

D.5.4 *Directives applicables à la représentation en fonction des états et aux éléments graphiques*

Le présent paragraphe décrit la représentation en fonction des états du LDS/GR et des éléments graphiques utilisés.

Le § D.5.4.1 contient des observations d'ordre général sur la représentation en fonction des états: caractéristiques, applications pertinentes et variantes.

Le § D.5.4.2 explique la description d'états au moyen d'éléments graphiques.

D.5.4.1 *Observations d'ordre général sur la représentation en fonction des états*

Le LDS/GR offre trois versions différentes pour écrire un diagramme de processus.

La première est appelée version du LDS/GR en fonction des transitions, dans laquelle les transitions sont écrites exclusivement par des symboles d'action explicites.

La seconde est appelée version du LDS/GR orientée vers les états ou extension graphique du LDS orientée vers les états; les états d'un processus y sont écrits à l'aide d'illustrations d'états et les transitions ne sont données qu'implicitement, par les différences entre les états de départ et d'arrivée.

La dernière est appelée version mixte; il s'agit d'une combinaison des deux versions précédentes.

On trouvera des exemples de ces trois versions dans l'annexe E du présent fascicule.

La version orientée vers les transitions convient lorsque la séquence des actions présente plus d'importance que la description détaillée des états.

La version orientée vers les états décrit en détail les états par des énoncés; elle convient donc au cas où un état de processus présente plus d'importance que la séquence des actions à l'intérieur de chaque transition, alors que l'explication graphique intuitive est souhaitable et qu'il est intéressant de connaître les ressources ainsi que leurs relations avec les états.

Les illustrations d'état sont généralement exprimées par des éléments graphiques indiquant les ressources pertinentes de l'état en cours du processus. Cette version convient à des applications dans lesquelles sont définis des éléments graphiques appropriés; l'utilisateur peut donc employer cette représentation pour n'importe quelle application en définissant des éléments graphiques appropriés selon les besoins.

La version mixte convient lorsqu'il faut connaître à la fois la séquence des actions intérieures à chaque transition et les descriptions détaillées des états.

D.5.4.2 *Illustration d'état et élément graphique*

D.5.4.2.1 *Élément graphique et texte qualificatif*

Si l'on a choisi l'option illustration d'état, celle-ci se compose d'éléments graphiques et d'un texte qualificatif, comme indiqué dans la figure D-5.4.1 a) à D-5.4.1 d).

Cette combinaison rend compréhensibles les illustrations d'état. A titre d'exemple, la figure D-5.4.1 a) donne la signification d'un récepteur à cadran manipulé par le processus, l'exemple b) celle d'un émetteur de tonalité de numérotation émettant un signal permanent vers l'environnement.

A noter que les signaux de sortie (signaux non permanents) et les ressources pertinentes ne sont pas écrits dans les illustrations d'état; les signaux de sortie peuvent être écrits dans un diagramme de transition.

L'exemple c) montre un temporisateur dont l'expiration est toujours représentée par une entrée. A noter que l'illustration graphique recommandée pour le temporisateur comporte le signal d'entrée pertinent t_1 .

Le dernier exemple, d), signifie qu'un enregistreur de messages vocaux est en cours de fonctionnement.

L'identité de la ressource peut être considérablement abrégée et devrait, si possible, être placée à l'intérieur de l'illustration graphique appropriée. Ainsi, les éléments graphiques qui sont qualifiés sont tout à fait évidents.

D.5.4.2.2 *Illustrations d'état complètes*

Chaque illustration d'état doit comporter un nombre suffisant d'éléments graphiques afin de montrer:

- a) quelles ressources le processus met en oeuvre au cours de l'état représenté. Exemples: trajets de commutation, récepteurs de signalisation, émetteurs de signaux permanents et modules de commutation;
- b) s'il y a en ce moment un ou plusieurs temporisateurs qui contrôlent le processus;
- c) dans le cas où le processus concerne le traitement des appels, si la taxation est ou non actuellement en cours et quels abonnés sont taxés au cours de cette phase de l'appel;

- d) quels objets appartenant effectivement à un autre processus (environnement) sont considérées comme en relation avec des ressources du processus pendant l'état en cours;
- e) les signaux permanents de sortie qui sont émis dans cet état;
- f) la relation entre les signaux et ressources existants dans l'état;
- g) l'inventaire des ressources concernant l'état en cours du processus.

Figura D-5.4.1, (N), p. 1

D.5.4.2.3 *Exemple*

A titre d'exemple d'application des principes exposés ci-dessus, considérons l'état de la figure D-5.4.2. On peut voir que, dans cet état:

- a) les ressources affectées aux processus sont: un récepteur de chiffres à cadran, un émetteur de tonalités de numérotation, un poste d'abonné appartenant à l'environnement et des trajets de communication reliant ces organes;
- b) un temporisateur t_0 surveille le processus;
- c) aucune taxation n'est en cours;
- d) l'abonné est identifié comme l'abonné A mais aucune autre information de catégorie n'est prise en considération;

e) les signaux d'entrée suivants sont attendus: A_on_hook (A raccroché), digit (chiffre) (chiffre numéroté) et t0 (le temporisateur de supervision t0 fonctionne);

f) le signal permanent de sortie DT (tonalité de numérotation) a été mis avant cet état et pendant celui-ci.

Figure D-5.4.2, (N), p. 2

D.5.4.2.4 Vérification de la cohérence de diagrammes LDS avec éléments graphiques

On constate que l'illustration d'état est plus compacte et que, d'une certaine manière, elle offre au lecteur plus d'informations; cependant, l'identification de la série exacte d'opérations accomplies au cours de la transition exige un examen très attentif des ressources.

En outre, une simple observation de l'illustration d'état ne permet pas de déterminer l'ordre des actions dans la transition.

Les éléments graphiques représentés à l'extérieur du bloc sont des éléments qui ne sont pas directement commandés par le processus donné: ceux qui sont représentés à l'intérieur du symbole <<limites du bloc>> sont directement commandés par ce processus. Par exemple, le processus d'appel partiellement spécifié dans la figure D-5.4.3 peut allouer ou libérer l'émetteur de tonalité d'appel, l'émetteur de sonnerie et les trajets de conversation; il peut également déclencher ou arrêter le temporisateur T4, mais il ne peut commander directement la condition du combiné de l'abonné.

En concevant la logique à partir d'une spécification LDS avec éléments graphiques, seuls les éléments graphiques représentés à l'intérieur des limites du bloc ont une influence sur les actions exécutées pendant les séquences de la transition. Les éléments graphiques complexes représentés à l'intérieur des limites de ce bloc sont normalement inclus dans une illustration d'état:

- a) soit parce qu'ils indiquent les ressources et l'état de l'environnement concernées par le signal d'entrée du processus pendant l'état donné;
- b) soit pour améliorer l'intelligibilité du diagramme.

D.5.4.2.5 Utilisation du symbole <<temporisateur>>

Que l'on emploie ou non des éléments graphiques, l'expiration d'un délai de temporisation est toujours représentée par une entrée.

La présence d'un symbole de temporisateur dans une illustration d'état implique qu'un temporisateur fonctionne pendant cet état.

Figure D-5.4.3, (N), p. 3

Conformément au principe général exposé dans les Recommandations, le démarrage, l'arrêt, le redémarrage et l'expiration du temporisateur sont représentés à l'aide d'éléments graphiques de la manière suivante:

- a) pour montrer qu'une temporisation commence au cours d'une transition donnée, le symbole temporisateur doit apparaître sur l'illustration d'état qui correspond à la fin de cette transition et non sur celle qui correspond à son début;
- b) inversement, pour montrer qu'une temporisation s'arrête au cours d'une transition, le symbole temporisateur doit apparaître sur l'illustration d'état qui correspond au début de cette transition et non sur celle qui correspond à sa fin;
- c) pour montrer qu'une temporisation est relancée au cours d'une transition un symbole explicite de temps doit être représenté dans cette transition (on en voit deux exemples sur la figure D-5.4.4);
- d) l'expiration du délai d'une temporisation donnée est représentée par un symbole d'entrée associé à un état dont l'illustration porte le symbole <<temporisateur>> correspondant. Il peut naturellement arriver que plusieurs temporisateurs surveillent à la fois le même processus (voir la figure D-5.4.5).

Figure D-5.4.4, (N), p. 4

Figure D-5.4.5, (N), p. 5

D.5.5 *Diagrammes auxiliaires*

Pour faciliter la lecture et la compréhension de diagrammes de processus de grande taille et/ou complexes, l'auteur peut y ajouter des diagrammes auxiliaires informels. De tels documents ont pour but de donner une description synoptique ou simplifiée du comportement du processus (ou d'une partie de celui-ci). Les documents auxiliaires ne remplacent pas les documents en LDS mais constituent une introduction à ceux-ci.

On trouvera dans la présente section des exemples de certains diagrammes auxiliaires couramment utilisés, notamment des diagrammes synoptiques d'état, des matrices état/signal, et des diagrammes de séquençement. (Le diagramme en arbre de bloc décrit au § D.4.4 est également un diagramme auxiliaire.)

D.5.5.1 *Diagramme synoptique d'état*

Son objectif est de donner une vue d'ensemble des états d'un processus, et d'indiquer quelles transitions sont possibles entre eux. Etant donné qu'il s'agit de donner un aperçu, l'on peut négliger les états ou les transitions de peu

d'importance.

Les diagrammes se composent de symboles d'état, de flèches représentant des transitions et, éventuellement, de symboles de début et d'arrêt.

Le symbole d'état doit indiquer le nom de l'état référencé. Plusieurs noms d'état peuvent être inscrits dans le symbole, et il est possible d'employer un astérisque (*) pour désigner tous les états.

Chacune des flèches peut, de même que chacune des sorties possibles pendant la transition, être associée au nom du signal ou de l'ensemble de signaux qui déclenche la transition.

On trouvera dans la figure D-5.5.1 un exemple de diagramme synoptique d'état.

Figure D-5.5.1, (MC), p. 6

Il est possible de répartir sur plusieurs diagrammes le diagramme synoptique d'état d'un processus; chacun des diagrammes obtenus porte alors sur un aspect particulier, comme <<cas normal>>, traitement en cas de défaillance, etc.

Figure D-5.5.2, (N), p. 7

D.5.5.2 *Matrice ´etat/signal*

La matrice ´etat/signal doit servir de document <<pr´eliminaire>> à un diagramme de processus important. Elle indique les endroits où existent des combinaisons entre un ´etat et un signal dans le diagramme.

Le diagramme se compose d'une matrice bidimensionnelle; celle-ci présente sur un axe tous les ´etats d'un processus, et sur l'autre tous les signaux d'entrée valides d'un processus. L'´etat suivant est donné pour chaque ´élément de matrice, de même que les sorties possibles au cours de la transition. Une référence peut être indiquée pour permettre de trouver la combinaison donnée par les indices, si elle existe.

L'´élément correspondant à l'´etat fictif ou <<START>> et au signal fictif <<CREATE>> sert à indiquer l'´etat initial du processus.

Figure D-5.5.3, (MC), p. 8

Il est possible de fractionner la matrice en sous-parties réparties sur plusieurs pages. Les références sont celles qu'emploie normalement l'utilisateur dans la documentation.

Les signaux et les ´etats doivent être de préférence regroupés de façon que chaque partie de la matrice porte sur un aspect particulier du comportement du processus.

D.5.5.3 *Diagramme de séquençement*

Le diagramme de séquençement peut servir à montrer l'échange des séquences de signaux autorisées entre un ou plusieurs processus et leur environnement.

Ce diagramme doit donner une vue d'ensemble de l'échange de signaux entre les parties du système. Ce diagramme peut représenter l'ensemble ou une partie de l'échange de signaux, en fonction des aspects à mettre en évidence.

Les colonnes du diagramme indiquent les entités en communication (services, processus, blocs ou l'environnement).

Leurs interactions sont illustrées par un ensemble de lignes fléchées, dont chacune représente un signal ou un ensemble de signaux.

Figure D-5.5.4, (N), p. 9

On peut annoter chaque séquence afin de faire apparaître clairement l'ensemble d'informations échangées. Chaque ligne est accompagnée d'une annotation qui donne les renseignements requis (noms des signaux ou de procédures).

On peut placer un symbole de décision dans les colonnes pour indiquer que la séquence suivante est valide si la condition indiquée est vraie. Dans ce cas, le symbole de décision apparaît généralement plusieurs fois; il indique les différentes séquences produites par chacune des valeurs de la condition.

Ce diagramme peut représenter la totalité ou seulement un sous-ensemble significatif des séquences de signaux échangés.

La représentation de l'interaction réciproque des services résultant de la subdivision d'un processus représente une application utile de ce type de diagramme.

Les diagrammes de séquençage ne comprennent généralement pas toutes les séquences possibles; ils constituent souvent un préalable à la définition complète.

D.6 *Définition des données en LDS*

D.6.1 *Directives applicables aux données en LDS*

On trouvera dans la présente section des renseignements complémentaires sur les concepts définis au § 5 de la Recommandation concernant le LDS. La principale différence entre la présente section et l'ancienne Recommandation Z.104 est que cette dernière a fait l'objet d'une révision à des fins de clarification et d'harmonisation avec l'ISO. Certains des mots clés ont été modifiés et des adjonctions ont été faites mais la cohérence de la sémantique a

été assurée avec le Livre rouge. L'emploi des données décrites aux § 2, 3 et 4 de la nouvelle Recommandation (auparavant Z.101-Z.103) est resté tel quel.

D.6.1.1 Introduction générale

Les types de données du LDS sont fondés sur l'approche de <<types de données abstraits>>: on ne décrit pas la manière dont un type doit être mis en oeuvre, mais on se borne à dire quel sera le résultat des opérateurs appliqués aux valeurs.

Lorsque l'on définit des données abstraites, chaque segment de la définition, appelée <<définition de type partielle>> est introduit par le mot-clé NEWTYPE. Chaque définition de type partielle a une incidence sur les autres, de sorte que toutes les définitions de type partielles au niveau du système constituent une définition de type de données unique. Si plusieurs définitions de type partielles sont introduites à un niveau inférieur (niveau de bloc, par exemple), leur ensemble constitue, avec les définitions de niveau supérieur, une définition de type de données. Cela signifie qu'en un point quelconque de la spécification, il n'existe qu'une seule définition de type de données.

En substance, la définition de type de données comprend trois parties:

- a) définitions de sortes;
- b) définitions d'opérateurs;
- c) équations.

Chacune de ces parties fait l'objet d'explications dans les paragraphes qui suivent. La définition de type de données est structurée en définitions de type de données partielles, chacune introduisant une sorte. Les définitions d'opérateurs et les équations recouvrent les définitions de type de données partielles.

D.6.1.2 Sortes

Une sorte est un ensemble de valeurs qui peut avoir un nombre fini ou infini d'éléments mais ne peut être vide.

Exemples:

- a) l'ensemble de valeurs de la sorte booléenne est { vrai (vrai), False (faux) }
- b) l'ensemble de valeurs de la sorte Natural (naturel) est l'ensemble infini des nombres naturels { 0, 1, 2, ... }
- c) l'ensemble de valeurs de la sorte Couleur _primaire est { vert, Rouge, Bleu }

Tous les éléments d'une sorte ne doivent pas être directement fournis par l'utilisateur (cela exigerait un temps infini dans le cas des nombres naturels), mais le nom de la sorte doit être indiqué. Dans la syntaxe concrète, le mot-clé NEWTYPE est directement suivi par le nom de la sorte (certaines autres possibilités seront indiquées plus loin). Ce nom est surtout utilisé dans les définitions d'opérateurs, comme expliqué au § D.6.1.3, et dans les déclarations de variables.

D.6.1.3 Opérateurs, littéraux et termes

Les valeurs d'une sorte peuvent être définies de trois manières:

- a) par une énumération: les valeurs sont définies dans la section des littéraux;
- b) par des opérateurs: les valeurs sont données comme les résultats d'«applications» d'opérateurs;
- c) par une combinaison d'énumérations et d'opérateurs.

La combinaison de littéraux et d'opérateurs donne des termes. Les relations entre les termes sont exprimées à l'aide d'équations. Les paragraphes qui suivent traitent des littéraux, des opérateurs et des termes; le § D.6.1.4 traite des équations.

D.6.1.3.1 Littéraux

Les littéraux sont des valeurs énumérées d'une sorte. Une définition de type partiel ne doit pas nécessairement comporter de littéraux: tous les éléments de la sorte peuvent être définis au moyen d'opérateurs. Les littéraux peuvent être considérés comme des opérateurs dépourvus d'arguments. Une relation entre les littéraux peut être exprimée dans des équations. Dans la syntaxe concrète, les littéraux sont introduits après le mot-clé LITERALS.

Exemples:

a) La définition de la sorte booléenne contient deux littéraux, à savoir True (vrai) et False (faux). Ainsi, la définition du type booléen se présente comme suit:

```
NEWTYPE Boolean
  LITERALS True, False
. | |
ENDNEWTYPE Boolean;
```

b) la sorte naturelle peut être définie à l'aide d'un littéral, le zéro. Les autres valeurs peuvent être générées au moyen de ce littéral et d'opérateurs;

c) les valeurs de la sorte couleur _primaire peuvent être définies de la même manière que les littéraux booléens;

```
NEWTYPe Primary _colour
```

```
LITERALS Red, Green, Blue
```

```
. | |
```

```
ENDNEWTYPe Primary _colour;
```

d) au § D.6.1.3.2, le troisième exemple c) présente une définition de type partielle sans littéraux.

D.6.1.3.2 Opérateurs

Un opérateur est une fonction mathématique qui met en concordance une ou plusieurs valeurs (éventuellement de sortes différentes) avec une valeur résultante. Les opérateurs sont introduits après le mot-clé OPERATORS, la ou les sortes de leur ou leurs arguments et la sorte de résultat sont également indiquées (on parle de signature des opérateurs).

Exemples:

a) dans le type booléen, un opérateur appelé <<not>> peut être défini; il aura un argument de sorte booléen et également un résultat de sorte booléen. Dans la définition du type booléen, il se présente comme suit:

```
NEWTYPe Boolean
```

```
LITERALS True, False
```

```
OPERATORS <<Not>>: Boolean->Boolean;
```

```
. | |
```

```
ENDNEWTYPe Boolean;
```

b) l'opérateur susmentionné nécessaire pour construire tous les nombres naturels est Next. Cet opérateur prend un argument de sorte naturelle et donne une valeur naturelle (la valeur supérieure suivante) en tant que résultat;

c) il est possible de définir pour des couleurs un nouveau type qui n'a pas de littéraux mais utilise des littéraux de la sorte couleur primaire et certains opérateurs:

```
NEWTYPe Color
```

```
OPERATORS
```

```
Take: Primary _colour->Colour;
```

```
Mix: Primary _colour, Colour->Colour;
```

```
. | |
```

```
ENDNEWTYPe Colour;
```

Il s'agissait de prendre une couleur primaire et de la considérer comme une couleur puis de commencer à mélanger plusieurs couleurs primaires afin d'obtenir d'autres couleurs.

D.6.1.3.3 Termes

A l'aide de littéraux et d'opérateurs, il est possible de construire comme suit l'ensemble des termes:

- 1) Rassembler tous les littéraux dans un ensemble de la sorte dans laquelle ils sont définis — chaque littéral est un terme.
- 2) Un nouvel ensemble de termes est créé pour chaque opérateur lorsque l'opérateur est appliqué à toutes les combinaisons possibles de termes de la sorte correcte créés précédemment:
 - a) pour la sorte <<Boolean>>, l'ensemble de littéraux est { rue (vrai), False (faux) } Le résultat de cette étape est { ot(True), Not(False) } parce que nous avons seulement l'opérateur Not (non);
 - b) pour la sorte <<Natural>>, le résultat de cette étape est { ext(0) }
 - c) pour la sorte <<Colour>> l'ensemble de littéraux est vide mais le résultat de cette étape est { ake(Red), Take(green), Take(Blue) }
- 3) Les termes des ensembles créés au cours de l'étape précédente sont tous la sorte du résultat de l'opérateur appliqué; par exemple, tous les résultats de l'opérateur Not sont de la sorte booléenne. Alors, on réunit tous les ensembles de la même sorte, aussi bien des ensembles initiaux que les ensembles nouvellement créés:
 - a) pour la sorte booléenne, on obtient l'ensemble { rue, False, Not(True), Not(False) }
 - b) pour les nombres naturels cette étape donne l'ensemble { , Next(0) }

4) Les deux dernières étapes sont répétées à maintes reprises, et définissent en général un ensemble infini de termes:

a) l'ensemble de termes booléens générés par les littéraux True et False et l'opérateur Not est { True, False, Not(True), Not(False), Not(Not(True)), Not(Not(False)), Not(Not(Not(True))), . . . }

b) l'ensemble des termes naturels générés par le littéral 0 et l'opérateur Next est { 0, Next(0), Next(Next(0)), Next(Next(Next(0))), . . . }

c) l'ensemble des termes de couleur générés par les littéraux Red, Green et Blue de la sorte Primary_colour et les opérateurs Take et Mix est { Take(Red), Take(Green), Take(Blue), Mix(Red, Take(Red)), Mix(Red, Take(Green)), Mix(Red, Take(Blue)), Mix(Green, Take(Red)), Mix(Green, Take(Green)), Mix(Green, Take(Blue)), Mix(Blue, Take(Red)), Mix(Blue, Take(Green)), Mix(Blue, Take(Blue)), . . . }

D.6.1.4 Equations et axiomes

D'une manière générale, le nombre de termes générés au paragraphe précédent est supérieur au nombre de valeurs de la sorte. A titre d'exemple, il existe deux valeurs booléennes mais l'ensemble de termes booléens a un nombre infini d'éléments. Il existe cependant une possibilité d'établir des règles spécifiant quels sont les termes considérés comme désignant la même valeur. Ces règles sont appelées équations et font l'objet du paragraphe suivant. Deux types particuliers d'équations, les axiomes et les équations conditionnelles, font l'objet des § D.6.1.4.2 et D.6.1.4.3.

Les équations, les axiomes et les équations conditionnelles sont données, dans la syntaxe concrète après le mot-clé AXIOMS. Ce mot-clé a été conservé pour des raisons d'ordre historique.

D.6.1.4.1 Equations

Une équation indique quels sont les termes considérés comme désignant la même valeur. Une équation relie deux termes séparés par le symbole d'équivalence ==.

Par exemple <<Not(True) == False>> indique que les termes Not(True) et False sont équivalents; chaque fois que l'on trouve Not(True), on peut le remplacer par False sans changement de sens et vice versa.

Dans certaines équations, l'ensemble de termes est divisé en sous-ensembles discontinus de termes qui désignent la même valeur. Ces sous-ensembles sont appelées classes d'équivalence. Dans le langage courant, les classes d'équivalence sont identifiées aux valeurs.

Exemples:

a) L'ensemble de termes de la sorte booléenne est divisé en deux classes d'équivalence de termes par les deux axiomes suivants:

Not(True) == False;

Not(False) == True;

Les classes d'équivalence qui en résultent sont:

{ True, Not(False), Not(Not(True)), Not(Not(Not(False))),

Not(Not(Not(Not(True))))), Not(Not(Not(Not(Not(False))))), . . . }

et

{ False, Not(True), Not(Not(False)), Not(Not(Not(True))),

Not(Not(Not(Not(False))), Not(Not(Not(Not(Not(True))))) } . | |

Ces deux classes d'équivalence sont identifiées aux valeurs True (Vrai) et False (Faux);

b) Dans le cas de couleurs, on peut désire spécifier que le mélange d'une couleur primaire avec une couleur qui contient cette couleur primaire ne fait pas de différence. De plus, l'ordre dans lequel les primaires sont mélangés est sans importance. Cela peut être énoncé dans les équations suivantes:

Mix(Red, Take(Red)) == Take(Red);

Mix(Red, Mix(Red, Take(Green))) == Mix(Red, Take(Green));

Mix(Red, Mix(Red, Take(Blue))) == Mix(Red Take(Blue));

Mix(Red, Mix(Green, Take(Red))) == Mix(Green, Take(Red));

Mix(Red, Mix(Blue, Take(Red))) == Mix(Blue, Take(Red)); etc.

Cela demande beaucoup de travail car des équations similaires apparaissent pour toutes les permutations de Red, Green et Blue. C'est pourquoi le LDS comporte la construction FOR-ALL qui introduit les noms de valeur représentant une classe d'équivalence arbitraire (ou la valeur associée à cette classe d'équivalence). Cela peut être très utile dans la situation ci-dessus; toutes les équations susmentionnées et celles qui sont indiquées par etc. peuvent être écrites en quelques lignes sous la forme suivante:

```
FOR ALL p1, p2 IN Primary _colour
/*1*/ Mix(p1, Take(p1)) == Take(p1);
/*2*/ Mix(p1, Mix(p1, Take(p2))) == Mix(p1, Take(p2));
/*3*/ Mix(p1, Mix(p2, Take(p1))) == Mix(p2, Take(p1));
/*4*/ Mix(p1, Take(p2)) == Mix(p2, Take(p1));
/*4*/ FOR ALL c IN Colour
/*5*/ ( Mix(p1, Mix(p2, c)) == Mix(p2, Mix(p1, c)));
/*6*/ ( Mix(p1, Mix(p2, c)) == Mix(Mix(p1, Take(p2)) c))
/*4*/ )
```

Dans les équations ci-dessus, il y a chevauchement mais cela ne pose pas de problèmes pour autant que les équations ne se contredisent pas mutuellement.

Les équations susmentionnées créent 7 classes d'équivalence dans l'ensemble de termes de la sorte Colour, de sorte qu'avec ces équations, il y a sept valeurs de couleurs. Les termes suivants sont dans les classes d'équivalence différentes:

Take(Red), Take(Green), Take(Blue),
 Take(Red, Take(Green)),
 Mix(Green, Take(Blue)),
 Mix(Blue, Take(Red)),
 Mix(Blue, Mix(Green, Take(Red))).

Tous les autres termes de la sorte Colour sont équivalents à l'un des termes ci-dessus.

Dans les exemples d'équations comportant la construction FOR-ALL, appelées équations explicitement quantifiées, l'information que p1 et p2 sont des identificateurs de valeur de sorte Primary _colour est redondante; l'argument de l'opérateur Take et le premier argument de l'opérateur Mix ne peuvent être de la sorte Primary _colour. En général, les équations explicitement quantifiées deviennent plus lisibles mais il est possible d'omettre la quantification si la sorte des identificateurs de valeur peut être déduite du contexte. Dans ce cas, on dit que l'équation est implicitement quantifiée.

Exemple:

Les équations 4 et 5 ci-dessus sont les mêmes que:

```
Mix(p1, Take(p2)) , c) == Mix(p2, Take(p1));
Mix(p1, Mix(p2, c)) == Mix(p2, Mix(p1, c));
```

D.6.1.4.2 Axiomes

Les axiomes sont simplement des espèces particulières d'équation, introduites parce que dans des situations pratiques, de nombreuses équations se rapportent à des booléens. Dans ce cas, les équations tendent à prendre la forme $|| \text{ == True}$ (vrai), c'est-à-dire qu'elles indiquent qu'un terme donné est équivalent à True.

Exemple:

Admettons qu'un opérateur soit défini pour une couleur type: `Contains: Colour, Primary _colour-> Boolean`; ce qui doit donner la réponse True (vrai) si la couleur primaire est contenue dans la couleur et False (faux) dans le cas contraire. Voici un exemple des équations dont il s'agit:

```
FOR ALL p IN Primary _colour
```

```
( Contains(Take(p), p) == True;
```

```
FOR ALL c IN Colour
```

```
( Contains(Mix(p,c), p) == True)
```

```
)
```

La partie $\langle\langle \text{ == True} \rangle\rangle$ de ces équations peut être omise et les résultats sont appelés axiomes. Des axiomes peuvent se reconnaître à l'absence du symbole d'équivalence == ; ils désignent des termes qui sont équivalents à la valeur True (vrai) de la sorte booléenne.

La construction de la seconde équation peut paraître quelque peu forcée. Une meilleure manière d'écrire ces équations est indiquée après l'introduction de certaines constructions utiles.

D.6.1.4.3 Equations conditionnelles

Les équations conditionnelles constituent un moyen d'écrire des équations qui ne sont valables que dans certaines conditions. Ces conditions sont désignées par la même syntaxe que les équations non conditionnelles et sont séparées par un symbole `==>` de l'équation qui est valable si la condition est remplie.

Exemple:

L'exemple type d'une équation conditionnelle est la définition de la division en type réel où:

```
FOR ALL x, z IN Real
```

```
( z/=0 == True ==> (x/z) * z == x)
```

indique que, si la condition `z n'est pas égale à 0` est valable, la division par `z` suivie de la multiplication par `z` donne la valeur originale. Cette équation conditionnelle n'indique rien quant à ce qui devrait arriver si une valeur de la sorte `Real` était divisée par 0. Si l'on veut spécifier ce qui se passe en cas de division par zéro, il faudrait créer une équation conditionnelle de la forme suivante:

```
FOR ALL x, z IN Real
```

```
( z = 0 == True ==> (x/z) * z == . | |).
```

En pareil cas, cependant, il est recommandé de placer un `<<terme conditionnel>>` du côté droit, pour faciliter la lecture. Dans le cas ci-dessus l'équation deviendrait:

```
FOR ALL x, z IN Real
```

```
( (x/z) * z == IF z/=0
```

```
( (x/z) * z == THEN x
```

```
( (x/z) * z == ELSE . | |
```

```
( (x/z) * z == FI
```

```
)
```

D.6.1.5 Informations complémentaires concernant les équations et les axiomes

Les deux sections qui suivent traitent de certaines difficultés que l'on peut rencontrer lorsque des opérateurs donnent des résultats appartenant à une sorte déjà définie. Le § D.6.1.5.3 explique la notion d'erreur en tant que terme d'une équation.

D.6.1.5.1 Cohérence de la hiérarchie

En un point quelconque d'une spécification en LDS, il existe une seule et unique définition de type de données. Cette définition de type de données contient les sortes, opérateurs et équations prédéfinis et l'ensemble des sortes, opérateurs et équations définis par l'utilisateur dans les définitions de type partielles visibles en ce point. (C'est la raison pour laquelle un texte `NEWTTYPE . | | ENDNEWTTYPE` est appelé définition de type *partielle*.)

Il en résulte certaines conséquences pour les définitions de type aux niveaux inférieurs. Cette influence sur le type pourrait être peu souhaitable. A titre d'exemple, on pourrait spécifier de façon erronée que deux termes sont équivalents, les rendant ainsi équivalents alors qu'ils ne sont pas dans une portée englobante.

Il n'est pas admis de donner des équations telles que:

a) les valeurs d'une sorte qui sont différentes dans une portée à un niveau supérieur soient rendues équivalentes;

b) de nouvelles valeurs soient ajoutées à une sorte définie dans une portée de niveau supérieur.

Cela signifie par exemple que, dans un bloc au niveau du système, des définitions de type partielles spécifiées par l'utilisateur contenant un opérateur ayant un résultat prédéfini doivent rapporter tous les termes produits par cet opérateur à des valeurs de cette sorte de résultat.

Exemples:

a) Si, pour une raison quelconque, on donne l'axiome:

FOR ALL n, m IN Integer

((Fact(n) = Fact(m)) => (n = m))

afin de spécifier que si les résultats de l'opérateur Fact sont les m | mes, les arguments sont alors les m | mes. (A noter que => est l'implication booléenne; cela a peu de relation avec le signe d'équation conditionnelle ==>). Ainsi, par accident, les valeurs sont unifiées. Des équations des exemples précédents on peut déduire que Fact(0) = Fact(1) et cette dernière équation indique que 0 et 1 sont des notations différentes de la m | me valeur. Sur la base de cette dernière équation, on peut prouver que les nombres d'éléments de la sorte Integer sont réduits à un.

Avec l'aide d'une équation conditionnelle, on peut indiquer que, pourvu que n et m ne soient pas égaux à 0, le m | me résultat de l'opérateur Fact sur n et m implique $n = m$. En Led:

FOR ALL n, m IN Integer

$(n \neq 0, m \neq 0 \implies (\text{Fact}(n) = \text{Fact}(m)) \implies (n = m))$

A noter que cette dernière équation n'ajoute rien à la sémantique des nombres entiers; c'est un théorème qui peut | tre déduit des autres équations. Par ailleurs, l'adjonction d'une équation prouvable ne présente pas d'inconvénient.

b) Admettons que l'on découvre la nécessité d'un opérateur pour les factorielles lors de la spécification d'un type donné. Dans la définition de type partielle de ce type, l'opérateur Fact est introduit:

Fact: Integer \rightarrow Integer;

et les équations suivantes sont données pour définir cet opérateur:

Fact(0) == 1;

FOR ALL n IN Integer

$(n > 0 \implies \text{Fact}(n) == n * \text{Fact}(n-1))$

Ces équations ne définissent pas Fact(-1) et ainsi, c'est un terme de la sorte Integer qui n'a pas de relation avec d'autres termes de cette sorte. En conséquence, Fact(-1) est une nouvelle valeur de la sorte entier (et il en va de m | me pour Fact(-2), Fact(-3), etc). Cela n'est pas admis. L'exemple b) du § D.6.1.5.3 donne une définition correcte de fact.

D.6.1.5.2 Egalité et inégalité

Dans chaque type, les opérateurs d'égalité et d'inégalité sont implicites. Ainsi, si une définition de type partielle introduit une sorte S , il y a alors des définitions implicites d'opérateur:

"=": $S, S \rightarrow$ Boolean;

"!=": $S, S \rightarrow$ Boolean;

(Remarque — Les guillemets spécifient que = et != sont utilisées comme opérateurs infixes.)

L'opérateur d'égalité présente les propriétés prévisibles:

$a = a,$

$a = b \implies b = a,$

$a = b \text{ AND } b = c \implies a = c,$

$a = b \implies a == b,$

$a = b \implies \text{op}(a) = \text{op}(b)$ for all operators $\text{op}.$

Ces propriétés ne sont pas écrites en syntaxe LDS et ne doivent pas | tre énoncées dans des axiomes ou des équations car elles sont implicites. La valeur booléenne obtenue lorsque cet opérateur est appliquée est True (vrai) si les termes de la partie gauche et de la partie droite sont de m | me classe d'équivalence; sinon la valeur obtenue est False (faux). S'il n'est pas explicitement spécifié, que la valeur est True (vrai) ou False (faux), la spécification est incomplète.

Pour l'opérateur d'inégalité, c'est par une équation en LDS que l'on peut le mieux expliquer la sémantique:

FOR ALL a, b IN S

$(a \neq b \implies \text{Not}(a = b))$

Il n'y a pas de différence entre l'égalité et l'équivalence. Deux termes qui sont équivalents désignent la même valeur et l'opérateur d'égalité entre eux donne le résultat True (vrai).

D.6.1.5.3 *Erreur*

On a jugé nécessaire, pour les exemples qui précèdent, de spécifier que l'application de l'opérateur à certaines valeurs est considérée comme une erreur. Le LDS a un moyen de le spécifier formellement: l'élément ERROR. L'erreur devrait servir à exprimer:

<<l'application de cet opérateur à cette valeur n'est pas admise et lorsqu'il se présente, le comportement futur est indéfini>>.

Dans la syntaxe concrète, cela est indiqué par le terme Error!, qui ne peut être utilisé comme argument d'un opérateur.

Lorsque Error résulte de l'application d'un opérateur et que cette application est un argument d'un autre opérateur, l'application d'opérateur extérieur porte également Error dans son résultat (propagation-d'erreurs). Dans un terme conditionnel, la partie THEN ou la partie ELSE est évaluée, de sorte que l'une d'elle peut être une erreur sans que celle-ci soit évaluée (étant donné que l'autre partie de l'alternative est évaluée).

Exemples:

a) Dans l'exemple de division de valeurs de sorte Real, les poids peuvent être remplacés comme suit:

```
FOR ALL x, z IN Real
  ((x/z) * z == IF z/=0
  ((x/z) * z == THEN x
  ((x/z) * z == ELSE Error!
  ((x/z) * z == FI
)
```

Pour plus de clarté, on pourrait ajouter:

```
FOR ALL x IN Real
  ( x/0 == Error!)
```

b) Dans l'exemple comportant l'opérateur Fact, on pourrait spécifier que l'application de cet opérateur sur des entiers négatifs est considérée comme une erreur (Error). Cela permet d'éviter que Fact(-1), Fact(-2), ... ne deviennent de nouvelles valeurs de la sorte Integer (entier). Il conviendrait de définir l'opérateur Fact comme suit:

```
n < 0    ==>    Fact(n) == Error!;
Fact(0) == 1;
n > 0    ==>    Fact(n) == Fact(n-1) * n;
```

Ces trois lignes sont beaucoup plus claires que l'équation du style programmation indiquée ci-après. En général, le terme conditionnel devrait être utilisé s'il y a deux cas complémentaires; l'emboîtement de termes conditionnels rend l'équation illisible, comme on peut le voir ci-dessous:

```
Fact(n) == IF n > 0
Fact(n) == THEN Fact(n-1) * n
Fact(n) == ELSE IF n = 0
Fact(n) == ELSE THEN 1
Fact(n) == ELSE ELSE Error!
Fact(n) == ELSE FI
Fact(n) == FI
```

D.6.2 Générateurs et héritage

Le présent paragraphe traite de deux constructions qui peuvent être utilisées pour spécifier des types ayant des parties communes. Le générateur spécifie non un type mais un schéma, qui devient un type lorsque les sortes, opérateurs, littéraux et constantes formels sont remplacés par des termes réels.

L'héritage offre la possibilité de créer un nouveau type en partant d'un type déjà existant. Les noms de littéraux et d'opérateurs peuvent être renommés et il est possible de spécifier des littéraux, des opérateurs et des équations supplémentaires.

D.6.2.1 *Générateurs*

Une définition de générateur définit un schéma paramétré par des noms formels de sortes, de littéraux, de constantes et d'opérateurs. Les générateurs sont destinés à des types qui représentent des <<variations sur un thème>>, tels que des ensembles d'éléments, des chaînes d'éléments, des fichiers d'enregistrement, des tables de consultation, des tableaux.

On peut l'expliquer à l'aide d'un exemple pour lequel des variations peuvent être envisagées. Admettons qu'il soit nécessaire qu'un type ressemble à la notion mathématique d'un ensemble d'entiers. La partie suivante du texte fait partie de la définition de type de cet ensemble d'entiers.

Figure D.6.2.1 (comme tableau) [T37.100], p. 10

Toutes les équations ont une quantification implicite. La première équation indique que la suppression d'un élément de l'ensemble vide donne l'ensemble vide pour résultat. La seconde équation indique que la suppression après insertion du même élément donne pour résultat l'ensemble tel qu'il était avant l'insertion (à condition que l'ensemble ne contienne pas l'élément); sinon l'ordre d'insertion et de suppression peut être interchangeable. La troisième équation indique que l'élément vide ne contient pas d'éléments. La quatrième équation indique qu'un élément se trouve dans un ensemble s'il est le dernier élément ajouté ou s'il se trouvait dans l'ensemble avant l'adjonction du dernier élément. La dernière équation indique que l'ordre d'addition des éléments ne fait aucune différence.

Dans l'exemple de la figure D-6.2.1 `Int_set` (ensemble d'entiers) n'est qu'un exemple d'un ensemble et si l'on a également besoin d'un `Pid_set`, d'un `Subscriber_set` (ensemble d'abonnés) et d'un `Exchange_name_set` (ensemble de noms de central) dans la spécification, personne ne sera surpris qu'il contienne tous les opérateurs `Add` (ajouter), `Delete` (supprimer) et `Is_in` (est dans) et un littéral pour l'ensemble vide. Les équations données pour ces opérateurs facilitent la généralisation à d'autres ensembles.

Tel est le cas où la notion de générateur se révèle utile; le texte commun peut être donné une fois et être utilisé plusieurs fois. La figure D-6.2.2 présente le générateur. (A noter que les noms de sortes formels sont introduits par le mot clé `TYPE`, cela uniquement pour des raisons d'ordre historique.)

Au lieu d'utiliser `Integer` (entier), on utilise le type formel `Item` et, pour pouvoir donner différents noms à l'ensemble entier vide et aux ensembles vides dans d'autres types, on fait également de ce littéral un paramètre formel.

Figure D.6.2.2 (comme tableau) [T38.100], p. 11

Avec ce g n rateur, le type `Int _set` peut  tre construit comme suit:

```
NEWTYPE Int _set Set (Integer, empty _int _set)
ENDNEWTYPE Int _set;
```

Si l'on compare les figures D-6.2.1 et D-6.2.2, on constate que:

- a) `GENERATOR` et `ENDGENERATOR` sont remplac es par `NEWTYPE` et `ENDNEWTYPE` respectivement;
- b) les param tres formels de g n rateur (le texte entre parenth ses apr s le nom de g n rateur) sont supprim es;
- c) `Set`, `Item` et `empty _set` sont remplac es dans tout le g n rateur par `Int _set`, `Integer` et `empty _int _set`, respectivement.

Ainsi, il n'y a en fait aucune diff rence entre cet `Int _set` et celui de la figure D-6.2.1, mais . | |

— si l'on a besoin d'un ensemble de valeurs `PId`, on peut cr er le type   l'aide de:

```
NEWTYPE PId-set Set(PId, empty _pid _set)
ENDNEWTYPE PId _set;
```

— si l'on a besoin d'un ensemble d'abonn es, dans lequel les abonn es sont repr sent es par un type introduisant la sorte `Subscr`, l'ensemble d'abonn es peut  tre cr e   l'aide de:

```
NEWTYPE Subscr _set Set(Subscr, empty _subscr _set)
ENDNEWTYPE Subscr _set;
```

Cela permet d'économiser du papier, de plus, le travail est facilité parce qu'il suffit de penser une fois aux ensembles et que ce travail peut être délégué à des spécialistes qualifiés sur les types de données abstraites.

Exemple:

Cet exemple montre un g n rateur utilisant une sorte, un op rateur, un litt ral et une constante formels. Il d crit une rang e d' l ments ayant une longueur maximale max_length. La sorte comprend un litt ral d'esignant la rang e vide et les op rateurs pour insertion et suppression d' l ments dans une rang e ou de celle-ci, l'encha nement de rang es, le choix d'une sous-rang e et la d termination de la longueur d'une rang e. Ce dernier op rateur est rendu formel ce qui permet de la nommer   nouveau.

```
GENERATOR Row (TYPE Item, OPERATOR Length, LITERAL Empty,
```

```
    CONSTANT max_length)
```

```
LITERALS Empty
```

```
OPERATORS
```

```
Length: Row      -> Integer;
```

```
Insert: Row, Item, Integer      -> Row;
```

```
Delete: Row, Integer, Integer      -> Row;
```

```
"/": Row, Row      -> Row;
```

```
Select: Row, Integer, Integer      -> Row
```

```
/* and other operators relevant for rows of items */
```

```
AXIOMS
```

```
/* The equations for the operators above, among *
```

```
/* which the following two (or equivalents) */
```

```
Length(r) = max_length ==> Insert(r, itm, int) == Error!;
```

```
Length(r1) + Length(r2) > max_length ==> r1//r2 == Error!
```

```
ENDGENERATOR Row;
```

A noter que l'op rateur formel Length (longueur) et le litt ral Empty (vide) sont donn es une fois de plus dans le corps du g n rateur parce qu'ils sont renomm es lors de leur instantiation. Dans le cas de l'op rateur, les arguments et la sorte de r sultat ne sont donn es que dans le corps. Le g n rateur Row (rang e) peut servir   faire des lignes, des pages et des livres, comme suit:

```
NEWTYPE Line Row(Character, Width, Empty_line, 80)
```

```
ENDNEWTYPE Line;
```

```
NEWTYPE Page Row(Line, Length, Empty_page, 66)
```

```
ENDNEWTYPE Page;
```

```
NEWTYPE Book Row(Page, Nr_of_pages, Empty_book, 10000)
```

```
ENDNEWTYPE Book;
```

D.6.2.2 *H ritage*

L'h ritage constitue un moyen d' tablir toutes les valeurs de la sorte dite parente, certains ou tous les op rateurs du type parent et toutes les  quations du type parent. Pour les litt raux et les op rateurs, il existe une possibilit  de les

nommer à nouveau. En général, c'est une méthode satisfaisante parce que, dans ce cas, le lecteur peut déduire du contexte qu'il s'agit d'un autre type, même si les littéraux sont les mêmes.

Si un opérateur n'est pas hérité, on lui attribue systématiquement un autre nom inaccessible à l'utilisateur. Le fait que les opérateurs sont encore présents signifie que toutes les équations du type parent sont encore présentes (avec des opérateurs portant un autre nom). Cela garantit que les valeurs parentes sont héritées.

Avec la possibilité d'empêcher l'utilisation d'un opérateur (lorsqu'il n'est pas hérité), on assure la possibilité d'ajouter de nouveaux opérateurs. Après le mot clé `ADDING`, on peut donner des littéraux, des opérateurs et des équations comme dans un type ordinaire. Toutefois, il faut faire attention aux nouveaux littéraux et aux confusions possibles entre opérateurs hérités et ajoutés.

Lorsque les littéraux sont ajoutés, le résultat des opérateurs hérités, appliqués à des nouveaux littéraux, doit être défini (par des équations). Lorsque des opérateurs sont ajoutés, il ne faut pas oublier les opérateurs nommés à nouveau de manière invisible et les équations associées. Les équations de définition des opérateurs ajoutés devraient être compatibles avec les équations comportant des opérateurs hérités et non hérités.

Après cette liste d'avertissements, prenons quelques exemples:

a) Supposons que le newtype couleur est complet et disponible. Ce type est fondé sur le choix et le mélange de faisceaux de lumière de couleur primaire. Il faudrait de longues réflexions et un long texte et/ou copie pour définir quelque chose de semblable pour le choix et le mélange de peinture.

Une solution commode à ce problème consiste à faire du newtype Colour (couleur) un générateur en effectuant uniquement deux remplacements:

1) la première ligne

```
NEWTYPED Colour
```

devient

```
GENERATOR Colour (TYPE Primary _colour)
```

2) le mot-clé ENDNEWTYPED devient ENDGENERATOR.

On peut maintenant nommer à nouveau le générateur lorsqu'il est instancié. Supposons que la sorte antérieure Primary _colour soit appelée Light _primary, et que la sorte Paint _primary soit définie comme:

```
NEWTYPED Paint _primary
```

```
LITERALS Red, Yellow, Blue
```

```
ENDNEWTYPED Paint _primary;
```

Il est maintenant très facile de définir deux types similaires, un pour la lumière et un pour la peinture:

```
NEWTYPED Light _colours Colour (Light _primary) ENDNEWTYPED;
```

```
NEWTYPED Paint _colours Colour (Paint _primary) ENDNEWTYPED;
```

Il n'y a pas de problème jusqu'ici, mais comment peut-on voir la différence entre Take (Red) de Light _colour et celui de Paint _colour avec la même syntaxe? S'il est nécessaire de distinguer entre ces deux termes, on peut avoir recours à l'héritage. Au lieu de Light _colours et Paint _colours, les types Light (lumière) et Palette sont définis par héritage et le nom de l'opérateur Take (prendre) est modifié:

```
NEWTYPED Light
```

```
INHERITS Light _colours
```

```
OPERATORS (Beam=Take, Mix, Contains)
```

```
ADDING
```

```
LITERALS White
```

```
AXIOMS
```

```
White == Mix (Red, Mix (Yellow, Beam (Blue)))
```

```
ENDNEWTYPED Light;
```

Maintenant le newtype Light (lumière) a les littéraux de Light _colours et le littéral White (blanc). Light _colours n'a pas de littéraux qui lui soient propres (car il utilise les littéraux de Light _primary), de sorte que White est le seul littéral de Light. Les opérateurs et les équations de Light sont les mêmes que ceux de Light _colours, à l'exception du fait que le nom d'opérateur Take est remplacé par Beam (faisceau) et que l'équation pour White a été ajoutée. L'axiome ajoutée indique que ce littéral ajoutée devient un élément de l'ensemble de termes dans lequel les trois couleurs primaires sont mélangées.

Le newtype Palette a les littéraux de Paint _colours et l'opérateur Take est remplacé par Paint (peinture):

NEWTYPER Palette

INHERITS Paint _colours

OPERATORS (Paint _Take, Mix, Contains)

ENDNEWTYPER Palette;

b) Admettons que l'on veuille 'étendre' ensemble de types entiers (sorte Int _set), introduit dans la section précédente, par un opérateur qui trouve le plus petit entier de l'ensemble. Tout d'abord, il faut se demander si cet opérateur peut être introduit dans la définition de générateur pour le rendre disponible à tous les ensembles et autres choses.

S'il est vrai que cela peut être fait, cela limiterait l'élément à la définition de $>$ et $<$. Cela ne convient pas à tous les éléments (Pid par exemple) et il peut être préférable de créer un newtype ayant la sorte `New_int_set` comportant un opérateur `Min`.

```
NEWTYPE New_int_set
  INHERITS Int_set
  OPERATORS ALL
  ADDING
  OPERATORS
    Min: New_int_set-> Integer
  AXIOMS
    Min(Empty_int_set) == Error!;
    Min(Add(Empty_int_set, x)) == x;
    Min(Add(Add(nis,x),y)) ==
      IF y < Min(Add(nis,x))
      THEN y
      ELSE Min(Add(nis,x))
    FI
ENDNEWTYPE New_int_set;
```

Etant donné que l'adjonction après une instantiation de générateur est relativement courante, le texte commençant par `ADDING` et se terminant juste avant le `ENDNEWTYPE` peut être donné à l'intérieur de l'instanciation de générateur. On en trouvera un exemple au § 5.4.1.12 de la Recommandation.

D.6.3 *Observations relatives aux équations*

Lorsque l'on introduit un nouveau type de données, il est essentiel d'introduire suffisamment d'équations. Dans les paragraphes qui suivent, trois observations sont présentées concernant les équations qui en faciliteront l'établissement.

D.6.3.1 *Conditions générales*

De quelque manière que l'on construise les équations, il faut tenir compte des faits suivants:

- a) chaque opérateur apparaît au moins une fois dans l'ensemble des équations (sauf pour les cas «pathologiques»);
- b) tous les énoncés vrais peuvent être tirés des équations. Ils sont soit indiqués comme des axiomes, soit déduits par substitution de termes équivalents dans les équations;
- c) aucune incohérence doit être décelée, c'est-à-dire que l'on ne peut déduire des équations que `Vrai = Faux`.

Une procédure permettant de trouver des équations peut être exprimée en LDS informel, comme indiquée dans la figure D-6.3.1.

D.6.3.2 Application de fonctions aux constructeurs

D'une manière générale, l'ensemble d'opérateurs possède un sous-ensemble d'opérateurs appelés <<constructeurs>> et <<fonctions>>. Les constructeurs peuvent servir à générer toutes les valeurs (classes d'équivalence) de la sorte. Dans cette approche, les littéraux sont considérés comme des opérateurs sans argument.

Exemples:

- a) le type booléen a ses littéraux pour constructeurs;
- b) le type naturel a le littéral 0 et l'opérateur Next comme constructeurs; un naturel quelconque peut être construit seulement avec 0 et Next;
- c) le générateur pour les ensembles a le littéral ensemble _vide et l'opérateur _additif comme constructeurs; un ensemble quelconque ne peut être construit qu'en utilisant Empty _set (ensemble _vide) et Add (ajouter);
- d) le type entier peut être construit au moyen des littéraux 0 et 1, des opérateurs + et moins unaires.

Figura D-6.3.1, (N), p. 12

A noter qu'il y a parfois plusieurs choix possibles pour l'ensemble de constructeurs. Tout choix sera valable pour le reste de la présente section mais les petits ensembles sont généralement les meilleurs.

Ensuite, les fonctions sont traitées une par une. Pour chaque argument d'une fonction, seuls sont énumérés tous les termes possibles composés de constructeurs. Pour éviter le problème que posent les nombres infinis de termes, il faut appliquer la quantification.

Exemples:

a) Pour les nombres naturels, cette liste peut être réduite à

0

Next (n) où n est tout nombre naturel.

b) Pour les ensembles, la liste éventuelle peut être:

empty _set

Add (s,i) où s est tout ensemble et i tout élément (item).

Si, dans le terme de droite d'une équation ayant (s,i) du côté gauche, il y a une différence entre s étant vide ou n'étant pas vide, on peut réécrire la liste comme suit:

empty_set

$\text{Add}(\text{empty_set},i)$ où i est un élément (item) quelconque,

$\text{Add}(\text{Add}(s,i),j)$ où s est un ensemble quelconque et i, j , un élément quelconque.

Après la création de cette liste, on obtient les parties de gauche des équations en appliquant chaque fonction à une combinaison d'arguments tirés de la liste. Les identificateurs de valeurs de différents arguments reçoivent des noms différents. La procédure indiquée ci-dessus pour les fonctions peut être appliquée aux constructeurs; dans ce cas, elle donne les relations entre des termes où les constructeurs sont utilisés dans différents ordres.

Exemples:

a) Pour l'opérateur de multiplication de nombres naturels portant la signature

$\langle\langle * \rangle\rangle: \text{Natural}, \text{Natural} \rightarrow \text{Natural}$

Cette procédure donne la partie de gauche des équations (incomplètes) suivantes. L'utilisateur devra ajouter la partie de droite.

$\text{Next}(n) * 0 == . | | ;$

$\text{Next}(n) * \text{Next}(n) == . | | ;$

$\text{Next}(n) * 0 == . | | ;$

$\text{Next}(n) * \text{Next}(m) == . | | ;$

b) Pour les opérateurs *Is_in* (est dans) et *Delete* (supprimer, dans le générateur *Set* (voir le § D.6.2.1), cette approche est déjà appliquée.

c) Pour la sorte *Colour*, les constructeurs sont *Take* et *Mix*. Un opérateur analogue à *Contains* dans le § D.6.1.4.2 doit être défini pour les arguments.

Take(p) où p est toute couleur primaire

Mix(p,c) où p est toute couleur primaire et c une couleur quelconque.

Etant donné que l'on avait annoncé au § D.6.1.4.2 que les équations complètes seraient données pour cet opérateur:

$\text{Contains}(\text{Take}(p),q) == p=q;$

$\text{Contains}(\text{Mix}(p,c),q) == (p=q) \text{ OR } \text{Contains}(c,q);$

Cette procédure de construction peut donner plus d'équations que cela n'est nécessaire, mais elle est très sûre. Dans l'exemple susmentionné de multiplication des nombres naturels, il est vraisemblable que la propriété de commutativité de la multiplication sera indiquée et qu'en conséquence, seule la dernière (ou la seconde) équation des trois premières sera nécessaire.

La procédure décrite dans cette section peut être appliquée en combinaison avec la procédure décrite dans la section précédente, où elle est utile pour la tâche $\langle\langle \text{Penser_à_un_énoncé} \rangle\rangle$.

D.6.3.3 *Spécification d'ensemble d'essai*

On peut aussi considérer les équations du point de vue de la mise en oeuvre. Si les opérateurs sont mis en oeuvre sous la forme de fonctions dans un langage de programmation, les équations montrent comment ces fonctions doivent être testées.

Il convient d'évaluer les expressions correspondant à la partie de gauche d'une équation, de faire de même pour la partie droite de cette équation et de voir si elles sont équivalentes. C'est la construction FOR ALL qui pourrait poser certains problèmes. Ceux-ci peuvent souvent être résolus de manière pragmatique:

Au lieu de test peut utiliser FOR ALL i IN Integer

le dispositif de test peut utiliser FOR ALL i IN { (em10,—1,0,1,1) et procéder ainsi dans la plupart des cas.

Considérer des équations comme nécessaires à la mise en oeuvre peut être utile pour ce qui est de la t | che <<Penser à un énoncé>> dans la procédure du § D.6.3.1.

D.6.4 *Caractéristiques*

La présente section décrit certains dispositifs du LDS qui sont rarement nécessaires dont on peut pratiquement se passer, mais qui rendent quelquefois la t | che plus facile.

D.6.4.1 *Opérateurs cachés*

Il arrive que l'ensemble des équations puisse être simplifié ou rendu plus lisible grâce à l'introduction d'un opérateur supplémentaire, mais cet opérateur ne devrait pas être utilisé dans les processus. Cela signifie que l'opérateur est visible de l'intérieur mais caché en dehors de la définition de type.

On peut atteindre ce résultat en définissant un <<type caché>>, c'est-à-dire un type que l'utilisateur ne doit pas employer. A partir de ce type caché, l'utilisateur peut hériter de tous les opérateurs auxquels il peut avoir accès; c'est le type hérité qui doit être utilisé. On peut s'assurer qu'il est correctement employé en examinant toutes les déclarations de variables (aucune variable de sorte introduite par le type caché ne doit apparaître).

Ce qui caractérise les opérateurs cachés, c'est qu'ils peuvent être atteints par une restriction de leur visibilité aux seules équations. On peut le faire en plaçant un point d'exclamation après l'opérateur.

Exemple:

La manière courante de faire un ensemble d'un élément à partir du générateur Set est la suivante:

```
Add(empty_set,x)
```

et c'est ainsi que doit le faire chaque utilisateur. Dans les équations, le spécificateur peut utiliser un opérateur spécial, par exemple:

```
Mk_set!:Item->Set;
```

défini par l'équation:

```
Mk_set!(itm) == Add(empty_set,itm);
```

qui peut être utilisé dans des définitions de type partiel mais non dans le corps de processus en LDS.

D.6.4.2 *Relations d'ordre*

Lorsqu'il faut spécifier une relation d'ordre sur les éléments d'une sorte, cela signifie en général qu'il faut définir quatre opérateurs (<,<=,>,>=) et les propriétés mathématiques ordinaires (transitivité, etc.). S'il y a de nombreux littéraux, il faut également donner de nombreuses équations. Par exemple, le type de données prédéfinies Character est défini de cette manière.

Le LDS comporte une caractéristique qui permet d'abrégier ces définitions de type longues, peu lisibles et ennuyeuses: c'est l'abréviation ORDERING (relation d'ordre).

ORDERING est donné dans la liste d'opérateurs, de préférence au début ou à la fin de celle-ci. Cela permet d'introduire des opérateurs de relation d'ordre et les équations normales. Lorsque ORDERING est spécifié, il faut donner les littéraux, s'il en existe, dans l'ordre ascendant.

Exemple:

```
NEWTYPE Even _decimal _digit
```

```
LITERALS 0,2,4,6,8
```

```
OPERATORS
```

```
ORDERING
```

```
ENDNEWTYPE Even _decimal _digit;
```

Maintenant, l'ordre 0<2<4<6<8 est implicite.

Au § D.6.2.2 (Héritage), on soulignait qu'il fallait s'assurer que les littéraux étaient ajoutés à une sorte héritée. Il convient d'en indiquer ici la raison.

Admettons que l'on désire l'extension suivante de la sorte Even _decimal _digit:

NEWTTYPE Decimal _digit

INHERITS Even _decimal _digit

OPERATORS ALL

ADDING

LITERALS 1,3,5,7,9

AXIOMS

$0 < 1; 1 < 2;$

$2 < 3; 3 < 4;$

$4 < 5; 5 < 6;$

$6 < 7; 7 < 8;$

$8 < 9$

ENDNEWTTYPE Decimal _digit;

Les axiomes donnés ici ne peuvent être omis. Sans ces axiomes, il ne peut y avoir qu'un ordre dit partiel:

$$0 < 2 < 4 < 6 < 8$$

et

$$1 < 3 < 5 < 7 < 9.$$

Avec les axiomes ci-dessus, on obtient un ordre complet:

$$0 < 1 < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9$$

mais avec l'axiome $9 < 0$ au lieu de l'ensemble d'axiomes ci-dessus, l'ordre complet serait le suivant:

$$1 < 3 < 5 < 7 < 9 < 0 < 2 < 4 < 6 < 8.$$

D.6.4.3 Sortes avec champs

Comme indiqué au § 5.4.1.10 de la Recommandation, on peut définir une sorte structurée sans constructions spéciales, mais les sortes structurées sont à la fois courantes et utiles, ce qui justifie certaines constructions supplémentaires dans le langage.

Une sorte structurée devrait être utilisée lorsqu'une valeur d'objet est formée par l'association de valeurs de plusieurs sortes. Chaque valeur de cette association est caractérisée par un nom, appelé nom de champ. La sorte d'un champ est fixe.

Exemples:

```
NEWTYPE Subscriber
```

```
  STRUCT numbers Number _key;
```

```
    name Name _key;
```

```
    admin Administrative;
```

```
ENDNEWTYPE Subscriber;
```

```
NEWTYPE Name _key
```

```
  STRUCT name,
```

```
    street Charstring;
```

```
    number Integer;
```

```
    city Charstring;
```

```
ENDNEWTYPE Name _Key;
```

Avec une sorte structurée, certains opérateurs sont définis implicitement:

a) l'opérateur constructeur, $\langle\langle(. \langle\langle\text{avant, et}\rangle\rangle.)\rangle\rangle$ après les valeurs de champ;

b) les opérateurs de sélection de champ, les variables de la sorte structurée suivies par un ! et le nom de champ, ou suivies par le nom de champ entre parenthèses. Il ne faut pas confondre la variable suivie d'un ! avec l'opérateur caché (§ D.6.4.1).

On trouvera un exemple dans la figure D-6.4.1.

Figure D.6.4.1 (comme tableau) [T39.100], p. 13

D.6.4.4 *Sortes indexées*

Une sorte indexée est une sorte pour laquelle le type a pour nom d'opérateur Extract! (extraction). Dans les types de données prédéfinies, le générateur Array est un tel type. Array est l'un des exemples les plus courants de type indexé.

Pour l'opérateur caché Extract!, il existe une syntaxe concrète spéciale qui doit être appliquée en dehors des définitions de type.

On peut penser que le type Index dans le générateur prédéfini Array doit être un type <<simple>> comme Integer, Natural ou Character. Toutefois, il n'y a pas de raison pour qu'une structure comme Name_key ne puisse pas être utilisée comme Index.

Exemple:

```
NEWTTYPE Subsc_data_base
  Array (Name_key, Subscriber)
ENDNEWTTYPE Subsc_data_base;
```

Les sortes Name_key et Subscriber sont celles qui ont été définies dans la section précédente. Supposons qu'il existe une procédure Bill comportant un paramètre de sorte Subscriber et que cette procédure soit définie dans un processus qui comporte aussi une variable Sub_db de la sorte Subsc_data_base. Dans ce processus, l'appel suivant pourrait apparaître.

```
CALL Bill (Sub_db) (. 'P.M.', 'Downingstreet'10. 'Londres'.));
```

D.6.4.5 *Valeur par défaut de variables*

Comme indiqué dans la section concernant la déclaration de variables (§ D.3.10.1), il est possible d'affecter des valeurs à une variable immédiatement après la déclaration. Cependant, certains types ont une valeur qui sera (presque) toujours la valeur initiale d'une variable. Il existe une caractéristique qui permet d'éviter d'écrire la valeur initiale pour chaque déclaration: la clause DEFAULT.

A titre d'exemple, on peut considérer l'ensemble. Il est très probable que presque toutes les variables, de tout ensemble imaginaire, seront initialisées avec empty_set.

La notation:

```
DEFAULT empty_set
```

après la liste d'équations indique que chaque variable de chaque instantiation de ce générateur sera initialisée à la valeur empty_set de cette instantiation, sauf s'il y a une initialisation explicite (voir le § D.3.10.1.)

S'il n'est pas sûr que la valeur initiale de toutes les variables d'une sorte soit la même, il ne faut pas utiliser la clause DEFAULT, sinon il est difficile d'éviter des surprises.

D.6.4.6 *Opérateurs actifs*

Les utilisateurs qui connaissent la Recommandation Z.104 de 1984 concernant le LDS pourraient se demander ce qui est arrivé aux opérateurs dits actifs. En fait, cette caractéristique a été supprimée, pour les raisons suivantes:

a) elle n'est pas nécessaire car les opérateurs courants et les procédures et/ou macros offrent la même capacité d'expression;

- b) elle compromet la lisibilité des équations;
- c) de nombreux utilisateurs ont eu des difficultés à l'utiliser correctement;
- d) elle ne s'intègre pas au modèle de type de données abstrait fondé sur les algèbres initiales qui constituent le modèle choisi comme base mathématique de cette partie du LDS.

MONTAGE: § D.7 SUR LE RESTE DE CETTE PAGE

