LDS.

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 variable est

Définition-de-variable :: *Nom-de-variable Identificateur-de-référence-de-sorte*

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

La définition en Méta IV

Identificateur-de-référence-de-sorte = *Identificateur*

indique qu'un *identificateur-de-référence-de-sorte* est un *identificateur* 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 LDS, ces objets sont:

a) Des objets entiers

Exemple:

Nombre-d'instances :: *entier entier*

Le terme *nombre-d'instances* désigne un domaine composite contenant deux des valeurs entières (*entier*) indiquant le nombre initial et le nombre maximal d'instances.

b) Mots clés

Les mots clés sont représentés par une séquence en caractères gras de majuscules et de chiffres.

Exemple:

Processus-de-destination = *Identificateur-de-processus* | **ENVIRONNEMENT**

Le *processus-de-destination* est soit un *identificateur-de-processus* soit l'environnement qui est désigné par le mot clé **ENVIRONNEMENT**.

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:

Nom :: *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:

Texte-informel :: ...

Texte-informel contient un objet qui n'est pas interprété.

Les opérateurs ci-après (constructeurs) dans la forme BNF (voir le § 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:

graphe-de-processus :: *nœud-de-départ-de-processus nœud-d'état*-**set**

Un *graphe-de-processus* est constitué par un *nœud-de-départ-de-processus* et d'un ensemble de *nœud-de-processus.*

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 <nom> et <chaîne de caractères>. Notons que les deux terminaux spéciaux <nom> et <chaîne de caractères> 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 <chaîne de caractères> ou <nom>. 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:

<expression de visibilité> ::=

VIEW (<identificateur de variable>, <expression>)

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 <expression de visibilité>, <identificateur de variable> et <expression> dans l'exemple ci-dessus sont des symboles non-terminaux; VIEW, les parenthèses et le point 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 <identificateur de variable> est syntaxiquement identique à <identificateur>, mais sur le plan sémantique, il spécifie que l'identificateur doit être un identificateur de variable.

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

<zone de bloc> ::=

<référence de bloc graphique>

| <diagramme de bloc>

indique qu'une <zone de bloc> est soit une <référence de bloc graphique> ou un <diagramme de bloc>.

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

<zone d'interaction de bloc> ::=

{<zone de bloc> | <zone de définition de canal>}+

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 d'interaction de bloc> contient au moins une <zone de bloc> ou une <zone de définition de canal> et peut contenir plusieurs autres <zones de bloc> et <zones de définition de canal>.

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

<en-tête de processus> ::=

PROCESS <identificateur de processus> [<paramètres formels>]

indique qu'un <en-tête de processus> peut, mais pas nécessairement, contenir des <paramètres formels>.

1.5.3 *Métalangage applicable à la grammaire graphique*

En ce qui concerne la grammaire graphique, le métalangage décrit au § 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:

{{<zone de texte de système>}* {<diagramme de macro>}* <zone d'interaction de blocs>} **set**

est un ensemble de zéro, d'une ou de plusieurs <zone de texte de système>; de zéro, d'un ou plusieurs <diagramme de macro> et une <zone d'interation de blocs>.

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 nom terminal ayant le mot «symbole» placé immédiatement avant le signe <.

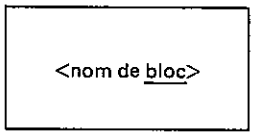
Le métasymbole **contains** indique que son argument de droite doit être placé à l'intérieur de son argument de gauche et, le cas échéant, à l'intérieur du <symbole d'extension de texte> associé. Par exemple:

<référence de bloc graphique> ::=

<symbole de bloc> **contains** <nom de bloc>

<symbole de bloc> ::=

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.

2 Le LDS de base

2.1 *Introduction*

Un système LDS 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.

Le § 2 a été subdivisé en huit principaux sujets:

a) *Règles générales*

Les concepts de base du LDS 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 en LDS telles les valeurs, les variables, les expressions.

c) *Structure des systèmes*

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

d) *Communication*

Décrit les mécanismes de communication utilisés dans le LDS 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, mise en réserve, étiquette, transition.

f) *Action*

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

g) *Temporisateurs*

Définition des temporisateurs et primitives des temporisateurs.

h) *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.

```
<unité lexicale> ::=
        <nom>
    |   <chaîne de caractères>
    |   <spécial>
    |   <spécial composite>
    |   <note>
    |   <mot clé>

<nom> ::=
        <mot> { <souligné> <mot> }*

<mot> ::=
        {<alphanumérique> | <point>}*
        <alphanumérique>
        {<alphanumérique> | <point> }*

<alphanumérique> ::=
        <lettre>
    |   <chiffre décimal>
    |   <national>

<lettre> ::=
        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
```

<chiffre décimal> ::=
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<national> ::=
#
| `
| ¤
| @
| <crochet gauche>
| \
| <crochet droit>
| <accolade gauche>
| <ligne verticale>
| <accolade droite>
| <tilde>
| <pointe de flèche vers le haut>

<crochet gauche> ::=
[

<crochet droit> ::=
]

<accolade gauche> ::=
{

<ligne verticale> ::=
|

<accolade droite> ::=
}

<tilde> ::=
~

<pointe de flèche vers le haut> ::=
^

<point> ::=
.

<souligné> ::=
_

<chaîne de caractères> ::=
<apostrophe> {<alphanumérique>
| <autre caractère>
| <spécial>
| <point>
| <souligné>
| <espace>
| <apostrophe> <apostrophe>}* <apostrophe>

<texte> ::=
{<alphanumérique>
| <autre caractère>
| <spécial>
| <point>
| <souligné>
| <espace>
| <apostrophe>}*

<apostrophe> ::=
'

<autre caractère> ::=
? | & | %

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

```
<spécial composite> ::=
          ==
        | ==>
        | /=
        | <=
        | >=
        | //
        | :=
        | =>
        | ->
        | (.
        | .)

<note> ::=
          /* <texte> */

<mot clé> ::=
          ACTIVE
        | ADDING
        | ALL
        | ALTERNATIVE
        | AND
        | AXIOMS
        | BLOCK
        | CALL
        | CHANNEL
        | COMMENT
        | CONNECT
        | CONSTANT
        | CONSTANTS
        | CREATE
        | DCL
        | DECISION
        | DEFAULT
        | ELSE
        | ENDALTERNATIVE
        | ENDBLOCK
        | ENDCHANNEL
        | ENDDECISION
        | ENDGENERATOR
        | ENDMACRO
        | ENDNEWTYPE
        | ENDPROCEDURE
        | ENDPROCESS
        | ENDREFINEMENT
        | ENDSELECT
        | ENDSERVICE
        | ENDSTATE
        | ENDSUBSTRUCTURE
        | ENDSYNTYPE
        | ENDSYSTEM
        | ENV
        | ERROR
        | EXPORT
        | EXPORTED
        | EXTERNAL
        | FI
        | FOR
        | FPAR
        | FROM
        | GENERATOR
        | IF
        | IMPORT
        | IMPORTED
        | IN
        | INHERITS
        | INPUT
        | JOIN
```

LITERAL
LITERALS
MACRO
MACRODEFINITION
MACROID
MAP
MOD
NAMECLASS
NEWTYPE
NEXTSTATE
NOT
NOW
OFFSPRING
OPERATOR
OPERATORS
OR
ORDERING
OUT
OUTPUT
PARENT
PRIORITY
PROCEDURE
PROCESS
PROVIDED
REFERENCED
REFINEMENT
REM
RESET
RETURN
REVEALED
REVERSE
SAVE
SELECT
SELF
SENDER
SERVICE
SET
SIGNAL
SIGNALLIST
SIGNALROUTE
SIGNALSET
SPELLING
START
STATE
STOP
STRUCT
SUBSTRUCTURE
SYNONYM
SYNTYPE
SYSTEM
TASK
THEN
TIMER
TO
TYPE
VIA
VIEW
VIEWED
WITH
XOR

<espace> représente le caractère espace de l'Alphabet n° 5 du CCITT.

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 definition des représentations nationales de ces caractères relève des organismes nationaux de normalisation.

Toutes les <lettre> sont toujours traitées comme des majuscules sauf celles qui sont à l'intérieur d'une <chaîne de caractères>. (Le traitement des <national> peut être défini par les organismes nationaux de normalisation.)

Une <unité lexicale> se termine par le premier caractère qui ne peut pas faire partie de l'<unité lexicale> conformément à la syntaxe spécifiée ci-dessus. Lorsqu'un caractère <souligné> est suivi d'un ou plusieurs caractères de commande (les caractères de commande sont définis comme dans la Recommandation T.50) ou espaces, tous ces caractères (y compris le <souligné>) sont ignorés, par exemple A_B correspond au même <nom> que AB. Cet emploi de <souligné> permet de répartir des <unité lexicale> sur plus d'une ligne.

Lorsqu'un caractère <souligné> est suivi d'un <mot> dans un <nom>, il est autorisé à spécifier un ou plusieurs caractères de commande ou espaces, au lieu du caractère <souligné>, pour autant que l'un des <mot> englobant le caractère <souligné> ne forme pas un <mot clé>, par exemple A B désigne le même <nom> que A_B.

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

1) les <souligné> dans le <nom> dans un <élément de trajet> doivent être spécifiés explicitement;

2) lorsqu'un ou plusieurs <nom> ou <identificateur> peuvent être suivis directement par une <sorte> (par exemple, <définition de variable>, <définition de vue>), les <souligné> dans ces <nom> ou <identificateur> doivent être spécifiés explicitement;

3) lorsqu'une <définition de données> contient des <instantiations de générateur>, les <souligné> du <nom de sorte> suivant le mot clé NEWTYPE doivent être spécifiés explicitement.

Un caractère de commande a la même signification qu'un espace.

Les caractères de commande et les espaces peuvent apparaître un nombre quelconque de fois entre deux <unité lexicale>. Un nombre quelconque de caractères de contrôle et d'espaces entre deux <unité lexicale> ont la même signification qu'un espace.

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 lexicale>.

Des règles lexicales particulières s'appliquent dans un <corps de macro> (voir le § 4.2.1).

2.2.2 *Règles de visibilité et identificateurs*

Grammaire abstraite

Identificateur	::	*Qualificatif-Nom*
Qualificatif	=	*Elément-de-chemin* +
Elément-de-chemin	=	*Qualificatif-de-système* |
		Qualificatif-de-bloc |
		Qualificatif-de-sous-structure-de-bloc |
		Qualificatif-de-signal |
		Qualificatif-de-processus |
		Qualificatif-de-procédure |
		Qualificatif-de-sorte |
Qualificatif-de-système	::	*Nom-de-système*
Qualificatif-de-bloc	::	*Nom-de-bloc*
Qualificatif-de-sous-structure-de-bloc	::	*Nom-de-sous-structure-de-bloc*
Qualificatif-de-processus	::	*Nom-de-processus*
Qualificatif-de-procédure	::	*Nom-de-procédure*
Qualificatif-de-signal	::	*Nom-de-signal*
Qualificatif-de-sorte	::	*Nom-de-sorte*
Nom	::	*Token*

Grammaire textuelle concrète

<identificateur> ::=
[<qualificatif>] <nom>

<qualificatif> ::=
<élément de chemin> {/<élément de chemin>}*

<élément de chemin> ::=
<classe d'unité de portée> <nom>

<classe d'unité de portée> ::=
SYSTEM
| BLOCK
| SUBSTRUCTURE
| SIGNAL
| PROCESS
| PROCEDURE
| TYPE
| SERVICE

Il n'y a pas de syntaxe abstraite correspondante pour la <classe d'unité de portée> indiquée par service. Les <nom> et <identificateur> d'entités définies dans une <définition de service> sont transformés en <nom> uniques ou en <identificateur> uniques définis dans la <définition de processus> contenant la <définition de service>.

Le <qualificatif> reflète la structure hiérarchique à partir du niveau du système vers le contexte de définition, et de manière telle que le niveau du système est la partie textuelle la plus à gauche. Toutefois, on peut omettre certains <élément de chemin> les plus à gauche, lorsque le premier <élément de chemin> restant le plus à gauche dans le <qualificatif> est unique à l'intérieur de toute la <définition de système>.

Il est permis d'omettre certains <élément de chemin> les plus à gauche (sauf pour les <définition différée>, voir le § 2.4.1) ou bien tout le <qualificatif>. Lorsque tout le <qualificatif> est omis et que le <nom> 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 <nom> avec une définition doit pouvoir être résolue par le contexte réel. Dans d'autres cas, l'<identificateur> est associé à une entité qui a son contexte de définition dans l'unité de portée englobante la plus proche dans laquelle le <qualificatif> de l'<identificateur> est le même que la partie la plus à droite du <qualificatif> complet désignant cette unité de portée. Si l'<identificateur> ne contient pas de <qualificatif>, la nécessité de mise en correspondance des <qualificatif> est omise.

Un sous-signal doit être qualifié par ses signaux parents, à moins qu'aucun autre signal visible n'existe à cet endroit qui porte le même <nom>.

La résolution par contexte est possible dans les cas suivants:

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

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

Seuls les identificateurs visibles peuvent être utilisés, à l'exception de l'<identificateur de variable> dans une <définition de visibilité> et de l'<identificateur> utilisé à la place d'un <nom> dans une définition référencée (c'est-à-dire une définition extraite de la <définition de système>).

Sémantique

Les unités de portée sont définies par le schéma suivant:

Grammaire textuelle concrète	*Grammaire graphique concrète*
<définition de système>	<diagramme de système>
<définition de bloc>	<diagramme de bloc>
<définition de processus>	<diagramme de processus>
<définition de procédure>	<diagramme de procédure>
<définition de sous-structure de bloc>	<diagramme de sous-structure de bloc>
<définition de sous-structure de canal>	<diagramme de sous-structure de canal>
<définition de service>	<diagramme de service>
<définition partielle de type>	
<affinage de signal>	

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. Pour une <définition partielle de type>, la liste de définitions associée comprend les <signature d'opérateur>, les <signature de littéral> et toutes <signature d'opérateur> et <signature de littéral> provenant d'une sorte parente, d'un générateur d'instance ou imposé par l'utilisation d'abréviations tel le mot clé ORDERING (voir le § 5.4.1.8). Il faut remarquer qu'une <définition de visibilité> ne définit pas une entité.

Bien que les <opérateur infixé>, les <opérateur> avec un point d'exclamation et les <chaîne de caractères> aient leur propre notation syntaxique, ils sont en réalité des <nom>, ils sont représentés dans la syntaxe abstraite représentée par un nom. Dans ce qui suit, ils sont étudiés comme s'ils étaient (syntaxiquement aussi) des <nom>. Toutefois, les <nom d'état>, les <nom de connecteur>, les <nom formel de générateur>, les <identificateur de valeur> dans les équations, les <nom formel de macro> et les <nom de macro> ont des règles de visibilité particulières et ne peuvent par conséquent être qualifiés. Leurs règles de visibilité sont expliquées dans les sections concernées.

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

Le <qualificatif> dans un <identificateur> spécifie uniquement le contexte de définition du <nom>.

Les classes d'entités sont les suivantes:

a) système
b) blocs
c) canal, acheminement de signaux
d) signaux, temporisateurs
e) processus
f) procédures
g) variables (y compris les paramètres formels), synonymes, littéraux, opérateurs
h) sortes
i) générateurs
j) entités importées
k) listes de signaux
l) services
m) sous-structures de bloc, sous-structures de canal

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

a) si la partie nom de l'<identificateur> a son contexte de définition dans cette unité de portée, 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 <définition partielle de type> dans laquelle l'<identificateur> est défini, ou
d) si l'unité de portée contient une <définition de signal> dans laquelle l'<identificateur> se trouve défini.

On ne peut avoir deux définitions dans la même unité de portée et appartenant à la même classe d'entités portant le même <nom>. Il existe cependant une exception: les définitions de <signature d'opérateur> et de <signature de littéral> dans la même <définition partielle de type> (voir le § 5.2.2): plusieurs opérateurs et littéraux peuvent avoir le même <nom> avec différentes <sorte d'argument> ou différentes sortes de <résultat>.

On notera une autre exception: les entités importées. Pour cette classe, les paires de (<nom d'import>, <sorte>) dans <définition d'import> dans l'unité de portée doivent être distinctes.

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

Texte informel :: ...

Grammaire textuelle concrète

<texte informel> ::=
<chaîne de caractères>

Sémantique

Lorsqu'un texte informel est utilisé dans une spécification LDS de système, il signifie que le texte n'est pas en LDS formel, c'est-à-dire que le LDS ne donne 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 <symbole de canal>, le <symbole d'acheminement de signal>, le <symbole de création de ligne>, le <symbole de ligne de flot>, le <symbole d'association continu> et le <symbole d'association pointillé>, qui peuvent se couper. Il n'existe pas d'association logique entre les symboles qui se coupent.

Le métasymbole **is followed by** implique un <symbole de ligne de flot>.

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

Les flèches sont nécessaires chaque fois qu'un <symbole de ligne de flot> entre dans un autre <symbole de ligne de flot>, dans un <symbole de connecteur de sortie> ou dans un <symbole d'état suivant>. Dans d'autres cas, ces flèches sont facultatives sur les <symbole de ligne de flot>. Les <symbole de ligne de flot> sont horizontaux ou verticaux.

On peut utiliser des images symétriques verticales des <symbole d'entrée>, des <symbole de sortie>, des <symbole de commentaire> et des <symbole d'extension de texte>.

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> ::=
<symbole de cadre> **contains**
{<zone d'en-tête> <zone de numéro de page>
{<unité syntaxique>} *}

<zone d'en-tête> ::=
<symbole implicite de texte> **contains** <en-tête>

<zone de numéro de page> ::=
<symbole implicite de texte> **contains** [<numéro de page> [(<nombre de pages>)]]

<numéro de page> ::=
<nom de littéral>

<nombre de pages> ::=

<nom de littéral de naturel>

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 <symbole de cadre> délimitant le diagramme et l'<en-tête> du diagramme est remplacé par un <symbole de cadre> et un <en-tête> pour chaque <page>.

L'utilisateur du LDS peut choisir les <symbole de cadre> imposés par la limite du support sur lequel les diagrammes sont reproduits.

Afin d'avoir une séparation nette entre la <zone d'en-tête> et la <zone de numéro de page>, le <symbole de texte implicite> n'est pas matérialisé, il est sous-entendu. La <zone d'en-tête> est placée au coin supérieur gauche du <symbole de cadre>, la <zone de numéro de page> est située au coin supérieur droit du <symbole de cadre>. L'<en-tête> et l'<unité syntaxique> dépendent du type de diagramme.

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> définie au § 2.2.1.

Des exemples sont donnés aux figures 2.9.1 et 2.9.3.

La syntaxe concrète de la deuxième forme est:

<fin> ::=
[<commentaire>];

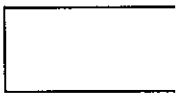
<commentaire> ::=
COMMENT <chaîne de caractères>

Un exemple est donné à la figure 2.9.2.

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

<zone de commentaire> ::=
<symbole de commentaire> **contains** <texte>
is connected to <symbole d'association pointillé>

<symbole de commentaire> ::=



<symbole d'association pointillé> ::=

Une des extrémités du <symbole d'association pointillé> doit être reliée au milieu du segment vertical du <symbole de commentaire>.

Un <symbole de commentaire> peut être relié à tout symbole graphique au moyen d'un <symbole d'association pointillé>. Le <symbole de commentaire> 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.

Un exemple est donné à la figure 2.9.4 dans le § 2.9.

2.2.7 *Extension de texte*

<zone d'extension de texte> ::=
<symbole d'extension de texte> **contains** <texte>
is connected to <symbole d'association continu>

<symbole d'extension de texte> ::=
<symbole de commentaire>

<symbole d'association continu> ::=

Une des extrémités du <symbole d'association continu> doit être reliée au milieu du segment vertical du <symbole d'extension de texte>.

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

Le texte contenu dans le <symbole d'extension de texte> 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 <symbole de texte> est utilisé dans tout <diagramme>. Le contenu dépend du diagramme.

<symbole de texte> ::=

2.3 *Concepts de base concernant les données*

Le concept de données dans le LDS est exposé dans le détail au § 5; plus précisément en ce qui concerne la terminologie du LDS pour ce qui est des données et la facilité à définir des nouveaux types de données et les facilités concernant les données prédéfinies.

Les données apparaissent dans les définitions du type de données, les expressions, l'application d'opérateurs, les variables, les valeurs et les littéraux.

2.3.1 *Définitions des types de données*

Les données dans le LDS sont principalement abordées sous l'aspect type de données. Un type de données 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 du type de données. Ces propriétés sont définies par les définitions de types de données.

Le LDS permet la définition de tout type de données 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 types de données 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.

Toutes les sortes ont au moins une valeur. 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, 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 peut être interprétée comme différentes valeurs durant l'interprétation d'un système LDS selon la valeur associée aux variables.

2.4 *Structure du système*

2.4.1 *Définitions différées*

Une <définition différée> est une définition qui a été supprimée de son contexte de définition pour accroître la visibilité. Elle est analogue à une définition de macro (voir le § 4.2), mais elle est «appelée» d'un seul endroit exactement (le contexte de définition) en utilisant une référence.

Grammaire concrète

<définition différée> ::=
<définition> | <diagramme>

<définition de système> ::=
{<définition textuelle de système> | <diagramme de système>}
{<définition différée>}*

<définition> ::=
<définition de bloc>
| <définition de processus>
| <définition de procédure>
| <définition de sous-structure de bloc>
| <définition de sous-structure de canal>
| <définition de service>
| <définition de macro>

<diagramme> ::=
<diagramme de bloc>
| <diagramme de processus>
| <diagramme de procédure>
| <diagramme de sous-structure de bloc>
| <diagramme de sous-structure de canal>
| <diagramme de service>
| <diagramme de macro>

Pour chaque <définition différée>, sauf pour des <définition de macro> et des <diagramme de macro>, on doit avoir une référence dans la <définition de système>, le <diagramme de système>, ou une autre <définition différée>.

Pour chaque référence on doit avoir une <définition différée> correspondante.

Dans chaque <définition différée>, on doit avoir un <identificateur> placé immédiatement après le mot clé initial. Dans cet <identificateur>, le <qualificatif> doit être soit complet, soit omis. Si le <qualificatif> est omis, le <nom> doit être unique dans la définition de système, à l'intérieur de la classe d'entité pour la <définition différée>. On ne peut pas spécifier un qualificatif dans l'<identificateur> après le mot clé initial pour les définitions qui ne sont pas des <définition différée> (c'est-à-dire qu'un <nom> doit être spécifié pour les définitions normales).

Sémantique

Avant de pouvoir analyser une <définition de système concret>, chaque référence doit être remplacée par la <définition différée> correspondante. Dans cette substitution, l'<identificateur> de la <définition différée> est remplacé par le <nom> dans la référence.

2.4.2 *Système*

Grammaire abstraite

Définition-de-système	::	*Nom-de-système* *Définition-de-bloc*-**set** *Définition-de-canal*-**set** *Définition-de-signal*-**set** *Définition-de-type-de-données* *Définition-de-type-de-synonyme*-**set**
Nom-de-système	=	*Nom*

Une *définition-de-système* a un nom qui peut être utilisé dans les qualificatifs.

La *définition-de-système* doit au moins contenir une *définition-de-bloc.*

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-de-système.* Toutes les données prédéfinies sont considérées comme étant définies au niveau du système.

Grammaire textuelle concrète

```
<définition textuelle de système> ::=
                    SYSTEM <nom de système> <fin>
                    {<définition de bloc>
                    |    <référence textuelle de bloc>
                    |    <définition de canal>
                    |    <définition de signal>
                    |    <définition de liste de signaux>
                    |    <définition de sélection>
                    |    <définition de macro>
                    |    <définition de données>} +
                              ENDSYSTEM [<nom de système>] <fin>

<référence textuelle de bloc>    ::=
                    BLOCK <nom de bloc> REFERENCED <fin>
```

<définition de sélection> est définie au § 4.3.3, <définition de macro> au § 4.2, <définition de données> au § 5.5.1, <définition de bloc> au § 2.4.3, <définition de canal> au § 2.5.1, <définition de signal> au § 2.5.4, <définition de liste de signaux> au § 2.5.5.

Un exemple de <définition de système> est donné à la figure 2.9.5 du § 2.9.

Grammaire graphique concrète

```
<diagramme de système> ::=
                    <symbole de cadre> contains
                         {<en-tête de système>
                         {    {<zone texte de système>}*
                              {<diagramme de macro>}*
                               <zone interaction de bloc>} set }

<symbole de cadre> ::=
```

```
<en-tête de système> ::=
                    SYSTEM <nom de système>

<zone de texte de système> ::=
                    <symbole de texte> contains
                         {<définition de signal>
                         |    <définition de liste de signaux>
                         |    <définition de données>
                         |    <définition de macro>
                         |    <définition de sélection>}*
```

<zone interaction de bloc> ::=
{<zone de bloc>
| <zone de définition de canal>}+

<zone de bloc> ::=
<référence de bloc graphique>
| <diagramme de bloc>

<référence de bloc graphique> ::=
<symbole de bloc> **contains** <nom de bloc>

<symbole de bloc> ::=

<définition de sélection> est définie au § 4.3.3, <définition de macro> au § 4.2, <diagramme de macro> au § 4.2, <définition de données> au § 5.5.1, <diagramme de bloc> au § 2.4.3, <zone de définition de canal> au § 2.5.1, <définition de signal> au § 2.5.4, <définition de liste de signaux> au § 2.5.5.

La *définition-de-bloc*-**set** dans la *grammaire abstraite* correspond aux <zone de bloc>, la *définition-de-canal*-**set** à la <zone de définition de canal>.

Un exemple d'un <diagramme de système> est donné à la figure 2.9.6.

Sémantique

Une *définition-de-système* est la représentation en LDS 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 *définition-de-système*, on choisit un sous-ensemble cohérent (voir le § 3.2.1). Ce sous-ensemble est appelé une instance de la *définition-de-système*. Une instance de système est une instanciation d'un type de système défini par une *définition-de-système*. L'interprétation d'une instance d'une *définition-de-système* est réalisée par une machine abstraite LDS qui en conséquence donne la sémantique aux concepts du LDS. Pour interpréter une instance d'une *définition-de-système*, 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

Définition-de-bloc	::	*Nom-de-bloc* *Définition-de-processus*-**set** *Définition-de-signal*-**set** *Connexion-de-canal-vers-acheminement*-**set** *Définition-d'acheminement-de-signal*-**set** *Définition-de-type-de-données* *Définition-de-type-de-synonyme*-**set** [*Définition-de-sous-structure-de-bloc*]
Nom-de-bloc	=	*Nom*

A moins qu'une *définition-de-bloc* ne contienne une *définition-de-sous-structure-de-bloc*, on doit avoir au moins une *définition-de-processus* et une *définition-d'acheminement-de-signal* à l'intérieur du bloc.

Il est possible de subdiviser les blocs en spécifiant la *définition-de-sous-structure-de-bloc*; cet aspect du langage est étudié au § 3.2.2.

Grammaire textuelle concrète

<définition de bloc> ::=

BLOCK {<nom de bloc>|<identificateur de bloc>}<fin>
{<définition de signal>
| <définition de liste de signaux>
| <définition de processus>
| <référence textuelle de processus>
| <définition d'acheminement de signal>
| <définition de macro>
| <définition de données>
| <définition de sélection>
| <connexion de canal vers acheminement>}*
[<définition de sous-structure de bloc>|<référence textuelle de sous-structure de bloc>]
ENDBLOCK [<nom de bloc>|<identificateur de bloc>] <fin>

<référence textuelle de processus> ::=

PROCESS <nom de processus>[<nombre d'instances>] REFERENCED <fin>

<définition de signal> est définie au § 2.5.4, <définition de liste de signaux> au § 2.5.5, <définition de processus> au § 2.4.4, <définition d'acheminement de signal> au § 2.5.2, <connexion de canal vers acheminement> au § 2.5.3, <définition de sous-structure de bloc> au § 3.2.2, <référence de sous-structure de bloc textuelle> au § 3.2.2, <définition de macro> au § 4.2.2, <définition de données> au § 5.5.1.

Un exemple de <définition de bloc> est montré à la figure 2.9.7 au § 2.9.

Grammaire graphique concrète

<diagramme de bloc> ::=
<symbole de cadre>
contains {<en-tête de bloc>
{{<zone texte de bloc>}*{<diagramme de macro>}*
[<zone d'interaction de processus>] [<zone de sous-structure de bloc>]} **set**}
is associated with {<identificateur de canal>}*

L'<identificateur de canal> identifie un canal relié à un acheminement de signaux dans le <diagramme de bloc>. Il se trouve placé à l'extérieur du <symbole de cadre>, à proximité de l'extrémité du trajet du signal arrivant au <symbole de cadre>. Si le <diagramme de bloc> ne contient pas une <zone d'interaction de processus>, il doit alors contenir une <zone de sous-structure de bloc>.

<en-tête de bloc> ::=
BLOCK {<nom de bloc>|<identificateur de bloc>}

<zone texte de bloc> ::=
<zone texte de système>

<zone d'interaction de processus> ::=
{<zone de processus>
| <zone de ligne de création>
| <zone de définition d'acheminement de signal>}+

<zone de processus> ::=
<référence graphique de processus>|<diagramme de processus>

<référence graphique de processus> ::=
<symbole de processus> **contains** {<nom de processus>[<nombre d'instances>]}

<symbole de processus> ::=

Le terme <nombre d'instances> est défini au § 2.4.4.

<zone de ligne de création> ::=
<symbole de ligne de création>
is connected to {<zone de processus> <zone de processus>}

<symbole de ligne de création> ::=

La tête de flèche placée sur le <symbole de ligne de création> indique la <zone de processus> sur laquelle le processus de création a lieu.

Le <diagramme de processus> est défini au § 2.4.4, la <zone de définition d'acheminement de signal> au § 2.5.2, la <zone de sous-structure de bloc> au § 3.2.2, le <diagramme de macro> au § 4.2.2.

Un exemple de <diagramme de bloc> est donné à la figure 2.9.8 au § 2.9.

Sémantique

Une définition de bloc contient une ou plusieurs définitions d'un système et éventuellement une sous-structure de bloc. La définition de bloc permet de regrouper les processus qui tous ensemble assurent une certaine fonction, soit directement soit par l'intermédiaire d'une sous-structure de bloc.

Une définition de 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 processus initiaux dans le bloc.

2.4.4 *Processus*

Grammaire abstraite

Définition-de-processus	::	*Nom-de-processus* *Nombre-d'instances* *Paramètre-formel-de-processus* * *Définition-de-procédure*-**set** *Définition-de-signal*-**set** *Définition-de-type-de-signal* *Définition-de-type-de-synonyme*-**set** *Définition-de-variable*-**set** *Définition-de-visibilité*-**set** *Définition-de-temporisateur*-**set** *Graphe-de-processus*
Nombre-d'instances	::	*Entier Entier*
Nom-de-processus	=	*Nom*
Graphe-de-processus	::	*Nœud-de-départ-de-processus* *Nœud-d'état*-**set**
Paramètre-formel-de-processus	::	*Nom-de-variable* *Identificateur-de-référence-de-sorte*

Grammaire textuelle concrète

```
<définition de processus> ::=
                    PROCESS {<identificateur de processus>|<nom de processus>}
                        [<nombre d'instances>] <fin>
                            [<paramètres formels> <fin>] [<ensemble des signaux d'entrée
                            valides>]
                                {<définition de signal>
                                |  <définition de liste de signaux>
                                |  <définition de procédure>
                                |  <référence textuelle de procédure>
                                |  <définition de macro>
                                |  <définition de données>
                                |  <définition de variable>
                                |  <définition de visibilité>
                                |  <définition de sélection>
                                |  <définition d'import>
                                |  <définition de temporisateur>}*
                                        {<corps de processus>
                                        |  <décomposition de service>}
                    ENDPROCESS [<nom de processus>|<identificateur de processus>] <fin>

<référence textuelle de procédure> ::=
                    PROCEDURE <nom de procédure> REFERENCED <fin>

<ensemble de signaux d'entrée valides> ::=
                    SIGNALSET [<liste de signaux>] <fin>

<corps de processus> ::=
                <départ>{<état>}*
```

<paramètres formels> ::=
FPAR <nom de variable>{,<nom de variable>}*<sorte>
{,<nom de variable>{,<nom de variable>}*<sorte>}*

<nombre d'instances> ::=
([<nombre initial>], [<nombre maximal>])

<nombre initial> ::=
<expression simple de naturel>

<nombre maximal> ::=
<expression simple de naturel>

Le nombre initial d'instances et le nombre maximal d'intances contenus dans le nombre-d'instances sont tirés du <nombre d'instances>. Si le <nombre initial> est omis, le <nombre initial> est 1. Si le <nombre maximal> est omis, le <nombre maximal> n'est pas limité.

Le <nombre d'instances> utilisé dans la dérivation est le suivant:

a) s'il n'y a pas de <référence textuelle de processus> pour le processus, le <nombre d'instances> dans la <définition de processus> est utilisé. S'il ne contient pas un <nombre d'instances>, on utilise le <nombre d'instances> où le <nombre initial> et le <nombre maximal> sont omis;

b) si à la fois le <nombre d'instances> dans la <définition de processus> et le <nombre d'instances> dans une <référence textuelle de processus> sont omis, on utilise le <nombre d'instances> où sont omis à la fois le <nombre initial> et le <nombre maximal>;

c) si soit le <nombre d'instances> dans la <définition de processus> soit le <nombre d'instances> dans une <référence textuelle de processus> est omis, le <nombre d'instances> est celui qui est présent;

d) si à la fois le <nombre d'instances> dans la <définition de processus> et le <nombre d'instances> dans une <référence textuelle de processus> sont spécifiés, les deux <nombre d'instances> doivent être égaux lexicalement et ce <nombre d'instances> est utilisé.

Une relation analogue s'applique au <nombre d'instances> spécifié dans le <diagramme de processus> et dans la <référence textuelle de processus> définis ci-dessous.

<définition de signal> est définie au § 2.5.4, <définition de liste de signaux> au § 2.5.5, <définition de visibilité> au § 2.6.1.2, <définition de variable> au § 2.6.1.1, <définition de procédure> au § 2.4.5, <définition de temporisateur> au § 2.8, <définition de macro> au § 4.2.2, <définition d'import> au § 4.1.3, <définition de sélection> au § 4.3.3, <expression simple> au § 4.3.2, <décomposition de service> au § 4.10.1, <définition de données> au § 5.5.1.

Le <nombre initial> d'instances doit être inférieur ou égal au <nombre maximal> et le <nombre maximal> doit être supérieur à zéro.

L'utilisation du terme <ensemble des signaux d'entrée valides> est précisée au § 2.5.2, *modèle.*

Un exemple de <définition de processus> est donné à la figure 2.9.9 du § 2.9.

Grammaire graphique concrète

<diagramme de processus> ::=
<symbole de cadre>
contains {<en-tête de processus>
{{<zone de texte de processus>}*
{<zone de processus>}*
{<diagramme de macro>}*
{<zone graphique de processus>|<zone d'interaction de service>}} **set**}
[is associated with {<identificateur d'acheminement de signal>}+]

L'<identificateur d'acheminement de signal> identifie le trajet externe d'un signal relié à un trajet de signal dans le <diagramme de processus>. Il est placé à l'extérieur du <symbole de cadre> à proximité de l'extrémité du trajet du signal au <symbole de cadre>.

<zone de texte de processus> ::=
<symbole de texte> **contains** {
[<ensemble des signaux d'entrée valides>]
{<définition de signal>
| <définition de liste de signaux>
| <définition de variable>
| <définition de visibilité>
| <définition d'import>
| <définition de données>
| <définition de macro>
| <définition de temporisateur>
| <définition de sélection>}*}

<en-tête de processus> ::=
PROCESS {<nom de processus>|<identificateur de processus>}
[<nombre d'instances> [<fin>]]
[<paramètres formels>]

<zone graphique de processus> ::=
<zone de départ>{<zone d'état>|<zone de connecteur d'entrée>}*

<définition de signal> est définie au § 2.5.4, <définition de liste de signaux> au § 2.5.5, <définition de visibilité> au § 2.6.1.2, <définition de variable> au § 2.6.1.1, <zone de procédure> au § 2.4.5, <définition de temporisateur> au § 2.8, <définition de macro> et <diagramme de macro> au § 4.2.2, <définition d'import> au § 4.1.3, <définition de sélection> au § 4.3.3, <définition de données> au § 5.5.1, <zone de départ> au § 2.6.2, <zone d'état> au § 2.6.3, <zone de connecteur d'entrée> au § 2.6.6 et <zone d'interaction de service> au § 4.10.1.

Un exemple de <diagramme de processus> est donné à la figure 2.9.10 du § 2.9.

Sémantique

Une définition de processus introduit le type d'un processus qui est destiné à représenter un comportement dynamique.

Dans le *nombre-d'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 en ce sens que l'état résultant d'une transition, à côté du signal provoquant la transition, peut être affecté par des décisions prises à partir de variables connues du processus.

Plusieurs instances du même type de processus peuvent exister au même instant et s'exécuter de manière asynchrone et en parallèle les unes aux autres, et avec d'autres instances de différents types de processus dans le système.

Lorsqu'un système est créé, les processus initiaux sont créés dans un ordre aléatoire. 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 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 aux 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 au processus, de l'<ensemble de signaux d'entrée valides>, des signaux implicites 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 le § 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 mise en réserve est extrait de la file d'attente, et traité par le processus.

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 auxquelles on affecte des valeurs.

Les paramères 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 ne sont pas initialisés), soit lorsque l'instance de processus est dynamiquement créée.

Pour toutes les instances de processus, on peut utiliser quatre expressions produisant un PId (voir le § 5.6.10): 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 le processus (OFFSPRING);

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

Ces expressions sont expliquées dans le détail au § 5.5.4.3.

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 *Procédure*

Les procédures sont définies au moyen de définitions de procédure. On fait appel à une procédure au moyen d'un appel de procédure faisant référence à la définition de procédure. Des paramètres sont associés à un appel de procédure: on les emploie à la fois pour fournir les valeurs et pour limiter la portée des variables pour l'exécution de la procédure. C'est du mécanisme de transfert des paramètres que dépendent les variables affectées par l'interprétation d'une procédure.

Grammaire abstraite

Définition-de-procédure	::	*Nom-de-procédure* *Paramètre-formel-de-procédure* ***** *Définition-de-procédure*-**set** *Définition-de-type-de-données* *Définition-de-type-de-synonyme*-**set** *Définition-de-variable*-**set** *Graphe-de-procédure*
Nom-de-procédure	=	*Nom*
Paramètre-formel-de-procédure	=	*Paramètre-d'entrée* | *Paramètre-d'entrée-et-de-sortie*
Paramètre-d'entrée	::	*Nom-de-variable* *Identificateur-de-référence-de-sorte*

Paramètre-d'entrée-et-de-sortie :: *Nom-de-variable*
Identificateur-de-référence-de-sorte

Graphe-de-procédure :: *Nœud-de-départ-de-procédure*
Nœud-d'état-**set**

Nœud-de-départ-de-procédure :: *Transition*

Grammaire textuelle concrète

```
<définition de procédure> ::=
        PROCEDURE {<identificateur de procédure>|<nom de procédure>} <fin>
            [<paramètres formels de procédure> <fin>]
                {<définition de données>
                | <définition de variable>
                | <référence textuelle de procédure>
                | <définition de procédure>
                | <définition de sélection>
                | <définition de macro>}*
                    <corps de procédure>
        ENDPROCEDURE [<nom de procédure>|<identificateur de procédure>]
        <fin>

<paramètres formels de procédure> ::=
                    FPAR <paramètre formel de variable>
                        {,<paramètre formel de variable>}*

<paramètre formel de variable>::=
                    [IN/OUT
                | IN]
                <nom de variable>{,<nom de variable>}*<sorte>

<corps de procédure> ::=
                <corps de processus>
```

<définition de variable> est définie au § 2.6.1.1, <référence textuelle de procédure> au § 2.4.4, <définition de macro> au § 4.2, <définition de sélection> au § 4.3.3, <définition de données> au § 5.5.1, <sorte> au § 5.2.2.

Dans une <définition de procédure>, la <définition de variable> ne peut contenir les expressions REVEALED, EXPORTED, REVEALED EXPORTED, EXPORTED REVEALED <nom de variable> (voir le § 2.6.1). Un exemple de <définition de procédure> est donné à la figure 2.9.11.

Grammaire graphique concrète

```
<diagramme de procédure> ::=
        <symbole de cadre> contains {<en-tête de procédure>
            {{<zone de texte de procédure>
            | <zone de procédure>
            | <diagramme de macro>}*
                <zone de graphe de procédure>} set}

<zone de procédure> ::=
        <référence graphique de procédure>
        | <diagramme de procédure>

<zone de texte de procédure> ::=
        <symbole de texte> contains
            {<définition de variable>
            | <définition de données>
            | <définition de sélection>
            | <définition de macro>}*
```

<référence graphique de procédure> ::=
<symbole de procédure> **contains** <nom de procédure>

<symbole de procédure> ::=

<en-tête de procédure> ::=
PROCEDURE {<nom de procédure>|<identificateur de procédure>}
[<paramètres formels de procédure>]

<zone de graphe de procédure> ::=
<zone de départ de procédure>
{<zone de d'état>|<zone de connecteur d'entrée> }*

<zone de départ de procédure> ::=
<symbole de départ de procédure> **is followed by** <zone de transition>

<symbole de départ de procédure> ::=

<définition de variable> est définie au § 2.6.1.1, <zone de transition> au § 2.6.7.1, <zone d'état> au § 2.6.3, <zone de connecteur d'entrée> au § 2.6.6, <définition de macro> et <diagramme de macro> au § 4.2, <définition de sélection> au § 4.3.3, <définition de données> au § 5.5.1.

Un exemple de <diagramme de procédure> est montré à la figure 2.9.12 au § 2.9.

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.

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.

Lorsqu'une définition de procédure est interprétée, son graphe de procédure est également interprété.

Une définition de procédure est interprétée seulement lorsqu'une instance de processus l'appelle, et l'interprétation est faite par l'instance de processus.

L'interprétation d'une définition de procédure 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-d'entrée*, ayant le *nom* et la *sorte* du *paramètre-d'entrée*. A cette variable on affecte la valeur de l'expression donnée par le paramètre effectif correspondant, qui peut ne pas être défini;

b) lorsqu'un paramètre effectif est vide, la valeur attribuée au paramètre formel correspondant n'est pas définie;

c) un paramètre formel n'ayant pas d'attribut explicite, a un attribut IN implicite;

d) une variable locale est créée pour chaque *définition-de-variable* dans la *définition-de-procédure* ayant le *nom* et la *sorte* de la *définition-de-variable*;

e) chaque *paramètre-d'entrée-sortie* décrit un nom de synonyme pour la variable qui est donnée dans l'expression des paramètres effectifs. Ce nom de synonyme est utilisé tout au long de l'interprétation du *graphe-de-procédure* lorsqu'on se réfère à la valeur de la variable ou lorsque l'on affecte une nouvelle valeur à la variable;

f) la *transition* contenu dans le *nœud-de-départ-de-procédure* est interprétée.

2.5 *Communication*

2.5.1 *Canal*

Grammaire abstraite

Définition-de-canal	::	*Nom-de-canal* *Chemin-de-canal* [*Chemin-de-canal*]
Chemin-de-canal	::	*Bloc-d'origine* *Bloc-destinataire* *Identificateur-de-signal*-**set**
Bloc-d'origine	=	*Identificateur-de-bloc* | ENVIRONMENT
Bloc-destinataire	=	*Identificateur-de-bloc* | ENVIRONMENT
Identificateur-de-bloc	=	*Identificateur*
Identificateur-de-signal	=	*Identificateur*
Nom-de-canal	=	*Nom*

L'*identificateur-de-signal*-**set** doit contenir la liste de tous les signaux qui peuvent être acheminés sur le ou les chemins-de-canal définis par le canal.

Une extrémité du canal doit au moins être un bloc.

Si les deux points extrêmes sont des blocs, ceux-ci 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.

Grammaire textuelle concrète

```
<définition de canal> ::=
        CHANNEL <nom de canal>
            <trajet de canal>
            [<trajet de canal>]
                [<définition de sous-structure de canal>
            |    <référence textuelle de sous-structure de canal>]
        ENDCHANNEL [<nom de canal>] <fin>

<trajet de canal> ::=
        {   FROM <identificateur de bloc> TO <identificateur de bloc>
        |   FROM <identificateur de bloc> TO ENV
        |   FROM ENV TO <identificateur de bloc>}
        WITH <liste de signaux> <fin>
```

<liste de signaux> est définie au § 2.5.5, <définition de sous-structure de canal> et <référence textuelle de sous-structure de canal> au § 3.2.3.

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

Grammaire graphique concrète

<zone de définition de canal> ::=
<symbole de canal>
is associated with {<nom de canal>
{[{<identificateur de canal>|<identificateur de bloc>}]
<zone de liste de signaux>[<zone de liste de signaux>]} **set**}
is connected to {<zone de bloc>{<zone de bloc>|<symbole de cadre>}
[<zone d'association de sous-structure de canal>]} **set**

L'<identificateur de canal> identifie un canal externe relié au <diagramme de sous-structure de bloc> délimité par le <symbole de cadre>. L'<identificateur de bloc> identifie un bloc externe comme étant une extrémité de canal pour le <diagramme de sous-structure de canal> délimité par le <symbole de cadre>.

<symbole de canal> ::=
<symbole de canal 1>
| <symbole de canal 2>
| <symbole de canal 3>

<symbole de canal 1> ::=

<symbole de canal 2> ::=

<symbole de canal 3> ::=

<zone de liste de signaux> est définie au § 2.5.5, <zone de bloc> et <symbole de cadre> au § 2.4.1 et <zone d'association de sous-structure de canal> au § 3.2.3.

Pour chaque flèche placée sur le <symbole de canal>, il doit y avoir une <zone de liste de signaux>. Une <zone de liste de signaux> doit être suffisamment proche de la flèche à laquelle elle est associée pour éviter toute ambiguïté.

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.

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 peut retarder l'acheminement des signaux sur le canal. Cela signifie qu'une file d'attente du type premier-entré-premier-sorti (FIFO) 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.

2.5.2 *Acheminement du signal*

Grammaire abstraite

Définition-d'acheminement-de-signal	::	*Nom-d'acheminement-de-signal* *Trajet-d'acheminement-du-signal* [*Trajet-d'acheminement-du-signal*]
Trajet-d'acheminement-du-signal	::	*Processus-d'origine* *Processus-destinataire* *Identificateur-de-signal*-**set**
Processus-d'origine	=	*Identificateur-de-processus* | ENVIRONMENT
Processus-destinataire	=	*Identificateur-de-processus* | ENVIRONMENT
Nom-d'acheminement-de-signal	=	*Nom*

Un des points extrêmes au moins du *trajet-d'acheminement-du-signal* doit être un processus.

Si les deux points extrêmes sont des processus, les *identificateur-de-processus* doivent être différents.

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

Grammaire textuelle concrète

<définition d'acheminement de signal> ::=
SIGNALROUTE <nom d'acheminement de signal>
<trajet d'acheminement du signal>
[<trajet d'acheminement du signal>]

<trajet d'acheminement du signal> ::=
{ FROM <identificateur de processus> TO <identificateur de processus>
| FROM <identificateur de processus> TO ENV
| FROM ENV TO <identificateur de processus> }
WITH <liste de signaux> <fin>

La <liste de signaux> est définie au § 2.5.5.

Lorsque deux <trajet d'acheminement du signal> sont définis, l'un doit être en sens opposé à l'autre.

Grammaire graphique concrète

<zone de définition d'acheminement de signal> ::=
<symbole d'acheminement de signal>
is associated with {<nom d'acheminement de signal>
{[<identificateur de canal>] <zone de liste de signaux> [<zone de liste de signaux>]} **set**}
is connected to
{<zone de processus> {<zone de processus> | <symbole de cadre>}} **set**

<symbole d'acheminement de signal> ::=
<symbole d'acheminement de signal 1> | <symbole d'acheminement de signal 2>

<symbole d'acheminement de signal 1> ::=

<symbole d'acheminement de signal 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 d'acheminement de signal>, il doit exister une <zone de liste de signaux>. Une <zone de liste de signaux> doit être suffisamment proche de la flèche à laquelle elle est associée afin d'éviter toute ambiguïté.

Lorsque le <symbole d'acheminement de signal> est relié au <symbole de cadre>, l'<identificateur de canal> identifie un canal auquel l'acheminement de signal est relié.

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 processus ou entre un processus et son environnement.

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

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

Aucun acheminement de signal ne peut relier des instances de processus du même type. Si tel est le cas, l'interprétation du nœud-de-sortie a pour conséquence implicite que le signal est appliqué directement à l'accès d'entrée du processus de destination.

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

Modèle

Un <ensemble de signaux d'entrée valides> contient les signaux que le processus est autorisé à recevoir. Un <ensemble de signaux d'entrée valides> ne doit toutefois pas contenir de signaux de temporisateur. Si une <définition de bloc> contient des <définition d'acheminement de signal>, l'<ensemble de signaux d'entrée valides>, s'il y en a un, n'a pas besoin de contenir des signaux dans des acheminements de signaux conduisant au processus.

Si une <définition de bloc> ne contient aucune <définition d'acheminement de signal>, toutes les <définition de processus> dans cette <définition de bloc> doivent contenir un <ensemble de signaux d'entrée valides>. Dans ce cas, les <définition d'acheminement de signal> et les <connexion de canal à acheminement> sont tirées de l'<ensemble de signaux d'entrée valides>, des <sortie> et des canaux qui se terminent à la frontière des blocs. Les signaux correspondant à une direction donnée entre deux processus dans l'acheminement de signal implicite constituent l'intersection des signaux spécifiés dans l'<ensemble de signaux d'entrée valides> 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 dans la direction donnée.

2.5.3 *Connexion*

Grammaire abstraite

Connexion-de-canal-à-acheminement :: *Identificateur-de-canal*
Identificateur-d'acheminement-de-signal-**set**

Identificateur-d'acheminement-de-signal = *Identificateur*

D'autres constructions sont données au § 3.

Chaque *identificateur-de-canal* relié au bloc englobant doit être mentionné dans une seule *connexion-de-canal-à-acheminement*. L'*identificateur-de-canal* dans une *connexion-de-canal-à-acheminement* doit dénoter un canal connecté au bloc entre crochets.

Chaque *identificateur-d'acheminement-de-signal* dans une *connexion-de-canal-à-acheminement* doit être défini dans le même bloc où la *connexion-de-canal-à-acheminement* se trouve définie et doit avoir la frontière de ce bloc située à l'un de ses points extrêmes. Chaque *identificateur-d'acheminement-de-signal* 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 seule et unique *connexion-de-canal-à-acheminement*.

Pour une direction donnée, l'union des ensembles d'*identificateur de signal* dans les acheminements de signaux dans une *connexion-de-canal-à-acheminement* doit être égale aux signaux acheminés par l'*identificateur-de-canal* dans la même *connexion-de-canal-à-acheminement* et correspondant à la même direction.

Grammaire textuelle concrète

<connexion de canal à acheminement> ::=
CONNECT <identificateur de canal>
AND <identificateur d'acheminement de signal> {,<identificateur d'acheminement de signal>}* <fin>

On ne peut mentionner deux fois un <identificateur d'acheminement de signal dans une <connexion de canal à acheminement>.

Grammaire graphique concrète

Sur le plan graphique, l'élément connexion est représenté par l'<identificateur de canal> associé au trajet de signal et contenu dans la <zone de définition d'acheminement de signal> (voir le § 2.5.2 *grammaire graphique concrète*).

2.5.4 *Signal*

Grammaire abstraite

Définition-de-signal :: *Nom-de-signal*
Identificateur-de-référence-de-sorte *
[*Affinage-de-signal*]

Nom-de-signal :: *Nom*

L'*identificateur-de-référence-de-sorte* est défini au § 5.2.2.

Grammaire textuelle concrète

<définition de signal> ::=
SIGNAL {<nom de signal> [<liste de sorte>] [<affinage de signal>]}
{,<nom de signal> [<liste de sorte>] [<affinage de signal>]}* <fin>

<liste de sorte> ::=
(<sorte> {, <sorte>}*)

L'<affinage de signal> est défini au § 3.3, la <sorte> au § 5.2.2.

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.

On associe à chaque instance de signal deux valeurs de PId (voir le § 5.6.10) indiquant les processus de départ et de destination, l'<identificateur de signal> spécifié dans la sortie correspondante et les autres valeurs, dont les sortes sont définies dans la définition d'un signal.

2.5.5 *Définition de liste de signaux*

Un <identificateur de liste de signaux> peut être utilisé dans une <définition de canal>, une <définition d'acheminement de signal>, une <définition de liste de signaux>, un <ensemble de signaux d'entrée valides> et <liste de mise en réserve> comme moyen abrégé pour énumérer les identificateurs de signaux et les signaux de temporisation.

Grammaire textuelle concrète

<définition de liste de signaux> ::=
SIGNALLIST <nom de liste de signaux> = <liste de signaux> <fin>

<liste de signaux> ::=
<objet signal> {,<objet signal>}*

<objet signal> ::=
<identificateur de signal> | <identificateur de signal prioritaire> (<identificateur de liste de signaux>) | <identificateur de temporisation>

La <liste de signaux> qui est établie en remplaçant tous les <identificateur de liste de signaux> dans la liste par les <identificateur de signal> qu'ils décrivent, correspond à un *identificateur-de-signal-set* dans la grammaire abstraite. Dans chaque <liste de signaux> ainsi établie, chaque <identificateur de signal> doit être distinct.

Grammaire graphique concrète

<zone de liste de signaux> ::=
<symbole de liste de signaux> **contains** <liste de signaux>

<symbole de liste de signaux> ::=

2.6 *Comportement*

2.6.1 *Variables*

2.6.1.1 *Définition de variable*

Grammaire abstraite

Définition-de-variable	::	*Nom-de-variable* *Identificateur-de-référence-de-sorte* [REVEALED]
Nom-de-variable	=	*Nom*

Grammaire textuelle concrète

<définition de variable> ::=
DCL [REVEALED|EXPORTED|REVEALED EXPORTED|EXPORTED REVEALED]
<nom de variable> {,<nom de variable>}* <sorte> [:=<expression close>].
{,<nom de variable> {,<nom de variable>}* <sorte> [:=<expression close>]}* <fin>

La variable exportée est définie au § 4.13.

Sémantique

La semantique des variables est définie au § 2.3.2. La valeur d'une variable ne peut être modifiée que par son propriétaire. Le propriétaire d'une variable est le processus (ou la procédure) où la variable est déclarée. La valeur d'une variable n'est connue que de son usager à moins que la variable possède l'attribut REVEALED. L'attribut REVEALED permet à tous les autres processus du même bloc de voir la variable, à condition qu'ils aient été dotés de la définition de visibilité lors de la déclaration de cette variable.

Modèle

L'<expression close> dans une <définition de variable> ou une valeur par défaut dans une <sorte> n'ont pas de syntaxe abstraite correspondante. Il s'agit d'une syntaxe dérivée pour spécifier une séquence d'instructions d'affectation dans la transition initiale de l'unité de portée environnante. Les instructions d'affectation affectent l'<expresssion close> à tous les <nom de variable> figurant dans la <définition de variable>. Si à la fois une valeur par défaut dans une <sorte> et une <expression close> dans la <définition de variable> sont spécifiées, l'<expression close> dans la <définition de variable> s'applique.

2.6.1.2 *Définition de visibilité*

Grammaire abstraite

Définition de visibilité :: *Identificateur-de-variable*
Identificateur-de-référence-de-sorte

La *définition-de-variable* désignée par l'*Identificateur-de-variable* doit avoir l'attribut REVEALED, et elle doit être de la même sorte que l'*Identificateur-de-référence-de-sorte* décrit.

Grammaire textuelle concrète

<définition de visibilité> ::=
VIEWED
<identificateur de variable> {, <identificateur de variable>}* <sorte>
{,<identificateur de variable>{, <identificateur de variable>}* <sorte>}* <fin>

Le qualificatif dans l'<identificateur de variable> dans la <définition de visibilité> ne peut être omis que s'il existe une seule <définition de processus> dans le bloc dont la <définition de variable> définissant un <nom de variable> est la même que le <nom de variable> mentionné dans la <définition de visibilité> et qui a l'attribut REVEALED et qui est de la même <sorte> que celle indiquée par la <sorte> dans la <définition de visibilité>.

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

Nœud-de-départ-de-processus :: *Transition*

Grammaire textuelle concrète

<départ> ::=
START<fin> <transition>

Grammaire graphique concrète

<zone de départ> ::=
<symbole de départ> **is followed by** <zone de transition>

<symbole de départ> ::=

Sémantique

La *transition* du *nœud-de-départ-de-processus* est interprétée.

2.6.3 *Etat*

Grammaire abstraite

Nœud-d'état :: *Nom-d'état*
Ensemble-de-signaux-mis-en-réserve
*Nœud-d'entrée-***set**

Nom-d'état = *Nom*

Les *nœud-d'état* dans un *graphe-de-processus* ou un *graphe-de-procédure* ont des *nom-d'état* différents.

Pour chaque *nœud-d'état*, tous les *identificateur-de-signaux* (dans l'ensemble complet des signaux d'entrée valides) apparaissent soit dans un *ensemble-de-signaux-mis-en-réserve*, ou dans un *nœud-d'entrée*.

Les *identificateurs-de-signaux* dans le *nœud-d'entrée-***set** doivent être distincts.

Grammaire textuelle concrète

<état> ::=
STATE <liste d'état> <fin>
{<partie entrée>
| <entrée prioritaire>
| <partie de mise en réserve>
| <signal continu>}*
[ENDSTATE [<nom d'état>] <fin>]

<liste d'état> ::=
{<nom d'état> {, <nom d'état>}*}
| <liste d'état astérisque>

La <partie entrée> est définie au § 2.6.4, la <partie de mise en réserve> au § 2.6.5, le <signal continu> au § 4.11, la <liste d'état astérisque> au § 4.4 et l'<entrée prioritaire> au § 4.10.2.

Lorsque la <liste d'états> contient un <nom d'état>, alors le <nom d'état> représente un *nœud-d'état.* Pour chaque *nœud-d'état,* l'*ensemble-des-signaux-de-mise-en-réserve* est représenté par la <partie de mise en réserve> et tout signal implicite mis en réserve. Pour chaque *nœud-d'état,* le *nœud-d'entrée-***set** est représenté par la <partie entrée> et tout signal d'entrée implicite.

Le <nom d'état> facultatif terminant un <état> peut être spécifié seulement si la <liste d'états> dans l'<état> consiste en un seul <nom d'état>, auquel cas il doit avoir le même <nom d'état> que celui qui figure dans la <liste d'états>.

Grammaire graphique concrète

<zone d'état> ::=
<symbole d'état> **contains** <liste d'états> **is associated with**
{<zone d'association d'entrée>
| <zone d'association d'entrée prioritaire>
| <zone d'association de signal continu>
| <zone d'association de mise en réserve>}*

<symbole d'état> ::=

<zone d'association d'entrée> ::=
<symbole d'association continu> **is connected to** <zone d'entrée>

<zone d'association de mise en réserve> ::=
<symbole d'association continu> **is connected to** <zone de mise en réserve>

Les termes <zone d'entrée>, <zone de mise en réserve>, <zone d'association de signal continu>, <zone d'association des entrées prioritaires> sont respectivement définis aux § 2.6.4; 2.6.5; 4.11 et 4.10.2.

Une <zone d'état> représente un ou plusieurs *nœuds-d'état.*

Les <symboles d'association continu> issus d'un <symbole d'état> peuvent avoir un trajet de départ commun.

Sémantique

Un état représente une condition particulière dans laquelle une instance de processus peut traiter une instance de signal en exécutant une transition. S'il n'y a pas d'instances de signaux en attente, le processus attend dans l'état jusqu'à ce qu'une instance de signal soit reçue.

Modèle

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

2.6.4 *Entrée*

Grammaire abstraite

Nœud-d'entrée :: *Identificateur-de-signal*
[*Identificateur-de-variable*]*
Transition

Identificateur-de-variable = *Identificateur*

La longueur de l'[*identificateur-de-variable*]* doit être égale au nombre d'*identificateurs-de-sorte-de-référence* de la *définition-de-signal* correspondant à l'*identificateur-de-signal.*

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

On ne peut pas spécifier un nombre de variables à recevoir plus grand que le nombre de valeurs acheminées par l'instance de signal.

Grammaire textuelle concrète

<partie entrée> ::=
INPUT <liste d'entrée> <fin>
[<condition de validation>]<transition>

<liste d'entrée> ::=
<liste d'entrée astérisque>
| <stimulus>{,<stimulus>}*

<stimulus> ::=
{<identificateur de signal>
| <identificateur de temporisateur>} [([<identificateur de variable>] {,[<identificateur de variable>]}*)]

La <transition> est définie au § 2.6.7, la <condition de validation> au § 4.12 et la <liste d'entrée astérisque> au § 4.6.

Lorsque la <liste d'entrée> contient un <stimulus>, la <partie entrée> représente un <nœud d'entrée>. Dans la *grammaire abstraite*, les signaux de temporisation (<identificateur de temporisation>) sont également representés par l'*identificateur-de-signal.* 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 au § 2.8.

Une <transition> doit avoir un terminateur de transition conformément à ce qui est indiqué au § 2.6.7.2.

Grammaire graphique concrète

<zone d'entrée> ::=
<symbole d'entrée> **contains** <liste d'entrée>
is followed by {[<zone de condition de validation>] <zone de transition>}

<symbole d'entrée> ::=

La <zone de transition> est définie au § 2.6.7, la <zone de condition de validation> au § 4.12.

Une <zone d'entrée> dont la <liste d'entrée> contient un <stimulus> correspond à un *nœud-d'entrée.* Chacun des <identificateurs de signal> contenu dans un <symbole d'entrée> donne le nom de l'un des *nœud-d'entrée* que ce <symbole d'entrée> représente.

Sémantique

Une entrée permet le traitement de l'instance de signal d'entrée spécifiée. Le traitement de l'instance 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. S'il n'y a pas de variables associées à l'entrée pour une sorte spécifiée dans le signal, on ne tient pas compte de la valeur de cette sorte.

A l'expression SENDER du processus récepteur est donnée la valeur PId de l'instance 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 auront toujours une valeur PId différente de toutes les valeurs dans le système. On y accède en utilisant l'expression SENDER.

Modèle

Lorsque la liste des <stimulus> d'une <partie entrée> donnée contient plus d'un <stimulus>, une copie de la <partie entrée> est créée pour chacun de ces <stimulus>. Ensuite, la <partie entrée> est remplacée par ces copies.

2.6.5 *Mise en réserve*

Une mise en réserve spécifie un ensemble d'identificateurs de signaux dont les instances ne peuvent pas être traitées par le processus dans l'état auquel la mise en réserve est associée, et qui nécessitent une mise en réserve pour un traitement ultérieur.

Grammaire abstraite

Ensemble-de-signaux-mis-en-réserve :: *Identificateur-de-signal*-**set**

Dans chaque *nœud-d'état*, les *identificateur-de-signal* contenus dans l'*ensemble-des-signaux-mis-en-réserve* doivent être différents.

Grammaire textuelle concrète

<partie mise en réserve> ::=
SAVE <liste de mise en réserve> <fin>

<liste de mise en réserve> ::=
{<liste de signaux> | <liste de mise en réserve astérisque>}

Une <liste de mise en réserve> représente l'*identificateur-de-signal*-**set**. La <liste de mise en réserve astérisque> est une notation abrégée expliquée au § 4.8.

Grammaire graphique concrète

<zone de mise en réserve> ::=

<symbole de mise en réserve> **contains** <liste de mise en réserve>

<symbole de mise en réserve> ::=

Sémantique

Les signaux mis en réserve sont bloqués à l'accès d'entrée dans l'ordre de leur arrivée.

La mise en réserve n'a d'effet que sur les états auxquels la mise en réserve est associée. Dans l'état suivant, les instances de signaux qui ont été «mis en réserve» sont traitées comme des instances de signaux normales.

2.6.6 *Etiquette*

Grammaire textuelle concrète

<étiquette> ::=
<nom de connecteur>:

Tous les <nom de connecteur> définis dans un <corps de processus> doivent être distincts.

Une étiquette représente le point d'entrée d'un «saut» à partir des instructions de branchement correspondantes avec les mêmes <nom de connecteur> dans le même <corps de processus>.

Les «sauts» ne sont autorisés que pour les étiquettes à l'intérieur du même <corps de processus>.

Grammaire graphique concrète

<zone de connecteur d'entrée> ::=
<symbole de connecteur d'entree> **contains** <nom de connecteur> **is followed by** <zone de transition>

<symbole de connecteur d'entrée> ::=

La <zone de transition> est définie au § 2.6.7.1.

Une <zone de connecteur d'entrée> représente la continuation d'un <symbole de ligne de flot> à partir d'une <zone de connecteur de sortie> correspondante avec le même <nom de connecteur>, dans la même <zone de graphe de processus> ou la même <zone de graphe de procédure>.

2.6.7 *Transition*

2.6.7.1 *Corps de transition*

Grammaire abstraite

Transition	::	*Nœud-de-graphe* * (*Terminateur* | *Nœud-de-décision*)
Nœud-de-graphe	::	*Nœud-de-tâche* | *Nœud-de-sortie* | *Nœud-de-demande-de-création* | *Nœud-d'appel* | *Nœud-d'initialisation* | *Nœud-de-réinitialisation* |
Terminateur	::	*Nœud-d'état-suivant* | *Nœud-d'arrêt* | *Nœud-de-retour*

Grammaire textuelle concrète

<transition> ::=
{<chaîne de transition> [<instruction de terminateur>]}
| <instruction de terminateur>

<chaîne de transition> ::=
{<instruction d'action>}+

<instruction d'action> ::=
[<étiquette>] <action> <fin>

<action> ::=
<tâche>
| <sortie>
| <sortie prioritaire>
| <demande de création>
| <décision>
| <option de transition>
| <initialisation>
| <réinitialisation>
| <exportation>
| <appel de procédure>

<instruction de terminateur> ::=
[<étiquette>]<terminateur> <fin>

<terminateur> ::=
<état suivant>
| <branchement>
| <arrêt>
| <retour>

La <tâche> est définie au § 2.7.1, la <sortie> au § 2.7.4, la <demande de création> au § 2.7.2, la <décision> au § 2.7.5, l'<initialisation> et la <réinitialisation> au § 2.8, l'<appel de procédure> au § 2.7.3, l'<état suivant> au § 2.6.7.2.1, le <branchement> au § 2.6.7.2.2, l'<arrêt> au § 2.6.7.2.3, le <retour> au § 2.6.7.2.4, la <sortie prioritaire> au § 4.10.2, l'<option transition> au § 4.3.4 et l'<export> au § 4.13.

Lorsque le <terminateur> d'une <transition> est omis, la dernière action dans la <transition> doit contenir une <décision> terminale (voir le § 2.7.5) ou une <option de transition> terminale, sauf pour toutes les <transition>s contenues dans les <décision>s et les <option de transition>s (le terme <option de transition> est défini au § 4.3.4).

Un <terminateur>, une <option de transition> terminale ou une <décision> terminale ne peuvent pas être suivis par un <terminateur> ou une <action>.

Grammaire graphique concrète

<zone de transition> ::=
[<zone de chaîne de transition>] **is followed by**
{<zone d'état>
| <zone d'état suivant>
| <zone de décision>
| <symbole d'arrêt>
| <zone de fusion>
| <zone de connecteur de sortie>
| <symbole de retour>
| <zone d'option de transition> }

<zone de chaîne de transition> ::=
{<zone de tâche>
|<zone de sortie>
|<zone de sortie prioritaire>
|<zone d'initialisation>
|<zone de réinitialisation>
|<zone d'export>
|<zone de demande de création>
|<zone d'appel de procédure>}
[**is followed by** <zone de chaîne de transition>]

La <zone de tâche> est définie au § 2.7.1, la <zone de sortie> au § 2.7.4, la <zone de demande de création> au § 2.7.2, la <zone de décision> au § 2.7.5, la <zone d'initialisation> et la <zone de réinitialisation> au § 2.8, la <zone d'appel de procédure> au § 2.7.3, la <zone d'état suivant> au § 2.6.7.2.1, la <zone de fusion> au § 2.6.7.2.2, le <symbole d'arrêt> au § 2.6.7.2.3, le <symbole de retour> au § 2.6.7.2.4, la <zone de sortie prioritaire> au § 4.10.2, la <zone d'option de transition> au § 4.3.4, la <zone d'export> au § 4.13 et la <zone de connecteur de sortie> au § 2.6.7.2.2.

Une transition consiste en une séquence d'actions qui doivent être effectuées par le processus.

La <zone de transition> correspond à *transition* et la <zone de chaîne de transition> correspond au *nœud-de-graphe**.

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.

2.6.7.2 *Terminateur de transition*

2.6.7.2.1 *Etat suivant*

Grammaire abstraite

Nœud-d'état-suivant :: *Nom-d'état*

Le *nom-d'état* spécifié dans un état suivant doit porter le même nom que l'état à l'intérieur du même *graphe-de-processus* ou du même *graphe-de-procédure.*

Grammaire textuelle concrète

<état suivant> ::=
NEXTSTATE <corps d'état suivant>

<corps d'état suivant> ::=
{<nom d'état> | <état suivant pointillé> }

L'<état suivant pointillé> est défini au § 4.9.

Grammaire graphique concrète

<zone d'état suivant> ::=
<symbole d'état> **contains** <corps d'état suivant>

Sémantique

Un état suivant représente un terminateur d'une transition. Il spécifie l'état qu'aura l'instance du processus lors de l'achèvement d'une transition.

2.6.7.2.2 *Branchement*

Un branchement modifie le flot dans un <diagramme de processus> ou dans un <corps de processus> exprimant que la <déclaration d'action> suivante qui doit être interprétée est celle qui contient le même <nom de connecteur>.

Grammaire textuelle concrète

<branchement> ::=
JOIN <nom de connecteur>

On doit avoir un seul <nom de connecteur> correspondant à un <branchement> à l'intérieur du même <corps de processus>, <corps de procédure> ou <corps de service> selon le cas.

Grammaire graphique concrète

<zone de fusion> ::=
<symbole de fusion> **is connected to** <symbole de ligne de flot>

<symbole de fusion> ::=
<symbole de ligne de flot>

<symbole de ligne de flot> ::=

<zone de connecteur de sortie> ::=
<symbole de connecteur de sortie> **contains** <nom de connecteur>

<symbole de connecteur de sortie> ::=
<symbole de connecteur d'entrée>

Pour chaque <zone de connecteur de sortie> d'une <zone de graphe de processus> ou d'une <zone de graphe de procédure>, on doit avoir une seule et unique <zone de connecteur d'entrée> respectivement dans cette <zone de graphe de processus> ou cette <zone de graphe de procédure> avec le même <nom de connecteur>.

Une <zone de connecteur de sortie> correspond à un <branchement> dans la *grammaire textuelle concrète*. Si une <zone de fusion> est incluse dans une <zone de transition>, elle est équivalente à la spécification d'une <zone de connecteur de sortie> dans la <zone de transition> qui contient un unique <nom de connecteur> et à l'adjonction d'une <zone de connecteur d'entrée> avec le même <nom de connecteur> au <symbole de ligne de flot> dans la <zone de fusion>.

Modèle

Dans la syntaxe abstraite, un <branchement> ou une <zone de connecteur de sortie> provient de la <chaîne de transition> dans laquelle la première <instruction d'action> ou zone a le même <nom de connecteur> associé.

2.6.7.2.3 *Arrêt*

Grammaire abstraite

Nœud-d'arrêt :: ()

Un *nœud-d'arrêt* ne doit pas être situé à l'intérieur d'un *graphe-de-procédure.*

Grammaire textuelle concrète

<arrêt> ::=
STOP

Grammaire graphique concrète

<symbole d'arrêt> ::=

Sémantique

L'arrêt provoque l'arrêt immédiat de l'instance du processus qui l'émet. 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.

2.6.7.2.4 *Retour*

Grammaire abstraite

Nœud-de-retour :: ()

Un *nœud-de-retour* ne doit pas être situé à l'intérieur d'un *graphe-de-processus.*

Grammaire textuelle concrète

<retour> ::=
RETURN

Grammaire graphique concrète

<symbole de retour> ::=

Sémantique

Un *nœud-de-retour* est interprété de la manière suivante:

a) toutes les variables créées par l'interprétation du *nœud-de-départ-de-procédure* cesseront d'exister;

b) l'interprétation du *nœud-de-retour* termine l'interprétation du *graphe-de-procédure* et l'instance de procédure cesse d'exister;

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

2.7 *Action*

2.7.1 *Tâche*

Grammaire abstraite

Nœud-de-tâche :: *Instruction-d'affectation* |
Texte-informel

Grammaire textuelle concrète

<tâche> ::=
TASK <corps de tâche>

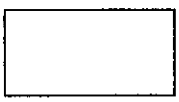
<corps de tâche> ::=
{<instruction d'affectation>{,<instruction d'affectation>}*}
| {<texte informel>{,<texte informel>}*}

Le terme <instruction d'affectation> est défini au § 5.5.3.

Grammaire graphique concrète

<zone de tâche> ::=
<symbole de tâche> **contains** <corps de tâche>

<symbole de tâche> ::=



Sémantique

L'interprétation d'un *nœud-de-tâche* est l'interprétation de l'*instruction-d'affectation* qui se trouve expliquée au § 5.5.3, ou l'interprétation du *texte-informel* qui est expliquée en § 2.2.3.

Modèle

Une <tâche> et une <zone de tâche> peuvent contenir plusieurs <instruction d'affectation>s ou <texte informel>. Dans ce cas, il s'agit d'une syntaxe dérivée pour spécifier une séquence de <tâche>, une pour chaque <instruction d'affectation> ou pour chaque <texte informel> de façon que l'ordre d'origine dans lequel ils étaient spécifiés dans le <corps de tâche> soit maintenu.

Cette abréviation est développée avant tout développement d'<expression d'import> (voir le § 4.13).

2.7.2 *Création*

Grammaire abstraite

Nœud-de-demande-de-création :: *Identificateur-de-processus*
*[Expression]**

Identificateur-de-processus = *Identificateur*

Le nombre d'*expressions* dans le terme *[expression]** doit être le même que le nombre de *paramètres-formels-de-processus* dans la *définition-de-processus* de l'*identificateur-de-processus.* Chaque *expression* doit avoir la même sorte que le *paramètre-formel-de-processus* de même position dans la *définition-de-processus* dénommée *identificateur-de-processus.*

Grammaire textuelle concrète

<demande de création> ::=
CREATE <corps de création>

<corps de création> ::=
<identificateur de processus> [<paramètres réels>]

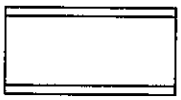
<paramètres réels> ::=
([<expression>] {,[<expression>]}*)

Le terme <expression> est défini au § 5.

Grammaire graphique concrète

<zone de demande de création> ::=
<symbole de demande de création> **contains** <corps de création>

<symbole de demande de création> ::=



Une <zone de demande de création> représente un *nœud-de-demande-de-création.*

Sémantique

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 au § 5.5.3) aux paramètres formels correspondants. Si un paramètre réel est vide, on attribue une valeur indéfinie au paramètre formel correspondant. 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.

L'action de création provoque la création d'une instance de processus dans le même bloc. Le processus créé PARENT a la même valeur de PId que le processus de création SELF. Les expressions du processus créé SELF et du processus de création OFFSPRING ont toutes les deux une valeur de PId nouvellement créée (voir le § 5.6.10.1).

Si l'on tente de créer un nombre d'instances de processus supérieur à celui qui est spécifié par le nombre maximum 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.

2.7.3 *Appel de procédure*

Grammaire abstraite

Nœud-d'appel :: *Identificateur-de-procédure*
*[Expression]**

Identificateur-de-procédure = *Identificateur*

La longueur de l'*[expression]** doit être la même que le nombre de *paramètres-formels-de-procédure* dans la *définition-de-procédure* de l'*identificateur-de-procédure.*

Chaque *expression* correspondant par sa position à un *paramètre-formel-de-processus* IN doit avoir la même sorte que le *paramètre-formel-de-processus.*

Chaque *expression* correspondant par sa position à un *paramètre-formel-de-processus* IN/OUT doit avoir un *identificateur-de-variable* avec le même *identificateur-de-référence-de-sorte* que le *paramètre-formel-de-proccessus.*

Il doit y avoir une *expression* pour chaque *paramètre-formel-de-processus* IN/OUT.

Grammaire textuelle concrète

<appel de procédure> ::=
CALL <corps d'appel de procédure>

<corps d'appel de procédure> ::=
<identificateur de procédure> [<paramètres réels>]

Les <paramètres réels> sont définis au § 2.7.2.

Un exemple d'<appel de procédure> est donné à la figure 2.9.13 au § 2.9.

Grammaire graphique concrète

<zone d'appel de procédure> ::=
<symbole d'appel de procédure> **contains** <cours d'appel de procédure>

<symbole d'appel de procédure> ::=

La <zone d'appel de procédure> représente le *nœud-d'appel.*

Un exemple de <zone d'appel de procédure> est donné à la figure 2.9.14 au § 2.9.

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. Le nœud du graphe de procédure est interprété de la même manière que les nœuds équivalents d'un graphe de processus.

L'interprétation du processus appelant est suspendue tant que l'interprétation de la procédure appelée n'est pas 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 au § 2.4.4).

2.7.4 *Sortie*

Grammaire abstraite

Nœud-de-sortie	::	*Identificateur-de-signal* [*Expression*]* [*Destination-de-signal*] *Trajet-direct*
Destination-de-signal	=	*Expression*
Trajet-direct	=	*Identificateur-d'acheminement-de-signal*-**set**

La longueur de l'[*expression*]* doit être la même que le nombre d'*identificateurs-de-référence-de-sorte* qui figure dans la *définition-de-signal* indiquée par l'*identificateur-de-signal.*

Chaque *expression* doit avoir la même sorte que l'*identificateur-de-référence-de-sorte* correspondante (par position) qui figure dans la *définition-de-signal.*

Pour chaque sous-ensemble homogène pouvant exister (voir le § 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 type de processus ayant un *identificateur-de-signal* dans son ensemble de signaux d'entrée valides et provenant du type de processus où le *nœud-de-sortie* est utilisé.

Pour chaque *identificateur-d'acheminement-de-signal* dans le *trajet-direct*, il est nécessaire que le *processus-d'origine* dans (l'un des) *trajets-d'acheminement-de-signal* dans le trajet de signal soit du même type de processus que le processus qui contient le *nœud-de-sortie* et que le *trajet-d'acheminement-de-signal* inclut l'*identificateur-de-signal* dans son ensemble d'*identificateurs-de-signal*.

Si aucun *identificateur-d'acheminement-de-signal* ne se trouve spécifié dans le *trajet-direct*, tout processus, pour lequel il y a un trajet de communication, peut recevoir le signal.

Grammaire textuelle concrète

<sortie> ::=
OUTPUT <corps de sortie>

<corps de sortie> ::=
<identificateur de signal>
[<paramètres réels>] {, <identificateur de signal> [<paramètres réels>]}*
[TO <expression PId>]
[VIA {<identificateur d'acheminement de signal>{, <identificateur d'acheminement de signal>}*
| {<identificateur de canal>{, <identificateur de canal>}* }]

Les <paramètres réels> sont définis au § 2.7.2, le terme <expression> au § 5.4.2.1.

On ne peut pas spécifier un <identificateur de canal> dans la construction VIA si des acheminements de signal sont spécifiés dans le bloc.

Pour chaque <identificateur de canal> d'une <sortie>, il doit exister un canal issu du bloc contenant et capable de transmettre les signaux définis par l'<identificateur de signal> contenu dans la <sortie>.

Les termes TO <expression PId> représentent la *destination-de-signal*.

La construction VIA représente le *trajet-direct*.

Grammaire graphique concrète

<zone de sortie> ::=
<symbole de sortie> **contains** <corps de sortie>

<symbole de sortie> ::=

Sémantique

L'expression PId *destination-de-signal* est interprétée après les autres expressions dans le *nœud-de-sortie*.

Les valeurs acheminées par l'instance du signal sont les valeurs des paramètres réels dans la sortie. S'il n'y a pas de paramètre réel à la sortie pour une sorte dans la définition des signaux, une valeur non définie est acheminée par le signal.

La valeur du PId d'origine acheminée par l'instance du signal donne la valeur associée au SELF (du processus effectuant l'action de sortie). La valeur du PId de destination acheminée par l'instance de signal est la valeur de l'expression PId du signal de destination contenu dans la sortie.

L'instance du signal est ensuite remise à un trajet de communication capable de l'acheminer vers l'instance du processus de destination spécifié.

Si aucune *destination-de-signal* n'est spécifiée, il doit exister un seul et unique récepteur qui peut recevoir le signal conformément aux acheminements de signal ou aux canaux spécifiés dans le *trajet-direct*. La valeur du PId de destination implicitement acheminée par l'instance de signal est la valeur du PId du récepteur.

L'environnement peut toujours recevoir un signal quelconque dans l'ensemble de signaux d'un canal qui conduit vers l'environnement.

Il faut noter que si l'on spécifie le même *identificateur-de-canal* ou le même *identificateur-d'acheminement-de-signal* dans le *trajet-direct* de deux *nœud-de-sortie* 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 *nœud-de-sortie* sont interprétés. Toutefois, l'ordre est préservé si les deux signaux sont acheminés par des canaux identiques reliant le *processus-origine* au *processus-destinataire* ou si les processus sont définis à l'intérieur du même bloc.

Si un syntype est spécifié dans la définition du signal et qu'une expression est spécifiée dans la sortie, la vérification d'intervalles définie au § 5.4.1.9.1 porte sur l'expression. Si la vérification d'intervalles est équivalente à False, la sortie est erronée et le comportement ultérieur du système n'est pas déterminé.

Une sortie envoyée vers une instance de processus qui n'existe pas (ou qui n'existe plus) provoque une erreur d'interprétation. L'évaluation de l'existence d'une instance de processus est effectuée au moment où la sortie est interprétée. Un arrêt consécutif de l'instance de processus réceptrice provoque l'élimination du signal de l'accès d'entrée et aucune condition d'erreur n'est indiquée.

Modèle

Si plusieurs paires de (<identificateur de signal> <paramètres réels>) se trouvent spécifiées dans un <corps de sortie>, on a une syntaxe dérivée pour spécifier la séquence des <sortie> ou des <zone de sortie>, dans le même ordre spécifié dans le <corps de sortie> d'origine, chacune contenant une seule paire de (<identificateur de signal> <paramètres réels>). La clause TO et la clause VIA sont répétées dans chacune des <sortie> ou <zone de sortie>. Cette abréviation est développée avant le développement d'une quelconque abréviation dans les expressions contenues.

2.7.5 *Décision*

Grammaire abstraite

Nœud-de-décision	::	*Question-de-décision* *Réponse-de-décision*-**set** [*Réponse-autre*]
Question-de-décision	=	*Expression* | *Texte-informel*
Réponse-de-décision	::	(*Condition-d'intervalle* | *Texte-informel*) *Transition*
Réponse-autre	::	*Transition*

Les *réponse-de-décision* s'excluent mutuellement.

Si la *question-de-décision* est une *expression*, la *condition-d'intervalle* des *réponse-de-décision* doit être de la même sorte que la *question-de-décision*

Grammaire textuelle concrète

<décision> ::=
DECISION <question> <fin> <corps de décision> ENDDECISION

<corps de décision> ::=
{<partie réponse> <partie autre>}
| {<partie réponse> {<partie réponse>}$^+$ [<partie autre>]}

<partie réponse> ::=
(<réponse>) : [<transition>]

<réponse> ::=
<condition d'intervalle> | <texte informel>

<partie autre> ::=
ELSE: [<transition>]

<question> ::=
<expression de question> | <texte informel>

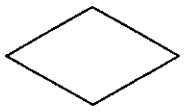
La <condition d'intervalle> est définie au § 5.4.1.9.1, la <transition> au § 2.6.7.1 et le <texte informel> au § 2.2.3.

Une <décision> ou une <option de transition> (définie au § 4.3.4) est terminale, si chaque <partie réponse> et <partie autre> dans le <corps de décision> contient une <transition> où une <instruction de terminateur> est spécifiée, ou contient une <chaîne de transition> dont la dernière <instruction d'action> contient une décision terminale ou une option terminale.

Grammaire graphique concrète

<zone de décision> ::=
    <symbole de décision> **contains** <question>
    is followed by
    {{<partie graphique réponse> <partie graphique autre>} **set**
  | {<partie graphique réponse> {<partie graphique réponse>}+ [<partie graphique autre>]} **set** }

<symbole de décision> ::=



<partie graphique réponse> ::=
    <symbole de ligne de flot> **is associated with** <réponse graphique>
    is followed by <zone de transition>

<réponse graphique> ::=
    <réponse> | (<réponse>)

<autre partie graphique> ::=
    <symbole de ligne de flot> **is associated with** ELSE
    is followed by <zone de transition>

Le terme <zone de transition> est défini au § 2.6.7.1 et le terme <symbole de ligne de flot> au § 2.6.7.2.2.

Les termes <réponse graphique> et ELSE peuvent être placés le long du <symbole de ligne de flot> associé, ou dans le <symbole de ligne de flot> interrompu.

Les <symbole de ligne de flot> provenant d'un <symbole de décision> peuvent avoir un trajet d'origine commun.

Une <zone de décision> représente un *nœud-de-décision.*

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-autre*, 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-autre* 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.

Il y a une ambiguïté syntaxique entre les expressions <texte informel> et <chaîne de caractères> dans <question> et <réponse>. Si la <question> et toutes les <réponse>s sont des <chaîne de caractères>, tous ces éléments sont interprétés comme <texte informel>. Si la <question> ou une <réponse> est une <chaîne de caractères> qui ne correspond pas au contexte de la décision, la <chaîne de caractères> désigne un <texte informel>. Le contexte de la décision (c'est-à-dire la sorte) est déterminé sans considération pour les <réponses>s qui sont <chaîne de caractères>.

Modèle

Si une <décision> n'est pas une décision terminale, il s'agit donc d'une syntaxe dérivée pour une <décision> lorsque toutes les <partie réponse>s et la <partie autre> ont inséré dans leur <transition> un <branchement> vers la première <instruction d'action> qui suit la décision ou, si la décision est la dernière <instruction d'action> dans une <chaîne de transition> un branchement vers l'<instruction de terminateur> suivante.

2.8 *Temporisateur*

Grammaire abstraite

Définition-de-temporisateur	::	*Nom-de-temporisateur* *Identificateur-de-référence-de-sorte**
Nom-de-temporisateur	=	*Nom*
Nœud-d'initialisation	::	*Expression-de-temps* *Identificateur-de-temporisateur* *Expression**
Nœud-de-réinitialisation	::	*Identificateur-de-temporisateur* *Expression**
Identificateur-de-temporisateur	=	*Identificateur*
Expression-de-temps	=	*Expression*

Les sortes de l'*expression** dans le *nœud-d'initialisation* et le *nœud-de-réinitialisation* doivent correspondre par leur position à l'*identificateur-de-référence-de-sorte** suivant directement le *nom-de-temporisateur* identifié par l'*identificateur-de-temporisateur.*

Les *expressions* dans un *nœud-d'initialisation* ou dans un *nœud-de-réinitialisation* doivent être calculées dans l'ordre donné.

Grammaire textuelle concrète

<définition de temporisateur> ::=
TIMER <nom de temporisateur> [<liste de sorte>]
{, <nom de temporisateur> [<liste de sorte>] }* <fin>

<réinitialisation> ::=
RESET (<instruction de réinitialisation>{,<instruction de réinitialisation>}*)

<instruction de réinitialisation> ::=
<identificateur de temporisateur> [(<liste d'expressions>)]

<initialisation> ::=
SET <instruction d'initialisation> { , <instruction d'initialisation> }*

<instruction d'initialisation> ::=
(<expression de temps>, <identificateur de temporisateur>
[(<liste d'expressions>)])

Les termes <liste de sortes> et <liste d'expressions> sont définis respectivement aux § 2.5.4 et § 5.5.2.1.

Une <instruction de réinitialisation> représente un *nœud-de-réinitialisation*; une <instruction d'initialisation> représente un *nœud d'initialisation.* Si une <réinitialisation> contient plusieurs <instructions de réinitialisation>s, elles doivent être interprétées dans l'ordre qui est donné. Si une <initialisation> contient plusieurs <instructions d'initialisation>s, elles doivent être interprétées dans l'ordre donné.

Grammaire graphique concrète

<zone d'initialisation> ::=
<symbole de tâche> **contains** <initialisation>

<zone de réinitialisation> ::=
<symbole de tâche> **contains** <réinitialisation>

Sémantique

Une instance de temporisateur est un objet, appartenant à une instance de processus, qui peut être actif ou inactif. Deux apparitions d'un identificateur de temporisateur suivies par une liste d'expressions se réfèrent à la même instance de temporisateur uniquement 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 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 au § 5.4.19.1 appliquée à l'expression correspondante dans une initialisation ou réinitialisation, doit être Vrai, dans le cas contraire, 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 rest actif.

Avant la première initialisation d'une instance de temporisateur, celle-ci est inactive.

2.9 *Exemples*

```
----------
INPUT S1/*exemple*/;
TASK/*exemple*/T1:=0;
----------
```

FIGURE 2.9.1

Exemple de commentaire (PR)

```
----------
INPUT I1 COMMENT 'exemple';
TASK T1:=0;
----------
```

FIGURE 2.9.2

Exemple de commentaire (PR)

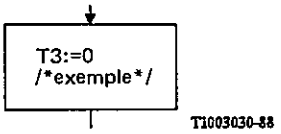


FIGURE 2.9.3

Exemple de commentaire (GR)

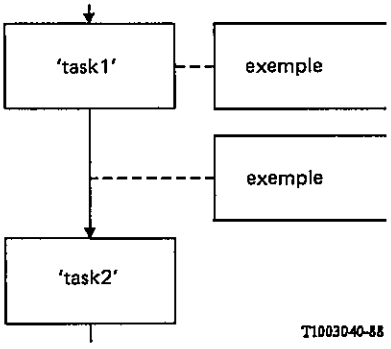


FIGURE 2.9.4

Exemple de commentaire (GR)

```
SYSTEM DAEMON_GAME;

/* Ceci est un jeu ..... Le joueur sort au moyen du signal Endgame */

SIGNAL Newgame, Probe, Result, Endgame, Gameid, Win, Lose, Score (Integer), Subscr,
Endsubscr, Bump;

CHANNEL C1
        FROM ENV TO Blockgame
        WITH Newgame, Probe, Result, Endgame;
        FROM Blockgame TO ENV
        WITH Gameid, Win, Lose, Score;
ENDCHANNEL C1;

CHANNEL C3 FROM Blockgame TO Blockdaemon
        WITH Subscr, Endsubscr;
ENDCHANNEL C3;

CHANNEL C4 FROM Blockdaemon TO Blockgame
        WITH Bump;
ENDCHANNEL C4;

BLOCK Blockgame REFERENCED;

BLOCK Blockdaemon REFERENCED;

ENDSYSTEM DAEMON_GAME;
```

FIGURE 2.9.5

Exemple de la spécification d'un système (PR)

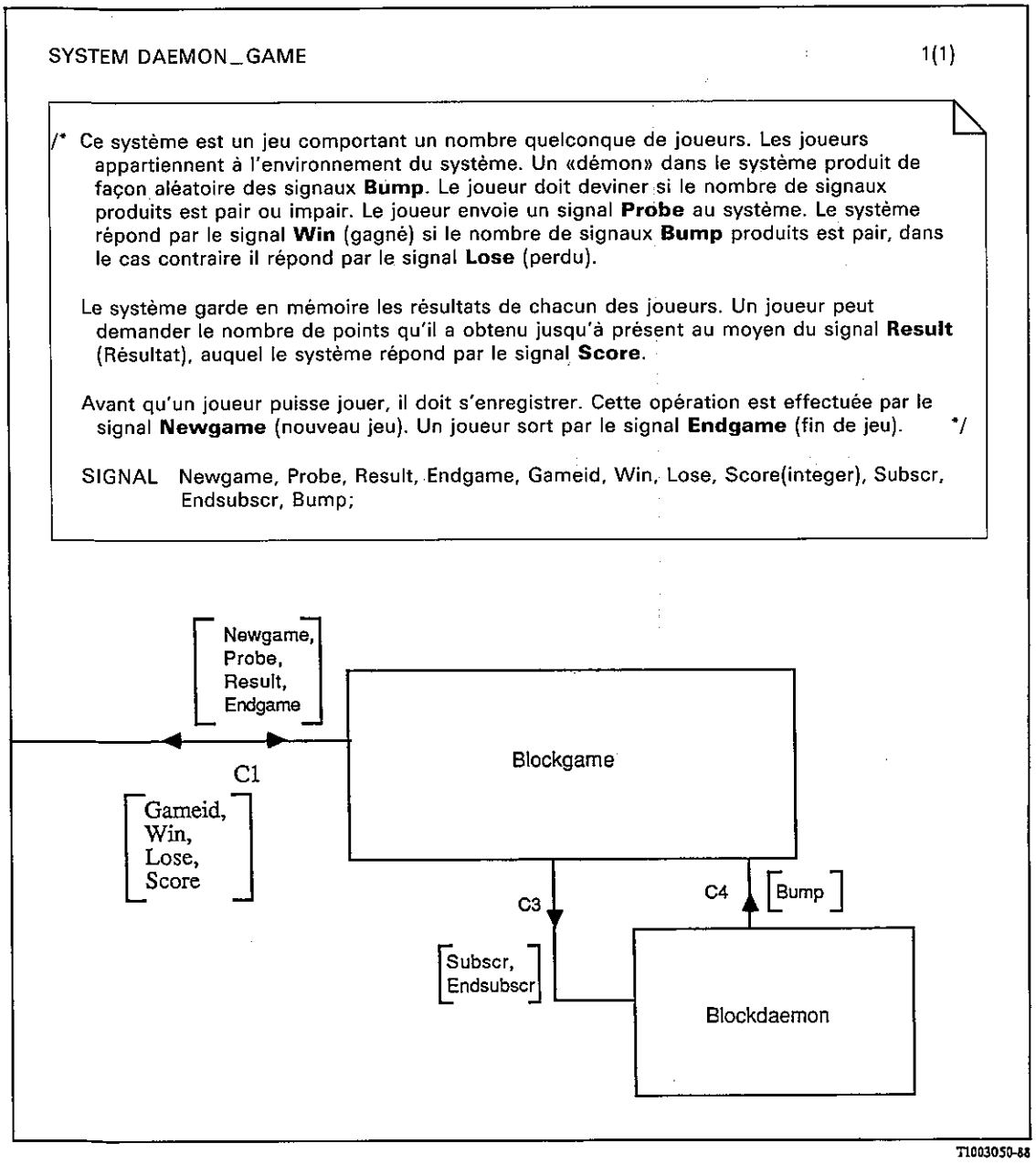


FIGURE 2.9.6

Exemple de spécification d'un système (GR)

```
BLOCK Blockgame;


CONNECT C1 AND R1,R2,R3;
CONNECT C3 AND R4;
CONNECT C4 AND R5;
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 ENV
        WITH Subscr, Endsubscr;
SIGNALROUTE R5 FROM ENV TO Game WITH Bump;


PROCESS Monitor (1,1) REFERENCED;


PROCESS Game (0,) REFERENCED;


ENDBLOCK Blockgame;
```

FIGURE 2.9.7

Exemple d'une spécification de bloc (PR)

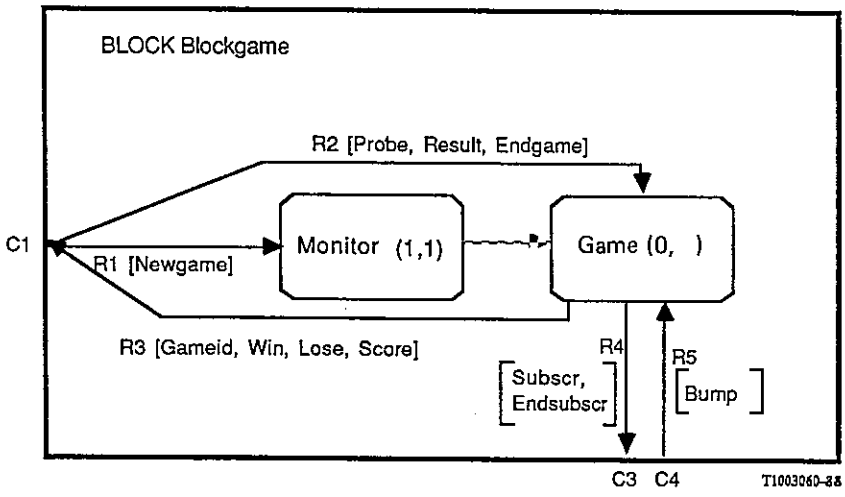


FIGURE 2.9.8

Exemple d'un diagramme de bloc

```
PROCESS Game (0,);
FPAR Player Pid;


DCL
      Count Integer;/*compteur mémorisant le résultat*/


START;
   OUTPUT Subscr;
   OUTPUT Gameid TO Player;
   TASK Count:=0;
NEXTSTATE Even;


STATE Even;


INPUT Probe;
   OUTPUT Lose TO Player;
   TASK Count:=Count-1;
NEXTSTATE-;


INPUT Bump;
NEXSTATE Odd;


STATE Odd;


INPUT Bump;
NEXSTATE Even;


INPUT Probe;
   OUTPUT Win TO Player;
   TASK Count:=Count+1;
NEXTSTATE-;


STATE*;


INPUT Result;
   OUTPUT Score(Count) TO Player;
NEXTSTATE-;


INPUT Endgame;
   OUTPUT Endsubscr;
STOP;


ENDPROCESS Game;
```

FIGURE 2.9.9

Exemple de processus de spécification (PR)

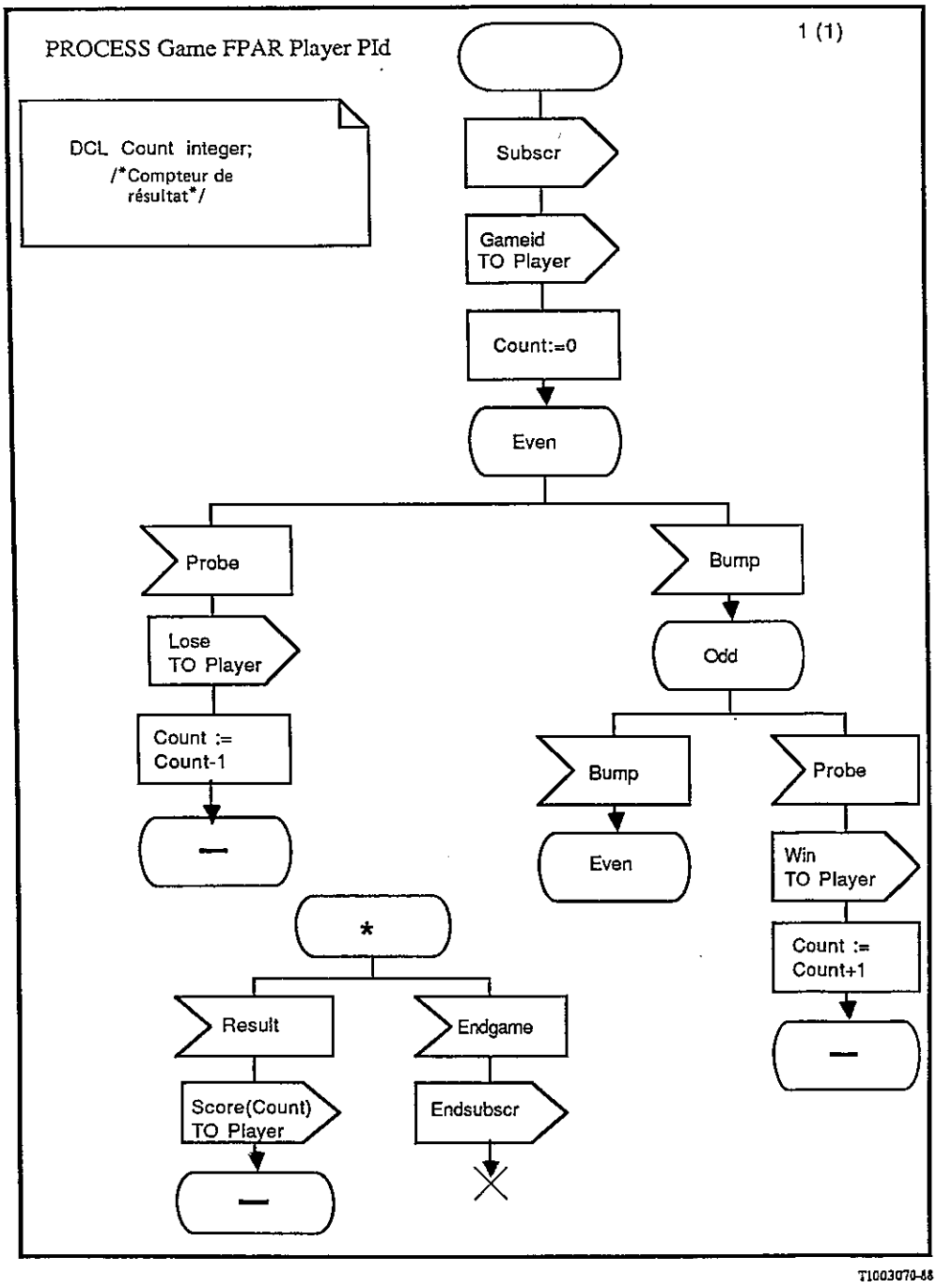


FIGURE 2.9.10

Exemple de spécification de processus (GR)

```
PROCEDURE check;
/* On suppose que les définitions des signaux sont les suivantes:
   SIGNAL sig1(Boolean), sig2, sig3(Integer,Pid); */
      FPAR IN/OUT x, y Integer;
      DCL sum, index Integer,
            nice Boolean;
      START;
      TASK sum:=0,
            index:=1;
      NEXTSTATE idle;
      STATE idle;
            INPUT sig1(nice);
            DECISION nice;
                  (true):  TASK 'calculer la somme';
                           OUTPUT sig3(sum, SENDER);
                           RETURN;
                  (false): NEXTSTATE Jaj;
            ENDDECISION;
            INPUT sig2;
            ......
      .........
      .........

ENDPROCEDURE check;
```

FIGURE 2.9.11

Exemple d'un fragment de spécification de procédure (PR)

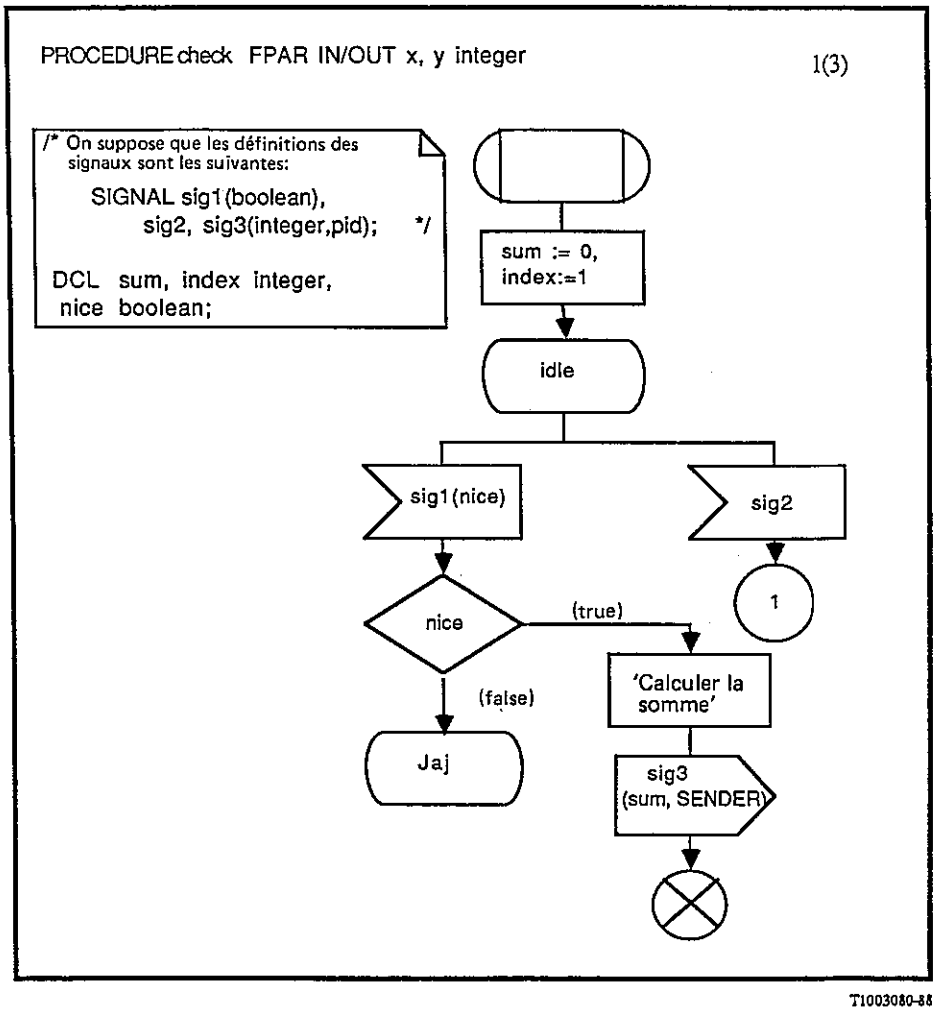


FIGURE 2.9.12

Exemple d'un fragment de spécification de procédure (GR)

```
/* On suppose que l'on dispose des définitions de signal
suivantes:
SIGNAL inquire(Integer,Integer,Integer); */
PROCESS alfa;
DCL a,b,c Integer;
        .....
        ......
        INPUT inquire (a,b,c);
        CALL check (a,b);
        ......
ENDPROCESS;
```

FIGURE 2.9.13

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

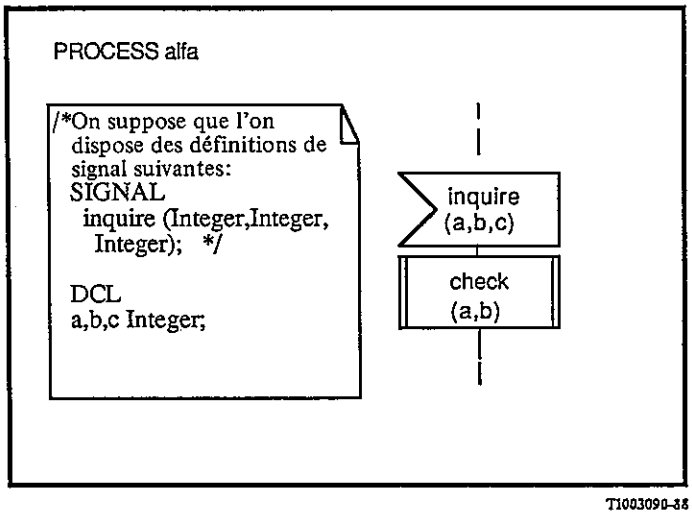


FIGURE 2.9.14

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

3 Concepts structurels dans le LDS

3.1 *Introduction*

La présente section a pour objet de définir un certain nombre de concepts dont on a besoin pour traiter des structures hiérarchiques dans le LDS. La base de ces concepts est définie au § 2 et ceux qui sont définis ci-après sont de strictes additions à ceux qui sont définis au § 2.

Les concepts présentés ci-après ont pour objet de fournir aux utilisateurs du LDS le moyen de spécifier des systèmes vastes et/ou complexes. Les concepts définis au § 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.

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 une définition de bloc en un ensemble de définitions de sous-bloc, de définitions de canal et de définitions de sous-canal. On peut aussi subdiviser une définition de canal en un ensemble de définitions de bloc, de définitions de canal et de définitions de sous-canal. Ainsi, chaque définition de bloc et chaque définition de canal peut exister en deux versions: une version non subdivisée et une version subdivisée dans les syntaxes concrètes. 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 *définition-de-bloc* dans une *définition-de-système* telle que:

a) s'il contient une *définition-de-bloc*, il doit contenir la définition de l'unité de portée qui le contient s'il y en a une;

b) il doit contenir toutes les *définition-de-bloc* définies au niveau du système et s'il contient une *définition-de-sous-bloc* d'une *définition-de-bloc*, il doit aussi contenir toutes les autres *définition-de-sous-bloc* de cette *définition-de-bloc*;

c) toutes les *définition-de-bloc* «feuille» de la structure résultante contiennent des *définition-de-processus*.

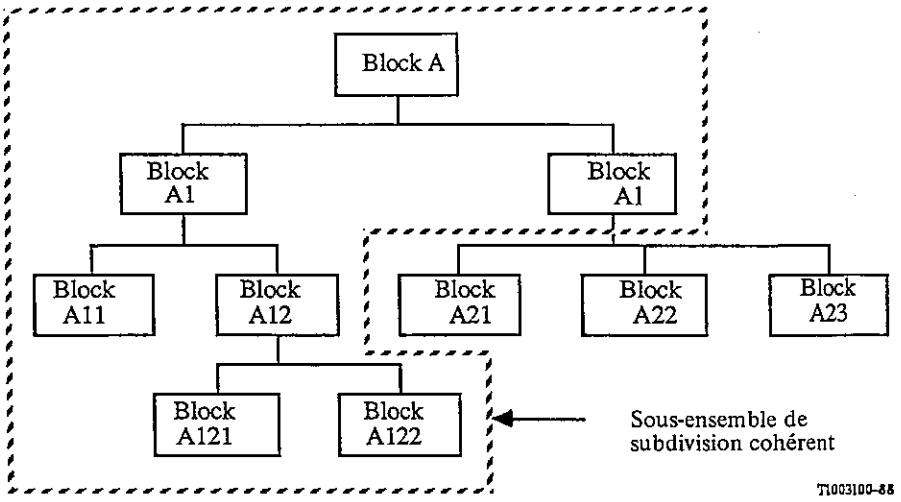


FIGURE § 3.2.1

Ensemble de subdivision cohérent illustré dans 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'éffet. 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.

3.2.2 *Subdivision des blocs*

Grammaire abstraite

Définition-de-sous-structure-de-bloc	::	*Nom-de-sous-structure-de-bloc* *Définition-de-sous-bloc*-**set** *Connexion-de-canal*-**set** *Définition-de-canal*-**set** *Définition-de-signal*-**set** *Définition-de-type-de-données* *Définition-de-type-de-synonyme*-**set**
Nom-de-sous-structure-de-bloc	=	*Nom*
Définition-de-sous-bloc	=	*Définition-de-bloc*
Connexion-de-canal	::	*Identificateur-de-canal* *Identificateur-de-sous-canal*-**set**
Identificateur-de-sous-canal	=	*Identificateur-de-canal*
Identificateur-de-canal	=	*Identificateur*

La *définition-de-sous-structure-de-bloc* doit contenir au moins une *définition-de-sous-bloc*. Dans ce qui suit, et sauf indication contraire, on suppose que la *définition-de-sous-structure-de-bloc* contient un terme de syntaxe abstraite.

Un *identificateur-de-bloc* contenu dans une *définition-de-canal* doit indiquer une *définition-de-sous-bloc*. Une *définition-de-canal* reliant une *définition-de-sous-bloc* à la frontière de la définition de *sous-structure-de-bloc* est appelée définition de sous-canal.

A chaque *définition-de-canal* externe rattachée à la *définition-de-sous-structure-de-bloc* doit correspondre exactement une *connexion-de-canal*. L'*identificateur-de-canal* dans la *connexion-de-canal* doit identifier cette *définition-de-canal* externe.

Pour les signaux qui sortent de la *définition-de-sous-structure-de-bloc*, l'association des *identificateur-de-signal* aux *identificateur-de-sous-canal*-set figurant dans une *connexion-de-canal* doit être identique aux *identificateur-de-signal* associés à l'*identificateur-de-canal* figurant dans la *connexion-de-canal*. La même règle s'applique aux signaux qui pénétrent dans la *définition-de-sous-structure-de-bloc*. Toutefois, cette règle est modifiée en cas d'affinement du signal, voir le § 3.3.

Chaque *identificateur-de-sous-canal* doit apparaître dans une seule *connexion-de-canal*.

Etant donné qu'une *définition-de-sous-bloc* est une *définition-de-bloc*, elle peut être subdivisée; cette opération peut être répétée un nombre quelconque de fois, et donne une structure hiérarchique en arbre de *définition-de-bloc* et leurs *définition-de-sous-bloc*. On dit que les *définition-de-sous-bloc* d'une *définition-de-bloc* existent au niveau inférieur suivant de l'arbre de blocs, voir aussi la figure ci-après.

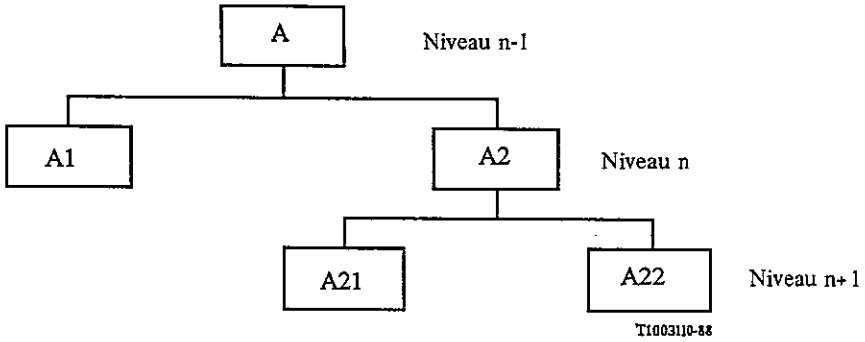


FIGURE 1/§ 3.2.2

Exemple d'un diagramme d'arbre de blocs

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

Grammaire textuelle concrète

<définition de sous-structure> ::=
SUBSTRUCTURE [[<nom de sous-structure de bloc>]
| <identificateur de sous-structure de bloc>] <fin>
| <définition de bloc>
| <référence textuelle de bloc>
| <définition de canal>
| <connexion de canal>
| <définition de signal>
| <définition de liste de signaux>
| <définition de données>
| <définition de sélection>
| <définition de macro>]+
ENDSUBSTRUCTURE [[<nom de sous-structure de bloc>
| <identificateur de sous-structure de bloc>]] <fin>

Le <nom de sous-structure de bloc> après le mot clé SUBSTRUCTURE peut être omis seulement s'il est le même que le <nom de bloc> dans la <définition de bloc> englobante.

<référence textuelle de sous-structure de bloc ::=
SUBSTRUCTURE <nom de sous-structure de bloc> REFERENCED <fin>

<connexion de canal> ::=
CONNECT <identificateur de canal> AND <identificateur de sous-canal>
[, <identificateur de sous-canal>]* <fin>

Grammaire graphique concrète

<diagramme de sous-structure de bloc> ::=
<symbole de cadre>
contains {<en-tête de sous-structure de bloc>
{{<zone de texte de sous-structure de bloc>}*
{<diagramme de macro>}*
<zone d'interaction de bloc> } **set** }
associated with {<identificateur de canal>}*

L'<identificateur de canal> identifie un canal relié à un sous-canal dans le <diagramme de sous-structure de bloc>. Il est placé à l'extérieur du <symbole de cadre>, à proximité du point d'extrémité du sous-canal, sur le <symbole de cadre>.

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

<en-tête de sous-structure de bloc> ::=
SUBSTRUCTURE {<nom de sous-structure de bloc> | <identificateur de sous-structure de bloc>}

<zone de texte de sous-structure de bloc> ::=
<zone de texte de système>

<zone de sous-structure de bloc> ::=
<référence graphique de sous-structure de bloc>
| <diagramme de sous-structure de bloc>
| <diagramme de sous-structure de bloc ouverte>

<référence graphique de sous-structure de bloc> ::=
<symbole de sous-structure de bloc> **contains** <nom de sous-structure de bloc>

<symbole de sous-structure de bloc> ::=
<symbole de bloc>

<diagramme de sous-structure de bloc ouverte> ::=
{{<zone de texte de sous-structure de bloc>}*
{<diagramme de macro>}*
<zone d'interaction de bloc>} **set**

Lorsqu'une <zone de sous-structure de bloc> est un <diagramme de sous-structure de bloc ouverte>, le <diagramme de bloc> ne doit pas contenir de <zone de texte de bloc>, de <diagramme de macro> ni de <zone d'interaction de processus>.

Sémantique

Voir le § 3.2.1.

Modèle

Un <diagramme de sous-structure de bloc ouverte> est transformé en un <diagramme de sous-structure de bloc> de manière telle que dans l'<en-tête de sous-structure de bloc>, le <nom de sous-structure de bloc>, ou l'<identificateur de sous-structure de bloc> est le même que respectivement le <nom de bloc> ou l'<identificateur de bloc> dans le <diagramme de bloc> englobant.

Exemple

Un exemple de <définition de sous-structure de bloc> est donné ci-après:

```
BLOCK A;
    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;
ENDBLOCK A;
```

Le <diagramme de sous-structure de bloc> correspondant à l'exemple est donné ci-après.

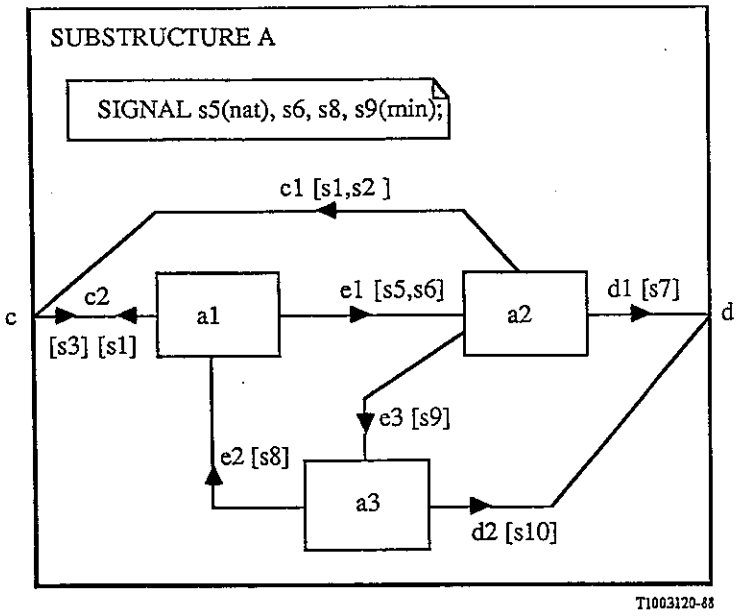


FIGURE 2/§ 3.2.2

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

<définition de sous-structure de canal> ::=
SUBSTRUCTURE {[<nom de sous-structure de canal>]
| <identificateur de sous-structure de canal> } <fin>
{ <définition de bloc>
| <référence textuelle de bloc>
| <définition de canal>
| <connexion de point d'extrémité de canal>
| <définition de signal>
| <définition de liste de signal>
| <définition de données>
| <définition de sélection>
| <définition de macro> }+
ENDSUBSTRUCTURE [[<nom de sous-structure de canal>
| <identificateur de sous-structure de canal>]] <fin>

Le <nom de sous-structure de canal> après le mot clé SUBSTRUCTURE peut être omis seulement s'il porte le même nom que le <nom de canal> dans la <définition de canal> englobante.

<référence textuelle de sous-structure de canal> ::=
SUBSTRUCTURE <nom de sous-structure de canal> REFERENCED <fin>

<connexion de point d'extrémité de canal> ::=
CONNECT {<identificateur de bloc | ENV} AND <identificateur de sous-canal>
{, <identificateur de sous-canal>}* <fin>

Pour chaque point d'extrémité de la <définition de canal> subdivisé, on doit avoir exactement une <connexion de point d'extrémité de canal>. L'<identificateur de bloc> ou l'ENVIRONMENT dans une <connexion de point d'extrémité de canal doit identifier un des points d'extrémité de la <définition de canal> subdivisée.

Grammaire graphique concrète

<diagramme de sous-structure de canal> ::=
<symbole de cadre>
contains {<en-tête de sous-structure de canal>
{{<zone de texte de sous-structure de canal>}*
{<diagramme de macro>}*
{<zone d'interaction de bloc> } **set** }
is associated with {<identificateur de bloc> | ENV}+

L'<identificateur de bloc> ou ENV désigne un point d'extrémité du canal subdivisé. L'<identificateur de bloc> est placé à l'extérieur du <symbole de cadre> au voisinage du point d'extrémité du sous-canal associé au <symbole de cadre>. Le <symbole de canal> à l'intérieur du <symbole de cadre> et qui est relié à ce dernier indique un sous-canal.

<en-tête de sous-structure de canal> ::=
SUBSTRUCTURE { <nom de sous-structure de canal>
| <identificateur de sous-structure de canal>}

<zone de texte de sous-structure de canal> ::=
<zone de texte de système>

<zone d'association de sous-structure de canal> ::=
<symbole d'association pointillé>
is connected to <zone de sous-structure de canal>

<zone de sous-structure de canal> ::=
<référence graphique de sous-structure de canal>
|<diagramme de sous-structure de canal>

<référence graphique de sous-structure de canal> ::=
<symbole de sous-structure de canal> **contains** <nom de sous-structure de canal>

<symbole de sous-structure de canal> ::=
<symbole de bloc>

Modèle

Une <définition de canal> qui contient une <définition de sous-structure de canal> est transformée en une <définition de bloc> et deux <définition de canal>s telles que:

a) les deux <définition de canal>s sont chacune connectées au bloc et à un point d'extrémité du canal d'origine. Les <définition de canal>s on des nouveaux noms distincts et chaque référence au canal d'origine dans les constructions VIA est remplacée par une référence au nouveau canal concerné;

b) la <définition de bloc> a un nouveau nom distinct et contient uniquement une <définition de sous-structure de bloc> portant le même nom et contenant les mêmes définitions que la <définition de sous-structure de canal> d'origine. Les qualificatifs de la nouvelle <définition de bloc> sont modifiés afin d'inclure le nom de bloc. Les deux <connexion de point d'extrémité de canal>s de la <définition de sous-structure de canal> d'origine sont représentées par deux <connexion de canal>s dans lesquelles l'<identificateur de bloc> ou le ENV est remplacé par le nouveau canal approprié.

Cette transformation doit avoir lieu immédiatement après celle d'un système générique (voir le § 4.3).

Exemple

Un exemple de <définition de sous-structure de canal> 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 de sous-structure de canal> pour cet exemple est donné ci-après.

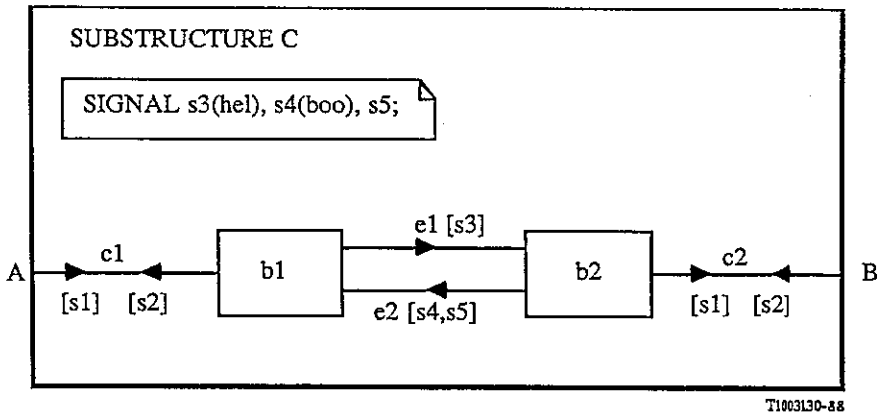


FIGURE § 3.2.3

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. Il faut remarquer qu'une définition de sous-signal d'une définition de signal n'est pas considérée comme étant une composante de la définition de signal.

Grammaire abstraite

Affinage-de-signal	::	*Définition-de-sous-signal*-**set**
Définition-de-sous-signal	::	[REVERSE] *Définition-de-signal*

Pour chaque *connexion-de-canal* il faut que pour chaque *identificateur-de-signal* associé à l'*identificateur-de-canal*, soit l'*identificateur-de-signal* soit associé à au moins un des *identificateur-de-sous-canal*, soit chacun de ses identificateurs de sous-signal soit associé à au moins un des *identificateur-de-sous-canal*. 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'une définition de processus ou des *noeuds-de-sortie* d'une définition de processus.

Grammaire textuelle concrète

<affinage-de-signal> ::=
REFINEMENT
{<définition-de-sous-signal>}+
ENDREFINEMENT

<définition-de-sous-signal> ::=
[REVERSE] <définition-de-signal>

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 scindé en sous-canaux. 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 lorsqu'un canal est scindé en trajets de signal.

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 auquel on apporte des restrictions au moyen de la règle suivante:

– Lors du choix du sous-ensemble de subdivisions cohérent, l'ensemble de signaux sur des acheminements de signaux reliés à un point d'extrémité d'un canal ne doit pas contenir de signaux ascendants de sous-signaux englobés et, à moins que l'autre point d'extrémité ne soit le système ENVIRONMENT, l'ensemble de signaux pour le premier point d'extrémité doit être égal à l'ensemble de signaux sur les acheminements de signaux reliés à l'autre point d'extrémité.

Exemple

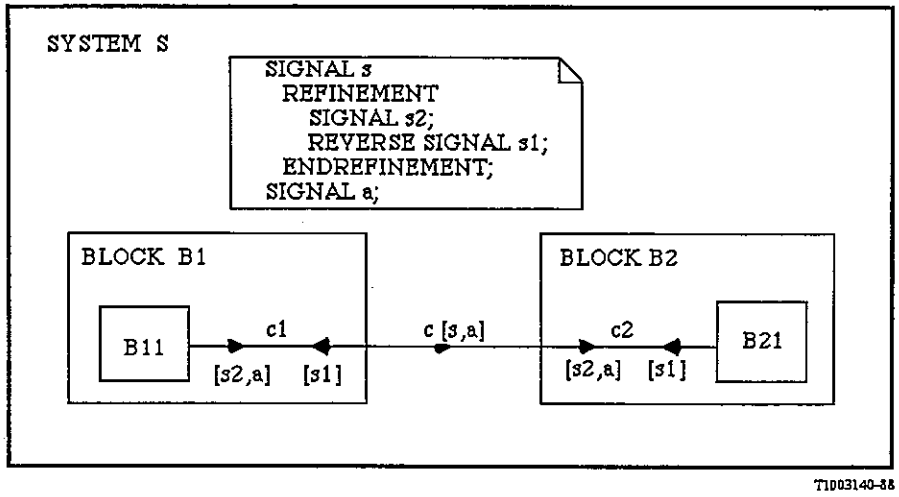


FIGURE § 3.3

Diagramme de système contenant un affinage de signal

Dans l'exemple 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.

Il faut remarquer que 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 Autres concepts dans le LDS

4.1 *Introduction*

Ce chapitre définit un certain nombre d'abréviations types qui sont modélisées en termes de concepts primitifs du LDS, utilisant la syntaxe concrète. Elles sont introduites pour améliorer la facilité d'utilisation du LDS, en sus des abréviations des autres chapitres de la Recommandation.

Les propriétés d'une abréviation dépendent de la façon dont elle est modélisée en termes de (ou transformé 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 comme suit:

1 Macro § 4.2
2 Systèmes génériques § 4.3
3 Etat astérisque § 4.4

4 Liste d'état § 2.6.3
5 Apparition multiple d'état § 4.5
6 Entrée astérisque § 4.6
7 Mise en réserve astérisque § 4.7
8 Liste de stimulus § 2.6.4
9 Liste de sortie § 2.7.4
10 Transition implicite § 4.8
11 Etat suivant pointillé § 4.9
12 Service § 4.10
13 Signal continu § 4.11
14 Condition de validation § 4.12
15 Valeur exportée et valeur importée § 4.13.

Cet ordre est également suivi pour la définition des concepts dans cette section. L'ordre de transformation spécifié signifie que dans la transformation d'une abréviation d'ordre *n*, une autre abréviation d'ordre *m* peut être utilisée si $m > n$.

Etant donné qu'il n'existe pas de syntaxe abstraite pour les abréviations, des termes de syntaxe graphique et de syntaxe textuelle sont utilisés dans leur définition. Le choix entre termes de la syntaxe graphique et termes de la syntaxe textuelle repose sur des considérations pratiques, et ne restreint pas l'utilisation des abréviations à une syntaxe concrète particulière.

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 LDS/GR et le LDS/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 <définition concrète de système>. Chacun de ces endroits est indiqué par un appel de macro. Avant de pouvoir analyser une <définition concrète de système>, chaque appel de macro doit être remplacé par la définition de macro correspondante.

4.2.1 *Règles lexicales*

```
<nom formel> ::=
        [<nom> %] <paramètre de macro>
        {% <nom> %<paramètre de macro> | %<paramètre de macro> }* [%<nom>]
```

4.2.2 *Définition de macro*

Grammaire textuelle concrète

```
<définition de macro> ::=
        MACRODEFINITION <nom de macro>
        [<paramètre formel de macro>] <fin>
        <corps de macro>
        ENDMACRO [<nom de macro>] <fin>

<paramètre formel de macro> ::=
        FPAR <paramètre formel de macro> {, <paramètre formel de macro>}*

<paramètre formel de macro> ::=
        <nom>

<corps de macro> ::=
        {<unité lexicale> | <nom formel>}*

<paramètre de macro> ::=
        <paramètre formel de macro>
    |   MACROID
```

Les <paramètre formel de macro> doivent être distincts. Les <paramètre réel de macro> doivent correspondre un à un avec les <paramètre formel de macro> correspondants.

Le <corps de macro> ne doit pas contenir les mots clé ENDMACRO et MACRODEFINITION.

Grammaire graphique concrète

<diagramme de macro> ::=
<symbole de cadre> **contains** {<en-tête de macro> <zone de corps de macro>}

<en-tête de macro> ::=
MACRODEFINITION < nom de macro> [<paramètres formels de macro>]

<zone de corps de macro>::=
{{<zone quelconque>}*
<zone quelconque> **[is connected to** <accès1 de corps de macro>] } **set**
|{ <zone quelconque> **is connected to** <accès2 de corps de macro>
<zone quelconque> **is connected to** <accès2 de corps de macro>
{ <zone quelconque> **[is connected to** <accès2 de corps de macro>]}* } **set**

<symbole de port d'entrée de macro>::=

<symbole de port de sortie de macro>::=

<accès1 de corps de macro>::=
<symbole de port de sortie> **is connected to** {<symbole de cadre>
[is associated with <étiquette de macro>]
| <symbole de port d'entrée de macro> [{**contains** |**is associated with** } <étiquette de macro>]
| <symbole de port de sortie de macro> [{**contains** |**is associated with** } <étiquette de macro>]}

<accès2 de corps de macro> ::=
<symbole de port de sortie> **is connected to** {<symbole de cadre>
is associated with <étiquette de macro>
| <symbole de port d'entreé de macro> {**contains** |**is associated with** } <étiquette de macro>
| <symbole de port de sortie de macro> {**contains** |**is associated with** } <étiquette de macro>}

<étiquette de macro> ::=
<nom>

<symbole de port de sortie> ::=
<symbole de port de sortie fictif>
| <symbole de ligne de flot>
| <symbole de canal>
| <symbole d'acheminement de signal>
| <symbole d'association continu>
| <symbole d'association pointillé>
| <symbole de ligne de création>

<symbole de port de sortie fictif> ::=
<symbole d'association continu>

<zone quelconque> ::=

<zone de texte de système>
| <zone d'interaction de bloc>
| <zone de liste de signal>
| <zone de bloc>
| <zone de texte de bloc>
| <zone d'interaction de processus>
| <référence graphique de procédure>
| <zone de texte de processus>
| <zone de graphe de processus>
| <zone de fusion>
| <zone de chaîne de transition>
| <zone d'état>
| <zone d'entrée>
| <zone de mise en réserve>
| <zone d'extension de texte>
| <zone d'association de sous-structure de canal>
| <zone de sous-structure de canal>
| <zone de sous-structure de bloc>
| <zone d'entrée prioritaire>
| <zone de signal continu>
| <zone de connecteur d'entrée>
| <zone d'état suivant>
| <zone de processus>
| <zone de définition de canal>
| <zone de ligne de création>
| <zone de définition d'acheminement de signal>
| <référence graphique de processus>
| <diagramme de processus>
| <zone de départ>
| <zone de sortie>
| <zone de sortie prioritaire>
| <zone de tâche>
| <zone de demande de création>
| <zone d'appel de procédure>
| <zone de procédure>
| <zone de décision>
| <zone de connecteur de sortie>
| <zone de texte de procédure>
| <zone de graphe de procédure>
| <zone de départ de procédure>
| <zone de texte de sous-structure de bloc>
| <zone d'interaction de bloc>
| <zone de service>
| <zone de définition d'acheminement de signal de service>
| <zone de texte de service>
| <zone de graphe de service>
| <zone de départ de service>
| <zone de commentaire>
| <zone d'appel de macro>
| <zone d'association d'entrée>
| <zone d'association de mise en réserve>
| <zone d'option>
| <zone de texte de sous-structure de canal>
| <zone d'option de transition>
| <zone d'interaction de service>
| <zone d'association d'entrée prioritaire>
| <zone d'association de signal continu>
| <zone de condition de validation>

Aucun élément ne doit être associé à un <symbole de port de sortie fictif> sauf pour ce qui est de l'<étiquette de macro>.

Pour un <symbole de port de sortie> qui n'est pas un <symbole de port de sortie fictif>, le <symbole de port d'entrée> correspondant dans l'appel de macro doit être un <symbole de port d'entrée fictif>.

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

Sémantique

Une <définition de macro> contient des unités lexicales, un <diagramme de macro> 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 de macro> 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.

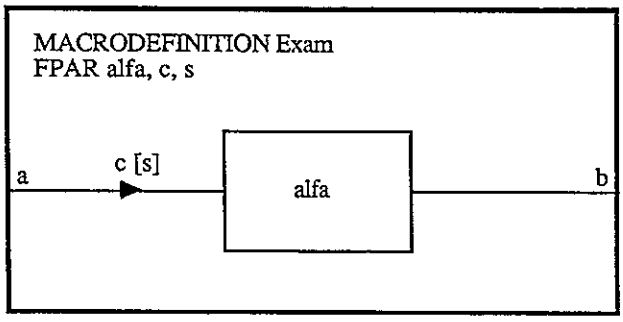
Le mot clé MACROID peut être utilisé comme un paramètre formel de pseudo-macro à l'intérieur de chaque définition de macro. Aucun <paramètre réel de macro> ne peut lui être attribué, et il est remplacé par un <nom> unique pour chaque développement de la définition de macro (à l'intérieur d'un développement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

Exemple

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

```
MACRODEFINITION Examen
FPAR alfa, c, s, x;
    BLOCK alfa REFERENCED;
    CHANNEL c FROM x TO alfa WITH s; ENDCHANNEL c;
ENDMACRO Examen;
```

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



T1003150-88

4.2.3 *Appel de macro*

Grammaire textuelle concrète

<appel de macro> ::=
MACRO <nom de macro> [<corps d'appel de macro>] <fin>

<corps d'appel de macro> ::=
(<paramètre réel de macro> {, <paramètre réel de macro>}*)

<paramètre réel de macro> ::=
{<unité lexicale>}*

L'<unité lexicale> ne peut être une virgule «,» ou une parenthèse droite «)». Si aucun de ces caractères n'est requis dans un <paramètre réel de macro>, alors le <paramètre réel de macro> doit être une <chaîne de caractères>. Si le <paramètre réel de macro> est une <chaîne de caractères>, la valeur de la <chaîne de caractères> est utilisée lorsque le <paramètre réel de macro> remplace un <paramètre formel de macro>.

Un <appel de macro> peut apparaître à un endroit quelconque où une <unité lexicale> est autorisée.

Grammaire graphique concrète

<zone d'appel de macro> ::=
<symbole d'appel de macro> **contains** {<nom de macro> [<corps d'appel de macro>]}
[**is connected to**
{<accès1 d'appel de macro> | <accès2 d'appel de macro> {<accès2 d'appel de macro>} + }]

<symbole d'appel de macro> ::=

<accès d'appel de macro 1> ::=
<symbole de port> [**is associated with** <étiquette de macro>]
is connected to <zone qulconque>

<accès d'appel de macro 2> ::=
<symbole de port> **is associated with** <étiquette de macro>
is connected to <zone quelconque>

<symbole de port> ::=
<symbole de port fictif>
| <symbole de ligne de flot>
| <symbole de canal>
| <symbole d'acheminement de signal>
| <symbole d'association continu>
| <symbole d'association pointillé>
| <symbole de ligne de création>

<symbole de port fictif> ::=
<symbole d'association continu>

Aucun élément ne peut être associé avec <symbole de port fictif> sauf en ce qui concerne l'<étiquette de macro>.

Pour chaque <symbole de port d'entrée> on doit avoir un <symbole de port de sortie> dans le <diagramme de macro> correspondant, associé avec la même <étiquette de macro>. Pour un <symbole de port d'entrée> qui n'est pas un <symbole de port fictif>, le <symbole de port de sortie> doit être un <symbole de port de sortie fictif>.

A l'exception du cas des <symbole de port d'entrée fictif>s et des <symbole de port de sortie fictif>s, il est possible d'avoir plusieurs <unité lexicale>s (textuelles) associées avec un <symbole de port d'entrée> ou un <symbole de port de sortie>. Dans ce cas, l'<unité lexicale> la plus proche du <symbole d'appel de macro> ou du <symbole de cadre> du <diagramme de macro> est prise pour être l'<étiquette de macro> associée au <symbole de port d'entrée> ou au <symbole de port de sortie>.

La <zone d'appel de macro> peut apparaître à un endroit quelconque où une zone est autorisée. Toutefois, un certain espace est requis entre le <symbole d'appel de macro> et tout autre symbole graphique clos. Si un tel espace ne peut pas être vide conformément aux règles syntaxiques, le <symbole d'appel de macro> est relié au symbole graphique clos avec un <symbole de port d'entrée fictif>.

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 de macro> 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 <paramètre formel de macro>s de la copie est remplacée par les <paramètre réel de macro>s correspondants de l'appel de macro, ensuite, les appels de macro dans la copie de le cas échéant sont développés. Tous les caractères pourcent (%) dans les <nom formel>s sont supprimés lorsque les <paramètre formel de macro>s sont remplacés par les <paramètre réel de macro>s.

Il doit y avoir une correspondance biunivoque entre <paramètre formel de macro> et <paramètre réel de macro>.

– *Règles applicables à la syntaxe graphique*

La <zone d'appel de macro> est remplacée par une copie du <diagramme de macro> de la manière suivante: tous les <symbole de port d'entrée de macro>s et tous les <symbole de port de sortie de macro>s sont supprimés. Un <symbole de port de sortie fictif> est remplacé par le <symbole de port d'entrée> ayant la même <étiquette de macro>. Ensuite, les <étiquette de macro>s attachées aux <symbole de port d'entrée>s et aux <symbole de port de sortie>s sont supprimées. L'<accès1 du corps de macro> et l'<accès2 du corps de macro> qui n'ont pas d'<accès1 d'appel de macro> ou d'<accès2 d'appel de macro> sont supprimés.

Exemple

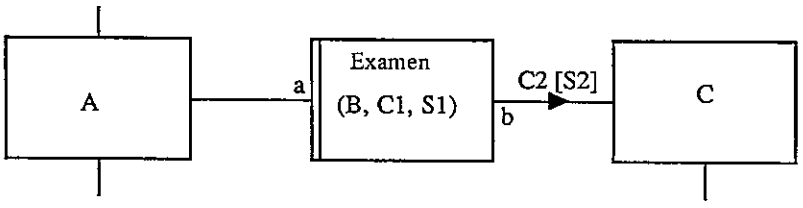
Un exemple d'<appel de macro>, dans une partie d'une <définition de bloc> est donnée ci-après.

```
.....
BLOCK A REFERENCED;
MACRO Examen (B, C1, S1, A);
BLOCK C REFERENCED;
CHANNEL C2 FROM B TO C WITH S2; ENDCHANNEL C2;
.....
```

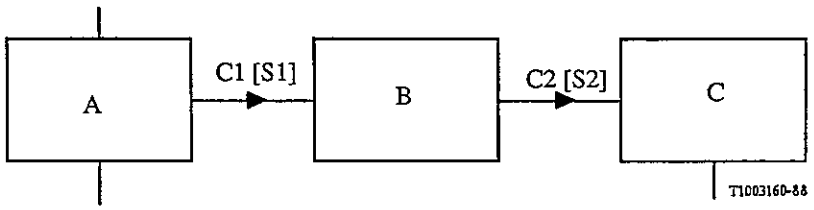
Le développement de cet appel de macro, utilisant l'exemple du § 4.2.2 donne le résultat suivant:

```
.....
BLOCK A REFERENCED;
BLOCK B REFERENCED;
CHANNEL C1 FROM A TO B WITH S1; ENDCHANNEL C1;
BLOCK C REFERENCED;
CHANNEL C2 FROM B TO C WITH S2; ENDCHANNEL C2;
.....
```

La <zone d'appel de macro> pour le même exemple, dans une partie de la <zone d'interaction de bloc> est donnée ci-après.



Le développement de cet appel de macro donne le résultat suivant:



4.3 *Systèmes génériques*

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 définies. Une telle spécification de système est appelée générique, sa propriété générique est spécifiée aux moyens de synonymes externes (qui sont les analogues des paramètres formels dans 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 appeléee spécification de système spécifique.

4.3.1 *Synonyme externe*

Grammaire textuelle concrète

<définition de synonyme externe> ::=
SYNONYM <nom de synonyme externe> <sorte prédéfinie> = EXTERNAL

<synonyme externe> ::=
<identificateur de synonyme externe>

Une <définition de synonyme externe> peut apparaître à tout endroit où une <définition de synonyme> est permise (voir le § 5.4.1.13). Un <synonyme externe> peut être utilisé à tout endroit où un <synonyme> est autorisé (voir le § 5.4.2.3). 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 externe> est un <synonyme> 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 de <expression simple>.

Une définition générique de système est une définition de système qui contient des <synonyme externe> ou du <texte informel> dans une option de transition (voir le § 4.3.4). Une définition particulière de système est créée à partir d'une définition générique de système en donnant des valeurs aux <synonyme externe>, et en transformant le <texte informel> 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

<expression simple> ::=
<expression close>

Une <expression simple> ne doit contenir que des opérateurs, des synonymes et des littéraux des sortes prédéfinies.

Sémantique

Une expression simple est une *expression-close.*

4.3.3 *Définition optionnelle*

Grammaire textuelle concrète

```
<définition de sélection> ::=
        SELECT IF (<expression simple booléenne>) <fin>
        {<définition de bloc>
            | <référence textuelle de bloc>
            | <définition de canal>
            | <définition de signal>
            | <définition de liste de signaux>
            | <définition de données>
            | <définition de processus>
            | <référence textuelle de processus>
            | <définition de temporisateur>
            | <définition d'acheminement de signal de service>
            | <connexion de canal>
            | <connexion terminale de canal>
            | <définition de variable>
            | <définition de visibilité>
            | <définition d'import>
            | <définition de procédure>
            | <référence textuelle de procédure>
            | <définition de service>
            | <référence textuelle de service>
            | <définition d'acheminement de signal>
            | <connexion de canal à acheminement>
            | <connexion d'acheminement de signal>
            | <définition de sélection>
            | <définition de macro>}+
        ENDSELECT <fin>
```

L'<expression simple booléenne> ne doit dépendre d'aucune définition dans la <définition de sélection>. Une <définition de sélection> ne peut contenir que les définitions syntaxiquement autorisées à cet endroit.

Grammaire graphique concrète

```
<zone d'option> ::=
        <symbole d'option> contains
        {SELECT IF (<expression simple booléenne>)
        {<zone de bloc>
            | <zone de définition de canal>
            | <zone de texte de système>
            | <zone de texte de bloc>
            | <zone de texte de processus>
            | <zone de texte de procédure>
            | <zone de texte de sous-structure de bloc>
            | <zone de texte de sous-structure de canal>
            | <zone de texte de service>
            | <diagramme de macro>
            | <zone de processus>
            | <zone de définition d'acheminement de signal>
            | <zone de ligne de création>
            | <zone de procédure>
            | <zone d'option>
            | <zone de service>
            | <zone de définition d'acheminement de signal de service> }+ }
```

Le <symbole d'option> est un polygone en pointillés ayant des angles droits par exemple:

Un <symbole d'option> contient logiquement la totalité d'un symbole graphique unidimentionnel quelconque coupé par sa frontière (c'est-à-dire avec un point d'extrémité à l'extérieur).

L'<expression simple booléenne> ne doit pase être dépendante d'une zone ou d'un diagramme quelconque à l'intérieur de la <zone d'option>.

Une <zone d'option> peut apparaître n'importe où, sauf à l'intérieur d'une <zone de graphe de processus>, de <zone de graphe de procédure> et de <zone de graphe de service>. Une <zone d'option> ne doit contenir que les zones et les diagrammes qui sont syntaxiquement autorisés à cet endroit.

Sémantique

Si la valeur de l'<expression simple booléenne> est fausse, toutes les constructions contenues dans la <définition de sélection> et dans le <symbole d'option> ne sont pas sélectionnées. Dans l'autre cas, ces constructions sont sélectionnées.

Modèle

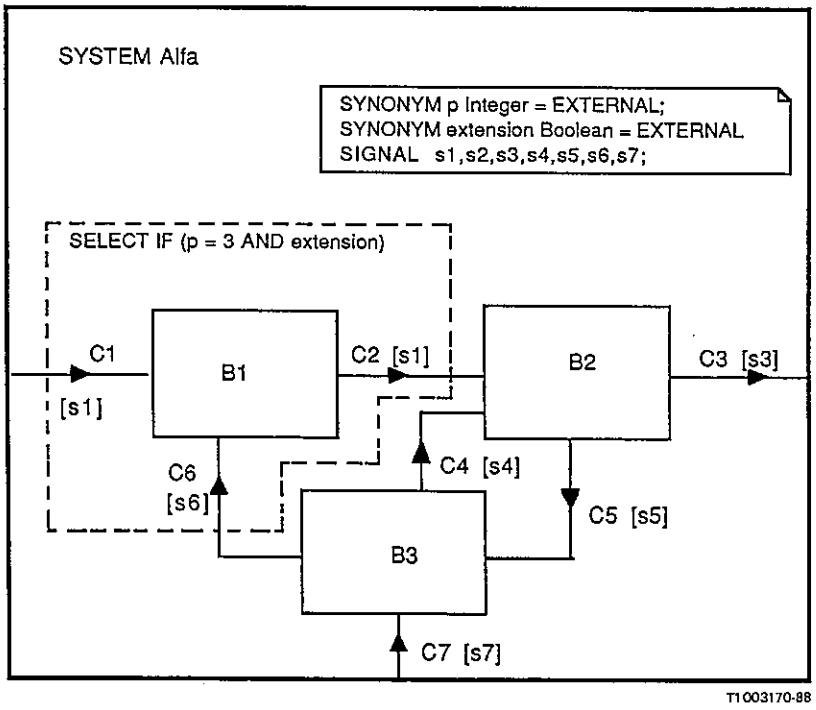
La <définition de sélection> et la <zone d'option> sont supprimées à la transformation et remplacées par les constructions sélectionnées qu'elles contiennent, s'il y en a.

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 LDS/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 dans la syntaxe LDS/GR se représente comme suit:



4.3.4 *Chaîne de transition optionnelle*

Grammaire textuelle concrète

<option de transition> ::=
ALTERNATIVE <question d'alternative> <fin>
{<partie réponse> <partie autre>
| <partie réponse> { <partie réponse> } + [<partie autre>] }
ENDALTERNATIVE

<question d'alternative> ::=
<expression simple>
| <texte informel>

Chaque <expression close> dans une <réponse> doit être une <expression simple>. Les <réponse> dans une <option de transition> doivent s'exclure mutuellement. Si la <question d'alternative> est une <expression>, la *Condition-d'intervalle* des <réponse> doit être de la même sorte que la <question d'alternative>.

Grammaire graphique concrète

<zone d'option de transition> ::=
<symbole d'option de transition> **contains** {<question d'alternative>}
is followed by {<port de sortie 1 d'option> {<port de sortie 1 d'option 1>|<port de sortie 2 d'option>}
{<port de sortie 1 d'option>}* }**set**

<symbole d'option de transition> ::=

<port de sortie 1 d'option> ::=
<symbole de ligne de flot> **is associated with** <répone graphique>
is followed by <zone de transition>

<port de sortie 2 d'option> ::=
<symbole de ligne de flot> **is associated with** ELSE
is followed by <zone de transition>

Le <symbole de ligne de flot> dans le <port de sortie 1 d'option> et le <port de sortie 2 d'option> est relié au bas du <symbole d'option de transition>. Les <symbole de ligne de flot> issus d'un <symbole d'option de transition> peuvent avoir un trajet de départ commun. La <réponse graphique> et le terme ELSE peuvent être placés le long du <symbole de ligne de flot> associé, ou dans le <symbole de ligne de flot> discontinu.

Les <réponse graphique> dans une <zone d'option de transition> doivent s'exclure mutuellement.

Sémantique

Les constructions dans un <port d'entrée 1 d'option> sont sélectionnées si la <réponse> contient la valeur de la <question d'alternative>. Si aucune des <question> ne contient la valeur de la <question d'alternative> ce sont les constructions dans le <port de sortie 2 d'option> qui sont sélectionnées.

Si aucun <port de sortie 2 d'option> et aucun des trajets sortants n'est sélectionné, la sélection n'est pas valable.

Modèle

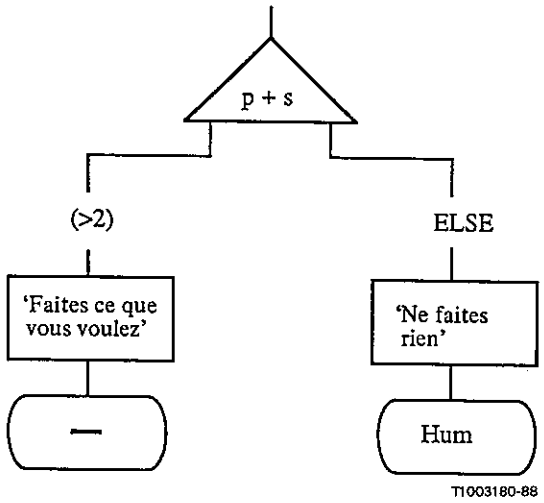
L'<option de transition> et la <zone d'option de transition> sont effacées à la transformation et remplacées par les constructions sélectionnées contenues.

Exemple

Une partie d'une <définition de processus> contenant une <option de transition> est montrée ci-après. *p* et *s* sont des synonymes.

```
.....
ALTERNATIVE p + s;
    (>2) : TASK 'Faites ce que vous voulez';
        NEXTSTATE -;
    ELSE: TASK 'Ne faites rien';
        NEXTSTATE Hum;
ENDALTERNATIVE;
.....
```

Le même exemple en syntaxe graphique concrète est montré ci-après:



4.4 *Etat astérisque*

Grammaire textuelle concrète

<liste d'état astérisque> ::=
<astérisque> [(<nom d'état> {,<nom d'état>}*)]

<astérisque> ::=

*

Dans un <corps de processus>, un <corps de procédure> ou un <corps de service>, il faut qu'au moins une <liste d'état> soit différente de la <liste d'état astérisque>. Les <nom d'état> dans une <liste d'état astérisque> doivent être distincts et doivent être contenus dans d'autres <liste d'état> du <corps de processus>, du <corps de procédure> ou du <corps de service> englobant.

Les <nom d'état> contenant une <liste d'état astérisque> ne doivent pas inclure tous les états dans un <corps de processus> ou dans un <diagramme>.

Grammaire graphique concrète

Une <zone d'état> contenant une <liste d'état astérisque> ne doit pas coïncider avec une <zone d'état suivant>.

Modèle

Une <liste d'état astérisque> est transformée en une <liste d'état> contenant tous les <nom d'état> du <corps de processus>, du <corps de service> ou du <corps de procédure> en question, sauf pour ceux des <nom d'état> qui sont contenus dans la <liste d'état astérisque>.

4.5 *Apparition multiple d'état*

Grammaire textuelle concrète

Un <nom d'état> peut apparaître dans plusieurs <état> d'un <corps de processus>, d'un <corps de service> ou d'un <corps de procédure>.

Modèle

Lorsque plusieurs <état> contiennent le même <nom d'état>, ces <état> sont concaténés en un <état> ayant ce <nom d'état>.

4.6 *Entrée astérisque*

Grammaire textuelle concrète

<liste d'entrée astérisque> ::=
<astérisque>

Un <état> peut contenir au plus une <liste d'entrée astérisque>. Un <état> ne doit pas contenir à la fois une <liste d'entrée astérisque> et une <liste de mises en réserve astérisque>.

Modèle

Une <liste d'entrée astérisque> est transformée en une liste de <stimulus> contenant la totalité de l'ensemble des signaux d'entrée valides de la <définition de processus>, englobante ou de la <définition de service> sauf pour les <identificateur de signal>, de signaux implicites et pour les <identificateur de signal> contenus dans les autres <liste d'entrées> et <liste de mises en réserve> de l'<état>, et dans toutes les <entrée prioritaire> de la <définition de service> (voir le § 4.10).

4.7 *Mise en réserve astérisque*

Grammaire textuelle concrète

<liste de mise en réserve astérisque> ::=
<astérisque>

Un <état> peut contenir au plus une <liste de mises en réserve astérisque>. Un <état> ne peut pas contenir à la fois une <liste d'entrée astérisque> et une <liste de mises en réserve astérisque>.

Modèle

Une <liste d'entrée astérisque> est transformée en une liste de <stimulus> contenant la totalité de l'ensemble des signaux d'entrée valides de la <définition de processus>, englobante ou de la <définition de service> sauf pour les <identificateur de signal>, de signaux implicites et pour les <identificateur de signal> contenus dans les autres <liste d'entrées> et <liste de mises en réserve> de l'<état>, et dans toutes les <entrée prioritaire> de la <définition de service> (voir le § 4.10).

4.8 *Transition implicite*

Grammaire textuelle concrète

Un <identificateur de signal> contenu dans l'ensemble complet des signaux d'entrée valides d'une <définition de processus>, d'une <définition de procédure> ou d'une <définition de service> peut être omis dans l'ensemble des <identificateur de signal> contenus dans les <liste d'entrées>, les <liste d'entrées prioritaires> et la <liste de mises en réserve> d'un <état>.

Modèle

Pour chaque <état>, il y a une <partie entrée> implicite contenant une <transition> qui ne contient qu'un <état suivant> ramenant au même <état>.

4.9 *Etat suivant pointillé*

Grammaire textuelle concrète

<état suivant pointillé> ::=
<trait d'union>

<trait d'union> ::=
-

La <transition> contenue dans un <départ> ne doit pas conduire, directement ou indirectement, dans un <état suivant pointillé>.

Modèle

Dans chaque <état suivant> d'un <état>, l'<état suivant pointillé> est remplacé par le <nom d'état> de l'<état>.

4.10 *Service*

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

4.10.1 *Décomposition de service*

Grammaire textuelle concrète

<décomposition de service> ::=
{<définition d'acheminement de signal de service>
| <connexion d'acheminement de signal>
| <définition de service>
| <définition de sélection>
| <référence textuelle de service>}+

<définition d'acheminement de signal de service> ::=
SIGNALROUTE <nom d'acheminement de signal de service>
<trajet d'acheminement de signal de service>
[<trajet d'acheminement de signal de service>]

<trajet d'acheminement de signal de service> ::=
{FROM <identificateur de service> TO <identificateur de service>
| FROM <identificateur de service> TO ENV
| FROM ENV TO <identificateur de service>}
WITH <liste de signaux> <fin>

<connexion d'acheminement de signal> ::=
CONNECT <identificateur d'acheminement de signal>
AND <identificateur d'acheminement de signal de service> {, <identificateur d'acheminement de signal de service>}* <fin>

<référence textuelle de service> ::=
SERVICE <nom de service> REFERENCED <fin>

Lorsqu'une <définition de processus> contient une <décomposition de service>, elle ne doit pas contenir de <définition de temporisateur> à l'exterieur de la <décomposition de service>.

Une <décomposition de service> doit contenir au moins une <définition de service>.

Pour l'<acheminement de signal de service>, des règles de bonne formation analogues à celles de l'<acheminement de signal> s'appliquent.

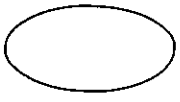
Grammaire graphique concrète

<zone d'interaction de service> ::=
{ <zone de service> | <zone de définition d'acheminement de signal de service> }+

<zone de service> ::=
<référence graphique de service>
| <diagramme de service>

<référence graphique de service> ::=
<symbole de service> **contains** <nom de service>

<symbole de service> ::=



<zone de définition d'acheminement de signal de service> ::=
<symbole d'acheminement de signal>
is associated with {<nom d'acheminement de signal de service>
[<identificateur d' acheminement de signal>]
<zone de liste de signal>
[<zone de liste de signal>] }**set**
is connected to {<zone de service>
{<zone de service> | <symbole de cadre>} }**set**

Lorsque le <symbole d'acheminement de signal> est connecté au <symbole de cadre>, l'<identificateur d'acheminement de signal> identifie un trajet extérieur de signal auquel le trajet de signal est connecté.

Sémantique

La <décomposition de service> est un moyen de remplacement du <corps de processus>, et exprime le même comportement.

Modèle

Le concept de service est modélisé en transformant la <décomposition de service> en concepts primitifs. La transformation des <définition d'acheminement de signal de service> et des <connexion d'acheminement de signal> n'aboutissent à rien.

4.10.2 *Définition de service*

Grammaire textuelle concrète

```
<définition de service> ::=
        SERVICE {<nom de service> | <identificateur de service>} <end>
        [<ensemble de signaux d'entrée valides>]
        {<définition de variable>
            | <définition de données>
            | <définition de temporisateur>
            | <définition de visibilité>
            | <définition d'import>
            | <définition de sélection>
            | <définition de macro>
            | <définition de procédure>
            | <référence textuelle de procédure>}*
        <corps de service>
        ENDSERVICE [{<nom de service> | <identificateur de service>}] <fin>

<corps de service> ::=
        <corps de processus>

<entrée prioritaire> ::=
        PRIORITY INPUT <liste d'entrées prioritaires> <fin> <transition>

<liste d'entrées prioritaires> ::=
        <stimulus prioritaire> {, <stimulus prioritaire>}*

<stimulus prioritaire> ::=
        <identificateur de signal prioritaire [ ( [<identificateur de variable>]
        {, [<identificateur de variable>] }* ) ]

<sortie prioritaire> ::=
        PRIORITY OUTPUT <corps de sortie prioritaire>

<corps de sortie prioritaire> ::=
        <identificateur de signal prioritaire> [<paramètres réels>]
        {, <identificateur de signal prioritaire> [<paramètres réels>]}*
```

Un signal est un signal hautement prioritaire dans un processus si et seulement s'il est indiqué dans une <entrée prioritaire> d'une <définition de service> de ce processus.

Une <définition de variable> dans une <définition de service> ne doit pas contenir le mot clé EXPORTED ou REVEALED.

Un <identificateur de signal prioritaire> dans une <sortie prioritaire> ne doit pas être contenu dans une <partie entrée> ou dans une <partie de mise en réserve>. Un <identificateur de signal prioritaire> dans une <entrée prioritaire> ne doit pas être contenu dans une <sortie>.

La même règle concernant l'ensemble des signaux d'entrée valides et l'acheminement de signal de service telle qu'elle est énoncée au § 2.5.2 concernant le processus s'applique.

La <décomposition de service> ne peut contenir des <définitions d'acheminement de signal de service> que si la <définition de bloc> englobante contient des <définition d'acheminement de signal>.

Une seule des <définition de service> dans une <décomposition de service> est permise pour avoir un <départ> contenant une <chaîne> de transition>. Tous les autres <départ> doivent seulement contenir un <état suivant>.

Les ensembles complets de signaux d'entrée valides (chacun d'eux étant la combinaison de l'<ensemble de signaux d'entrée valides> et de l'ensemble de signaux transmis sur les <acheminement de signal de service> d'une <définition de service> des <définition de service> d'une <définition de processus>) doivent être distincts.

Une <définition de procédure> ne doit pas avoir d'<état> lorsque la <définition de processus> englobante contient une <définition de service>. Des <définitions de procédure> visibles pour plus d'un service ne doivent pas contenir de structure VIA.

L'ensemble des priorités associées au <signal continu> à l'intérieur des diverses <définition de service> de la <décomposition de service> ne doivent pas se recouvrir.

Des règles de définitions bien formées analogues s'appliquent pour la <connexion d'acheminement de signal> et pour la <connexion de canal vers acheminement>.

Si la <décomposition de service> englobante contient des <définition d'acheminement de signal de service>, pour chaque <identificateur d'acheminement de signal> d'une <sortie> il doit exister un acheminement de signal de service issu du service englobant et connecté à l'acheminement de signal, et capable de transmettre les signaux désignés par les <identificateur de signal> contenus dans la <sortie>.

Si une <sortie> ne contient pas de construction VIA, il doit exister au moins un trajet de communication (soit implicite vers le même service, soit via des acheminements de signal de service (éventuellement implicites), et éventuellement des acheminements de signal et des canaux), issu du service, qui puisse transmettre les signaux désignés par les <identificateur de signal> contenus dans la <sortie>.

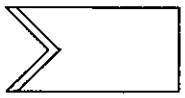
Pour chaque <sortie prioritaire>, il doit exister au moins un trajet de communication (soit implicite vers le même service, soit via (des acheminements de signal de service éventuellement implicites)), issu du service, qui puisse transmettre les signaux désignés par les <identificateur de signal prioritaire> contenus dans la <sortie prioritaire>.

L'<entrée prioritaire> est autorisée seulement dans un <corps de service>. La <sortie prioritaire> est autorisée seulement dans un <corps de service> et dans un <corps de procédure>.

Grammaire graphique concrète

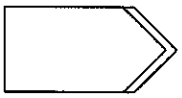
```
<diagramme de service> ::=
        <symbole de cadre> contains
        {<en-tête de service>
        {    {<zone de texte de service> }*
             {<référence graphique de procédure>}*
             {<diagramme de procédure>}*
             {<diagramme de macro>}*
             <zone de graphe de service> }set  }

<en-tête de service> ::=
        SERVICE {<nom de service> | <identificateur de service>}

<zone de texte de service> ::=
        <symbole de texte> contains
        { <définition de variable>
             | <définition de données>
             | <définition de temporisateur>
             | <définition de visibilité>
             | <définition d'import>
             | <définition de sélection>
             | <définition de macro> }*

<zone de graphe de service> ::=
        <zone de graphe de processus>

<zone d'association d'entrée prioritaire> ::=
        <symbole d'association continu> is connected to <zone d'entrée prioritaire>

<zone d'entrée prioritaire> ::=
        <symbole d'entrée prioritaire> contains <liste d'entrées prioritaires>
        is followed by <zone de transition>
```

<symbole d'entrée prioritaire> ::=



<zone de sortie prioritaire> ::=
<symbole de sortie prioritaire> **contains** <corps de sortie prioritaire>

<symbole de sortie prioritaire> ::=



Sémantique

Les propriétés d'un service sont obtenues à partir de la condition selon laquelle la <décomposition de service> qui remplace un <corps de processus> exprime le même comportement que le <corps de processus>.

Dans une instance de processus, il existe une instance de service pour chaque <définition de service> dans la <définition de processus>. Les instances de service sont des composantes de l'instance de processus et ne peuvent être manipulées (créées, invoquées ou détruites) comme des objets séparés. Ils partagent l'accès d'entrée et les expressions SELF, PARENT, OFFSPRING and SENDER de l'instance de processus.

Une instance de service est une machine à états finis, mais elle ne peut fonctionner en parallèle avec d'autres instances de service de l'instance de processus, c'est-à-dire qu'à l'intérieur d'une instance de processus, une seule instance de service peut effectuer une transition à un instant donné quelconque.

Dans le <corps de sortie prioritaire> la construction TO SELF est sous-entendue. Les signaux prioritaires sont une catégorie spéciale de signaux qui ont une priorité supérieure à celle des signaux ordinaires. Ces signaux peuvent être envoyés seulement entre des instances de service à l'intérieur de la même instance de processus.

Un signal d'entrée provenant d'un accès d'entrée est donné à l'instance de service qui est en mesure de recevoir ce signal.

Modèle

a) *Transformation de définitions*

Les définitions locales à l'intérieur d'une <définition de service> sont transformées au niveau du processus en remplaçant chaque occurence d'un nom dans le service par le même nouveau nom distinct. Chaque référence au service dans les qualificatifs disparaît.
Des définitions de visibilité ou des définitions d'import contenant la même variable de visibilité ou d'import sont fusionnées en une seule définition de visibilité ou d'import.

b) *Transformation des <corps de service>*

L'ensemble des <corps de service> est transformé en un <corps de processus>. Cette opération peut être effectuée de différentes façons. Dans le cas présent, une transformation simple a été choisie, puisque le principal objectif est de définir le concept de service en utilisant une syntaxe concrète stricte. Pour des raisons pratiques, un <corps de service> et un <corps de processus> est considéré comme un graphe composé d'états, de chaînes de transitions entre états, de chaînes de transitions d'arrêt et d'une chaîne de transition de départ. Une chaîne de transition est uniquement définie par un état de départ, une entrée et un état final.

1) *Etats*

Un état dans le graphe de processus résultant est identifié par un multiplet noms. La dimension du multiplet est égal au nombre de graphes de service. Chaque composante du multiplet se refère uniquement à un des graphes de services d'origine et la valeur de la composante du multiplet est un des noms d'état du graphe de service en question. Les noms d'état du graphe de processus seront donc l'ensemble des multiplets qu'il est possible de construire en utilisant ces règles. Exemple:

Soit deux graphes de service et leurs états correspondants:

f1: <a> <b>
f2: <A> <B> <C>

le graphe de processus résultant a donc les états suivants:

<a.A> <a.B> <a.C> <b.A> <b.B> <b.C>

Cette explosion d'état peut normalement être réduite de manière sensible, mais cet aspect n'est pas abordé ici.

2) *Chaînes de transition*

Chaque chaîne de transition dans un graphe de service est copiée dans le graphe de processus à un ou plusieurs endroits. Elle est copiée afin de relier chaque paire de multiplets d'état qui satisfont aux conditions suivantes:

- une composante du multiplet état de départ concerne l'état de départ de la chaîne de transition;
- une composante du multiplet d'état final concerne l'état final de la chaîne de transition;
- les autres valeurs des composantes doivent être les mêmes pour les deux multiplets d'état.

Exemple:

Dans l'exemple précédent, nous avons une chaîne de transition en f2 entre <B> et <C>. Dans le graphe du processus résultant, cette chaîne de transition reliera <a.B> à <a.C> et <b.B> à <b.C>. Cela peut s'exprimer de façon plus concise (en utilisant la notation abrégée de la syntaxe concrète):

<*.B> est transformé en <-.C>

3) *Chaînes de transition de départ*

Si un des graphes de service contient une chaîne de transition de départ, cette chaîne de transition est transformée en chaîne de transition de départ du graphe de processus. La chaîne de transition de départ du graphe de processus conduit à un multiplet d'état ayant comme composante tous les noms d'état initiaux du graphe de service.

4) *Chaînes de transition d'arrêt*

Chaque transition conduisant à un <arrêt> est copié dans le graphe de processus et est reliée à chaque multiplet d'état ayant une composante qui se réfère à l'état de départ de la transition.

5) *Signaux prioritaires*

Les signaux prioritaires sont transformés comme suit.

Chaque état du graphe de processus résultant est scindé en deux états. Les entrées prioritaires de l'état d'origine sont reliées au premier état, toutes les autres entrées sont reliées au second état et sont mises en réserve dans le premier état. La chaîne de transition conduisant vers l'état original conduit maintenant vers le premier état. On ajoute à cette chaîne de transition la chaîne d'action suivante:

- une unique valeur de marque est produite et affectée à la variable implicite SAME_TOKEN
- le signal implicite X_CONT est envoyé à SELF, transportant la valeur de la marque.

Une entrée pour le signal implicite X_CONT est ajoutée au premier état, suivie par la chaîne de transition suivante:

- une décision compare la valeur reçue de la marque avec la valeur de SAME_TOKEN. Si les deux valeurs sont égales, on choisit le trajet conduisant au deuxième état, dans le cas contraire on choisit un trajet renvoyant au premier état.

Exemple

Un exemple d'une <définition de processus> contenant une <décomposition de service> est donné ci-après ainsi que la <définition de service> correspondante. Ce processus a le même comportement que celui qui est donné à la figure 2.9.9 au § 2.9.

```
PROCESS Game;
        FPAR Player pid;
        SIGNAL Proberers (integer);
        DCL A integer;

        SIGNALROUTE IR1 FROM Game_handler TO ENV WITH Score,Gameid;
        SIGNALROUTE IR2 FROM Game_handler TO ENV WITH Subscr,Endsubscr;
        SIGNALROUTE IR3 FROM ENV TO Game_handler WITH Result,Endgame;
        SIGNALROUTE IR4 FROM ENV TO Bump_handler WITH Probe;
        SIGNALROUTE IR5 FROM ENV TO Bump_handler WITH Bump;
        SIGNALROUTE IR6 FROM Bump_handler TO ENV WITH Lose,Win;
        SIGNALROUTE IR7 FROM Bump_handler TO Game_handler WITH Proberers;

        CONNECT R5 AND IR5;
        CONNECT R2 AND IR3,IR4;
        CONNECT R3 AND IR1,IR6;
        CONNECT R4 AND IR2;

        SERVICE Game_handler REFERENCED;
        SERVICE Bump_handler REFERENCED;

ENDPROCESS Game;
```

```
SERVICE Game_handler;

    /*Le service gère un jeu avec des actions pour démarrer un jeu, le terminer, conserver trace du résultat et
    communiquer le résultat*/

    DCL Count integer;
    /*Compteur pour garder trace du résultat*/

    START;
        OUTPUT Subscr;
        OUTPUT Gameid TO Player;
        TASK Count:=0;
        NEXTSTATE STARTED;
    STATE STARTED;
        PRIORITY INPUT Proberers (A);
            TASK Count:=Count+A;
            NEXTSTATE_;
        INPUT Result;
            OUTPUT Score (Count) TO Player;
            NEXTSTATE_;
        INPUT Endgame;
            OUTPUT Endsubscr;
            STOP;
    ENDSTATE STARTED;
ENDSERVICE Game_handler;

SERVICE Bump_handler;

    /*Le service peut enregister les «bumps» et gérer les tentatives du joueur.
    Le résultat des tentatives est envoyé au joueur mais également au service «Game_handler»*/

    START;
        NEXTSTATE EVEN;
    STATE EVEN;
        INPUT Probe;
            OUTPUT Lose TO Player;
            PRIORITY OUTPUT Proberers(-1);
            NEXTSTATE_;
        INPUT Bump;
            NEXTSTATE ODD;
    ENDSTATE EVEN;
    STATE ODD;
        INPUT Bump;
            NEXTSTATE EVEN;
        INPUT Probe;
            OUTPUT Win TO Player;
            PRIORITY OUTPUT Proberers(+1);
            NEXTSTATE_;
    ENDSTATE ODD;
ENDSERVICE Bump_handler;
```

Le même exemple dans le LDS/GR est montré dans les diagrammes suivants:

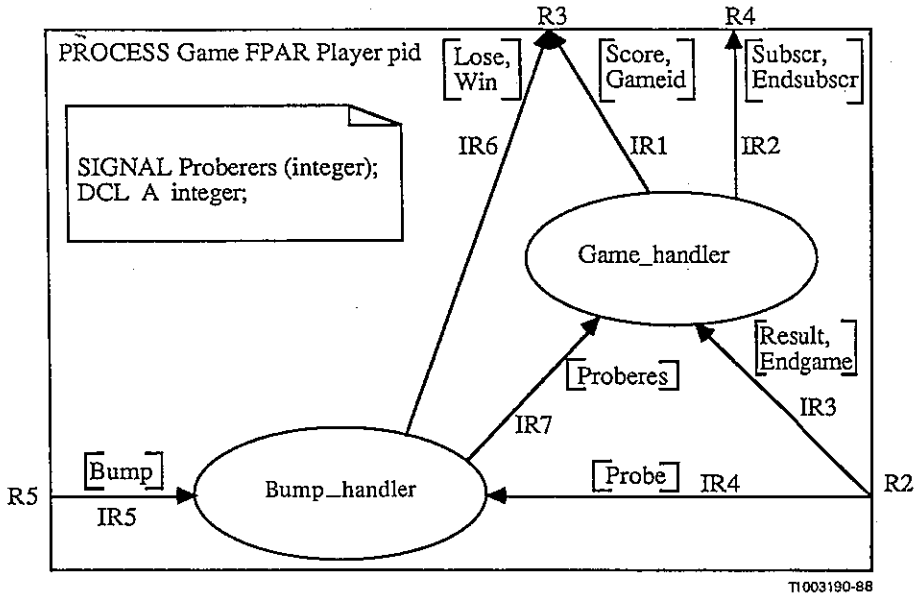


FIGURE 4.10.1

Exemple d'un diagramme de service

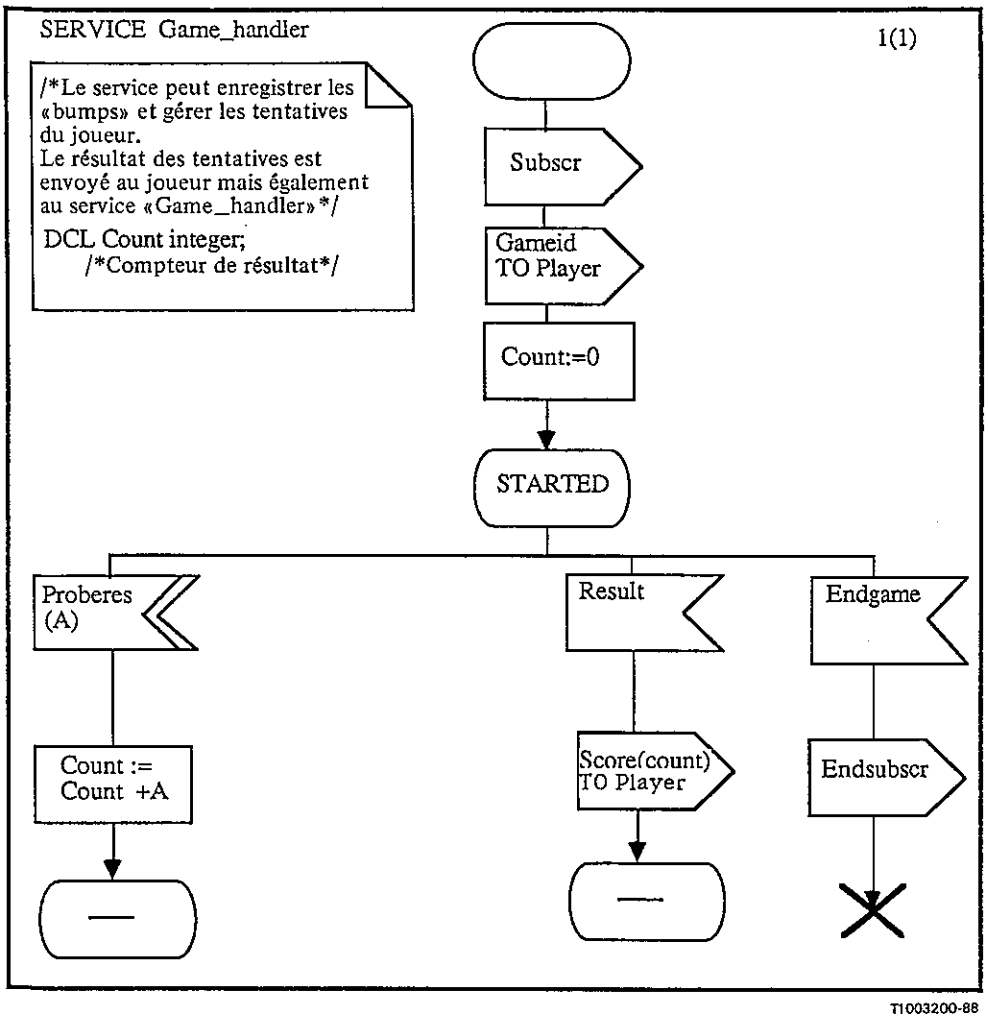


FIGURE 4.10.2

Exemple d'un diagramme de service

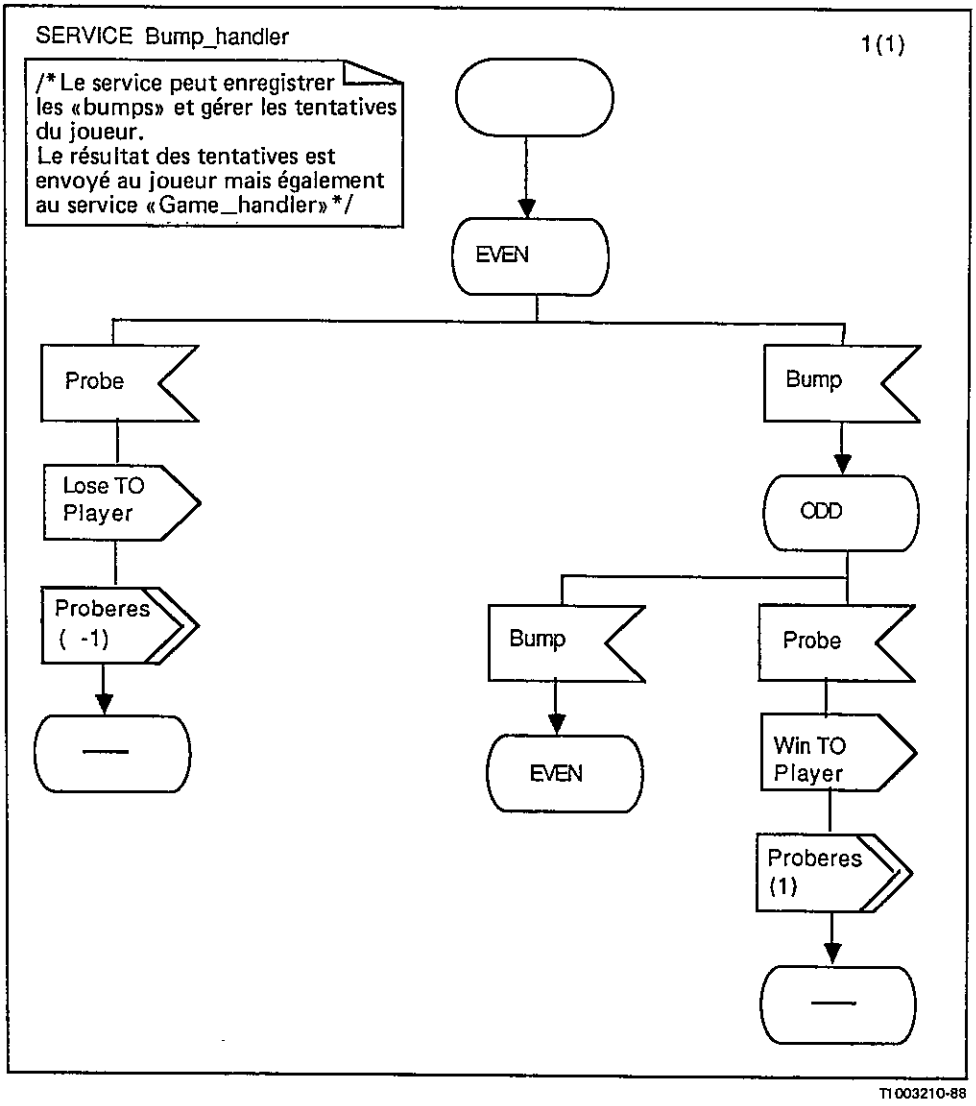


FIGURE 4.10.3

Exemple d'un diagramme de service

En appliquant les règles 1 à 4 de transformation, on obtient le graphe de processus de la figure 4.10.4. Il contient toujours les signaux prioritaires non encore transformés. En simplifiant de manière évidente les conditions qui contiennent les signaux prioritaires et utilisant le concept d'état astérisque, le même processus de la figure 2.9.10 du § 2.9 peut être obtenu. (Noter que les états EVEN et ODD correspondent respectivement aux états STARTED.EVEN et STARTED.ODD.)

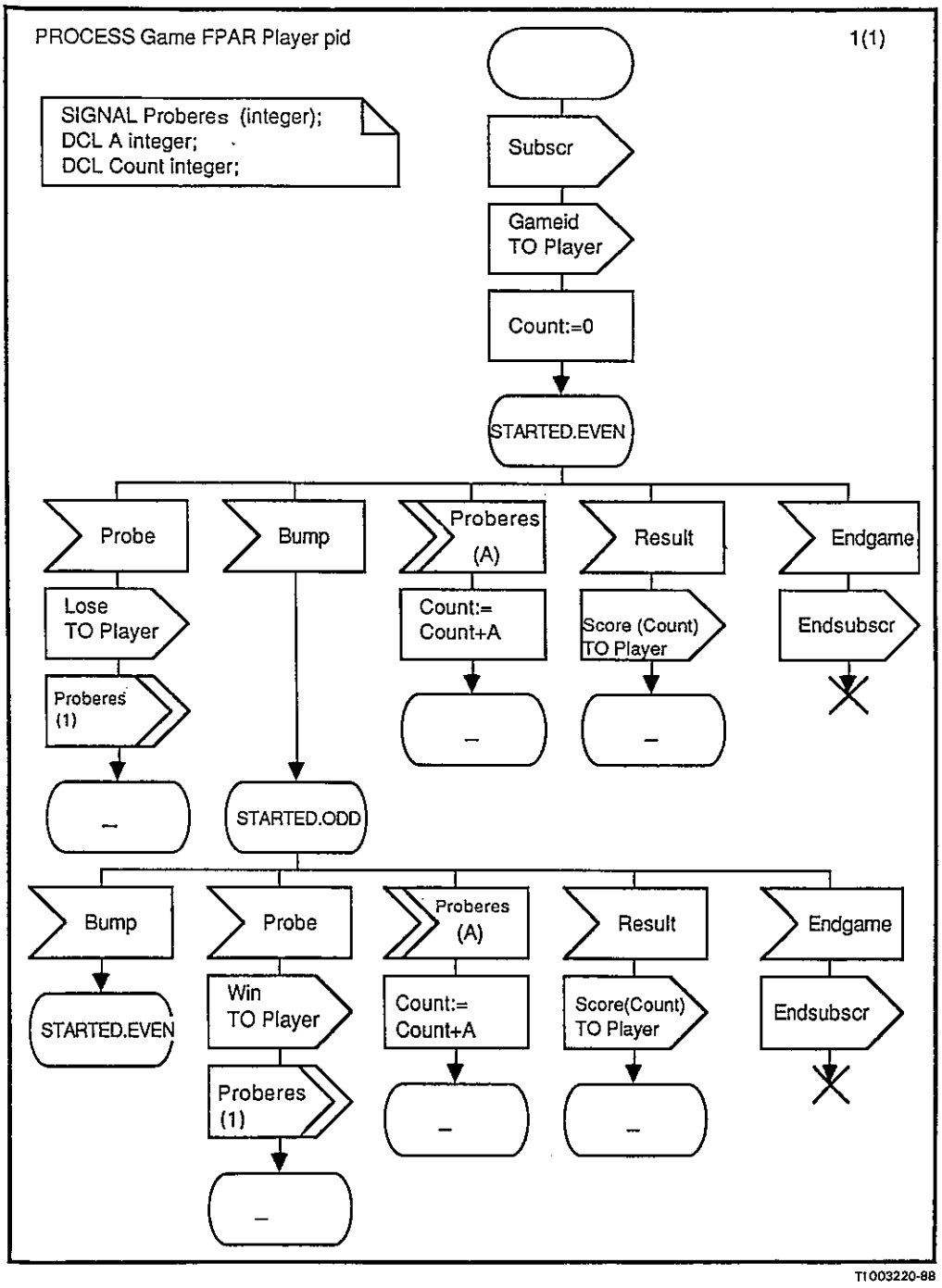


FIGURE 4.10.4

Exemple de transformation partielle

4.11 *Signal continu*

Lorsqu'on décrit un signal à l'aide du LDS, on se trouve souvent dans un cas où un usager aimerait décrire une transition causée directement par la valeur vraie 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 Vrai. 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

<signal continu> ::=
PROVIDED <expression booléenne> <fin>
[PRIORITY <nom de littéral entier> <fin>] <transition>

Les valeurs des <nom de littéral entier> dans les <signal continu> d'un <état> doivent être distinctes. La construction PRIORITY ne peut être omise que si l'<état> contient un <signal continu>.

Grammaire graphique concrète

<zone d'association de signal continu> ::=
<symbole d'association continue> **is connected to** <zone de signal continu>

<zone de signal continu> ::=
<symbole de condition de validation>
contains {<expression booléenne> [[<fin>] PRIORITY <nom de littéral entier>]}
is followed by <zone de transition>.

Sémantique

L'<expression booléenne> dans le <signal continu> 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 d'entrée> adjointe. Si la valeur de l'<expression booléenne> est Vrai, la transition a lieu. Si la valeur de l'<expression booléenne> est Vrai dans plus d'un <signal continu>, la transition qui doit être engagée est déterminée par le <signal continu> présentant la priorité la plus élevée, c'est-à-dire la valeur la plus faible pour le <nom de littéral entier>.

Modèle

L'état avec le nom state_name contenant les <signal continu> 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 emptyQ acheminant une valeur entière est nécessaire.

1) Tous les <état suivant> qui mentionnent state_name sont remplacés par JOIN 1;

2) La transition suivante est insérée:

```
1:  TASK n:=n+1;
    OUTPUT emptyQ (n) TO SELF;
    NEXTSTATE state_name;
```

3) La <partie entrée> suivante est ajoutée au <nom> state_name:

```
    INPUT emptyQ (newn);
```

et une <décision> contenant la <question>
(newn=n)

4a) La <partie réponse> correspondant à faux contient

```
    NEXTSTATE state_name;
```

4b) La <partie réponse> correspondant à vrai contient une séquence de <décision> correspondant aux <signal continu> dans l'ordre de priorité. La priorité la plus élevée est indiquée par la valeur la plus faible du <nom de littéral entier>.
La <partie réponse> correspondant à faux contient la <décision> suivante, sauf pour ce qui est de la dernière <décision> pour laquelle cette <partie réponse> contient: JOIN 1;
Chaque <partie réponse> vrai de ces <décision> conduit à la <transition> du <signal continu> correspondant.

Exemple

Voir le § 4.12

4.12 *Condition de validation*

Dans le LDS, la réception d'un signal 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

<condition de validation> ::=
PROVIDED <expression booléenne> <fin>

Grammaire graphique concrète

<zone de condition de validation> ::=
<symbole de condition de validation> **contains** <expression booléenne>

<symbole de condition de validation> ::=

Sémantique

L'<expression booléenne> dans la <condition de validation> est calculée 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 évalués séquentiellement dans un ordre non déterministe 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 d'entrée> qui précède la <condition de validation> peut déclencher la transition seulement si la valeur de l'<expression booléenne> correspondante est Vrai. Si cette valeur est Faux, le signal est alors mis en réserve.

Modèle

L'état nom state_name contenant les <conditions de validation> 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 emptyQ acheminant une valeur entière est requis.

1) Tous les <état suivant> qui mentionnent state_name sont remplacés par JOIN 1;

2) La transition suivante est insérée:

   1: TASK n:=n+1;
   OUTPUT emptyQ (n) TO SELF;

   Un certain nombre de décisions, dont chacune contient une seule <expression booléenne correspondant à une <condition de validation> associée à l'état, sont ajoutées hiérarchiquement dans un ordre non déterministe 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 <partie entrée> consistant en une copie de ces <partie entrée> de l'état sans les conditions de validation plus les <partie entrée> pour lesquelles les <expression booléenne> des <condition de validation> sont évaluées en donnant vrai pour cet état.

   Les <stimulus> pour les <partie entrée> restantes constituent la <liste de mise en réserve> pour une nouvelle <partie de mise en réserve> adjointe à cet état. Les <partie de mise en réserve> de l'état d'origine sont également copiées dans ce nouvel état.

4) Ajouter à chacun des nouveaux états:

   INPUT emptyQ (newn);

   Une <décision> contenant la <question> (newn=n);
   La <partie réponse> faux contient un <état suivant> ramenant à ce même nouvel état.

5) La <partie réponse> vrai contient un JOIN 1;

6) Si les <signaux continus> et les <conditions de validation> sont utilisés dans le même <état>, les évaluations des <expression booléenne> à partir des <signal continu> sont effectuées en remplaçant l'étape 5 du modèle pour la <condition de validation> par l'étape 4b du modèle pour le <signal continu>.

Exemple

L'exemple illustrant la transformation d'un signal continu et d'une condition de validation apparaissant dans un état est donné ci-après.

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.

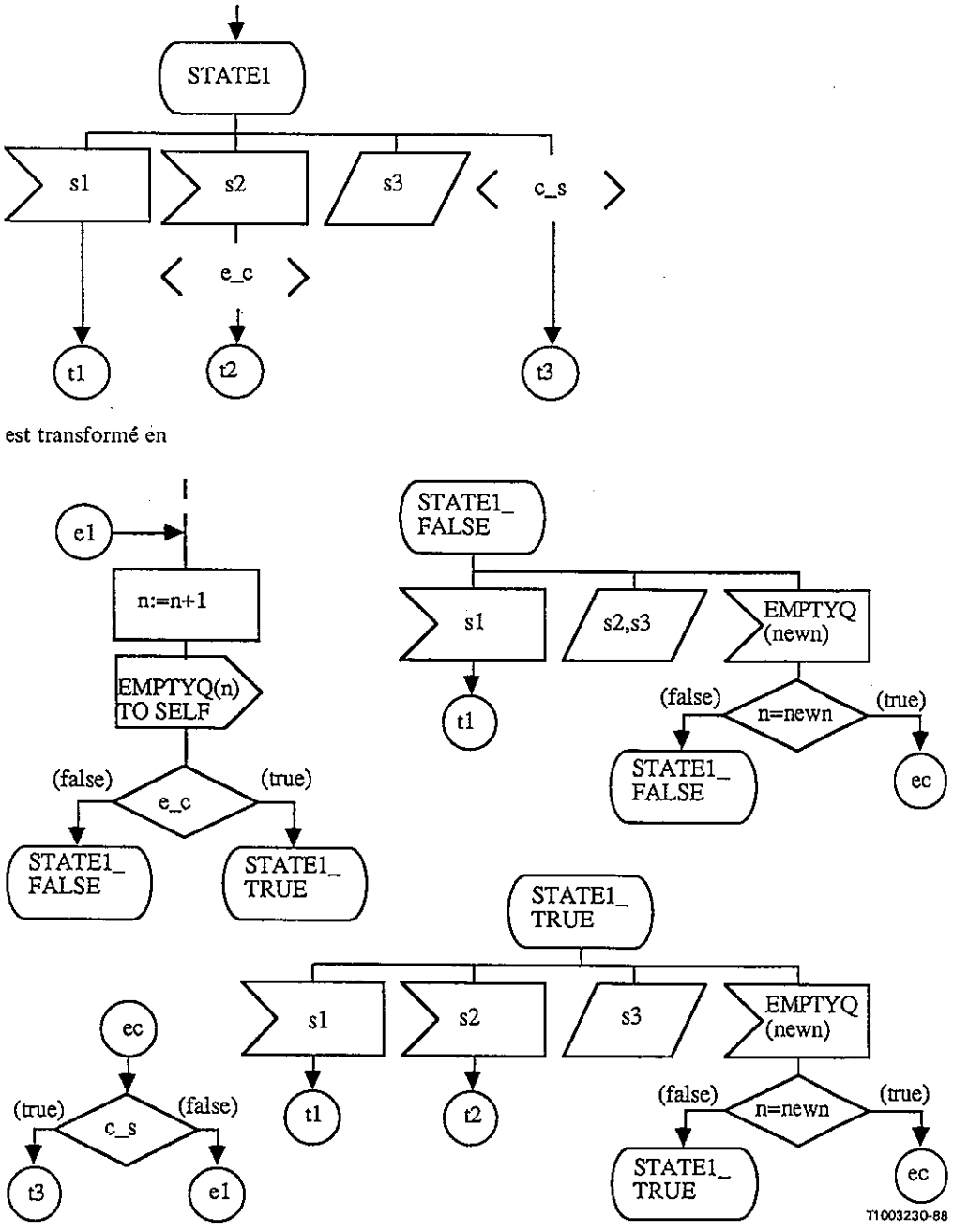


FIGURE 4.12.1

Transformation d'un signal continu et d'une condition de validation dans le même état

4.13 *Valeur importée et valeur exportée*

Dans le LDS, 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 partagée (voir le § 2) 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 valeur partagées.

Grammaire textuelle concrète

<définition d'import> ::=
IMPORTED <nom d'import> {, <nom d'import> }* <sorte>
{, <nom d'import> {, <nom d'import> }* <sorte>}* <fin>

<expression d'import> ::=
IMPORT (<identificateur d'import> [, <expression pid>])

<export> ::=
EXPORT (<identificateur de variable> {, <identificateur de variable> }*)

Grammaire graphique concrète

<zone d'export> ::=
<symbole de tâche> **contains** < export>

Sémantique

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 leur <définition de variable> 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 <définition d'import> dans un importateur, il y a un ensemble de variables implicites ayant le nom et la sorte donnés dans la <définition d'import>. Ces variables implicites sont utilisés pour le stockage des valeurs importées.

Une opération d'import est l'exécution d'une <expression d'import> dans laquelle un importateur accède à la valeur d'une variable exportée. La valeur est stockée dans une variable implicite indiquée par l'<identificateur d'import> dans l'<expression d'import>. L'exportateur contenant la variable exportée est spécifié par l'<expression pid> dans l'<expression d'import>. Si aucune <expression PId> n'est spécifiée, il ne doit y avoir qu'une instance exportant cette variable. L'association entre la variable exportée dans l'exportateur et la variable implicite dans l'importateur est spécifiée comme ayant le même <identificateur> dans l'<export> et dans l'<expression d'import>. De plus, la variable exportée et la variable implicite doivent avoir la même sorte.

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 affectation 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 <définition de signal> implicites pour chaque combinaison de <nom d'import> et de <sorte> contenue dans toutes les <définition d'import> dans une définition de système. Les <nom de signal dans ces <définition de signal> sont décrits par *xtQUERY* et *xtREPLY*, où *x* est un <nom d'import> et *t* une <sorte>. La copie implicite de la variable exportée est appelée *imcx*.

a) *Importateur*

L'<expression d'import> 'IMPORT (x,pidexp)' est transformée en ce qui suit:

```
OUTPUT xtQUERY TO pidexp;
Attendre dans l'état xtWAIT, en mettant en réserve tous les autres signaux;
INPUT xtREPLY (x);
Remplacer l'<expression d'import> par x, (le <nom> de la variable implicite);
```

Si une <expression d'import> apparaît plus d'une fois dans une <expression>, il y a une variable implicite distincte portant le même <nom> et qui est utilisée pour chaque occurrence.

b) *Exportateur*

A tous les <états>, y compris les états implicites, de l'exportateur, la <partie entrée> suivante est ajoutée:

```
INPUT xtQUERY;
OUTPUT xtREPLY (imcx) TO SENDER;
/* état suivant identique*/
```

L'<export> 'EXPORT (x)' est transformé en ce qui suit:

```
TASK imcx := x;
```

5 Données dans le LDS

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 œuvre 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 œuvre 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 LDS, 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 œuvre concrète. On trouvera au § 5.6 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 LDS 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 regroupe toutes les valeurs possibles qui peuvent être produites par l'ensemble associé d'opérateurs. Chaque valeur peut être décrite par au moins une expression dans le langage contenant seulement des littéraux et des opérateurs (à l'exception du cas spécial des valeurs Pid). Les littéraux constituent un cas particulier d'opérateurs sans argument.

Les sortes et les opérateurs, ainsi que le comportement (spécifié au moyen de règles algébriques) 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 déjà existante.

Un générateur est une description NEWTYPE incomplète: avant de décrire l'état d'une sorte, elle doit être «instanciée» en fournissant l'information manquante.

Certains opérateurs 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.

Dans le LDS, une fonction est connue comme étant un opérateur passif et qui peut n'avoir aucun effet sur les valeurs associées aux variables données en paramètres. Le LDS définit également l'affection qui peut modifier les valeurs associées aux variables.

5.1.3 *Terminologie*

La terminologie utilisée au § 5 ou le modèle de données est choisi 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'une valeur booléenne est équivalente à Vrai. Toutefois, le terme «axiome» est utilisé pour désigner des «axiomes» ou des «équations»; et une «équation» peut être un «axiome».

5.1.4 *Structure du texte sur les données*

Le modèle d'algèbre initiale utilisé dans le cas des données pour le LDS, est décrit de manière à permettre la définition de la plupart des concepts relatifs aux données en terme de noyau de données du langage de données abstrait du LDS.

Le texte du § 5 est divisé en plusieurs parties: l'introduction (§ 5.1), le langage de noyau de données (5.2), le modèle d'algèbre initiale (§ 5.3), utilisation passive des données (§ 5.4), utilisation active des données (§ 5.5) et les données prédéfinies (§ 5.6).

Le langage du noyau de données définit la partie des données dans le LDS qui correspond directement avec l'approche d'algèbre initiale sous-jacente.

Le texte concernant l'algèbre initiale est une introduction plus détaillée aux bases mathématiques de cette méthode. On trouvera à l'appendice 1 une formulation mathématique plus précise.

L'utilisation passive du LDS comprend les caractéristiques implicites et abrégées des données du LDS 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 LDS 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.

Le thème final concerne les données prédéfinies.

5.2 *Le langage de noyau de données*

Le noyau de données peut être utilisé pour définir des types abstraits de données.

On peut utiliser des constructions plus appropriées pour définir les types de données à l'aide des constructions définies pour le noyau de données, sauf pour les cas où il est nécessaire d'utiliser les concepts d'affectation à une variable (les concepts d'erreur et de syntype peuvent être définis en terme de noyau, mais dans les § 5.4.1.7 et § 5.4.1.9 d'autres définitions, plus concises sont utilisées).

5.2.1 *Définitions des types de données*

A un point quelconque d'une spécification LDS, 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 le § 5.3).

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 COMMENT '«nombres» pairs avec plus-dépendent de impair';
NEWTYPE odd LITERAL 1;
    OPERATORS  plusoe        : odd, even    -> odd;
               pluseo        : even, odd    -> odd;
    AXIOMS     plusoe(a,0)         ==a;
               pluseo(a,b)         ==plusoe(b,a);
    ENDNEWTYPE odd; /* les «nombres» impairs avec plus-dépendent de pair*/
```

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 d'une 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 au niveau du système, les sortes prédéfinies (par exemple booléennes ou entières) peuvent être librement utilisées dans le système.

Grammaire abstraite

Définition-de-type-de-données	::	*Type-de-données* *Union-de-type* *Sorte* *Signature*-**set** *Equations*
Union-de-types	=	*Identificateur-de-type*-**set**
Identificateur-de-type	=	*Identificateur*
Sortes	=	*Nom-de-sorte*-**set**
Nom-de-type	=	*Nom*
Nom-de-sorte	=	*Nom*
Equations	=	*Equation*-**set**

A l'intérieur d'une *définition-de-type-de-données* pour chaque sorte il doit y avoir au moins une signature avec un *résultat* (voir le § 5.2.2) qui est le même que la *sorte.*

Une *définition-de-type-de-données* ne doit pas ajouter de nouvelles valeurs à une *sorte* du type de données identifié par l'*union-de-type.*

Si un *terme* (voir le § 5.2.3) est non-équivalent à un autre *terme* selon le type de données identifié par l'*union-de-type* d'une *définition-de-type-de-données*, ces *termes* ne doivent pas être définis comme étant équivalents par la *définition-de-type-de-données.*

De plus, les deux *termes* booléens Vrai et Faux ne doivent pas être (directement ou indirectement) définis come étant équivalents (voir le § 5.4.3.1).

Remarque — La syntaxe abstraite permet d'harmoniser plus d'une identité de type pour une *union-de-type*, avec la catégorie plus générale des algèbres utilisées pour le modèle sous-jacent — en LDS, un seul type est référencé car dans la syntaxe concrète, le type de données visibles est implicitement défini par la <catégorie d'unité de portée> environnante: par conséquent l'<union de type> est uniquement référencée dans la syntaxe abstraite et est soit l'*identificateur-de-type* de l'unité de portée environnante ou dans le cas d'une <définition de système> un ensemble vide.

Grammaire textuelle concrète

<définition partielle de type> ::=
NEWTYPE <nom de sorte> [<propriétés étendues>] <expression des propriétés>
ENDNEWTYPE [<nom de sorte>]

<expression des propriétés> ::=
<opérateur> [AXIOMS <axiomes>] [<correspondence de littéraux>] [<affectation par défaut>]

Les <propriétés étendues>, la <correspondance de littéraux> et l'<affectation par défaut> optionnelles ne font pas partie du noyau de données et sont respectivement définies dans les § 5.4.1, § 5.4.1.15 et § 5.5.3.3.

La *définition-de-type-de-données* est représentée par l'ensemble de toutes les <définition partielle de type> dans la <catégorie d'unité de portée> courante associée avec la *définition-de-type-de-données* identifiée par l'*union de type* de la <catégorie d'unité de portée> environnante. Le nom de type d'une <définition de type de données> est implicite et n'a pas de représentation dans la syntaxe concrète. L'*Identificateur-de-type* d'une *union-de-type* est implicitement l'identité de la *définition-de-type-de-données* de l'unité de portée environnante.

Chacune des <catégorie d'unité de portée> suivantes (voir le § 2.2.2) représente un point dans la syntaxe abstraite qui contient une *définition-de-type-de-données*: <définition de système>, <définition de bloc>, <définition de processus>, <définition de procédure>, <définition de sous-structure de canal> ou <définition de sous-structure de bloc> ou les diagrammes correspondants dans la syntaxe graphique. La <définition partielle de type> dans une <définition de service> représente une partie de la *définition-de-type-de-données* dans la <définition de processus> englobante de la <définition de service> (voir le § 4.10).

Les *sortes* pour une <catégorie d'unités de portée> sont représentées par l'ensemble des <nom de sorte> introduits par l'ensemble des <définition partielle de type> de la <catégorie d'unités de portée>.

L'ensemble *signature* et *équations* pour une <catégorie d'unités de portée> sont représentées par les <expression de propriétés> des <définition partielle de type> de la <catégorie d'unité de portée>.

Les <opérateur> d'une <expression de propriété> représentent une partie de l'ensemble *signature* dans la syntaxe abstraite. L'ensemble complet *signature* est l'union des ensembles *signature* définis par les <définition partielle de type> dans la <catégorie d'unité de portée>.

Les <axiomes> d'une <expression de propriétés> représentent la partie de l'ensemble d'*équation* dans la syntaxe abstraite. Les *équations* sont l'union des ensembles d'*équation* définis par la <définition partielle de type> dans la <catégorie d'unité de portée>.

Les sortes de données prédéfinies ont leur <définition partielle de type> implicite au niveau du système.

Si un <nom de sorte> est donné après le mot clé ENDNEWTYPE, il doit être le même que le <nom de sorte> donné après le mot clé NEWTYPE.

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 se rapportant à la sorte introduite par la définition partielle de type. 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. La définition du système contient la définition des sortes de données prédéfinies.

Sauf à l'intérieur d'une <définition partielle de type>, d'un <affinage de signal> ou d'une <définition de service>, 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 partielle de type> ou d'un <affinage de signal>, 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 partielle de type> ou l'«affinage de signal>. A l'intérieur d'une <définition de service> il y a la *définition-de-type-de-données* de la <définition de processus> englobante de la <définition de service> concernée (voir le § 4.10).

L'ensemble des sortes d'un type de données est l'ensemble des sortes introduites dans l'unité de portée courante auquel on ajoute l'ensemble des sortes de type de données identifié par l'union de type. L'ensemble des opérateurs d'un type de données est l'ensemble des opérateurs introduits dans l'unité de portée courante auquel on ajoute l'ensemble des opérateurs du type de données identifié par l'union de type. L'ensemble des équations d'un type de données est l'ensemble des équations introduites dans l'unité de portée courante auquel on ajoute l'ensemble des équations du type de données identifié par l'union de type.

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.

Un type de données a un identificateur qui est le nom de type unique dans la syntaxe abstraite qualifié par l'identité de l'unité de portée. Il n'y a pas de nom pour un type de données dans la syntaxe concrète.

Exemple

```
NEWTYPE téléphone
    /*opérateurs et construction des valeurs définis ailleurs*/
ENDNEWTYPE téléphone;
```

5.2.2 *Littéraux et opérateurs paramétrisés*

Grammaire abstraite

Signature	=	*Signature-de-littéral* | *Signature-d-opérateur*
Signature-de-littéral	::	*Nom-de-littéral-opérateur* *Résultat*
Signature-d-opérateur	::	*Nom-d-opérateur* *Liste-d-arguments* *Résultat*
Liste-d-arguments	=	*Identificateur-de-référence-de-sorte* $^+$
Résultat	=	*Identificateur-de-référence-de-sorte*
Identificateur-de-référence-de-sorte	=	*Identificateur-de-sorte* | *identificateur-de-syntype*

Nom-d-opérateur-littéral = *Nom*

Nom-d-opérateur = *Nom*

Identificateur-de-sorte = *Identificateur*

Les syntypes et les *identificateur-de-syntype* ne font pas partie du noyau (voir le § 5.4.1.9).

Grammaire textuelle concrète

<opérateurs> ::=
[<liste de littéraux>] [<liste d'opérateurs>]

<liste de littéraux> ::=
LITERALS <signature de littéral> {, <signature de littéral> }* [<fin>]

<signature de littéral> ::=
<nom d' opérateur de littéral>
| <nom de littéral étendu>

<liste d'opérateurs> ::=
OPERATORS
<signature d'opérateur> { <fin> <signature d'opérateur> }* [<fin>]

<signature d'opérateur> ::=
<nom d'opérateur> : <liste d'arguments> -> <résultat>
| <relation d'ordre>

<nom d'opérateur> ::=
<nom d'opérateur>
| <nom d'opérateur étendu>

<liste d'arguments> ::=
<sorte d'argument> {, <sorte d'argument> }*

<sorte d'argument> ::=
<sorte étendue>

<résultat> ::=
<sorte étendue>

<sorte étendue> ::=
<sorte>
| <sorte de générateur>

<sorte> ::=
<identificateur de sorte>
| <syntype>

Les alternatives <nom d'opérateur étendu>, <nom de littéral étendu> <relation d'ordre>, <sorte de générateur>, <sorte de générateur> et <syntype> ne font pas partie du noyau des données et sont définies dans les § 5.4.1, § 5.4.1, § 5.4.1.8, § 5.4.1.12.1, § 5.4.1.12.1 et § 5.4.1.9 respectivement.

Les littéraux sont introduits par les <signature de littéral> figurant après le mot clé LITERALS. Le *résultat* d'une *signature-de-littéral* est la sorte introduite par la <définition partielle de type> définissant le littéral.

Chaque <signature d'opérateur> dans la liste des <signature d'opérateur> après le mot clé OPERATORS représente une *signature-d-opérateur* avec un *nom-d-opérateur*, une *liste-d-arguments* et un *résultat.*

Le <nom d'opérateur> correspond à un *nom-d-opérateur* 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-d-opérateur-de-littéral*, dans la syntaxe abstraite est obtenu à partir:

a) du <nom d'opérateur> (ou du <nom d'opérateur de littéral>), 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 d'opérateur de littéral>) est défini.

Chaque fois qu'un <identificateur d'opérateur>, se trouve spécifié, on obtient l'unique *nom-d-opérateur* dans l'*identificateur-d-opérateur* 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> qui diffèrent par une ou plusieurs sortes d'arguments ou de résultats ont des *noms* différents.

Chaque <sorte d'argument> dans une <liste d'arguments> représente un *identificateur-de-référence-de-sorte* dans une *liste-d-arguments.* Un <résultat> représente l'*identificateur-d-référence-de-sorte* d'un *résultat.*

Lorsqu'un <qualificatif> d'un <identificateur d'opérateur> (ou d'un <identificateur d'opérateur littéral>) contient un <élément de chemin> avec le mot clé TYPE, le <nom de sorte> après ce mot clé ne fait pas partie du *qualificatif* de l'*identificateur-d-opérateur* (ou de l'*identificateur-d-opérateur-de-littéral*) mais est utilisé pour obtenir le *nom* unique de l'*identificateur.* Dans ce cas, le qualificatif est formé à partir de la liste des <élément de chemin> 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 la 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.

Remarque – A titre d'indication: une <signature d'opérateur> doit mentionner la sorte introduite par la <définition partielle de type> englobante comme étant soit un <argument>, soit un <résultat>.

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	=	*Equation-non-quantifiée* | *Equations-quantifiées* | *Equation-conditionnelle* | *Texte-informel*
Equation-non-quantifiée	::	*Terme* *Terme*
Equations-quantifiées	::	*Nom-de-valeur*-**set** *Identificateur-de-sorte* *Equations*
Nom-de-valeur	=	*Nom*
Terme	=	*Terme-clos* | *Terme-composite* | *Terme-d-erreur*
Terme-composite	::	*Identificateur-de-valeur* | *Terme-identificateur-d'opérateur*$^{+}$ | *Terme-composite-conditionnel*
Identificateur-de-valeur	=	*Identificateur*
Identification-d-opérateur	=	*Identificateur*

Terme-clos :: *Identificateur-d-opérateur-littéral* |
Terme-clos-identificateur-d-opérateur^{+} |
Terme-clos-conditionnel

Identificateur-d-opérateur-de-littéral = *Identificateur*

Les possibiltés *terme-composite-conditionnel* et *terme-clos-conditionnel* dans les règles *terme-composite* et *terme-clos* 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 le § 5.4.1.6). La possibilité *terme-d-erreur* règle *terme* ne fait pas partie du noyau de données et est définie au § 5.4.1.7.

Les définitions de *texte-informel* y *équation-conditionnelle* sont respectivement données aux § 2.2.3 et 5.2.4.

Chaque *terme* (ou *terme-clos*) dans la liste des termes après un *identificateur-d-opérateur* doit avoir la même sorte que la sorte correspondante (position) dans la *liste-d-arguments* de la *signature-d-opérateur.*

Les deux *termes* dans une *équation-non-quantifiée* doivent être de la même sorte.

Grammaire textuelle concrète

<axiomes> ::=
<équation> { <fin> <équation>}* [<fin>]

<équation> ::=
<équation non quantifiée>
| <équation quantifiée>
| <équation conditionnelle>
| <texte informel>

<équation quantifiée> ::=
<quantification>(<axiomes>)

<quantification> ::=
FOR ALL <nom de valeur> {, <nom de valeur> }* IN <sorte étendue>

<équation non quantifiée> ::=
<terme> == <terme>
| <axiome booléen>

<terme> ::=
<terme clos>
| <terme composé>
| <terme d'erreur>
| <terme d'énoncé>

<terme composite> ::=
<identificateur de valeur>
| <identificateur d'opérateur>(<liste de termes composites>)
| (<terme composite>)
| <terme composite étendu>

<liste de terme composite> ::=
<terme composite> {, <terme>}*
| <terme>, <liste de termes composites>

<terme clos> ::=
<identificateur de littéral>
| <identificateur d'opérateur> (<terme clos> {,<terme clos>}*)
(<terme clos>)
| (<terme clos étendu>)

<identificateur de littéral> ::=
<identificateur d'opérateur de littéral>
| <identificateur de littéral étendu>

Les possibilités <axiome booléen> de la règle <équation non quantifiée>, <terme d'erreur> et <terme d'énoncé> de la règle <terme>, <terme composite étendu> de la règle <terme composite>, <terme clos étendu> de la règle <terme clos> et <identificateur de littéral étendu> de la règle <identificateur de littéral> ne font pas partie du noyau de données et sont respectivement définies aux § 5.4.1.5, 5.4.1.7, 5.4.1.15, 5.4.1, 5.4.1 et 5.4.1.

La <sorte> dans une <quantification> représente l'*identificateur-de-* sorte dans les *équations-quantifiées.* Le <nom de valeur> dans une <quantification> représente l'ensemble *nom-de-valeur* dans les *équations-quantifiées.*

Une <liste de termes composites> représente une liste de *terme*. Un *identificateur-d-opérateur* suivi par une liste de *terme-composite* seulement si la liste de *terme* contient au moins un *identificateur-de-valeur*.

Un <identificateur> qui est un nom qualifié apparaissant dans un *terme* représente:

a) un *identificateur-d-opérateur* s'il précède une parenthèse ouverte (ou si c'est un <nom d'opérateur> qui est un <nom d'opérateur étendu> (voir le § 5.4.1), sinon

b) un *identificateur-de-valeur* s'il y a une définition de ce nom dans une <quantification> d'<équation quantifiée> englobant le <terme> d'une sorte adaptée à ce contexte, sinon

c) un *identificateur-d-opérateur-de-littéral* s'il y a un littéral visible portant ce nom d'une sorte adaptée au contexte, sinon

d) un *identificateur-de-valeur* qui a une *équation-quantifiée* implicite dans la syntaxe asbtraite pour l'<équation non quantifiée>.

Deux occurrences au moins du même <identificateur de valeur> non liées dans une <équation> impliquent seulement une *quantification*.

Un *identificateur-d-opérateur* est obtenu à partir du contexte de manière à ce que si le <nom d'opérateur> est surchargé (c'est-à-dire que le même <nom> est utilisé pour plusieurs opérateurs), il sera alors le *nom-d-opérateur* 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 d'opérateur> 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 d'opérateur*.

Dans une <équation non quantifiée> 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 d'opérateur> ou <identificateur d'opérateur littéral> non qualifié à exactement un *identificateur-d-opérateur* ou un *identificateur-d-opérateur-de-littéral* défini qui satisfait la condition dans la construction dans laquelle l'<identificateur> est utilisé. C'est-à-dire que la liaison doit être unique.

Remarque – A titre d'information: 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 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 LDS.

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-de-référence-de-sorte*. 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 littéraux désignent la même valeur, chaque littéral 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

Equation-conditionnelle	::	*Restriction*-**set** *Equation-restreinte*
Restriction	=	*Equation-non-quantifiée*
Equation-restreinte	=	*Equation-non-quantifiée*

Grammaire textuelle concrète

<équation conditionnelle> ::=
<restriction> {, <restriction> }* == > <équation restreinte>

<équation restreinte> ::=
<équation non quantifiée>

<restriction> ::=
<équation non quantifiée>

Sémantique

Une équation restreinte 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. Une valeur satisfera une restriction seulement si la restriction peut être déduite des autres équations pour cette valeur.

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) Définissons une équation conditionnelle prouvable: c'est une équation conditionnelle pour laquelle 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 correpondant 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 *Modèle d'algèbre initiale (description informelle)*

La définition des données dans le LDS est fondée sur le noyau de données défini au § 5.2. Il est nécessaire de donner une signification plus approfondie aux opérateurs et aux valeurs en addition à la définition précédente, afin que l'on puisse interpréter les expressions; citons à titre d'exemple, les expressions utilisées dans les signaux continus, les conditions de validation, les appels de procédure, les actions de sortie, les demandes de création, les instructions d'affectation, les instructions d'initialisation et de réinitialisation, les instructions d'export, les instructions d'import, les décisions et la visibilité.

La signification supplémentaire nécessaire est donnée aux expressions en utilisation le formalisme de l'algèbre initiale qui est expliqué dans les § 5.3.1 à 5.3.6 ci-après [1].

En tout point d'une spécification LDS, le dernier type de données hiérarchiquement défini s'appliquera mais il y aura un ensemble de sortes visibles. L'ensemble de sortes sera l'union de toutes les sortes à des niveaux hiérarchiques situés hiérarchiquement au-dessus du point en question ainsi qu'il est expliqué au § 5.2.

(Dans la présente section le symbole = est utilisé comme un symbole d'équivalence d'équation tandis que le symbole LDS = = est utilisé pour une équivalence d'équation si bien que le symbole = peut être utilisé comme opérateur d'égalité. Le symbole = est utilisé dans la présente section comme symbole conventionnel utilisé dans les travaux qui ont été publiés sur les algèbres initiales.)

5.3.1 *Introduction*

La signification et l'interprétation des données fondées sur l'algèbre initiale sont expliquées en trois étapes:

a) Signatures

b) Termes

c) Valeurs

5.3.1.1 *Représentation*

L'idée que différentes notations puissent représenter le même concept est courante. Par exemple, on accepte généralement que les nombres positifs arabes (1,2,3,4,...) et les chiffres romains (I,II,III,IV,...) représentent un même ensemble de nombres avec les mêmes propriétés. Voici un autre exemple, il est assez habituel d'accepter une notation fonctionnelle préfixée (plus (1,1)), une notation infixée (1 + 1) et une notation polonaise inverse (1 1 +) puissent représenter le même opérateur. De plus, différents utilisateurs peuvent employer différents noms (peut-être parce qu'ils utilisent différentes langues) pour les mêmes notions si bien que les paires {vrai, faux}, {V, F} {0,1}, {true, false} puissent être des représentations différentes de la sorte booléenne.

Ce qui est essentiel est la relation abstraite entre les identités et non la représentation concrète. Ainsi, pour les chiffres ce qui est intéressant est la relation entre 1 et 2 qui est la même que la relation entre I et II. Aussi pour les opérateurs ce qui est important est la relation entre l'identité d'un opérateur et les autres identités d'opérateur et la liste des arguments. Les constructions concrètes telles les parenthèses qui nous permettent de faire la distinction entre (a + b)*c et a + (b*c) ne présentent un intérêt que si la notion abstraite sous-jacente peut être déterminée.

Ces notions abstraites sont incorporées dans une syntaxe abstraite de la notion qui peut être réalisée par plusieurs syntaxes concrètes. Par exemple, dans ce qui suit, deux exemples concrets décrivent à la fois les mêmes propriétés de type de données mais dans des syntaxes concrètes différentes.

---

[1] Le texte des § 5.3.1 à 5.3.6 a été adopté par l'ISO et le CCITT comme étant une description commune informelle du modèle d'algèbre initiale pour les types de données abstraites. Apparaissant dans la présente Recommandation, cet texte (avec les modifications typographiques et de chiffres appropriées) constitue également une annexe à ISO IS8807.

```
NEWTYPE     bool LITERALS true, false;
OPERATORS   "not" :bool->bool;
AXIOMS
        not(true)           ==false;
        not(not(a))         ==a;
ENDNEWTYPE bool;

NEWTYPE int LITERALS zero, one;
OPERATORS   plus        :int,int->int;
            minus       :int,int->int;
AXIOMS
        plus(zero,a)                ==a;
        plus(a,b)                   ==plus(b,a);
        plus(a,plus(b,c))           ==plus(plus(a,b),c);
        minus(a,a)                  ==zero;
        minus(a,zero)               ==a;
        minus(a,minus(b,c)          ==minus(plus(a,c),b);
        minus(minus(a,b),c)         ==minus(a,plus(b,c));
        plus(minus(a,b),c)          ==minus(plus(a,c),b);
ENDNEWTYPE int;

NEWTYPE tree LITERALS nil;
OPERATORS
        tip     :int        ->tree;
        isnil   :tree       ->bool;
        istip   :tree       ->bool;
        node    :tree,tree  ->tree;
        sum     :tree       ->int;
AXIOMS
        istip(nil)              ==false;
        istip(tip(i))           ==true;
        istip(node(t1,t2))      ==false;
        isnil(nil)              ==true;
        isnil(tip(i))           ==false;
        isnil(node(t1,t2))      ==false;
        sum(node(t1,t2))        ==plus(sum(t1),sum(t2));
        sum(tip(i))             ==i;
        sum(nil)                ==zero;
ENDNEWTYPE tree;
```

EXEMPLE 1

```
TYPE      bool        IS
SORTS     bool
OPNS      true  :     ->bool
          false :     ->bool
          not   : bool->bool
EQNS OFSORT           bool FORALL a:bool
      not(true)       =false;
      not(not(a))     =a
ENDTYPE

TYPE      int  IS     bool WITH
SORTS     int
OPNS      zero :                ->int
          one  :                ->int
          plus :      int,int   ->int
          minus:      int,int   ->int
EQNS OFSORT           int FORALL a,b,c:int
      plus(zero,a)                =a;
      plus(a,b)                   =plus(b,a);
      plus(a,plus(b,c))           =plus(plus(a,b),c);
      minus(a,a)                  =zero;
      minus(a,zero)               =a;
      minus(a,minus(b,c))         =minus(plus(a,c),b);
      minus(minus(a,b),c)         =minus(a,plus(b,c));
      plus(minus(a,b),c)          =minus(plus(a,c),b)
ENDTYPE

TYPE      tree IS     int   WITH
SORTS     tree
OPNS      nil  :              ->tree
          tip  :int           ->tree
          isnil :tree         ->bool
          istip :tree         ->bool
          node:tree,tree      ->tree
          sum  :tree          ->int
EQNS OFSORT           bool FORALL i:int, t1,t2:tree
          istip(nil)                  =false;
          istip(tip(i))               =true;
          istip(node(t1,t2))          =false
          isnil(nil)                  =true
          isnil(tip(i))               =false
          isnil(node(t1,t2))          =false
      OFSORT          int FORALL i:int, t1,t2:tree
          sum (node(t1,t2))           =plus(sum(t1),sum(t2));
          sum(tip(i))                 =i;
          sum(nil)                    =zero
ENDTYPE
```

EXEMPLE 2

Nous allons utiliser cet exemple pour illustrer le sujet. Nous examinerons d'abord la définition des sortes et des littéraux.

Il faut remarquer que les littéraux sont considérés comme étant un cas particulier d'opérateurs, c'est-à-dire des opérateurs sans paramètre.

Nous pouvons introduire certaines sortes et certains littéraux sous la première forme par:

```
NEWTYPE int LITERALS zero, one; ...
NEWTYPE bool LITERALS true, false; ...
NEWTYPE tree LITERALS nil; ...
```

ou sous la seconde forme par:

```
...
SORTS   bool
OPNS    true  :     ->bool
        false :     ->bool
...
SORTS   int
OPNS    zero  :     ->int
        one   :     ->int
...
SORTS   tree
OPNS    nil   :     ->tree
...
```

Pour ce qui suit, on utilisera uniquement la deuxième forme car elle est la plus proche de la formulation utilisée dans de nombreuses publications relatives à l'algèbre initiale. Il faut remarquer que la forme des termes est la même dans les deux cas et que la différence la plus sensible repose sur la façon dont les littéraux sont présentés. Il faut se souvenir qu'il est nécessaire d'adopter une notation concrète pour communiquer les concepts, mais que la signification des algèbres est indépendante de la notation et qu'ainsi un changement systématique de nom (retenant la même unicité) et une modification due au passage de la notation avec préfixe à la notation polonaise ne modifiera pas la sémantique définie par les définitions de type.

5.3.2 *Signature*

Un ou plusieurs opérateurs seront associés à chaque sorte. Chaque opérateur a une fonctionnalité d'opérateur c'est-à-dire qu'il est défini pour établir une relation entre une ou plusieurs sortes d'entrée et une sorte résultante.

Par exemple, les opérateurs ci-après peuvent être ajoutés aux sortes définies ci-dessus

```
...
SORTS   bool
OPNS    true  :     ->bool
        false :     ->bool
        not   : bool->bool
...
SORTS   int
OPNS    zero  :                 ->int
        one   :                 ->int
        plus  :     int,int     ->int
        minus:      int,int     ->int
...
SORTS   tree
OPNS    nil   :                 ->tree
        tip   :     int         ->tree
        isnil :     tree        ->bool
        istip :     tree        ->bool
        node:       tree,tree   ->tree
        sum   :     tree        ->int
...
```

La signature du type qui s'applique ici est l'ensemble des sortes, et l'ensemble des opérateurs (à la fois des littéraux et des opérateurs avec des paramètres) qui sont visibles.

La signature d'un type est appelée complète (fermée) si pour chaque opérateur dans la signature, les sortes de la fonctionnalité de l'opérateur sont incluses dans l'ensemble des sortes du type.

5.3.3 *Termes et expressions*

Le langage étudié est celui qui autorise des expressions qui sont des variables, des littéraux ou des opérateurs appliqués à des expressions. Une variable est un objet qui est associé à une expression. L'interprétation d'une variable peut être remplacée par l'interprétation de l'expression associée à la variable. De cette manière, les variables peuvent être éliminées de manière à ce que l'interprétation d'une expression puisse être réduite à l'application de divers opérateurs sur des littéraux.

Ainsi, à l'interprétation, une expression ouverte (une expression faisant intervenir des variables) devient une expression close (expression sans variable) en fournissant à l'expression ouverte les arguments réels (c'est-à-dire des expressions closes).

Una expression close correspond à un terme clos.

L'ensemble de tous les termes clos possibles d'une sorte est appelé l'ensemble des termes clos de la sorte. Par exemple, pour bool tel qu'il est défini dans l'exemple ci-dessus, l'ensemble des termes clos contiendra:

{true, false, not(true), not(false), not(not(true)), . . .}

On peut constater que même pour cette sorte très simple, l'ensemble de termes clos est infini.

5.3.3.1 *Production de termes*

Etant donné une signature d'un type, il est possible de produire un ensemble de termes clos pour le type en question.

L'ensemble des littéraux du type est considéré comme étant l'ensemble de base des termes clos. Chaque littéral a une sorte, par conséquent chaque terme clos a une sorte. Pour les types qui ont été définis ci-dessus, l'ensemble de base des termes clos sera:

{zero, one, true, false, nil}

Pour chaque opérateur de l'ensemble des opérateurs du type, les termes clos sont produits en substituant pour chaque argument tous les termes clos précédemment produits de la sorte correcte pour cet argument. La sorte résultante pour chaque opérateur est la sorte des termes clos produits par cet opérateur. L'ensemble résultant des termes clos est ajouté à l'ensemble existant des termes clos afin de produire un nouvel ensemble de termes clos. Pour le type ci-dessus, cet ensemble est le suivant:

{zero,	one,	true,	false,	nil,
plus(zero,zero),	plus(one,one),	plus(zero,one),	plus(one,zero),	
minus(zero,zero),	minus(one,one),	minus(zero,one),	minus(one,zero),	
not(true),	not(false),	tip(zero),	tip(one),	
isnil(nil),	istip(nil),	node(nil,nil),	sum(nil),	}

Ce nouvel ensemble de termes clos est ensuite pris comme étant l'ensemble précédent des termes clos pour une nouvelle application du dernier algorithme pour produire un nouvel ensemble de termes clos. Cet ensemble de termes clos comprendra:

{zero,	one,	true,	false,	nil,
plus(zero,zero)),	plus(one,one),	plus(zero,one),	plus(one,zero),	. . .
plus(zero,plus(zero,zero)),		plus(zero,plus(one,one)),		. . .
plus(zero,sum(nil)),		. . .		
isnil(node(nil,nil)),		istip(node(nil,nil)),	node(nil,node(nil,nil)),	
. . . ,			sum(node(nil,nil)) }	

Cet algorithme est appliqué de manière répétitive afin de produire tous les termes clos possibles pour le type qui est l'ensemble des termes clos du type. L'ensemble des termes clos d'une sorte est l'ensemble des termes clos du type qui a cette sorte:

Normalement, la production continuera indéfiniment conduisant à un nombre infini de termes.

5.3.4 *Valeurs et algèbres*

Chaque terme d'une sorte représente une valeur de cette sorte. On peut voir de ce qui précède que même une sorte simple comme bool a un nombre infini de termes et par conséquent un nombre infini de valeurs, à moins qu'une certaine définition soit donnée indiquant comment les termes sont équivalents (représentent la même valeur). Cette définition est donnée au moyen d'équations définies sur les termes. En l'absence de istip et isnil, la sorte bool peut être limitée à deux valeurs par les équations.

not(true) = false;
not(false) = true

De telles équations définissent l'équivalence des termes et il est ensuite possible d'obtenir deux catégories équivalentes de termes:

{ true,	not(false),	not(not(true)),	not(not(not(false))), . . .}
{ false,	not(true),	not(not(false)),	not(not(not(true))), . . .}

Chaque classe d'équivalence représente alors une valeur et les membres de la classe sont différentes représentations de la même valeur.

Remarquons qu'à moins qu'on ne définisse leur équivalence par des équations, les termes ne sont pas équivalents (c'est-à-dire qu'ils ne représentent pas la même valeur).

Une algèbre définit l'ensemble des termes qui satisfont à la signature de l'algèbre. Les équations de l'algèbre établissent des relations entre les termes.

En général, il y aura plus d'une seule représentation pour chaque valeur d'une sorte dans une algèbre.

Une algèbre pour une signature donnée est une algèbre initiale si et seulement si toute autre algèbre qui confère les mêmes propriétés à la signature peut être systématiquement transformée dans l'algèbre initiale. (De manière formelle, une telle transformation est connue sous le nom d'homomorphisme).

Pourvu que not, istip et isnil produisent toujours des valeurs dans les classes d'équivalences de true et false, une algèbre initiale pour bool est la paire de littéraux.

{true, false}

et pas d'équation.

5.3.4.1 *Equations et quantification*

Pour une sorte comme bool, où il y a seulement un nombre limité de valeurs, toutes les équations peuvent être écrites en utilisant uniquement des termes clos, c'est-à-dire des termes qui contiennent seulement des littéraux et des opérateurs.

Lorsqu'une sorte contient beaucoup de valeurs, l'écriture de toutes les équations utilisant les termes clos n'est pas commode et pour les sortes ayant un nombre infini de valeurs (tel les entiers), une énumération explicite devient impossible. La technique d'écriture des équations quantifiées est utilisée pour représenter un ensemble éventuellement infini d'équations au moyen d'une équation quantifiée.

Une équation quantifiée contient des identificateurs de valeurs dans des termes. Ces termes sont appelés termes composites. L'ensemble des équations qui ne contiennent que des termes clos peut être obtenu à partir de l'équation quantifiée en produisant systématiquement des équations avec chaque identificateur de valeur substitué dans l'équation par un des termes clos de la sorte de l'identificateur de valeur. Ainsi:

```
FOR ALL b : bool      not(not(b)) = b
```

représente

```
not(not(true))    = true;
not(not(false))   = false
```

Un autre ensemble d'équations pour bool peut être maintenant le suivant:

```
FOR ALL b : bool
    not(not(b))       = b;
    not(true)         = false
```

Lorsque la sorte de l'identificateur de valeur quantifiée est évidente à partir du contexte, on omet habituellement la clause définissant l'identificateur de valeur si bien que l'exemple devient:

```
not(not(b))       = b;
not(true)         = false
```

5.3.5 *Spécification algébrique et sémantique (signification)*

Une spécification algébrique comprend une signature et un ensemble d'équations por chaque sorte de cette signature. Ces ensembles d'équations induisent des relations d'équivalence qui définissent la sémantique de la spécification.

Le symbole = indique une relation d'équivalence qui satisfait les propriétés de réflexivité, de symétrie et de transitivité et la propriété de substitution.

Les équations données avec un type permettent de placer les termes dans des classes d'équivalence. Deux termes dans une même classe d'équivalence sont interprétés comme ayant la même valeur. Ce mécanisme peut être utilisé pour identifier syntaxiquement des termes différents dont on veut qu'ils aient la même valeur.

Deux termes de la même sorte, TERM1 et TERM2, appartiennent à la même classe d'équivalence si:

a) il y a une équation
   TERM1 = TERM2,
   ou

b) une des équations obtenues à partir de l'ensemble donné des équations quantifiées est:
   TERM1 = TERM2,
   ou

c) i) TERM1 appartient à une classe d'équivalence contenant TERMA, et

   ii) TERM2 appartient à une classe d'équivalence contenant TERMB, et

   iii) il y a une équation ou une équation obtenue à partir de l'ensemble donné des équations quantifiées telles que

   TERMA = TERMB,

   ou

d) En substituant un sous-terme de TERM1 par un terme de la même classe que le sous-terme produisant un terme TERM1A, il est possible de montrer TERM1A appartient à la même classe que TERM2.

En appliquant toutes les équations, les termes de chaque sorte sont répartis en une ou plusieurs classes d'équivalence. Il y a autant de valeurs pour la sorte qu'il y a de classes d'équivalence. Chaque classe d'équivalence représente une valeur et chaque membre d'une classe représente la même valeur.

5.3.6 *Représentation des valeurs*

L'interprétation d'une expressionsignifie donc qu'il faut d'abord en déduire le terme clos en déterminant la valeur réelle des variables utilisées dans l'expression au point d'interprétation, puis trouver la classe d'équivalence pour ce terme clos. La classe d'équivalence de ce terme détermine la valeur de l'expression.

On donne ainsi une sémantique aux opérateurs utilisés dans les expressions en déterminant la valeur résultante étant donné un ensemble d'arguments.

Il est habituel de choisir un littéral dans la classe d'équivalence pour représenter la valeur de la classe. Par exemple, bool sera représenté par true et false et les nombres naturels par 0,1,2,3, etc. Quand il n'y a pas de littéral, on utilise habituellement un terme dont la complexité est la plus faible possible (le plus petit nombre d'opérateurs). Par exemple, pour les entiers négatifs la notation usuelle est -1, -2, -3, etc..

5.4 *Utilisation passive des données du LDS*

Le § 5.4.1 traite des extensions aux constructions de définitions de données du § 5.2. Dans le § 5.4.2 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 au § 5.5.

5.4.1 *Constructions de définitions de données étendues*

Les constructions définies au § 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. Au § 5.4.1 et à tous ses sous-paragraphes, la syntaxe abstraite applicable se trouve pratiquement au § 5.2.

Grammaire textuelle concrète

```
<propriétés étendues> ::=
        <règle d'héritage>
    |   <instanciation de générateur>
    |   <définition de structure>

<terme composite étendu> ::=
        <identificateur d'opérateur étendu> (<liste de termes composites>)
    |   <terme composite> <opérateur infixe> <terme>
    |   <terme> <opérateur infixe> <terme composite>
    |   <opérateur monadique> <terme composite>
    |   <terme composite conditionnel>

<terme clos étendu> ::=
        <identificateur d'opérateur étendu>
            (<terme clos> {, <terme clos> }*)
    |   <terme clos> <opérateur infixe> <terme clos>
    |   <opérateur monadique> <terme clos>
    |   <terme clos conditionnel>

<identificateur d'opérateur étendu> ::=
        <identificateur d'opérateur> <exclamation>
    |   <nom formel de générateur>
    |   [ <qualificatif> ] <opérateur entre quotes>

<nom d'opérateur étendu> ::=
        <nom d'opérateur> <exclamation>
    |   <nom formel de générateur>
    |   <opérateur entre quotes>

<exclamation> ::=
        !

<nom de littéral étendu> ::=
        <littéral de chaîne de caractères>
    |   <nom formel de générateur>
    |   <littéral de classe de nom>

<identificateur de littéral étendu> ::=
        <identificateur de littéral de chaîne de caractères>
    |   <nom formel de générateur>
```

Les règles <propriété étendue>, <terme composite étendu>, <terme clos étendu>, <nom d'opérateur étendu>, <nom de littéral étendu> <identificateur de littéral étendu> étendent les règles dans le noyau de données respectivement pour la <définition partielle de type> (§ 5.2.1), <terme composite> (§ 5.2.3), <terme clos> (§ 5.2.3), <nom d'opérateur> (§ 5.2.2), <littéral> (§ 5.2.2) et <identificateur de littéral> (§ 5.2.3). Les règles ci-dessus sont ensuite développées par les règles <règle d'héritage> (§ 5.4.1.11), <instanciation de générateur> (§ 5.4.1.12.2), <nom formel de générateur> (§ 5.4.1.12.1), <terme composite conditionnel> (§ 5.4.1.6), <terme clos conditionnel> (§ 5.4.1.6), <littéral de chaîne de caractères> et <identificateur de littéral de chaîne de caractères> (§ 5.4.1.2) et un <littéral de classe de nom> (§ 5.4.1.14). Les règles <opérateur infixe>, <opérateur monadique>, <opérateur infixe entre quotes> et <opérateur unaire entre quotes> sont définis au § 5.4.1.1.

Le choix du <nom formel de générateur> est uniquement valable dans une <expression de propriétés> dans un <texte de générateur> (voir le § 5.4.1.12) qui porte le même nom défini comme étant un paramètre formel.

Le choix de <terme composite étendu> et de <terme clos étendu> avec un <nom formel de générateur> précédent un «(» ne sont valables que si le <nom formel de générateur> est défini comme appartenant à la catégorie OPERATOR (voir le § 5.4.1.12).

Le choix du <nom de littéral étendu> avec un <nom formel de générateur> n'est valable que si le <nom formel de générateur> est défini comme appartenant à la catégorie LITERAL (voir le § 5.4.1.12).

Le choix d'<identificateur de littéral étendu> avec un <nom formel de générateur> n'est valable que si le <nom formel de générateur> est défini comme appartenant à la catégorie LITERAL ou à la catégorie CONSTANT (voir le § 5.4.1.12).

Lorsqu'un nom d'opérateur est défini avec une <exclamation>, cette <exclamation> fait sémantiquement partie du *nom.*

Les formes <nom d'opérateur> <exclamation> ou <identificateur d'opérateur> <exclamation> représentent respectivement le *nom-d'opérateur* (§ 5.2.2) et l'*identificateur-d'opérateur* (§ 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.

5.4.1.1 *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

```
<opérateur entre quotes> ::=
        <quote> <opérateur infixe> <quote>
    |   <quote> <opérateur monadique> <quote>

<quotes> ::=
        "

<opérateur infixe> ::=
        =>
    |   OR
    |   XOR
    |   AND
    |   IN
    |   /=
    |   =
    |   >
    |   <
    |   <=
    |   >=
    |   +
    |   /
    |   *
    |   //
    |   MOD
    |   REM
    |   -

<opérateur monadique> ::=
        -
    |   NOT
```

Sémantique

Un opérateur infixe dans un terme à 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 au § 5.4.2.1.

Lorsque le lien est ambigu 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 ;

A noter que les <opérateurs cités> MOD et REM n'ont pas de sémantique prédéfinie, car ils ne sont pas définis dans les sortes de données prédéfinies.

Modèle

Un terme de la forme

<terme 1> <opérateur infixe> <terme 2>

est une syntaxe dérivée pour

"<opérateur infixe>" (<terme 1>, <terme 2>)

avec "<opérateur infixe>" comme nom légal. "<opérateur infixe>" représente un *nom-d'opérateur.*

De manière analogue

<opérateur monadique> <terme>

est une syntaxe dérivée pour

"<opérateur monadique>" (<terme>)

avec "<opérateur monadique>" comme nom légal et représentant un *nom-d'opérateur.*

[Remarquons que l'opérateur d'égalité du LDS (=) ne doit pas être confondu avec le (symbole d'équivalence de terme du LDS (= =).].

5.4.1.2 *Littéral de chaîne de caractères*

Grammaire textuelle concrète

<identificateur de littéral de chaîne de caractères> ::=
[<qualificatif>] <littéral de chaîne de caractères>

<littéral de chaîne de caractères> ::=
<chaîne de caractères>

Une <chaîne de caractères> est une unité lexicale définie au § 2.2.1.

Un <identificateur de littéral de chaîne de caractères> représente un *identificateur-d'opérateur-de-littéral* dans la syntaxe abstraite.

Un <littéral de chaîne de caractères> représente un unique *nom-d'opérateur-de-littéral* (§ 5.2.2) dans la syntaxe abstraite obtenue à partir de la <chaîne de caractères>.

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 and character (voir le § 5.6). Ils ont également un lien particulier avec les littéraux de classe de nom (voir le § 5.4.1.14) et les mises en correspondance de littéraux (voir le § 5.4.1.15). Ces littéraux peuvent être également définis pour d'autres utilisations.

Un <littéral de chaîne de caractères> a une longueur qui est la longueur des <alphanumérique> plus les <autre caractère> plus les <spécial> plus <point> plus <souligné> plus <espace> plus les <apostrophe> <apostrophe> dans la <chaîne de caractères> (voir le § 2.2.1).

Un <littéral de chaîne de caractères> qui

a) a une longueur supérieure à un, et

b) a une sous-chaîne formée en supprimant le dernier caractère (<alphanumérique> ou <autre caractère> ou <spécial> ou <point> ou <souligné> ou <espace> ou <apostrophe> <apostrophe>) de la <chaîne de caractères>, et

c) cette sous-chaîne est définie comme un littéral tel que
sous-chaîne // caractère_supprimé_entre_quotes
est un terme valable avec la même sorte que le <littéral de chaîne de caractères>,

alors, il y a une équation implicite donnée par la syntaxe concrète telle que le <littéral de chaîne de caractères> est équivalent à la sous-chaîne suivie par l'opérateur infixe "//" suivi par le caractère supprimé avec les apostrophes pour former une <chaîne de caractères>.

Par exemple, les littéraux 'ABC', 'AB''' et 'AB' dans

```
NEWTYPE s
LITERALS 'ABC', 'AB''' 'AB', 'A', 'B', '''';
OPERATORS "//": s,s->s;
```

ont les équations implicites suivantes

```
'ABC'   == 'AB' // 'C';
'AB'''  == 'AB' // '''';
'AB'    == 'A' // 'B';
```

5.4.1.3 *Données prédéfinies*

Les données prédéfinies, y compris la sorte Boolean qui définissent les propriétés des deux littéraux True and False, sont définies au § 5.6. La sémantique pour l'égalité (§ 5.4.1.4), les axiomes booléens (§ 5.4.1.5), les termes conditionnels (§ 5.4.1.6), la relation ordre (§ 5.4.1.8), et les syntypes (§ 5.4.1.9) reposent sur la définition de la sorte booléenne (§ 5.6.1). La sémantique de littéraux de classe de nom (si des <intervalles réguliers> sont utilisés – § 5.4.1.14) et la mise en correspondance des littéraux (§ 5.4.1.15) reposent également respectivement sur la définition de character (§ 5.6.2) et charstring (§ 5.6.4).

Les données prédéfinies sont considérées comme étant définies au niveau du système.

5.4.1.4 *Egalité*

Grammaire textuelle concrète

Chaque nom de sorte introduit dans une <définition partielle de type> a une *signature-d'opérateur* implicite pour = et /=, et un ensemble d'*équation* pour ces opérateurs.

Une <définition partielle de type> introduisant une sorte S a une paire de *signature-d'opérateur* implicite équivalant à:

```
"=" : S, S -> Boolean;
"/=": S, S -> Boolean;
```

où Boolean est la sorte prédéfinie booléenne.

Une <définition partielle de type> introduisant une sorte de nom S a un ensemble d'*équation* implicite:

```
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 )
```

La dernière équation exprimant la propriété de substitution pour l'égalité.

S'il est possible de déduire à partir des équations (explicites, implicites et dérivées) que

```
True == False
```

cela est en contradiction avec les propriétés supposées du type de données booléen, la définition doit être non valable. Il ne doit pas être possible d'obtenir

```
True == False;
```

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.

Sémantique

Pour chaque sorte introduite par une définition partielle de type de données, il y a une définition implicite d'opérateurs et d'équations pour l'égalité.

Les symboles = et /= dans la syntaxe concrète représentent les noms des opérateurs qui sont appelés les opérateurs égal ou non égal.

5.4.1.5 *Axiomes booléens*

Grammaire textuelle concrète

```
<axiome booléen> ::=
        <terme booléen>
```

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 booléen>;

est une syntaxe dérivée pour l'équation en syntaxe concrète

<terme booléen> = = True;

qui a la relation normale d'une équation avec la syntaxe abstraite.

5.4.1.6 *Termes conditionnels*

Dans ce qui suit, l'équation contenant le terme conditionnel est appelée équation de terme conditionnel.

Grammaire abstraite

Terme-composite-conditionnel	=	*Terme-conditionnel*
Terme-clos-conditionnel	=	*Terme-conditionnel*
Terme-conditionnel	::	*Condition* *Conséquence* *Alternative*
Condition	=	*Terme*
Conséquence	=	*Terme*
Alternative	=	*Terme*

La sorte de la *condition* doit être la sorte booléenne prédéfinie et la condition ne doit pas être le terme d'erreur. La *conséquence* et l'*alternative* doivent avoir la même sorte.

Un *terme-conditionnel* est un *terme-composite-conditionnel* si et seulement si un ou plusieurs des *termes* dans la *condition*, la *conséquence* ou l'*alternative* est un *terme-composite.*

Un *terme-conditionnel* est un *terme-clos-conditionnel* si et seulement si tous les *termes* dans la *condition*, la *conséquence* ou l'*alternative* sont des *termes-clos.*

Grammaire textuelle concrète

<terme composite conditionnel> ::=
<terme conditionnel>

<terme clos conditionnel> ::=
<terme conditionnel>

<terme conditionnel> ::=
IF <condition> THEN <conséquence> ELSE <alternative> FI

<condition> ::=
<terme booléen>

<conséquence> ::=
<terme>

<alternative> ::=
<terme>

Sémantique

Un terme conditionnel utilisé dans une équation est sémantiquement équivalent à deux ensembles d'équations où tous les identificateurs de valeurs quantifiées dans le terme booléen ont été éliminés.

L'ensemble des équations peut être formé en substituant simultanément dans toute l'équation de terme conditionnel chaque *identificateur-de-valeur* dans la *condition* par chaque *terme-clos* de la sorte appropriée. Dans cet ensemble d'équations, la *condition* aura toujours été remplacée par un *terme-clos* booléen. Dans ce qui suit, on fera référence à cet ensemble d'équations comme étant l'ensemble clos développé.

Une équation de terme conditionnel est équivalente à l'ensemble d'*équations* qui contient:

a) pour chaque *équation* dans l'ensemble clos développé pour laquelle la *condition* est équivalente à True, cette *équation* obtenue à partir de l'ensemble clos développé avec le *terme-conditionnel,* remplacée par la *conséquence* (close) et

b) pour chaque *équation* dans l'ensemble clos développé pour laquelle la *condition* est équivalente à False, cette *équation* obtenue à partir de l'ensemble clos développé avec le *terme-conditionnel,* 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;

Remarque – Il y a de meilleures façons de spécifier ces propriétés, il ne s'agit que d'un exemple.

5.4.1.7 *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

Terme-d'erreur :: ( )

Un *terme-d'erreur* ne doit pas être utilisé comme *terme* argument pour un *identificateur-d'opérateur* dans un *terme-composite.*

Un *terme-d'erreur* ne doit pas être utilisé dans une *restriction.*

Il ne doit pas être possible de déduire à partir d'*équations* qu'un *identificateur-d'opérateur-littéral* est égal à un *terme-d'erreur.*

Grammaire textuelle concrète

<terme d'erreur> ::=
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.

5.4.1.8 *Relation d'ordre*

Grammaire textuelle concrète

<relation d'ordre> ::=
ORDERING

(<relation d'ordre> est référencé au § 5.2.2)

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 partielle de type> introduisant une sorte nommée S avec le mot clé ORDERING implique un ensemble de *signature-d'opérateur* équivalant aux définitions explicites:

```
"<"  : S,S -> Boolean;
">"  : S,S -> Boolean;
"<=" : S,S -> Boolean;
">=" : S,S -> Boolean;
```

où Boolean est la sorte prédéfinie booléenne, et implique les *axiomes* booléens:

```
FOR ALL a,b IN S
(
    "<"(a,a) = = False;
    "<"(a,b) = = ">"(b,a);
    "<="(a,b) = = "OR"("<"(a,b),"="(a,b));
    ">="(a,b) = = "OR"(">"(a,b),"="(a,b));
    "<"(a,b) = > NOT("<"(b,a));
    "<"(a,b) AND "<"(b,c) = > "<"(a,c);
);
```

Lorsqu'une <définition partielle de type> inclut à la fois <une liste de littéraux> et le mot clé ORDERING, les <signature de littéral> sont nommées selon l'ordre ascendant, c'est-à-dire

```
LITERALS A,B,C;
OPERATORS ORDERING;
```

implique A < B, B < C.

5.4.1.9 *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

Identificateur-de-syntype	=	*Identificateur*
Définition-de-syntype	::	*Nom-de-syntype* *Identificateur-de-sorte-parente* *Condition-d'intervalle*
Nom-de-syntype	=	*Nom*
Identificateur-de-la-sorte-parente	=	*Identificateur-de-sorte*

Grammaire textuelle concrète

<syntype> ::=
    <identificateur de syntype>

<définition de syntype> ::=
    SYNTYPE
        <nom de syntype> = <identificateur de la sorte parente>
        [<affectation par défaut>] [CONSTANTS <condition d'intervalle>]
        ENDSYNTYPE [<nom de syntype>]
  | NEWTYPE <nom de syntype> [<propriété étendue>]
        <expression de propriété> CONSTANTS <condition d'intervalle>
        ENDNEWTYPE [<nom de syntype>]

<identificateur de la sorte parente> ::=
    <sorte>

Un <syntype> est une alternative pour une <sorte> (voir le § 5.2.2).

Une <définition de syntype> avec le mot clé SYNTYPE et le terme « = <identificateur de syntype> » est la syntaxe dérivée définie ci-après.

Une <définition de syntype> avec le mot clé SYNTYPE dans la syntaxe concrète correspond à une *définition-de-syntype* dans la syntaxe abstraite.

Une <définition de syntype> avec le mot clé NEWTYPE peut être distinguée d'une <définition partielle de type> avec l'inclusion de CONSTANTS <condition d'intervalle>. Une telle <définition de syntype> est une abréviation pour introduire une <définition partielle de type> avec un nom anonyme suivi par une <définition de syntype> avec le mot clé SYNTYPE fondée sur cette sorte anonyme. C'est-à-dire que:

```
NEWTYPE X /* détails */
    CONSTANTS /* liste de constantes */
ENDNEWTYPE X;
```

est équivalent à

```
NEWTYPE anon /* détails */
ENDNEWTYPE anon;
```

suivi de

```
SYNTYPE X = anon
    CONSTANTS /* liste de constantes */
ENDSYNTYPE X;
```

Lorsqu'un <identificateur de syntype> est utilisé comme <argument> dans une <liste d'arguments> définissant l'opérateur, la sorte pour l'argument dans une *liste-d'arguments* est l'*identificateur-de-sorte-parente* du syntype.

Lorsqu'un <identificateur de syntype> est utilisé pour résultat pour un opérateur, la sorte du *résultat* est l'*identificateur-de-sorte-parente* du syntype.

Lorsqu'un <identificateur de syntype> est utilisé comme qualificatif pour un nom, le *qualificatif* est l'*identificateur-de-la-sorte-parente* du syntype.

Le <nom de syntype> optionnel donné à la fin d'une <définition de syntype> après le mot clé ENDSYNTYPE ou ENDNEWTYPE doit être le même que le nom de syntype> spécifié après SYNTYPE ou NEWTYPE respectivement.

Si le mot SYNTYPE est utilisé et que la <condition d'intervalle> est omise, toutes les valeurs de la sorte sont situées dans la condition d'intervalle de manière telle que l'<identificateur de syntype> ait exactement la même sémantique que l'identificateur de la sorte et que la condition d'intervalle soit toujours vraie.

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 le § 5.5.3),

b) sortie d'un signal si une des sortes spécifiées pour le signal est un syntype (voir le § 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 le § 2.4.5),

d) création d'un processus lorsque l'une des sortes spécifiées pour les paramètres de processus est un syntype (voir les § 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 le § 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 les § 5.4.2.2 et 5.5.2.4),

g) instruction d'initialisation ou de réinitialisation sur un temporisateur, une des sortes dans la définition de temporisateur étant un syntype (voir le § 2.8),

h) une définition d'import (voir le § 4.13).

Par exemple, une <définition de syntype> avec le mot clé SYNTYPE et "=<identificateur de syntype>" est équivalente à la substitution de l'<identificateur de sorte parente> par l'<identificateur de sorte parente> de la <définition de syntype> de l'<identificateur de syntype>. 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 de syntype>, les deux syntypes ne doivent pas être mutuellement définis.

Un syntype défini par une définition de syntype a une identité qui est le nom introduit par le nom de syntype qualifié par l'identité de l'unité de portée englobante.

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.

5.4.1.9.1 *Condition d'intervalle*

Grammaire abstraite

Condition-d'intervalle	::	*Identificateur-d'opérateur-ou* *Elément-de-condition*-**set**
Elément-de-condition	=	*Intervalle-ouvert* | *Intervalle-fermé*
Intervalle-ouvert	::	*Identificateur-d'opérateur* *Expression-close*
Intervalle-fermé	::	*Identificateur-d'opérateur-et* *Intervalle-ouvert* *Intervalle-ouvert*
Identificateur-d'opérateur-OU	=	*Identificateur*
Identificateur-d'opérateur-ET	=	*Identificateur*

Grammaire textuelle concrète

```
<condition d'intervalle> ::=
        { <intervalle fermé> | <intervalle ouvert> } { , { <intervalle fermé> | <intervalle ouvert> } }*
<intervalle fermé> ::=
        <constante> : <constante>
<intervalle ouvert> ::=
        <constante>
    |   { = | /= | < | > | <= | >= } <constante>
<constante> ::=
        <expression close>
```

Le symbole "<" ("<=", ">", ">=" respectivement) ne doit être utilisé que dans la syntaxe concrète de la <condition d'intervalle> si les symboles ont été définis avec une <signature d'opérateur>

P, P -> Boolean;

où P est la sorte du syntype. Ces symboles représentent un *identificateur-d'opérateur.*

Un <intervalle fermé> doit seulement être utilisé si le symbole "<=" est défini avec une <signature d'opérateur>

P, P Boolean;

où P est la sorte du syntype.

Une <constante> dans une <condition d'intervalle> 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 le § 5.4.1.9 et les cas où les syntypes ont des sémantiques différentes – voir les § 5.5.3, 2.6.4, 2.7.2, 2.5.4, 5.4.2.2 et 5.5.4). Une vérification d'intervalle est également utilisée pour déterminer l'interprétation d'une décision (voir le § 2.7.5).

La verification d'intervalle est l'application de l'opérateur formé à partir de la condition d'intervalle. 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 ouvert> ou <intervalle fermé>) dans la <condition d'intervalle> a un *intervalle-ouvert* ou un *intervalle-fermé* correspondant dans l'*élément-de-condition,*

b) Un <intervalle ouvert> de la forme <constante> est équivalent à un <intervalle ouvert> de la forme = <constante>.

c) Pour un terme donné, A,

   i) un <intervalle ouvert> de la forme = <constante>, /= <constante>, <<constante>, <= <constante>,> <constante>, et >= <constante>, a des sous-termes dans la vérification d'intervalle de la forme A = <constante>, A/=<constante>, A <<constante>, A <= <constante>, A > <constante>, et A >= <constante> respectivement.

   ii) L'<intervalle fermé> de la forme <première constante> : <seconde constante> a un sous-terme dans la vérification d'intervalle de la forme <première constante> <= A AND A <= <seconde constante> où AND correspond à l'opérateur AND booléen et correspond à l'*identificateur-d'opérateur-et* dans la syntaxe abstraite.

d) Il y a un *identificateur-d'opérateur-ou* pour l'opérateur distribué sur tous les éléments dans l'*élément-de-condition*-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) de tous les sous-termes obtenus à partir de la <condition d'intervalle>.

Si un syntype est spécifié sans une <condition d'intervalle>, la vérification d'intervalle est vraie.

5.4.1.10 *Sortes de structure*

Grammaire textuelle concrète

<définition de structure> ::=
STRUCT <liste de champs> [<fin>] [ADDING]

<liste de champs> ::=
<champ> { <fin> <champ> }*

<champ> ::=
<nom de champ> {, <nom de champ> }* <sorte de champ>

<sorte de champ> ::=
<sorte>

Chaque <nom de champ> d'une sorte structure doit être différent de chaque autre <nom de champ> de la même <définition de structure>.

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 d'arguments> pour l'opérateur Make! est la liste des <sorte de champ> apparaissant dans la liste des champs dans l'ordre avec lesquelles elles apparaissent.

Le <résultat> pour l'opérateur Make! est l'identificateur de sorte de la structure.

La <liste d'arguments> pour l'opérateur de modification de champ est l'identificateur de sorte de la structure suivie par la <sorte de champ> de ce champ. Le <résultat> pour un opérateur de modification de champ est l'identificateur de sorte de la structure.

La <liste d'arguments> pour un opérateur d'extraction de champ est l'identificateur de sorte de la structure. Le <résultat> pour un opérateur d'extration de champ est la <sorte de champ> 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.

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;
ENDNEWTYPE s;
```

5.4.1.11 *Héritage*

Grammaire textuelle concrète

<règle d'héritage> ::=
    INHERITS <sorte parente> [<renommage de littéral>]
        [[OPERATORS] { ALL | (<liste d'héritage>) } [<fin>]] [ADDING]

<sorte parente> ::=
    <sorte>

<liste d'héritage> ::=
    <operateur d'héritage> {, <opérateur d'héritage> }*

<opérateur d'héritage> ::=
    [<nom d'opérateur> =] <nom d'opérateur d'héritage>

<nom d'opérateur d'héritage> ::=
    <nom d'opérateur de la sorte parente>

<renommage de littéraux> ::=
LITERALS <liste de renommage de littéraux> <fin>

<liste de renommage de littéraux> ::=
<paire de renommage de littéral> {, <paire de renommage de littéral> }*

<paire de renommage de littéral> ::=
<signature de renommage de littéral> = <signature de renommage de littéral parent>

<signature de renommage de littéral> ::=
<nom d'opérateur littéral>
| <littéral chaîne de caractères>

Une sorte ne doit pas être basée sur une définition retournant sur elle-même par héritage.

Toutes les <signature de renommage de littéral> dans une <liste de renommage de littéraux> doivent être distinctes. Toutes les <signature de renommage de littéral parent> dans une <liste de renommage de littéraux> doivent être différentes.

Tous les <nom d'opérateur d'héritage> dans une <liste d'héritage> doivent être distincts. Tous les <nom d'opérateur> dans une <liste d'héritage> doivent être distincts.

Un <nom d'opérateur d'héritage> spécifié dans une <liste d'héritage> doit être un opérateur visible de la <sorte parente> défini dans la <définition partielle de type> définissant la <sorte parente>. Un nom d'opérateur n'est pas visible à ce point s'il est défini avec une <exclamation>.

Lorsque plusieurs opérateurs de la <sorte ascendante> ont le même nom, comme le <nom d'opérateur hérité>, alors tous ces opérateurs sont hérités.

Sémantique

Une sorte peut être fondée sur une autre sorte en utilisant NEWTYPE en association avec une règle d'héritage. La sorte définie au moyen d'une règle d'héritage est disjointe de la sorte parente.

Si la sorte parente a des littéraux définis, les noms de littéraux sont hérités comme des noms pour des littéraux de la sorte 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 parent apparaît comme deuxième nom dans une paire de renommage de littéral auquel cas, le littéral est renommé comme premier nom dans cette paire.

Il y a un opérateur d'héritage pour chaque opérateur de la sorte parente à l'exception de "=" et "/=". Un opérateur de la sorte parente est tout opérateur qui à la fois:

a) est défini au moyen d'une définition partielle de type ou d'une définition de syntype (à l'exception de celle qu'on est en train de définir) qui définit une sorte visible au point d'héritage, et également

b) a une sorte parente soit comme argument, soit comme résultat.

Les noms des opérateurs sont 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 de la sorte parente 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 la sorte parente, sinon

b) si l'identificateur d'opérateur parent 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 l'identificateur d'opérateur parent 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 de la sorte parente, sinon

d) si ALL n'est pas spécifié et que l'identificateur d'opérateur parent n'est pas mentionné dans la liste d'héritage, alors il est renommé en un nom invisible mais unique. De tels noms ne peuvent pas être explicitement utilisés dans les axiomes ou dans les expressions.

Los sortes d'argument et le résultat d'un opérateur d'héritage sont les mêmes que ceux des opérateurs correspondants de la sorte parente, sauf si la sorte d'argument ou de résultat est la sorte parente auquel cas elle est modifiée en la sorte en cours de définition. C'est-à-dire que chaque apparition de la sorte parente dans les opérateurs d'héritage est modifiée en la nouvelle sorte.

Pour chaque équation de la sorte parente, une équation est obtenue par héritage. Les équations de la sorte sont constituées par:

a) toute équation qui contient un opérateur (ou un littéral) de la sorte parente, et également

b) toute équation qui est définie par une quelconque définition partielle de type ou définition de syntype (à l'exception de celle qui est en train d'être définie) qui définit une sorte visible au point d'héritage.

Une équation d'héritage est la même que l'équation correspondante de la sorte parente avec les exceptions suivantes:

a) toute apparition de la sorte parente est changée en la nouvelle sorte, et

b) les opérateurs (ou les littéraux) de la sorte parente qui ont renommé les opérateurs (ou les littéraux) d'héritage, subissent le même renommage dans l'équation d'héritage.

Une conséquence de la modification des sortes telle qu'elle est indiquée en a) fait que les identités de littéraux et les identités d'opérateurs des littéraux hérités et des opérateurs hérités sont modifiées pour être qualifiées par l'identité de sorte de la nouvelle sorte.

Modèle

La syntaxe concrète d'une <règle d'héritage> est liée à la syntaxe concrète de l'<expression des propriétés> dans la <définition partielle de type> ou la <définition de syntype> contenant la <règle d'héritage>.

L'ensemble des <littéral> de la nouvelle sorte dans la syntaxe abstraite correspond à l'ensemble des <signature de littéral> dans l'<expression de propriétés> plus l'ensemble des littéraux hérités.

L'ensemble des <opérateur> de la nouvelle sorte dans la syntaxe abstraite correspond à l'ensemble des <signature d'opérateur> dans l'<expression de propriétés> plus l'ensemble des opérateurs hérités.

L'ensemble des <équation> de la nouvelle sorte dans la syntaxe abstraite correspond aux <axiome> de l'<expression des propriétés> plus l'ensemble des équations héritées.

Exemple

```
NEWTYPE     bit
    INHERITS  Boolean
        LITERALS      1 = True, 0 = False;
        OPERATORS ("NOT", "AND", "OR")
    ADDING
    OPERATORS
        EXOR: bit, bit -> bit;
    AXIOMS /* remarque – 2 différentes façons d'écrire NOT sont utilisées ici */
        EXOR (a,b) == (a AND "NOT"(b)) OR (NOT a AND b);
ENDNEWTYPE bit;
```

5.4.1.12 *Générateurs*

Un générateur permet de définir un squelette de texte paramétré qui est développé par instanciation avant que la sémantique des types de données ne soit considérée.

5.4.1.12.1 *Définition de générateur*

Grammaire textuelle concrète

<définition de générateur> ::=
GENERATOR <nom de générateur> (<liste de paramètres de générateur>) <texte de générateur>
ENDGENERATOR [<nom de générateur>]

<texte de générateur> ::=
[<instanciations de générateur>] <expression de propriétés>

<liste de paramètres de générateur> ::=
<paramètre de générateur> {, <paramètre de générateur> }*

<paramètre de générateur> ::=
{ TYPE | LITERAL | OPERATOR | CONSTANT }
<nom formel de générateur> {, <nom formel de générateur> }*

<nom formel de générateur> ::=
<nom formel de générateur>

<sorte de générateur> ::=
<nom formel de générateur>
| <nom de générateur>

Un <nom de générateur> ou <nom formel de générateur> doit uniquement être employé dans une <expression de propriétés> si l'<expression de propriétés> figure dans un <texte de générateur>.

Dans une <définition de générateur>, tous les <nom formel de générateur> 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 de générateur>. Le <nom de générateur> placé après le mot clé GENERATOR doit être distinct de tous les noms de sorte appartenant à la <définition de générateur> mais aussi de tous les <paramètre de générateur> TYPE de cette <définition de générateur>.

Une <sorte de générateur> n'est valable que si elle apparaît sous la forme d'une <sorte étendue> (voir le § 5.2.2) dans un <texte de générateur> et si le nom est soit le <nom de générateur> de cette <définition de générateur>, soit un <nom formel de générateur> décrit dans cette définition.

Si une <sorte de générateur> est un <nom formel de générateur>, il faut qu'il s'agisse d'un nom défini comme appartenant à la catégorie TYPE.

Le <nom de générateur> facultatif figurant après ENDGENERATOR doit être le même que le <nom de générateur> indiqué après GENERATOR.

Un <nom formel de générateur> ne doit pas être employé dans un <qualificatif>. Un <nom de générateur> ou un <nom formel de générateur> ne doit pas:

a) être qualifié, ou
b) être suivi d'une <exclamation>, ou
c) être employé dans une <affectation par défaut>.

Sémantique

Un générateur nomme une partie de texte qui peut être employée dans les instanciations de générateur.

On considère que les textes d'instanciations 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 instancié. 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;
```

Remarque – La définition formelle (annexe F.2) n'autorise pas l'emploi du <nom formel de générateur> dans les qualificatifs. La Recommandation a été corrigée à cet égard, après l'impression de l'annexe F.2. Cette annexe n'est donc pas valable sur ce sujet.

5.4.1.12.2 *Instanciation de générateur*

Grammaire textuelle concrète

```
<instanciation de générateur> ::=
        { <instanciation de générateur> [ <fin> ] [ADDING] }+

<instanciation de générateur> ::=
        <identificateur de générateur> ( <liste de générateurs réels> )

<liste de générateurs réels> ::=
        <générateur réel> {, <générateur réel> }*

<générateur réel> ::=
        <sorte étendue>
      | <signature de littéral>
      | <nom d'opérateur>
      | <terme clos>
```

Si la classe d'un <paramètre de générateur> est TYPE, le <générateur réel> correspondant doit alors être une <sorte étendue>.

Si la classe d'un <paramètre de générateur> est LITERAL, le <générateur réel> correspondant doit alors être une <signature de littéral>.

Une <signature de littéral> qui est un <littéral de classe de nom> peut être utilisée comme un <générateur réel> à la seule condition que le <nom formel de générateur> correspondant n'apparaisse pas dans les <axiomes>, ou dans la <mise en correspondance de littéral> de <l'expression de propriétés> dans le <texte de générateur>.

Si la classe d'un <paramètre de générateur> est OPERATOR, le <générateur réel> correspondant doit alors être un <nom d'opérateur>.

Si la classe d'un <paramètre de générateur> est CONSTANT, le <générateur réel> correspondant doit alors être un <terme clos>.

Si le <générateur réel> est un <nom formel de générateur>, la classe du <nom formel de générateur> doit alors être la même que la classe correspondant au <générateur réel>.

Sémantique

L'utilisation d'une instanciation de générateur dans des propriétés étendues ou dans un texte de générateur signifie l'instanciation du texte identifié par l'identificateur de générateur. Un texte instancié 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,

b) le nom du générateur est remplacé

   i) si l'instanciation 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 d'instanciation de générateur à l'intérieur d'un générateur, par le nom de ce générateur.

Le texte instancié applicable aux littéraux est le texte instancié à partir des littéraux figurant dans l'expression de propriétés du texte du générateur, le mot clé LITERALS étant omis.

Le texte instancié applicable aux opérateurs est le texte instancié 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 instancié applicable aux axiomes est le texte 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 instanciation de générateur dans la liste d'instanciations de générateur, les textes instanciés 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 instancié 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 instanciations de générateur seront placés dans l'ordre dans lequel ils ont été instanciés et avant tout autre littéral.

Le texte instancié 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 instancié 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 instanciation de générateur est déterminée après instanciation. La relation est déterminée à partir du texte instancié, au point d'instanciation.

Exemple

```
NEWTYPE boolbag bag(Boolean)
        ADDING
        OPERATORS
            yesvote : boolbag -> Boolean;
        AXIOMS
            yesvote(b) == count (True, b) > count(False, b);
ENDNEWTYPE boolbag;
```

5.4.1.13 *Synonymes*

Un synonyme donne un nom à une expression close qui représente une des valeurs d'une sorte.

Grammaire textuelle concrète

<définition de synonyme> ::=
SYNONYM <nom de synonyme> [<sorte>] = <expression close>
| <définition de synonyme externe>

La variante <définition de synonyme externe> est décrite au § 4.3.1.

Si la sorte de l'<expression close> ne peut être déterminée de manière unique, il faut alors spécifier une sorte dans la <définition de synonyme>.

La sorte identifiée par la <sorte> doit être l'une des sortes auxquelles l'<expression close> peut être liée.

L'<expression close> ne doit pas se référer directement ou indirectement (par l'intermédiaire d'un autre synonyme) au synonyme décrit dans la <définition de synonyme>.

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>.

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.

Modèle

Dans la syntaxe concrète, l'<expression close> désigne un *terme-clos* de la syntaxe abstraite, conformément à la définition donnée au § 5.4.2.2.

Si une <sorte> est spécifiée, le résultat de l'<expression close> est liée à cette *sorte*. L'<expression close> représente un *terme-clos* de la syntaxe abstraite qui comporte un *identificateur-d'opérateur* dont le nom et les sortes d'arguments sont les mêmes que ceux indiqués dans la syntaxe concrète et dont la sorte de résultat équivaut à la sorte spécifiée dans la syntaxe concrète.

5.4.1.14 *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

```
<littéral de classe de nom> ::=
        NAMECLASS <expression régulière>

<expression régulière> ::=
        <expression régulière partielle>
            { [OR] <expression régulière partielle> }*

<expression régulière partielle> ::=
        <élément régulier> [ <nom de littéral naturel> | + |*]

<élément régulier> ::=
        (<expression régulière>)
    |   <littéral chaîne de caractères>
    |   <intervalle régulier>

<intervalle régulier> ::=
        <littéral chaîne de caractères> : <littéral chaîne de caractères>
```

Les noms formés par le <littéral de classe de nom> doivent satisfaire aux conditions statiques normales applicables aux littéraux (voir le § 5.2.2) et, soit aux règles lexicales relatives aux noms (voir le § 2.2.1), soit à la syntaxe concrète relative au <littéral chaîne de caractères> (voir le § 5.4.1.2).

Dans un <intervalle régulier>, il faut que les <littéral chaîne de caractères> soient l'un et l'autre de longueur un et soient des littéraux définis par la sorte caractère (voir le § 5.6.2).

Sémantique

Un littéral de classe de nom est une autre façon de spécifier les signatures de littéraux.

Modèle

L'ensemble de noms dont l'équivalent est un littéral de classe de nom se définit par l'ensemble de noms conforme à la syntaxe spécifiée par l'<expression régulière>. Le littéral de classe de nom est l'équivalent de cet ensemble de noms dans la syntaxe abstraite.

Une <expression régulière> qui est une liste d'<expression régulière partielle> sans un OR spécifie que les noms peuvent être formés à partir des caractères définis par la première <expression régulière partielle> suivis des caractères définis par la seconde <expression régulière partielle>.

Lorsqu'un OR est spécifié entre deux <expression régulière partielle>, les noms sont alors formés à partir de la première ou de la seconde de ces <expression régulière partielle>. 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 régulier> est suivi d'un <nom de littéral naturel>, l'<expression régulière partielle> équivaut à l'<élément régulier> répété autant de fois que cela est spécifié par le <nom de littéral naturel>.

Par exemple,

NAMECLASS 'A' ('A' OR 'B') 2

définit les noms AAA, AAB, ABA et ABB.

Si un <élément régulier> est suivi d'un '*' l'<expression régulière partielle> équivaut à l'<élément régulier> 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 régulier> est suivi d'un '+' l'<expression régulière partielle> équivaut à l'<élément régulier> 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 régulier> qui est une <expression régulière> entre crochets définit les séquences de caractères décrits dans l'<expression régulière>.

Un <élément régulier> qui est un <littéral chaîne de caractères> définit la séquence de caractères indiquée dans le littéral chaîne de caractères (sans guillemets).

Un <élément régulier> qui est un <intervalle régulier> définit tous les caractères spécifiés par l'<intervalle régulier> comme séquences de caractères possibles. Les caractères définis par l'<intervalle régulier> 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 le § 5.6.2). Par exemple,

'a':'f'

définit les alternatives 'a' or 'b' or 'c' or 'd' or 'e' or '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 chaîne de caractères. Si les deux noms commencent par les mêmes caractères mais si leurs longueurs sont différentes, on considère que le nom le plus court est défini en premier.

5.4.1.15 *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

```
<mise en correspondance de littéral> ::=
        MAP <équation de littéral> { <fin> <équation de littéral> }* [<fin>]

<équation de littéral> ::=
        <quantification de littéral>
            ( <axiomes de littéral> { <fin> <axiomes de littéral> }* [<fin>] )

<axiomes de littéral> ::=
        <équation>
    |   <équation de littéral>

<quantification de littéral> ::=
        FOR ALL <nom de valeur> { , <nom de valeur> }* IN <sorte étendue> LITERALS

<terme orthographique> ::=
        SPELLING ( <identificateur de valeur> )
```

Les règles <mise en correspondance de littéral> et <terme orthographique> ne font pas partie du noyau de données mais apparaissent dans les règles <expression de propriétés> et <terme clos> aux § 5.2.1 et 5.2.3 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. Lorsque la sorte comprend un grand nombre (ou infini) de valeurs, la mise en correspondance de littéral offre le seul moyen pratique de définir la valeur correspondant à chaque littéral.

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 mise en correspondance de littéral.

Modèle

Une <mise en correspondance de littéral> est une notation abrégée d'un ensemble d'<axiomes>. Cet ensemble d'<axiomes> est obtenu à partir des <équation de littéral> dans la <mise en correspondance de littéral>. Les <équation> qui sont utilisées pour les obtenir sont toutes les <équation> contenues dans les <axiomes> des règles <axiomes de littéral>. Dans chacune de ces <équation>, on remplace les <identificateur de valeur> définis par le <nom de valeur> dans la <quantification de littéral>. Dans chaque <équation> obtenue, le même <identificateur de valeur> est remplacé, chaque fois qu'il apparaît, par le même <identificateur d'opérateur de littéral> de la <sorte> de la <quantification de littéral>. L'ensemble obtenu d'<axiomes> renferme toutes les <équations> possibles qui peuvent être obtenues de cette manière. Les <axiomes> obtenus pour les <équation de littéral> sont ajoutés aux <axiomes> (le cas échéant) définis après le mot clé AXIOMS et avant le mot clé MAP dans la même <définition partielle de type>.

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

```
NETWTYPE 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 de littéral> contient un ou plusieurs <terme orthographique>, il faut alors remplacer les <terme orthographique> par les littéraux charstring (voir le § 5.6.3).

Si la <signature de littéral> de l'<identificateur d'opérateur de littéral> d'un <terme orthographique> est un <nom d'opérateur de littéral> (voir le § 5.2.2), le <terme orthographique> est alors la notation abrégée d'une chaîne de caractères en lettres majuscules obtenue à partir de l'<identificateur d'opérateur de littéral>. La charstring contient l'orthographe en lettres majuscules du <nom d'opérateur de littéral> de l'<identificateur d'opérateur de littéral>.

Si la <signature de littéral> de l'<identificateur d'opérateur de littéral> d'un <terme orthographique> est un <littéral chaîne de caractères> (voir les § 5.2.2 et 5.4.1.2), le <terme orthographique> est alors la notation abrégée d'une charstring obtenue à partir du <littéral chaîne de caractères>. La charstring contient l'orthographe du <littéral chaîne de caractères>.

La charstring est utilisée pour remplacer l'<identificateur de valeur> une fois que l'<équation de littéral> contenant le <terme orthographique> 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 de

```
NETWTYPE abc LITERALS 'A',Bb,'c';
    OPERATORS
        "<":abc,abc -> Boolean;
    AXIOMS
        /* A noter que 'A', Bb, 'c' sont liés à la sorte locale abc*/
        /* '"A"', 'BB' et '"c"' doivent être précisés par l'identificateur de chaîne de caractères si ces
        littéraux sont ambigus – pour abréger, cette notation n'est pas utilisée ci-après*/
        '"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;
```

Chaque <équation non quantifiée> faisant partie des <axiomes de littéral> doit renfermer un <terme orthographique> ou un <identificateur d'opérateur de littéral>. Un <terme orthographique> doit faire partie d'une <mise en correspondance de littéral>.

L'<identificateur de valeur> faisant partie d'un <terme orthographique> doit être un <identificateur de valeur> défini par une <quantification de littéral>.

5.4.2 *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 ci-après.

5.4.2.1 *Expressions*

Les expressions sont des littéraux, des opérateurs, des accès de variables, des expressions conditionnelles et des opérateurs impératifs.

Grammaire abstraite

Expression = *Expression-close* | *Expression-active*

Une *expression* est une *expression active* si elle contient un *primaire-actif* (voir le § 5.5).

Une *expression* qui ne contient pas un *primaire-actif* est une *expression-close.*

Grammaire textuelle concrète

Pour simplifier la description, aucune distinction n'est établie entre la syntaxe concrète de l'*expression-close* et de l'*expression-active.* La syntaxe concrète de l'<expression> est indiquée au § 5.4.2.2 ci-après.

Sémantique

Une expression est interprétée en tant que valeur de l'expression close ou de l'expression active. Si la valeur est une errreur, le comportement ultérieur du système n'est pas défini.

L'expression a la sorte de l'expression close ou de l'expression active.

5.4.2.2 Expressions closes

Grammaire abstraite

Expression-close :: *Terme-clos*

Les conditions statiques applicables au *terme-clos* sont également valables pour l'*expression-close.*

Grammaire textuelle concrète

```
<expression close> ::=
        <expression close>

<expression> ::=
        <opérande0>
    |   <sous-expression> => <opérande0>

<sous-expression> ::=
        <expression>

<opérande0> ::=
        <opérande1>
    |   <sous-opérande0> { OR | XOR } <opérande1>

<sous-opérande0> ::=
        <opérande0>

<opérande1> ::=
        <opérande2>
    |   <sous-operande1> AND <opérande2>

<sous-opérande1> ::=
        <opérande1>

<opérande2> ::=
        <opérande3>
    |   <sous-opérande2> { = | /= | > | >= | < | <= | IN } <opérande3>

<sous-opérande2> ::=
        <opérande2>

<opérande3> ::=
        <opérande4>
    |   <sous-opérande3> { + | - | // } <opérande4>

<sous-opérande3> ::=
        <opérande3>

<opérande4> ::=
        <opérande5>
    |   <sous-opérande4> { * | / | MOD | REM } <opérande5>

<sous-opérande4> ::=
        <opérande4>

<opérande5> ::=
        [ - | NOT ] <primaire>

<primaire> ::=
        <primaire clos>
    |   <primaire actif>
    |   <primaire étendu>

<primaire clos> ::=
        <identificateur de littéral>
    |   <identificateur d'opérateur> ( <liste d'expressions closes> )
    |   ( <expression close> )
    |   <expression close conditionnelle>

<primaire étendu> ::=
        <synonyme>
    |   <primaire indexé>
    |   <primaire de champ>
    |   <primaire de structure>

<liste d'expressions closes> ::=
        <expression close> {, <expression close> }*

<identificateur d'opérateur> ::=
        <identificateur d' opérateur>
        |[<qualificatif>] <opérateur entre quotes>
```

Une <expression> qui ne contient aucun <primaire actif> représente une *expression-close* dans la syntaxe abstraite. Une <expression close> ne doit pas contenir un primaire <actif>.

Si une <expression> est un <primaire clos> avec un <identificateur d'opérateur> et si une <sorte d'argument> de la <signature d'opérateur> est un <syntype>, le contrôle de l'intervalle de ce syntype défini au § 5.4.1.9.1 est alors appliqué à la valeur d'argument correspondante. La valeur du contrôle de l'intervalle doit être vrai.

Si une <expression> est un <primaire clos> avec un <identificateur d'opérateur> et si la <sorte de résultat> de la <signature d'opérateur> est un <syntype>, le contrôle de l'intervalle de ce syntype défini au § 5.4.1.9.1 est alors appliqué à la valeur du résultat. La valeur du contrôle de l'intervalle doit être vrai.

Si une <expression> contient un <primaire étendu> (c'est-à-dire, un <synonyme>, un <primaire indexé>, un <primaire de champ> ou un <primaire de structure>), un remplacement est opéré au niveau de la syntaxe concrète selon la définition donnée au § 5.4.2.3, 5.4.2.4, 5.4.2.5 et 5.4.2.6, respectivement, avant que la relation avec la syntaxe abstraite ne soit considérée.

Le <qualificatif> facultatif placé avant un <opérateur entre quotes> possède la même relation avec la syntaxe abstraite qu'un <qualificatif> d'un <identificateur d' opérateur> (voir le § 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.

5.4.2.3 *Synonyme*

Grammaire textuelle concrète

<synonyme> ::=
    <identificateur de synonyme>
  | <synonyme externe>

La variante <synonyme externe> est décrite au § 4.3.1.

Sémantique

Un synonyme est une notation abrégée permettant de désigner une expression définie ailleurs.

Modèle

Un <synonyme> représente l'<expression close> décrite dans la <définition de synonyme> identifiée par l'<identificateur de synonyme>. Un <identificateur> utilisé dans l'<expression close> représente un *identificateur* en syntaxe abstraite, conformément au contexte de la <définition de synonyme>.

5.4.2.4 *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

<primaire indexé> ::=
    <primaire> (<liste d'expressions>)

Sémantique

Une expression indexée représente l'application d'un opérateur Extract!

Modèle

Un <primaire> suivi d'une <liste d'expressions> entre crochets est une syntaxe concrète représentant la syntaxe concrète

Extract!(<primaire>, <liste d'expressions>)

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 § 5.4.2.2.

5.4.2.5 *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 priamire comme paramètre.

Grammaire textuelle concrète

<primaire de champ> :: =
<primaire> <sélection de champ>

<sélection de champ> ::=
!<nom de champ>
| (<nom de champ> { , <nom de champ>}*)

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
<primaire> (<nom de champ>)
est la syntaxe dérivée de
<primaire> ! <nom de champ>

La forme
<primaire> (<nom de premier champ> { , <nom de champ>}*)
est la syntaxe dérivée de
<primaire> ! <nom de premier champ> { ! <nom de champ> }*
où l'ordre des noms de champ est préservé.

La forme
<primaire> ! <nom de champ>
est la syntaxe dérivée représentant
<nom d'opérateur d'extraction de champ> (<primaire>)

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 qu'il s'agit d'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 § 5.4.2.2.

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

5.4.2.6 *Primaire de structure*

Grammaire textuelle concrète

<primaire de structure> ::=
[<qualifier>] (. <liste d'expressions> .)

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
(. <liste d'expressions> .)

est la syntaxe concrète dérivée de

Make!(<liste d'expressions>)

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 § 5.4.2.2.

5.4.2.7 *Expression close conditionnelle*

Grammaire textuelle concrète

<expression close conditionnelle> ::=
IF <expression close booléenne>
THEN <expression close de conséquence>
ELSE <expression close d'alternative>
FI

<expression close de conséquence> ::=
<expression close>

<expression close d'alternative> ::=
<expression close>

L'<expression close conditionnelle> représente une *expression-close* dans la syntaxe abstraite. Si l'<expression close booléenne> représente la valeur vrai, l'*expression-close* est représentée par l'<expression close de conséquence>, autrement elle est représentée par l'<expression close d'alternative>.

La sorte de l'<expression close de fréquence> doit être la même que la sorte de l'<expression close d'alternative>.

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 close booléenne> a la valeur vrai, il s'ensuit que l'<expression close d'alternative> n'est pas interprétée. Si l'<expression close booléenne> a la valeur faux, il s'ensuit que l'<expression close de conséquence> n'est pas interprétée. Le comportement ultérieur du système n'est pas défini si l'<expression close> 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).

5.5 *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.

5.5.1 *Variables et définition de données*

Grammaire textuelle concrète

<définition de données> ::=
{ <définition partielle de type>
| <définition de syntype>
| <définition de générateur>
| <définition de synonyme> } <fin>

Une définition de données fait partie d'une *définition-de-type-de-données* s'il s'agit d'une <définition partielle de type> ou d'une <définition de syntype>, selon la définition donnée aux § 5.2.1 et 5.4.1.9, respectivement. Les règles <définition de générateur> et <définition de synonyme> sont définies aux § 5.4.1.12 et 5.4.1.13, respectivement.

La syntaxe qui régit l'introduction des variables de processus ainsi que les variables des paramètres de procédure est indiquée aux § 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 aux § 5.2.1, 5.4.1.9 ou 5.4.1.13.

Lorsqu'une variable est créée, elle contient une valeur spéciale qualifiée d'indéfinie qui se distingue de n'importe quelle autre valeur de la sorte de cette variable.

5.5.2 *Variables d'accès*

L'interprétation d'une expression comportant des variables est décrite ci-après.

5.5.2.1 *Expressions actives*

Grammaire abstraite

Expression active	=	*Accès-de-variable* |
		Expression-conditionnelle |
		Application-d'opérateur |
		Opérateur-impératif

Grammaire textuelle concrète

<expression active> ::=
<expression active>

<primaire actif> ::=
<accès de variable>
| <application d'opérateur>
| <expression conditionnelle>
| <opérateur impératif>
| (<expression active>)
| <primaire actif étendu>

<primaire étendu actif> ::=
<primaire actif étendu>

<liste d'expression> ::=
<expression> { , <expression> }*

Pour abréger, la syntaxe concrète de l'<expression active> correspond à l'<expression> au § 5.4.2.2. Une <expression> est une <expression active> si elle contient un <primaire actif>.

Pour abréger également, la syntaxe concrète du <primaire actif étendu> correspond à celle du <primaire étendu> au § 5.4.2.2. Un <primaire étendu> est un <primaire actif étendu> s'il contient un <primaire actif>. Dans le cas d'un <primaire étendu>, le remplacement au niveau de la syntaxe concrète intervient conformément à la définition donnée aux § 5.4.2.3, 5.4.2.4, 5.4.2.5 et 5.4.2.6, 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 sytème.

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

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 acif 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.

A l'intérieur d'une expression active, chaque opérateur est interprété dans l'ordre établi par la syntaxe concrète indiquée au § 5.4.2.2 ou, en cas d'ambiguïté, de gauche à droite. A l'intérieur d'une liste d'expressions actives ou d'une liste d'expressions, chaque élément de la liste est interprété de gauche à droite.

5.5.2.2 *Accès de variable*

Grammaire abstraite

Accès-de-variable	=	*Identificateur-de-variable*

Grammaire textuelle concrète

<accès de variable> ::=
<identificateur de variable>

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 une erreur s'il s'agissait de la valeur spéciale «non définie». Si la valeur d'un accès de variable est une erreur, il s'ensuit que le comportement futur du système n'est pas défini.

5.5.2.3 *Expression conditionnelle*

Une expression conditionnelle est une expression qui est interprétée comme conséquence ou comme alternative.

Grammaire abstraite

Expression-conditionnelle	::	*Expression-booléenne* *Expression-de-conséquence* *Expression-d'alternative*
Expression-booléenne	=	*Expression*
Expression-de-conséquence	=	*Expression*
Expression-d'alternative	=	*Expression*

La sorte de l'*expression-de-conséquence* doit être la même que celle de l'*expression-d'alternative.*

Grammaire textuelle concrète

```
<expression conditionnelle> ::=
        IF <expression booléenne active>
            THEN <expression de conséquence>
            ELSE <expression d'alternative>
            FI
    |   IF <expression booléenne>
            THEN <expression active de conséquence>
            ELSE <expression d'alternative>
            FI
    |   IF <expression booléenne>
            THEN <expression de conséquence>
            ELSE <expression active d'alternative>
            FI

<expression de conséquence> ::=
        <expression>

<expression d'alternative>  ::=
        <expression>
```

Une <expression conditionnelle> diffère d'une <expression close conditionnelle> par l'occurrence d'une <expression active> dans l'<expression conditionnelle>.

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.

5.5.2.4 *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

Application-d'opérateur	::	*Identificateur-d'opérateur* *Expression+*

Si une *sorte* d'argument de la *signature-d'opérateur* est un *syntype* et si l'*expression* correspondante de la liste d'*expression* est une *expression-close*, le contrôle de l'intervalle défini au § 5.4.1.9.1 qui est appliqué à la valeur de l'*expression* doit être vrai.

Grammaire textuelle concrète

```
<application d'opérateur> ::=
        <identificateur d'opérateur> ( <liste d'expressions actives> )

<liste d'expressions actives>  ::=
        <expression active> [ , <liste d'expressions> ]
    |   <expression close> , <liste d'expressions actives>
```

Une <application d'opérateur> diffère de l'<expression close> 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>. Si toutes les <expression> entourées de crochets sont des <expression close>, la construction représente une *expression-close* selon la définition du § 5.4.2.2.

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'au § 5.5.2.1.

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 au § 5.4.1.9.1 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 au § 5.4.1.9.1 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.

5.5.3 *Instruction d'affectation*

Grammaire abstraite

Instruction-d'affectation :: *Identificateur-de-variable*
Expression

La sorte de l'*identificateur-de-variable* doit être la même que celle de l'*expression.*

Si la *variable* est déclarée avec un *syntype* et si l'*expression* est une *expression-close,* il s'ensuit que le contrôle de l'intervalle tel que défini au § 5.4.1.9.1 et appliqué à l'*expression* doit être vrai.

Grammaire textuelle concrète

<instruction d'affectation> ::=
<variable> := <expression>

<variable> ::=
<identificateur de variable>
| <variable indexée>
| <variable de champ>

Si la <variable> est un <identificateur de variable> l'<expression> en syntaxe concrète représente alors l'<expression> en syntaxe abstraite. Les autres formes de <variable>, <variable indexée> et <variable de champ> correspondent à des syntaxes dérivées et l'<expression> en syntaxe abstraite est déduite de la syntaxe concrète équivalente définie aux § 5.5.3.1 et 5.5.3.2 ci-après.

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 au § 5.4.1.9.1 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.

5.5.3.1 *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

<variable indexée> ::=
<variable> (<liste d'expressions>)

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> (<liste d'expressions>) := <expression>

est la syntaxe concrète dérivée de

<variable> := Modify!(<variable>,<liste d'expressions>,<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 d'affectation> conformément au § 5.5.3. ci-dessus.

Le modèle relatif aux variables indexées doit être appliqué avant le modèle relatif à l'import (voir le § 4.13).

5.5.3.2 *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

<variable de champ> ::=

<variable> <selection de champ>

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 ! <nom de champ> sélection de champ conformément à la définition donnée au § 5.4.2.5.

La forme syntaxique concrète

<variable> ! <nom de champ> := <expression>

est la syntaxe concrète dérivée de

<variable> := <nom d'opérateur de modification de champ> (<variable>, <expression>)

où

a) la même <variable> est répétée, et

b) le <nom d'opérateur de modification de champ> 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 d'opérateur de modification de champ> n'est pas admis, dans la syntaxe concrète, comme nom d'opérateur pour les expressions.

S'il existe plus d'un <nom de champ> dans la sélection de champ, ils sont alors représentés de la façon indiquée ci-dessus, chaque ! <nom de champ>, étant développé tour à tour de droite à gauche et la partie restante de la <variable de champ> étant considéré comme une <variable>. Par exemple,

var ! fielda ! fieldb := expression;

est représenté tout d'abord par

var ! fielda := fieldbModify!(var ! fielda, expression);

puis par

var := fieldaModify!(var, fieldbModify!(var ! fielda, expression));

La syntaxe abstraite est déterminée pour l' <instruction d'affectation> formée à partir du modèle conformément au § 5.5.3 ci-dessus.

5.5.3.3 *Affectation par défaut*

Une affectation par défaut est une notation abrégée permettant d'affecter la même valeur à toutes les variables d'une sorte spécifiée immédiatement après leur création.

Grammaire textuelle concrète

<affectation par défaut> ::=

DEFAULT <expression close> [<fin>]

Une <définition partielle de type> ou une <définition de syntype> ne doit pas contenir plus d'une <affectation par défaut>. (On évite ainsi des affectations multiples provenant d'instanciations de générateur).

Sémantique

Une affectation par défaut est ajoutée, à titre facultatif, à une expression de propriétés d'une sorte. Une affectation par défaut spécifie que la valeur de l'expression close est immédiatement affectée à n'importe quelle variable déclarée avec la sorte introduite par la définition partielle de type ou la définition de syntype.

S'il n'existe pas d'affectation par défaut, lorsqu'une variable sera déclarée, elle sera donc associée à la valeur non définie.

Une autre valeur peut être affectée à une variable lorsqu'elle est déclarée, moyennant l'inclusion d'une affectation explicite dans la déclaration.

Les affectations par défaut ne sont pas héritées.

Modèle

La forme syntaxique concrète

DEFAULT <expression close>

utilisée dans une expression de propriétés où la sorte s'est introduite implique l'affectation de l'<expression close> à une variable. Cette affectation est interprétée immédiatement après la déclaration de la variable et avant l'interprétation de toute action explicitement spécifiée dans le même processus ou dans la même procédure.

Par exemple, si

DEFAULT 2*dnumber

est attribué à la sorte s et qu'il existe une déclaration dans la syntaxe concrète

DCL v s;

il y a alors une affectation implicite

v := 2*dnumber;

Si la déclaration a également une <valeur initiale>, cette <valeur initiale> est alors affectée à la variable après l'<expression close> dans l'<affectation par défaut>.

L'instruction d'affectation implicite a la relation normale d'une <instruction d'affectation> par rapport à la syntaxe abstraite (voir le § 5.5.3).

Si une <affectation par défaut> est spécifiée pour une <définition de données>, la <sorte> (représentant un syntype ou une sorte) a donc une valeur d'affectation par défaut qui est la valeur de l'<expression close> de l'<affectation par défaut>. Si aucune <affectation par défaut> n'est attribuée dans la <définition de syntype>, le syntype a alors une valeur d'affectation par défaut si l'*identificateur-de-sorte-parente* (identifiant un syntype ou une sorte) indiqué dans la *définition-de-syntype* a une valeur d'affectation par défaut.

Dans le cas d'une <définition de syntype>, les affectations ne sont interprétées qu'à la seule condition que le contrôle de l'intervalle défini au § 5.4.1.9.1 indique vrai lorsqu'il est appliqué à la valeur d'affectation par défaut. Autrement dit, pour chaque variable de syntype, il existe une décision implicite de la forme

DECISION <contrôle de l'intervalle>;
(vrai) : <affectation par défaut>
ELSE: ENDDECISION.

5.5.4 *Opérateurs impératifs*

Les opérateurs impératifs obtiennent des valeurs de l'état du système sous-jacent.

Grammaire abstraite

Opérateur-impératif = *Expression-maintenant* |
Expression-PId |
Expression-de-visibilité |
Expression-active-de-temporisateur

Grammaire textuelle concrète

<opérateur-impératif> ::=
<expression maintenant>
| <expression d'import>
| <expression PId>
| <expression de vue>
| <expression active de temporisateur>

La variante <expression d'import> est définie au § 4.13.

Les opérateurs impératifs sont des expresssions permettant de vérifier si les temporisateurs sont actifs ou d'accéder à l'horloge du système, aux valeurs PId associées à un processus ou à des variables importées.

5.5.4.1 *NOW*

Grammaire abstraite

Expression-maintenant	::	( )

Grammaire textuelle concrète

<expression maintenant> ::=
NOW

Sémantique

L'expression maintenant est une expression qui permet d'accéder à une variable d'horloge de système pour déterminer le temps absolu du sytème.

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.

Une expression maintenant a la sorte temps.

5.5.4.2 *Expression d'IMPORT*

Grammaire textuelle concrète

La syntaxe concrète d'une expression d'import est définie au § 4.3.

Sémantique

En plus de la sémantique définie au § 4.13, une expression d'import est interprétée comme un accès de variable (voir le § 5.5.2.2) 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 au § 4.13 et comporte également un *accès-de-variable* implicite de la variable implicite pour l'import dans le contexte où l'<expression d'import> apparaît.

5.5.4.3 *Expression PId*

Grammaire abstraite

Expression-PId	=	*Expression-identité* | *Expression-parente* | *Expression-de-descendant* | *Expression-d'émetteur*
Expression-identité	::	( )
Expression-parente	::	( )
Expression-de-descendant	::	( )
Expression-d'émetteur	::	( )

Grammaire textuelle concrète

<expression PId> ::=
SELF
| PARENT
| OFFSPRING
| SENDER

Sémantique

Une expression PId permet d'accéder à une des variables implicites de processus définies au § 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 au § 2.4.4.

5.5.4.4 *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

Expression-de-vue :: *Identificateur-de-variable*
Expression

L'*expression* doit être une expression PId.

L'*identificateur-de-variable* doit être l'un des identificateurs d'une des variables du processus identifié par l'*expression*.

Grammaire textuelle concrète

<expression de vue> ::=
VIEW (<identificateur de variable>, <expression PId>)

L'<identificateur de variable> doit être défini comme une variable visualisée dans une <définition de vue> du processus contenant l'<expression de vue>. On peut omettre le <qualificatif> dans l'<identificateur de variable> uniquement si aucune autre variable ayant la même partie de <nom> ne figure dans une <définition de processus> englobante.

Sémantique

Une expression de vue est interprétée de la même façon qu'un accès de variable (voir le § 5.5.2.2). La variable dont l'accès est obtenu est la variable faisant partie du processus identifié par l'expression PId (voir le § 5.5.4.3).

Une expression de vue a une valeur et une sorte qui sont la valeur et la sorte de l'accès de variable.

L'expression PId doit identifier un processus existant du même bloc comme étant le processus dans lequel l'expression de vue est interprétée, faute de quoi l'expression de vue est dans l'erreur et le comportement futur du système n'est pas défini. L'expression PId doit identifier le même type de processus que l'*identificateur-de-processus* dans la définition de vue correspondante.

5.5.4.5 *Expression active de temporisateur*

Grammaire abstraite

Expression-active-de-temporisateur :: *Identificateur-de-temporisateur*
*Expression**

Les sortes de l'*expression** contenues dans l'*expression-active-de-temporisateur* doivent correspondre par leur position à l'*identificateur-de-référence-de-sorte** suivant directement le *nom-de-temporisateur* (§ 2.8) identifié par l'*identificateur-de-temporisateur*.

Grammaire textuelle concrète

<expression active de temporisateur> ::=
ACTIVE (<identificateur de temporisateur> [(<liste d'expressions>)])

La <liste d'expressions> est définie au § 5.5.2.1.

Sémantique

Une expression active de temporisateur est une expression de la sorte booléenne 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 le § 2.8.2). Dans le cas contraire, l'expression active de temporisateur a la valeur faux. Les expressions sont interprétées dans l'ordre indiqué.

5.6 *Données prédéfinies*

Les sortes de données et les générateurs de données qui sont définis implicitement au niveau du système sont décrits dans le présent paragraphe.

Remarque – Le § 5.4.1.1 définit la syntaxe et l'ordre des opérateurs spéciaux (infixe et monadique), mais la sémantique de ces opérateurs (sauf REM et MOD) est définie par les définitions de données dans le présent paragraphe.

5.6.1 *Sorte booléenne*

5.6.1.1 *Définition*

```
NEWTYPE     Boolean

    LITERALS  True, False;
    OPERATORS

        "NOT"     : Boolean                 -> Boolean;
                                                    /*
        "="       : Boolean, Boolean        -> Boolean;    Les opérateurs "=" et "/=" sont
                                                           implicites.
        "/="      : Boolean, Boolean        -> Boolean;    Voir le § 5.4.1.4
                                                    */
        "AND"     : Boolean, Boolean        -> Boolean;
        "OR"      : Boolean, Boolean        -> Boolean;
        "XOR"     : Boolean, Boolean        -> Boolean;
        "=>"      : Boolean, Boolean        -> Boolean;
    AXIOMS
        "NOT" (True)              == False;
        "NOT" (False)             == True;

        "=" (True, True)          == True;
        "=" (True, False)         == False;
        "=" (False, True)         == False;
        "=" (False, False)        == True;

        "/=" (True, True)         == False;
        "/=" (True, False)        == True;
        "/=" (False, True)        == True;
        "/=" (False, False)       == False;

        "AND" (True, True)        == True;
        "AND" (True, False)       == False;
        "AND" (False, True)       == False;
        "AND" (False, False)      == False;

        "OR" (True, True)         == True;
        "OR" (True, False)        == True;
        "OR" (False, True)        == True;
        "OR" (False, False)       == False;

        "XOR" (True, True)        == False;
        "XOR" (True, False)       == True;
        "XOR" (False, True)       == True;
        "XOR" (False, False)      == False;

        "=>" (True, True)         == True;
        "=>" (True, False)        == False;
        "=>" (False, True)        == True;
        "=>" (False, False)       == True;

ENDNEWTYPE Boolean;
```

5.6.1.2 *Utilisation*

La sorte booléenne est utilisée pour représenter les valeurs vrai et faux. Elle est souvent utilisée comme résultat d'une comparaison.

La sorte Boolean est utilisée par de nombreuses formes abrégées de données en LDS telles que les axiomes sans le symbole "= =" et les opérateurs implicites d'égalité "=" et "/=".

5.6.2 *Sorte de caractère*

5.6.2.1 *Définition*

```
NEWTYPE Character

    LITERALS
        NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
        BS,  HT,  LF,  VT,  FF,  CR,  SO,  SI,
        DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB,
        CAN, EM,  SUB, ESC, IS4, IS3, IS2, IS1,
        ' ', '!', '"', '#', '¤', '%', '&', '''',
        '(', ')', '*', '+', ',', '-', '.', '/',
        '0', '1', '2', '3', '4', '5', '6', '7',
        '8', '9', ':', ';', '<', '=', '>', '?',
        '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
        'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
        'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
        'X', 'Y', 'Z', '[', '\', ']', '^', '_',
        '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
        'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
        'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
        'x', 'y', 'z', '{', '|', '}', '~', DEL;
/* ''' est une apostrophe, ' 'est un espace, '~' est une ligne de séparation supérieure ou un tilde */


OPERATORS
                                                    /*
    "="  : Character, Character -> Boolean;          Les signatures d'opérateur "=" et "/=" sont
    "/=" : Character, Character -> Boolean;          implicites – voir le § 5.4.1.4
                                                    */
    "<"  : Character, Character -> Boolean;
    "<=": Character, Character -> Boolean;
    ">"  : Character, Character -> Boolean;
    ">=": Character, Character -> Boolean;
AXIOMS
/*Les notations suivantes spécifient "inférieur à" entre littéraux de caractères adjacents*/
    NUL < SOH == True;      SOH < STX == True;
    STX < ETX == True;      ETX < EOT == True;
    EOT < ENQ == True;      ENQ < ACK == True;
    ACK < BEL == True;      BEL < BS  == True;
    BS  < HT  == True;      HT  < LF  == True;
    LF  < VT  == True;      VT  < FF  == True;
    FF  < CR  == True;      CR  < SO  == True;
    SO  < SI  == True;      SI  < DLE == True;
    DLE < DC1 == True;      DC1 < DC2 == True;
    DC2 < DC3 == True;      DC3 < DC4 == True;
    DC4 < NAK == True;      NAK < SYN == True;
```

```
SYN   < ETB    == True;     ETB   < CAN== True;
CAN   < EM     == True;     EM    < SUB== True;
SUB   < ESC    == True;     ESC   < IS4 == True;
IS4   < IS3    == True;     IS3   < IS2 == True;
IS2   < IS1    == True;     IS1   < ' ' == True;
' '   < '!'    == True;     '!'   < '"' == True;
'"'   < '#'    == True;     '#'   < '¤' == True;
'¤'   < '%'    == True;     '%'   < '&' == True;
'&'   < '''    == True;     ''''  < '(' == True;
'('   < ')'    == True;     ')'   < '*' == True;
'*'   < '+'    == True;     '+'   < ',' == True;
','   < '-'    == True;     '-'   < '.' == True;
'.'   < '/'    == True;     '/'   < '0' == True;
'0'   < '1'    == True;     '1'   < '2' == True;
'2'   < '3'    == True;     '3'   < '4' == True;
'4'   < '5'    == True;     '5'   < '6' == True;
'6'   < '7'    == True;     '7'   < '8' == True;
'8'   < '9'    == True;     '9'   < ':' == True;
':'   < ';'    == True;     ';'   < '<' == True;
'<'   < '='    == True;     '='   < '>' == True;
'>'   < '?'    == True;     '?'   < '@' == True;
'@'   < 'A'    == True;     'A'   < 'B' == True;
'B'   < 'C'    == True;     'C'   < 'D' == True;
'D'   < 'E'    == True;     'E'   < 'F' == True;
'F'   < 'G'    == True;     'G'   < 'H' == True;
'H'   < 'I'    == True;     'I'   < 'J' == True;
'J'   < 'K'    == True;     'K'   < 'L' == True;
'L'   < 'M'    == True;     'M'   < 'N' == True;
'N'   < 'O'    == True;     'O'   < 'P' == True;
'P'   < 'Q'    == True;     'Q'   < 'R' == True;
'R'   < 'S'    == True;     'S'   < 'T' == True;
'T'   < 'U'    == True;     'U'   < 'V' == True;
'V'   < 'W'    == True;     'W'   < 'X' == True;
'X'   < 'Y'    == True;     'Y'   < 'Z' == True;
'Z'   < '['    == True;     '['   < '\' == True;
'\'   < ']'    == True;     ']'   < '^' == True;
'^'   < '_'    == True;     '_'   < '`' == True;
'`'   < 'a'    == True;     'a'   < 'b' == True;
'b'   < 'c'    == True;     'c'   < 'd' == True;
'd'   < 'e'    == True;     'e'   < 'f' == True;
'f'   < 'g'    == True;     'g'   < 'h' == True;
'h'   < 'i'    == True;     'i'   < 'j' == True;
'j'   < 'k'    == True;     'k'   < 'l' == True;
'l'   < 'm'    == True;     'm'   < 'n' == True;
'n'   < 'o'    == True;     'o'   < 'p' == True;
'p'   < 'q'    == True;     'q'   < 'r' == True;
'r'   < 's'    == True;     's'   < 't' == True;
't'   < 'u'    == True;     'u'   < 'v' == True;
'v'   < 'w'    == True;     'w'   < 'x' == True;
'x'   < 'y'    == True;     'y'   < 'z' == True;
'z'   < '{'    == True;     '{'   < '|' == True;
'|'   < '}'    == True;     '}'   < '~' == True;
'~'   < DEL    == True;     .

FOR ALL a, b, c IN Character (
    a < a                                  == False
    a < b AND b < c => a < c               == True
    a < b                                  == b > a;
    a < b OR a > b                         ==a/= b;
    a < b => NOT(b < a);
    NOT ( a/= b )                          ==a = b;
    a < b OR a = b                         ==a <= b;
    a > b OR a = b                         ==a >= b;)

ENDNEWTYPE Character;
```

5.6.2.2 *Utilisation*

La sorte de caractère définit des chaînes de caractères de longueur 1, où les caractères sont ceux de l'Alphabet international n° 5. Ils sont définis comme chaînes ou comme abréviations selon la version de référence internationale de l'alphabet. La représentation imprimée peut varier selon l'utilisation nationale qui est faite de l'alphabet.

Il existe 128 littéraux et valeurs différentes définis pour les caractères. Le classement des valeurs est spécifié ainsi que les rapports d'égalité et d'inégalité.

5.6.3 *Générateur de chaîne*

5.6.3.1 *Définition*

```
GENERATOR String(TYPE Itemsort, LITERAL Emptystring) /*Les chaînes sont «indexées» à partir de un*/
    LITERALS Emptystring;
    OPERATORS
        MkString:Itemsort       -> String;                      /* former une chaîne à partir d'un élément*/
        Length   :String        -> Integer;                     /* longueur de la chaîne*/
        First    :String        -> Itemsort;                    /* premier élément de la chaîne*/
        Last     :String        -> Itemsort;                    /* dernier élément de la chaîne*/
        "//"     String, String      -> String;                 /* concaténation*/
        Extract! :String, Integer    -> Itemsort;               /* extraire un élément de la chaîne*/
        Modify! :String, Integer, Itemsort          -> String;  /* modifier la valeur de la chaîne*/
        SubString:String, Integer,Integer           -> String;  /* supprimer la sous-chaîne de la chaîne*/
            /*la sous-chaîne (s, i, j) donne une chaîne de longueur j à partir du ième élément*/
    AXIOMS
        FOR ALL item, itemi, itemj, item 1, item 2 IN Itemsort (
        FOR ALL s, s1, s2, s3 IN String (
        FOR ALL i, j IN Integer (
        type String Length (Emptystring)                        == 0;
        type String Length (MkString(item))                     == 1;
        type String Extract! (MKString(item),1)                 == item;
        First(s)                                                == Extract!(s,1);
        Last(s)                                                 == Extract!(s, Length(s));
        Length(s1 // s2)                                        == Length (s1) + Length (s2);
        Length(Modify!(s,i,item))                               == Length(s));
        (s1 // s2) // s3                                        == s1 // (s2 // s3);
        Emptystring // s                                        == s;
        s // Emptystring                                        == s;
        Emptystring = (MkString (item) // s2)                   == False;
        (MkString (item1) // s1) = (MkString (item2) // s2) == (item1 = item2) AND (s1 = s2);

        i > 0 AND i <=Length(s) == True ==>
                        Extract!(Modify!(s,i,item),i) == item;
        i /= j AND i > 0 AND i <= Length(s) AND j > 0 AND j <= Length(s) == True ==>
                        Extract!(Modify!(s,i,item),j) == Extract!(s,j);
        i <= 0 OR i > Length(s) == True ==> Extract!(s,i) == ERROR!;

        i/= j == True ==>
            Modify!(Modify!(s,i,itemi),j,itemj) == Modify!(Modify!(s,j,itemj),i,itemi);
        Modify!(Modify!(s,i,item1),i,item2) == Modify!(s,i,item2);
        i <= 0 OR i > Length(s) == True ==> Modify!(s,i,item) == ERROR!;

        i <= Length(s1) == True ==>
                        Extract!(s1 // s2, i)       == Extract!(s1,i);
        i > Length(s1) == True ==>
                        Extract!(s1 // s2, i)       == Extract!(s2,i - Length(s1));

        i > 0 AND i <= Length(s) == True ==> SubString(s,i,0)== Emptystring;
        i > 0 AND i <= Length(s) == True ==> SubString(s,i,1)==MkString(Extract!(s,i));
        i > 0 AND i <= Length(s) AND i-1+j <= Length(s) AND j > 1 == True ==>
                        SubString(s,i,j) == SubString(s,i,1) // SubString(s,i+1,j-1);
        i < 0 OR i > Length(s) OR j <= 0 OR i+j > Length(s) == True ==>
                        SubString(s,i,j) == ERROR!;


        i > 0 AND i <= Length(s) == True ==>
                Modify!(s,i,item)    ==
                        Substring(s,1,i-1) // MkString(item) // Substring(s,i+1,Length(s)-i );)));
ENDGENERATOR String;
```

5.6.3.2 *Utilisation*

Un générateur de chaîne peut être utilisé pour définir une sorte qui permet la construction de chaînes de n'importe quelle sorte d'éléments. L'utilisation la plus courante sera celle de la chaîne de caractères définie ci-après.

Les opérateurs Extract! et Modify! seront normalement utilisés avec les notations abrégées définies aux § 5.4.2.4 et 5.5.3.1 pour avoir accès aux valeurs des chaînes et affecter des valeurs aux chaînes.

5.6.4 *Sorte chaîne de caractères*

5.6.4.1 *Définition*

```
NEWTYPE Charstring (Character,'')
    ADDING LITERALS NAMECLASS "" ( ( ' ':'&') OR """" OR ( '(': '~' ) )+ "";
    /* les chaînes de caractères d'une longueur quelconque composées de caractères quelconques compris entre
    un espace ' ' et une ligne de séparation supérieure '~' */
    /* les équations de la forme
        'ABC' == 'AB' // 'C';
        sont implicites – voir le § 5.4.1.2 */
    MAP     FOR ALL c IN Character LITERALS (
                FOR ALL charstr IN Charstring LITERALS (
                        Spelling( charstr) == Spelling(c) ==> charstr == Mkstring(c);
            )   ); /* la chaîne 'A' est formée à partir du caractère 'A' etc. */
ENDNEWTYPE Charstring;
```

5.6.4.2 *Utilisation*

La sorte charstring définit des chaînes de caractères. Un littéral de charstring peut contenir des caractères d'imprimerie et des espaces. Un caractère non imprimable peut être utilisé comme chaîne en utilisant Mkstring, par exemple, Mkstring(DEL).

```
/*Exemple*/ SYNONYM newline_prompt charstring = Mkstring(CR) // Mkstring(LF) // '$>';
```

5.6.5 *Sorte entier*

5.6.5.1 *Définition*

```
NEWTYPE Integer
    LITERALS NAMECLASS ('0':'9')* ('0':'9') ;
        /*séquence de nombres facultatifs avant l'un des nombres compris entre 0 et 9 */
    OPERATORS
        "-"   : Integer              -> Integer;

        "+"  : Integer,Integer       -> Integer;
        "-"   : Integer,Integer      -> Integer;
        "*"   : Integer,Integer      -> Integer;
        "/"   : Integer,Integer      -> Integer;
                                        /*
        "="  : Integer,Integer       -> Boolean;      Les signatures d'opérateur "=" et "/=" sont
        "/=": Integer,Integer        -> Boolean;      implicites voir le § 5.4.1.4
                                        /*
        "<"  : Integer,Integer       -> Boolean;
        ">"  : Integer,Integer       -> Boolean;
        "<=": Integer,Integer        -> Boolean;
        ">=": Integer,Integer        -> Boolean;

        Float: Integer               -> Real;    /* axiomes de la définition NEWTYPE Réel */
        Fix   : Real                 ->Integer;  /* axiomes de la définition NEWTYPE Réel */
```

```
AXIOMS
    FOR ALL a, b, c IN Integer (
    /*négation*/
    0 - a                          == -a;
    /* addition:*/
    0 + a                          == a;
    a + b                          == b + a;
    a + (b + c)                    == (a + b) + c;
    /*soustraction*/
    a - a                          == 0;
    (a - b) - c                    == a - (b + c);
    (a - b) + c                    == (a + c) - b;
    a - (b - c)                    == (a + c) - b;
    /*multiplication*/
    a * 0                          == 0;
    a * 1                          == a;
    a * b                          == b * a;
    a * (b * c)                    == (a * b) *c;
    a * (b + c)                    == a * b + a * c;
    a * (b - c)                    == a * b - a * c;
    /*relation d'ordre*/
    a + 1 > a                      == True;
    a - 1 < a                      == True;
    /* égalité*/
    (a > b) OR (b > a) == NOT (a = b);
    /* axiomes normaux de relation d'ordre*/
    "<" (a,a)                      == False;
    "<" (a,b)                      == ">" (b,a);
    "<="(a,b)                      == "OR"("<"(a,b),"="(a,b));
    ">="(a,b)                      == "OR"(">"(a,b),"="(a,b));
    "<"(a,b) == True                    ==>    NOT("<"(b,a)) == True;
    "<"(a,b) AND "<"(b,c)== True   ==>    "<"(a,c)      == True;
    /*division*/
    a/0                            ==ERROR!;
    a>=0 AND b > a == True                                  ==> a/b == 0;
    a>=0 AND b <=a AND b > 0 == True                        ==>a/b == 1 + (a-b)/b;
    a>=0 AND b < 0 == True                                  ==> a/b == -(a/(-b));
    a< 0 AND b < 0 == True                                  ==> a/b == (-a)/(-b);
    a< 0 AND b > 0 == True                                  ==> a/b ==-( (-a)/b );
    /* Littéraux 2 à 9*/
    TYPE Integer 2 == 1 + 1; TYPE Integer 3 == 2 + 1;
    TYPE Integer 4 == 3 + 1; TYPE Integer 5 == 4 + 1;
    TYPE Integer 6 == 5 + 1; TYPE Integer 7 == 6 + 1;
    TYPE Integer 8 == 7 + 1; TYPE Integer 9 == 8 + 1;
MAP /* Littéraux autres que 0 à 9*/
    FOR ALL a,b,c IN Integer LITERALS
    ( Spelling(a) == Spelling(b) // Spelling(c), Length(Spelling(c)) == 1 ==>
            a  == b * (9 + 1) + c;


    );
ENDNEWTYPE Integer;
```

5.6.5.2 *Utilisation*

La sorte entier est utilisée pour les nombres entiers mathématiques avec notation décimale.

5.6.6 *Syntype naturel*

5.6.6.1 *Définition*

```
SYNTYPE Natural = Integer CONSTANTS >= 0 ENDSYNTYPE Natural;
```

5.6.6.2 *Utilisation*

Le syntype naturel est utilisé lorsqu'il faut recourir uniquement à des nombres entiers positifs. Tous les opérateurs seront des opérateurs de la sorte entier mais lorsqu'une valeur est utilisée comme paramètre ou qu'elle est affectée, la valeur est vérifiée. Une valeur négative sera une erreur.

5.6.7 *Sorte réel*

5.6.7.1 *Définition*

```
NEWTYPE Real
    LITERALS NAMECLASS ( ('0':'9')* ('0':'9') ) OR ( ('0':'9')* '.' ('0':'9')+ );
    OPERATORS
        "-"     : Real              -> Real;

        "+"     : Real, Real        -> Real;
        "-"     : Real, Real        -> Real;
        "*"     : Real, Real        -> Real;
        "/"     : Real, Real        -> Real;
                                        /*
        "="     : Real, Real        -> Boolean;     Les signatures d'opérateur "=" et "/="
        "/="    : Real, Real        -> Boolean;     sont implicites -- voir le § 5.4.1.4
                                        */
        "<"     : Real, Real        -> Boolean;
        ">"     : Real, Real        -> Boolean;
        "<="    : Real, Real        -> Boolean;
        ">="    : Real, Real        -> Boolean;
    AXIOMS
        FOR ALL a, b, c IN Real (
        /*négation*/
        0 - a                       ==  -a;
        /* addition*/
        0 + a                       ==  a;
        a + b                       ==  b + a;
        a + (b + c)                 ==  (a + b) + c;
        /*soustraction*/
        a - a                       ==  0;
        (a - b) -c                  ==  a - (b + c);
        (a - b) + c                 ==  (a + c) - b;
        a - (b - c)                 ==  (a + c) - b;
        /*multiplication*/
        a * 0                       ==  0;
        a * 1                       ==  a;
        a * b                       ==  b * a;
        a * (b * c)                 ==  (a * b) *c;
        a * (b + c)                 ==  a * b + a * c;
        a * (b - c)                 ==  a * b - a * c);
        /*relation d'ordre*/
        FOR ALL i, j, IN Integer (
        Float(i) > Float(j)         ==TYPE Integer ">"(i,j);
        Float(j) = 0 == False       ==>Float(i) / Float(j) > 0 == Float(i) > 0 AND Float(j) > 0
                                            OR Float(i) < 0 AND Float(j) < 0;
        Float(i) > 0 AND Float(j) > 0 AND Float(i) > Float(j)
                                    ==>Float(i) / Float(j) > 1 == True;);
        FOR ALL a,r,b IN Real (a + r < b + r == a < b;
        r > 0 ==> a * r < b * r == a < b;
        r < 0 ==> a * r < b * r == b < a;);
```

```
        /* axiomes normaux de relation d'ordre*/
        FOR ALL a, b, c, d IN Real
        /* égalité et relation d'ordre*/
        (a > b) OR (b > a) = NOT (a = b);
        "<"(a,a)                        = =False;
        "<"(a,b)                        = = ">" (b,a);
        "< ="(a,b)                      = = "OR" ("<"(a,b),"="(a,b));
        "> ="(a,b)                      = = "OR" (">"(a,b),"="(a,b));
        "<"(a,b) = = True                 = = >   NOT("<"(b,a))= = True;
        "<"(a,b) AND "<"(b,c) = =True= = >   "<"(a,c)       = = True;
        /*division*/
        a/0                             = =ERROR!;
        a = 0 = = False = = > a / a= = 1;
        a = 0 = = False = = > 0 / a= = 0;
        b = 0 = = False = = > (a / b) * b = = a;
        b = 0 OR c = 0 = = False = = > (a * b)/(c * b) = = a/c;
        b = 0 OR d = 0 = = False = = > a/b + c/d = = (a * d + b * c)/(b * d);
        b = 0 OR d = 0 = = False = = > a/b - c/d = = (a * d - b * c)/(b * d);
        b = 0 OR d = 0 = = False = = > (a/b) * (c/d) = = (a * c)/(b * d);
        b = 0 OR d = 0 = = False = = > (a/b) / (c/d) = = (a * d)/(b * c););
        /*conversions entre entier et réel*/
        FOR ALL a, i, j IN Integer (
        FOR ALL r IN Real (
        Fix(Float(a))                   = = a;
        r - 1.0 < Float(Fix(r))         = = True; /* Note Fix(1.5) = = 1, Fix(-0.5)= = -1 */
        Float(Fix(r)) < = r             = = True;
        Float(TYPE integer "+"(i,j)) = =        Float(i) + Float(j);));
    MAP
        FOR ALL r,s IN Real LITERALS (
            FOR ALL i,j IN Integer LITERALS (
                Spelling(r) = = Spelling(i) = = > r = = Float(i);
                Spelling(r) = = Spelling(i) = = > i = = Fix(r);
                Spelling(r) = = Spelling(i) //Spelling(s), Spelling(s)= ='.'// Spelling(j)
                                                = = > r = = Float(i) + s;
                Spelling(r) = = '.' //Spelling(i), Length(Spelling(i)) = = 1,
                                                = = > r = = Float(i) / 10;
                Spelling(r) = = '.' //Spelling(i) //Spelling(j), Length(Spelling(i)) = = 1,
                        Spelling(s) = = '.' //Spelling(j)
                                                = = > r = = (Float(i) + s) / 10;
        )    );
ENDNEWTYPE Real;
```

5.6.7.2 *Utilisation*

La sorte réel est utilisée pour représenter des nombres réels. Elle peut donc s'appliquer à tous les nombres qui peuvent être représentés sous la forme d'un entier divisé par un autre. Les nombres qui ne peuvent être représentés de cette manière (les nombres irrationnels – par exemple $\sqrt{2}$) ne font pas partie de la sorte réel. Toutefois, dans la pratique, on peut généralement recourir à une approximation suffisamment précise. Il faut obligatoirement faire appel à des techniques supplémentaires pour définir un ensemble de nombres qui englobent tous les irrationels.

5.6.8 *Générateur de tableau*

5.6.8.1 *Définition*

```
GENERATORS Array (TYPE Index, TYPE Itemsort)
    OPERATORS
        Make!   : Itemsort                  -> Array;
        Modify! : Array,Index,Itemsort      -> Array;
        Extract! : Array,Index              -> Itemsort;
    AXIOMS
        FOR ALL item, item1, item2, itemi, itemj IN Itemsort (
        FOR ALL i, j, ipos IN Index (
        FOR ALL a, s IN Array (
        type Array Extract!(Make!(item,i)) = = item;
        Modify!(Modify!(s,i,item1),i,item2) = = Modify!(s,i,item2);
        Extract!(Modify!(a,ipos,item),ipos) = = item;
        i = j = = False             = = > Extract!(Modify!(a,j,item),i) = = Extract!(a,i);
        i = j = = False             = = >
            Modify!(Modify!(s,i,itemi),j,itemj) = = Modify!(Modify!(s,j,itemj),i,itemi);));

        /* égalité*/
        type Array Make! (item1) = Make! (item2) = = item1 = item2;
        Modify! (a, i, item) = s                = = (Extract! (s, i) = item AND (a = s);
ENDGENERATOR Array;
```

5.6.8.2 *Utilisation*

Le générateur de tableau peut être utilisé pour définir une sorte qui est indexée par une autre. Par exemple,

```
NEWTYPE indexbychar Array(Character,Integer)
ENDNEWTYPE indexbychar;
```

définit un tableau contenant des entiers et indexé par des caractères.

Les tableaux sont généralement utilisés en combinaison avec les formes abrégées de Modify! et Extract! définies aux § 5.5.3.1 et § 5.4.2.4 pour l'indexation. Par exemple

```
DLC charvalue indexbychar;
.....
TASK charvalue('A') := charvalue('B')-1;
```

5.6.9 *Générateur de mode ensembliste*

5.6.9.1 *Définition*

```
GENERATOR Powerset (TYPE Itemsort)
    LITERALS Empty;
    OPERATORS
        "IN"    : Itemsort,   Powerset -> Boolean;   /* l'opérateur est membre de                    */
        Incl    : Itemsort,   Powerset -> Powerset;  /* inclure un élément dans l'ensemble           */
        Del     : Itemsort,   Powerset -> Powerset;  /* supprimer un élément de l'ensemble           */
        "<"     : Powerset,   Powerset -> Boolean;   /* l'opérateur est un sous-ensemble strict de   */
        ">"     : Powerset,   Powerset -> Boolean;   /* l'opérateur est un super ensemble strict de  */
        "<="    : Powerset,   Powerset -> Boolean;   /* l'opérateur est un sous-ensemble de          */
        ">="    : Powerset,   Powerset -> Boolean;   /* l'opérateur est un sur ensemble de           */
        "AND"   : Powerset,   Powerset -> Powerset;  /* intersection d'ensembles                     */
        "OR"    : Powerset,   Powerset -> Powerset;  /* union d'ensembles                            */
```

```
AXIOMS
    FOR ALL i, j IN Itemsort (
    FOR ALL p, ps, a, b, c IN Powerset (
    i IN type Powerset Empty                    == False;
    i IN Incl(i,ps)                             == True;
    i IN ps                                     == i IN Incl(j,ps);
    type Powerset Del(i,Empty)                  == Empty;
    NOT(i IN ps)                                == Del(i,ps) = ps;
    DEL(i,Incl(i,ps))                           == ps;
    i = j == False ==> Del(i,Incl(j,ps))        == Incl(j,Del(i,ps));
    Incl(i,Incl(j,p))                           == Incl(j,Incl(i,p));
    Incl(i,Incl(i,p))                           == Incl(i,p);
    a < b => (i IN a => i IN b)                 == True;
    i IN ( a AND b )                            == TYPE Boolean "AND"( i IN a, i IN b);
    i IN ( a OR b )                             == TYPE Boolean "OR"( i IN a, i IN b);

    /* égalité*/
    Empty = Incl (i, ps) == False;
    Incl (i, a) = b        ==(i IN b) AND (a=Del (i,b));

    /* Axiomes de relation d'ordre normale*/
    "<"(a,a) ==     False;
    "<"(a,b) ==     ">"(b,a);
    "<="(a,b) ==    "OR"("<"(a,b),"="(a,b));
    ">="(a,b) ==    "OR"(">"(a,b),"="(a,b));
    "<"(a,b) == True ==> NOT("<"(b,a)) == True;
    TYPE Boolean "AND"("<"(a,b),"<"(b,c)) == True ==> "<"(a,c) == True;))

ENDGENERATOR Powerset;
```

5.6.9.2 *Utilisation*

Les modes ensemblistes sont utilisés pour représenter des ensembles mathématiques. Par exemple,

NEWTYPE Boolset Powerset(Boolean) ENDNEWTYPE Boolset; peut être utilisé pour une variable qui peut être vide ou contenir (True), (False), ou (True), (False).

5.6.10 *Sorte PId*

5.6.10.1 *Définition*

```
NEWTYPE PId
    LITERALS Null;
    OPERATORS       unique!     : PId -> PId;

                    /*
                    "="         : Pid,Pid -> Boolean;     Les signatures d'opérateur "=" et "/="
                    "/="        : Pid,Pid -> Boolean;     sont implicites – voir le § 5.4.1.4
                    */
    AXIOMS
        FOR ALL p, p1, p2 IN PId (
        unique! (p) = Null                          == False;
        unique! (p1) = unique! (p2)                 == p1 = p2);

    DEFAULT Null;
    ENDNEWTYPE PId;
```

5.6.10.2 *Utilisation*

La sorte PId est utilisée pour les identités de processus. A noter qu'il n'existe pas d'autres littéraux que la valeur null. Lorsqu'un processus est créé, le système sous-jacent utilise l'opérateur unique! pour générer une nouvelle valeur unique.

5.6.11 *Sorte durée*

5.6.11.1 *Définition*

```
NEWTYPE Duration INHERITS Real ( "+","-",">")
ADDING
      OPERATORS
          "*"  : Duration, Real -> Duration;
          "/"  : Duration, Real -> Duration;
      AXIOMS /* dans ce qui suit chaque d doit être une valeur de durée déterminée à partir du contexte */
      FOR ALL d,z IN Duration (
      FOR ALL r IN Real (
          /* égalité*/
          (d > z) OR (z > d) == NOT (d = z);
          /* Durée multipliée par Réel*/
          d * 0     == 0;
          0 * r     == 0;
          d * TYPE Real "+"(1,r)      == d + (d * r);
          d * TYPE Real "-"(1,r)      == d - (d * r);
          d * TYPE Real "-"(r,1)      == (d * r) - d;
          d * TYPE Real "-"(r)        == 0 - (d * r);
          /* Durée divisée par Réel*/
          d / 0 == ERROR!;
          r = 0 == False ==> d / r == d * TYPE Real "/" (1,r);
          /* c'est-à-dire que la division est la même qu'en multipliant par la valeur inverse (réelle)*/
          r = 0 == False ==> z * r = d          == ( d / r) = z; ));
      MAP
          FOR ALL d IN Duration LITERALS (
          FOR ALL r IN Real LITERALS ( Spelling(d) == Spelling(r) ==> d == 1 * r ));
ENDNEWTYPE Duration;
```

5.6.11.2 *Utilisation*

La sorte durée est utilisée pour la valeur à ajouter au temps actuel pour initialiser les temporisateurs. Les littéraux de la sorte durée sont les mêmes que les littéraux de la sorte réel. La signification d'une unité de durée dépendra du système défini.

Les durées peuvent être multipliées et divisées par des réels.

5.6.12 *Sorte temps*

5.6.12.1 *Définition*

```
NEWTYPE Time INHERITS Real OPERATORS ("<","<=",">",">=") ADDING
      OPERATORS
          "+" : Time, Duration        -> Time;
          "-" : Time, Duration        -> Time;
          "-" : Time, Time            -> Duration;
      AXIOMS
          FOR ALL t, t1, t2 IN Time (
          FOR ALL d,d1,d2, IN Duration (
          (t1 > t2) OR (t2 > t1) == NOT (t1 = t2);
          t + 0                  == t;
          t - d                  == t + TYPE Duration "-"( 0 , d );
          (t + d1) + d2          == t + TYPE Duration "+"(d1,d2);
          (t + d1) - (t + d2 )   == TYPE Duration "-"(d1,d2);));
      MAP
          FOR ALL d IN Duration LITERALS (
          FOR ALL t IN Time LITERALS ( Spelling(d) == Spelling(t) ==> t == 0 + d ));
ENDNEWTYPE Time;
```

5.6.12.2 *Utilisation*

L'expression NOW renvoie une valeur de la sorte temps. Une valeur de temps peut avoir une durée qui lui est ajoutée ou soustraite pour donner un autre temps. Une valeur de temps qui est soustraite d'une autre valeur de temps donne une durée. Les valeurs de temps sont utilisées pour fixer le temps d'expiration des temporisateurs.

L'origine du temps dépend du système. Une unité de temps est la quantité de temps représentée en ajoutant une unité de durée à un temps.

APPENDICE I

(à la Recommandation Z.100)

Modèle formel applicable aux types de données non paramétrées[1)]

I.1 *Algèbres multisorte*

Une **algèbre multisorte** A est un multiplet <D,O> où

a) D est un ensemble d'ensembles et les éléments de D sont désignés comme **supports de données** (de A); les éléments d'un support de données **dc** sont désignées comme **valeurs-de-données**; et

b) O est un ensemble de fonctions totales, où le domaine de chaque fonction est un produit cartésien des supports de données de A et des codomaines de l'un des supports de données.

I.2 *Sémantiques des définitions de type de données*

I.2.1 *Principes généraux*

I.2.1.1 *Signature*

Une **signature** SIG est un ensemble <S,OP> où

a) S est un ensemble d'**identificateurs-de-sorte** (également désignés par sortes); et

b) OP est un ensemble d'**opérateurs.**

Un opérateur comprend un **identificateur-d'opération** op, une liste de sortes (argument) w avec des éléments dans S et une sorte (résultat) s S. La forme de représention généralement utilisée est op:w→s. Si w est égal à la liste vide, op:w→s est appelé opérateur **null-aire** ou **symbole constant** de la sorte s.

I.2.1.2 *Morphisme de signature*

Supposons que $SIG_1 = <S_1,OP_1>$ et $<SIG_2 = <S_2,OP_2>$ soient des signatures. Un **morphisme de signature** $g:SIG_1 \rightarrow SIG_2$ est une paire de mises en correspondance

$$g = <gs:S_1 \rightarrow S_2, gop:OP_1 \rightarrow OP_2>$$

de sorte que pour tous les $e\text{-}opid_1 = <opidf_1, <gs/e\text{-}sidf_1), ..., gs(e\text{-}sidf_k)>, gs(e\text{-}res), pos> \in OP_1$

$gop(e\text{-}opid_1) = <opidf_2, <(e\text{-}sidf_1), ..., (e\text{-}sidf_k)>, (e\text{-}res), pos>$

pour des identificateurs-d'opération $opidf_2$.

---

1) Le CCITT et l'ISO se sont mis d'accord sur le texte du présent appendice qui constitue une description formelle commune du modèle d'algèbre initiale applicable aux types de données abstraites. Ce texte qui figure dans la présente Recommandation (avec les modification appropriées quant à la typographie et à la numérotation) est également reproduit dans le document ISO IS8807. Les § I.1, I.2.1.1, I.2.1.2, I.2.1.3, I.2.1.4, I.2.1.5, I.2.1.6, I.3, I.4.1, I.4.2, I.4.3, I.4.4, I.4.5, et I.4.6 du présent appendice figurent aux § 5.2, 7.2.2.1, 7.3.2.8, 7.2.2.2, 7.2.2.3, 7.2.2.4, 7.2.2.5, 4.7, 7.4.2.1, 7.4.2.2, 7.4.3, 7.4.3 et 7.4.4, respectivement du document ISO IS8807. Les termes **identificateur de sorte, opérateur, identificateur de variable, variable, spécification algébrique SPEC** et **opérations** du présent appendice sont respectivement remplacés dans l'IS8807 par les termes **variable de sorte, variable d'opération, variable de valeur, variable de valeur, présentation de données pres** et **fonctions.**

I.2.1.3 *Termes*

Supposons que V soit un ensemble quelconque de variables, et $<S,OP>$ une signature. Les ensembles TERM(OP,V,s) de **termes** de sorte $s \in S$ avec des opérateurs dans OP et des variables dans V, sont définis inductivement par les étapes suivantes:

a) chaque variable $x:s \in V$ est dans TERM(OP,V,s);

b) chaque opérateur null-aire $op \in OP$ avec $res(op)=s$ est dans TERM(OP,V,s);

c) si les termes t_i de la sorte s_i figurent dans TERM(OP,V,s_i) pour $i=1,...,n$ puis pour chaque $op \in OP$ avec $arg(op)=<s_1,...,s_n>$ et $res(op)=s$, $op(t_1,...,t_n)$ dans TERM(OP,V,s).

Si le terme t est un élément de TERM(OP,V,s), s est appelé la sorte de t, et s'écrit sorte (t).

L'ensemble TERM(OP,s) de termes clos de $s \in S$ est défini comme l'ensemble TERM(OP,{},s).

I.2.1.4 *Equations*

Une **équation** de sorte s par rapport à une signature $<S,OP>$ est un triplet $<V,L,R>$ où

a) V est un ensemble d'identificateurs-de-variable;

b) $L,R \in T(OP,V,s)$; et

c) $s \in S$.

Une équation $e' = <\{\},L',R'>$ est une **instance close** d'une équation $e= <V,L,R>$ si L',R' peuvent être obtenus à partir de L,R pour chaque variable v:s dans V, toutes les occurrences de cette variable dans L,R étant remplacées par le même terme clos avec la sorte s.

La notation $L=R$ est utilisée pour l'instance close $<\{\},L,R>$ d'une équation.

Remarque – De même, une équation $<V,L,R>$ peut s'écrire sous la forme $L=R$ si aucune complication d'ordre sémantique n'est introduite.

I.2.1.5 *Equations conditionnelles*

Une **équation conditionnelle** de sorte s par rappport à la signature $<S,OP>$ est un triplet $<V,Eq,e>$, où

a) V est un ensemble d'identificateurs de variable; et

b) Eq est un ensemble d'équations avec des variables dans V; et

c) e est une équation de sorte s avec des variables dans V.

I.2.1.6 *Spécifications algébriques*

Une **spécification algébrique** SPEC est un triplet $<S,OP,E>$ où

a) $<S,OP>$ est une signature; et

b) E est un ensemble d'équations conditionnelles par rapport à $<S,OP>$.

I.3 *Systèmes de dérivation*

A **système de dérivation** est un multiplet $D=<A,Ax,I>$ avec:

a) A, ensemble dont les éléments sont appelés **affirmations,**

b) $A \supseteq Ax$, l'ensemble d'**axiomes,**

c) I, ensemble de **règles de déduction.**

Chaque règle de déduction $R \in I$ s'écrit sous la forme suivante

$$R: \frac{P_1,...,P_n}{Q}$$

où $P_1,...,P_n,Q \in A$.

Une **dérivation** d'une affirmation P dans un système de dérivation D est une séquence finie s d'affirmations satisfaisant aux conditions:

a) le dernier élément de s est P,

b) si Q est un élément de s, soit $Q \in Ax$, soit on peut appliquer une règle $R \in I$

$$R: \frac{P_1,...,P_n}{Q}$$

avec $P_1,...,P_n$ éléments de s précédant Q.

S'il existe une dérivation de P dans un système de dérivation D, on pose $D \mid\!- P$. Si D est uniquement déterminé par le contexte, on peut utiliser l'abréviation $\mid\!- P$.

I.4 *Sémantique des spécifications algébriques*

Toutes les occurrences d'un ensemble de sortes S, d'un ensemble d'opérations OP et d'un ensemble d'équations E au § I.4 se réfèrent à une spécification algébrique donnée SPEC = <S,OP,E>, conformément à la définition du § I.2.1.6.

Afin de définir la sémantique d'une spécification algébrique SPEC, on utilise un système de dérivation associé à SPEC qui est défini au § I.4.1-I.4.3. A l'aide de ce système de dérivation, on définit aux § I.4.4 et I.4.5 des classes de congruence. Cette relation est utilisée au § I.4.6 pour définir une algèbre (voir le § I.1) qui représente le type de données spécifié par <S,OP,E>.

I.4.1 *Axiomes engendrés par des équations*

Soit ceq une équation conditionnelle. L'ensemble d'axiomes engendrés par ceq, notation Ax(ceq), est défini comme suit:

a) si ceq = <V,Eq,e> avec Eq ≠ {}, il s'ensuit que Ax(ceq) = {}; et

b) si ceq = <V,{},e>, Ax(ceq) est alors l'ensemble de toutes les instances closes de e (voir le § I.2.1.3).

I.4.2 *Règles de déduction engendrées par des équations*

Soit ceq une équation conditionnelle. L'ensemble des règles de déduction engendrées par ceq, notation Inf(ceq), est défini comme suit:

a) si ceq = <V,{},e>, il s'ensuit que Inf(ceq) = {}, et

b) si ceq = <V,{e_1,...,e_n},e> avec n > 0, Inf(ceq) contient alors toutes les règles de la forme

$$\frac{e_1',\ldots,e_n'}{e'}$$

où

e_1',...,e_n',e' sont des instances closes de e_1,...,e_n,e respectivement, que l'on obtient pour chaque variable x apparaisant dans V, en remplaçant toutes les occurrences de cette variable dans e_1,...,e_n,e, par le même terme clos avec la **sorte (x)**.

I.4.3 *Système de dérivation engendré par une spécification algébrique*

Le **système de dérivation** D = <A,Ax,I> (voir § I.3) engendré par une spécification algébrique SPEC = <S,OP,E> est défini comme suit:

a) A est l'ensemble de toutes les instances closes des équations par rapport à <S,OP>; et

b) Ax = ∪{Ax(ceq) | ceq ∈ E} ∪ ID,
avec ID = {t = t | t est un terme clos}; et

c) $I = \cup\{Inf(ceq) \mid ceq \in E\} \cup SI$,
où SI est donné par le schéma suivant

i) $\dfrac{t_1 = t_2}{t_2 = t_1}$ pour tous les termes clos t_1, t_2; et

ii) $\dfrac{t_1 = t_2,\ t_2 = t_3}{t_1 = t_3}$ pour tous les termes clos t_1, t_2, t_3; et

iii) $\dfrac{t_1 = t_1',\ldots,\ t_n = t_n'}{op(t_1,\ldots,t_n) = op(t_1',\ldots,t_n')}$

pour tous les opérateurs $op: s_1,\ldots,s_n \rightarrow s \in OP$ avec $n > 0$ et tous les termes clos de t_i, t_i' de la sorte s_i pour $i = 1,\ldots,n$.

I.4.4 *Relation de congruence engendrée par une spéficification algébrique*

Soit D le système de dérivation engendré par une spécification algébrique $SPEC = <S,OP,E>$. Deux termes clos t_1 et t_2 sont appelés **congruents** par rapport à SPEC, notation $t_1 \equiv_{SPEC} t_2$, si et seulement si

$$D \mid\!- t_1 = t_2$$

I.4.5 *Classes de congruence*

La **classe de congruence SPEC** [t] d'un terme clos t est l'ensemble de tous les termes congruents à t par rapport à SPEC, c'est à dire:

$$[t] = \{ t' \mid t \equiv_{SPEC} t' \}$$

I.4.6 *Algèbre des termes quotient*

L'interprétation sémantique d'une spécification algébrique $SPEC = <S,OP,E>$ est l'algèbre multisortes suivante $Q = <Dq,O_q>$, appelée **algèbre des termes quotient**, où

a) Dq est l'ensemble $\{ Q(s) \mid s \in S \}$ où
$Q(s) = \{[t] \mid t$ est le terme clos de la sorte $s\}$ pour chaque $s \in S$; et

b) Oq est l'ensemble d'opérations $\{ op' \mid op \in OP \}$ où les op' sont définis par
$op'([t_1],\ldots,[t_n]) = [op(t_1,\ldots,t_n)]$.

SÉRIES DES RECOMMANDATIONS UIT-T

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

Imprimé en Suisse

Genève, 2010