5i’

D.5.4 *Directives applicables à la repr´esentation en fonction des ´etats et aux ´el´ements graphiques*

Le pr´esent paragraphe d´ecrit la repr´esentation en fonction des ´etats du LDS/GR et des ´el´ements graphiques utilis´es.

Le § D.5.4.1 contient des observations d’ordre g´en´eral sur la repr´esentation en fonction des ´etats: caract´eristiques, applications pertinentes et variantes.

Le § D.5.4.2 explique la description d’´etats au moyen d’´el´ements graphiques.

D.5.4.1 *Observations d’ordre g´en´eral sur la repr´esentation en fonction des ´etats*

Le LDS/GR offre trois versions diff´erentes pour d´ecrire un diagramme de processus.

La premi`ere est appel´ee version du LDS/GR en fonction des transitions, dans laquelle les transitions sont d´ecrites exclusivement par des symboles d’action explicites.

La seconde est appel´ee version du LDS/GR orient´ee vers les ´etats ou extension graphique du LDS orient´ee vers les ´etats; les ´etats d’un processus y sont d´ecrits à l’aide d’illustrations d’´etats et les transitions ne sont donn´ees qu’implicitement, par les diff´erences entre les ´etats de d´epart et d’arriv´ee.

La derni`ere est appel´ee version mixte; il s’agit d’une combinaison des deux versions pr´ec´edentes.

On trouvera des exemples de ces trois versions dans l’annexe E du pr´esent fascicule.

La version orient´ee vers les transitions convient lorsque la s´equence des actions pr´esente plus d’importance que la description d´etaill´ee des ´etats.

La version orient´ee vers les ´etats d´ecrit en d´etail les ´etats par des ´enonc´es; elle convient donc au cas où un ´etat de processus pr´esente plus d’importance que la s´equence des actions à l’int´erieur de chaque transition, alors que l’explication graphique intuitive est souhaitable et qu’il est int´eressant de conna | tre les ressources ainsi que leurs relations avec les ´etats.

Les illustrations d’´etat sont g´en´eralement exprim´ees par des ´el´ements graphiques indiquant les ressources pertinentes de l’´etat en cours du processus. Cette version convient à des applications dans lesquelles sont d´efinis des ´el´ements graphiques appropri´es; l’utilisateur peut donc employer cette repr´esentation pour n’importe quelle application en d´efinissant des ´el´ements graphiques appropri´es selon les besoins.

La version mixte convient lorsqu’il faut conna | tre à la fois la s´equence des actions int´erieures à chaque transition et les descriptions d´etaill´ees des ´etats.

D.5.4.2 *Illustration d’´etat et ´el´ement graphique*

D.5.4.2.1 *El´ement graphique et texte qualificatif*

Si l’on a choisi l’option illustration d’´etat, celle-ci se compose d’´el´ements graphiques et d’un texte qualificatif, comme indiqu´e dans la figure D-5.4.1 a) à D-5.4.1 d).

Cette combinaison rend compr´ehensibles les illustrations d’´etat. A titre d’exemple, la figure D-5.4.1 a) donne la signification d’un r´ecepteur à cadran manipul´e par le processus, l’exemple b) celle d’un ´emetteur de tonalit´e de num´erotation ´emettant un signal permanent vers l’environnement.

A noter que les signaux de sortie (signaux non permanents) et les ressources pertinentes ne sont pas d´ecrits dans les illustrations d'´etat; les signaux de sortie peuvent | tre d´ecrits dans un diagramme de transition.

L'exemple c) montre un temporisateur dont l'expiration est toujours repr´esent´ee par une entr´ee. A noter que l'illustration graphique recommand´ee pour le temporisateur comporte le signal d'entr´ee pertinent t1.

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

L'identit´e de la ressource peut | tre consid´erablement abr´eg´ee et devrait, si possible, | tre plac´ee à l'int´erieur de l'illustration graphique appropri´ee. Ainsi, les ´el´ements graphiques qui sont qualifi´es sont tout à fait ´evidents.

D.5.4.2.2 *Illustrations d'´etat compl`etes*

Chaque illustration d'´etat doit comporter un nombre suffisant d'´el´ements graphiques afin de montrer:

a) quelles ressources le processus met en oeuvre au cours de l'´etat repr´esent´e. Exemples: trajets de commutation, r´ecepteurs de signalisation, ´emetteurs de signaux permanents et modules de commutation;

b) s'il y a en ce moment un ou plusieurs temporisateurs qui contr | lent 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 abonn´es sont tax´es au cours de cette phase de l'appel;

d) quels objets appartenant effectivement à un autre processus (environnement) sont consid´er´es comme en relation avec des ressources du processus pendant l’´etat en cours;

e) les signaux permanents de sortie qui sont ´emis dans cet ´etat;

f) la relation entre les signaux et ressources existants dans l’´etat;

g) l’inventaire des ressources concernant l’´etat 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 expos´es ci-dessus, consid´erons l’´etat de la figure D-5.4.2. On peut voir que, dans cet ´etat:

a) les ressources affect´ees aux processus sont: un r´ecepteur de chiffres à cadran, un ´emetteur de tonalit´es de num´erotation, un poste d’abonn´e appartenant à l’environnement et des trajets de communication reliant ces organes;

b) un temporisateur t0 surveille le processus;

c) aucune taxation n’est en cours;

d) l’abonn´e est identifi´e comme l’abonn´e A mais aucune autre information de cat´egorie n’est prise en consid´eration;

e) les signaux d’entr´ee suivants sont attendus: A _on _hook (A raccroch´e), digit (chiffre) (chiffre num´erot´e) et t0 (le temporisateur de supervision t0 fonctionne);

f) le signal permanent de sortie DT (tonalit´e de num´erotation) a ´et´e mis avant cet ´etat et pendant celui-ci.

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

D.5.4.2.4 *V´erification de la coh´erence de diagrammes LDS avec ´el´ements graphiques*

On constate que l'illustration d'´etat est plus compacte et que, d'une certaine mani`ere, elle offre au lecteur plus d'informations; cependant, l'identification de la s´erie exacte d'op´erations accomplies au cours de la transition exige un examen tr`es attentif des ressources.

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

Les ´el´ements graphiques repr´esent´es à l'ext´erieur du bloc sont des ´el´ements qui ne sont pas directement command´es par le processus donn´e: ceux qui sont repr´esent´es à l'int´erieur du symbole <<limites du bloc>> sont directement command´es par ce processus. Par exemple, le processus d'appel partiellement sp´ecifi´e dans la figure D-5.4.3 peut allouer ou lib´erer l'´emetteur de tonalit´e d'appel, l'´emetteur de sonnerie et les trajets de conversation; il peut ´egalement d´eclencher ou arr | ter le temporisateur T4, mais il ne peut commander directement la condition du combin´e de l'abonn´e.

En concevant la logique à partir d'une sp´ecification LDS avec ´el´ements graphiques, seuls les ´el´ements graphiques repr´esent´es à l'int´erieur des limites du bloc ont une influence sur les actions ex´ecut´ees pendant les s´equences de la transition. Les ´el´ements graphiques complexes repr´esent´es à l'int´erieur des limites de ce bloc sont normalement inlus dans une illustration d'´etat:

a) soit parce qu'ils indiquent les ressources et l'´etat de l'environnement concern´es par le signal d'entr´ee du processus pendant l'´etat donn´e;

b) soit pour am´eliorer l'intelligibilit´e du diagramme.

D.5.4.2.5 *Utilisation du symbole <<temporisateur>>*

Que l'on emploie ou non des ´el´ements graphiques, l'expiration d'un d´elai de temporisation est toujours repr´esent´ee par une entr´ee.

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

Conform´ement au principe g´en´eral expos´e dans les Recommandations, le d´emarrage, l’arr | t, le red´emarrage et l’expiration du temporisateur sont repr´esent´es à l’aide d’´el´ements graphiques de la mani`ere suivante:

a) pour montrer qu’une temporisation commence au cours d’une transition donn´ee, le symbole temporisateur doit appara | tre sur l’illustration d’´etat qui correspond à la fin de cette transition et non sur celle qui correspond à son d´ebut;

b) inversement, pour montrer qu’une temporisation s’arr | te au cours d’une transition, le symbole temporisateur doit appara | tre sur l’illustration d’´etat qui correspond au d´ebut de cette transition et non sur celle qui correspond à sa fin;

c) pour montrer qu’une temporisation est relanc´ee au cours d’une transition un symbole explicite de t | che doit | tre repr´esent´e dans cette transition (on en voit deux exemples sur la figure D-5.4.4);

d) l’expiration du d´elai d’une temporisation donn´ee est repr´esent´ee par un symbole d’entr´ee associ´e à un ´etat dont l’illustration porte le symbole <<temporisateur>> correspondant. Il peut naturellement arriver que plusieurs temporisateurs surveillent à la fois le m | me 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 séquencement. (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’´etat, de fl`eches repr´esentant des transitions et, ´eventuellement, de symboles de d´ebut et d’arr | t.

Le symbole d’´etat doit indiquer le nom de l’´etat r´ef´erenc´e. Plusieurs noms d’´etat peuvent | tre inscrits dans le symbole, et il est possible d’employer un ast´erisque (*) pour d´esigner tous les ´etats.

Chacune des fl`eches peut, de m | me que chacune des sorties possibles pendant la transition, | tre associ´ee au nom du signal ou de l’ensemble de signaux qui d´eclenchent la transition.

On trouvera dans la figure D-5.5.1 un exemple de diagramme synoptique d’´etat.

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

Il est possible de r´epartir sur plusieurs diagrammes le diagramme synoptique d’´etat d’un processus; chacun des diagrammes obtenus porte alors sur un aspect particulier, comme <<cas normal>>, traitement en cas de d´efaillance, 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 pr´esente sur un axe tous les ´etats d'un processus, et sur l'autre tous les signaux d'entr´ee valides d'un processus. L'´etat suivant est donn´e pour chaque ´el´ement de matrice, de m | me que les sorties possibles au cours de la transition. Une r´ef´erence peut | tre indiqu´ee pour permettre de trouver la combinaison donn´ee par les indices, si elle existe.

L'´el´ement 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 r´eparties sur plusieurs pages. Les r´ef´erences sont celles qu'emploie normalement l'usager dans la documentation.

Les signaux et les ´etats doivent | tre de pr´ef´erence regroup´es de fa,con que chaque partie de la matrice porte sur un aspect particulier du comportement du processus.

D.5.5.3 *Diagramme de s´equencement*

Le diagramme de s´equencement peut servir à montrer l'´echange des s´equences de signaux autoris´ees entre un ou plusieurs processus et leur environnement.

Ce diagramme doit donner une vue d'ensemble de l'´echange de signaux entre les parties du syst`eme. Ce diagramme peut repr´esenter l'ensemble ou une partie de l'´echange de signaux, en fonction des aspects à mettre en ´evidence.

Les colonnes du diagramme indiquent les entit´es 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 séquencement 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 g´en´erale*

Les types de donn´ees du LDS sont fond´es sur l'approche de <<types de donn´ees abstraits>>: on ne d´ecrit pas la mani`ere dont un type doit | tre mis en oeuvre, mais on se borne à dire quel sera le r´esultat des op´erateurs appliqu´e aux valeurs.

Lorsque l'on d´efinit des donn´ees abstraites, chaque segment de la d´efinition, appel´ee <<d´efinition de type partielle>> est introduit par le mot-cl´e NEWTYPE. Chaque d´efinition de type partielle a une incidence sur les autres, de sorte que toutes les d´efinitions de type partielles au niveau du syst`eme constituent une d´efinition de type de donn´ees unique. Si plusieurs d´efinitions de type partielles sont introduites à un niveau inf´erieur (niveau de bloc, par exemple), leur ensemble constitue, avec les d´efinitions de niveau sup´erieur, une d´efinition de type de donn´ees. Cela signifie qu'en un point quelconque de la sp´ecification, il n'existe qu'*une seule* d´efinition de type de donn´ees.

En substance, la d´efinition de type de donn´ees comprend trois parties:

a) d´efinitions de sortes;

b) d´efinitions d'op´erateurs;

c) ´equations.

Chacune de ces parties fait l'objet d'explications dans les paragraphes qui suivent. La d´efinition de type de donn´ees est structur´ee en d´efinitions de type de donn´ees partielles, chacune introduisant une sorte. Les d´efinitions d'op´erateurs et les ´equations recouvrent les d´efinitions de type de donn´ees partielles.

D.6.1.2 *Sortes*

Une sorte est un ensemble de valeurs qui peut avoir un nombre fini ou infini d'´el´ements mais ne peut | tre vide.

Exemples:

a) l'ensemble de valeurs de la sorte bool´eenne est { rue (vrai), False (faux }

b) l'ensemble de valeurs de la sorte Natural (naturel) est l'ensemble infini des nombres naturesls { , 1, 2, . | }

c) l'ensemble de valeurs de la sorte Couleur _primaire est { ert, Rouge, Ble }

Tous les ´el´ements d'une sorte ne doivent pas | tre directement fournis par l'utilisateur (cela exigerait un temps infini dans le cas des nombres naturels), mais le nom de la sorte doit | tre indiqu´e. Dans la syntaxe concr`ete, le mot-cl´e NEWTYPE est directement suivi par le nom de la sorte (certaines autres possibilit´es seront indiqu´ees plus loin). Ce nom est surtout utilis´e dans les d´efinitions d'op´erateurs, comme expliqu´e au § D.6.1.3, et dans les d´eclarations de variables.

D.6.1.3 *Op´erateurs, litt´eraux et termes*

Les valeurs d'une sorte peuvent | tre d´efinies de trois mani`eres:

a) par une ´enum´eration: les valeurs sont d´efinies dans la section des litt´eraux;

b) par des op´erateurs: les valeurs sont donn´ees comme les r´esultats d'<<applications>> d'op´erateurs;

c) par une combinaison d'´enum´erations et d'op´erateurs.

La combinaison de litt´eraux et d'op´erateurs donne des termes. Les relations entre les termes sont exprim´ees à l'aide d'´equations. Les paragraphes qui suivent traitent des litt´eraux, des op´erateurs et des termes; le § D.6.1.4 traite des ´equations.

D.6.1.3.1 *Litt´eraux*

Les litt´eraux sont des valeurs ´enum´er´ees d'une sorte. Une d´efinition de type partiel ne doit pas n´ecessairement comporter de litt´eraux: tous les ´el´ements de la sorte peuvent | tre d´efinis au moyen d'op´erateurs. Les litt´eraux peuvent | tre consid´er´es comme des op´erateurs d´epourvus d'arguments. Une relation entre les litt´eraux peut | tre exprim´ee dans des ´equations. Dans la syntaxe concr`ete, les litt´eraux sont introduits apr`es le mot-cl´e LITERALS.

Exemples:

a) La d´efinition de la sorte bool´eenne contient deux litt´eraux, à savoir True (vrai) et False (faux). Ainsi, la d´efinition du type bool´een se pr´esente comme suit:

NEWTYPE Boolean

  LITERALS True, False

  . | |

ENDNEWTYPE Boolean;

b) la sorte naturelle peut | tre d´efinie à l’aide d’un litt´eral, le z´ero. Les autres valeurs peuvent | tre g´en´er´ees au moyen de ce litt´eral et d’op´erateurs;

c) les valeurs de la sorte couleur _primaire peuvent
| tre d´efinies de la m | me mani`ere que les litt´eraux bool´eens;

NEWTYPE Primary _colour

LITERALS Red, Green, Blue

. | |

ENDNEWTYPE Primary _colour;

d) au § D.6.1.3.2, le troisi`eme exemple c) pr´esente une d´efinition de type partielle sans litt´eraux.

D.6.1.3.2 *Op´erateurs*

Un op´erateur est une fonction math´ematique qui met en concordance une ou plusieurs valeurs (´eventuellement de sortes diff´erentes) avec une valeur r´esultante. Les op´erateurs sont introduits apr`es le mot-cl´e OPERATORS, la ou les sortes de leur ou leurs arguments et la sorte de r´esultat sont ´egalement indiqu´ees (on parle de signature des op´erateurs).

Exemples:

a) dans le type bool´een, un op´erateur appel´e <<not>> peut | tre d´efini; il aura un argument de sorte bool´een et ´egalement un r´esultat de sorte bool´een. Dans la d´efinition du type bool´een, il se pr´esente comme suit:

NEWTYPE Boolean

LITERALS True, False

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

. | |

ENDNEWTYPE Boolean;

b) l’op´erateur susmentionnn´e n´ecessaire pour construire tous les nombres naturels est Next. Cet op´erateur prend un argument de sorte naturelle et donne une valeur naturelle (la valeur sup´erieure suivante) en tant que r´esultat;

c) il est possible de d´efinir pour des couleurs un nouveau type qui n’a pas de litt´eraux mais utilise des litt´eraux de la sorte couleur primaire et certains op´erateurs:

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 consid´erer comme une couleur puis de commencer à m´elanger plusieurs couleurs primaires afin d’obtenir d’autres couleurs.

D.6.1.3.3 *Termes*

A l’aide de litt´eraux et d’op´erateurs, il est possible de construire comme suit l’ensemble des termes:

1) Rassembler tous les litt´eraux dans un ensemble de la sorte dans laquelle ils sont d´efinis — chaque litt´eral est un terme.

2) Un nouvel ensemble de termes est cr´e´e pour chaque op´erateur lorsque l’op´erateur est appliqu´e à toutes les combinaisons possibles de termes de la sorte correcte cr´e´es pr´ec´edemment:

a) pour la sorte <<Boolean>>, l’ensemble de litt´eraux est { rue (vrai), False (faux } Le r´esultat de cette ´etape est { ot (True), Not (False } parce que nous avons seulement l’op´erateur Not (non);

b) pour la sorte <<Natural>>, le r´esultat de cette ´etape est { ext(0 }

c) pour la sorte <<Colour>> l’ensemble de litt´eraux est vide mais le r´esultat de cette ´etape est { ake(Red), Take(green), Take(Blue }

3) Les termes des ensembles cr´e´es au cours de l’´etape pr´ec´edente sont tous la sorte du r´esultat de l’op´erateur appliqu´e; par exemple, tous les r´esultats de l’op´erateur Not sont de la sorte bool´eenne. Alors, on r´eunit tous les ensembles de la m | me sorte, aussi bien des ensemble initiaux que les ensembles nouvellement cr´e´es:

a) pour la sorte bool´eenne, on obtient l’ensemble { rue, False, Not(True), Not(False }

b) pour les nombres naturels cette ´etape donne l’ensemble { , Next(0 }

4) Les deux derni`eres ´etapes sont r´ep´et´ees à maintes reprises, et d´efinissent en g´en´eral un ensemble infini de termes:

a) l’ensemble de termes bool´eens g´en´er´es par les litt´eraux True et False et l’op´erateur Not est { rue, False, Not(True), Not(False), Not(Not(True)), Not(Not(False)), Not(Not(Not(True))), . | | }

b) l’ensemble des termes naturels g´en´er´es par le litt´eral 0 et l’op´erateur Next est { , Next(0), Next(Next(0)), Next(Next(Next(0))), . | | }

c) l’ensemble des termes de couleur g´en´er´es par les litt´eraux Red, Green et Blue de la sorte Primary _colour et les op´erateurs Take et Mix est { ake(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 mani`ere g´en´erale, le nombre de termes g´en´er´es au paragraphe pr´ec´edent est sup´erieur au nombre de valeurs de la sorte. A titre d’exemple, il existe deux valeurs bool´eennes mais l’ensemble de termes bool´eens a un nombre infini d’´el´ements. Il existe cependant une possibilit´e d’´etablir des r`egles sp´ecifiant quels sont les termes consid´er´es comme d´esignant la m | me valeur. Ces r`egles sont appel´ees ´equations et font l’objet du paragraphe suivant. Deux types particuliers d’´equations, les axiomes et les ´equations conditionnelles, font l’objet des § D.6.1.4.2 et D.6.1.4.3.

Les ´equations, les axiomes et les ´equations conditionnelles sont donn´es, dans la syntaxe concr`ete apr`es le mot-cl´e AXIOMS. Ce mot-cl´e a ´et´e conserv´e pour des raisons d’ordre historique.

D.6.1.4.1 *Equations*

Une ´equation indique quels sont les termes consid´er´es comme d´esignant la m | me valeur. Une ´equation relie deux termes s´epar´es par le symbole d’´equivalence ==.

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

Dans certaines ´equations, l’ensemble de termes est divis´e en sous-ensembles discontinus de termes qui d´esignent la m | me valeur. Ces sous-ensembles sont appel´es classes d’´equivalence. Dans le langage courant, les classes d’´equivalence sont identifi´ees aux valeurs.

Exemples:

a) L’ensemble de termes de la sorte bool´eenne est divis´e en deux classes d’´equivalence de termes par les deux axiomes suivants:

Not(True) == False;

Not(False) == True;

Les classes d’´equivalence qui en r´esultent sont:

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

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

et

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

Ces deux classes d'´equivalence sont identifi´ees aux valeurs True (Vrai) et False (Faux);

b) Dans le cas de couleurs, on peut d´esirer sp´ecifier que le m´elange d'une couleur primaire avec une couleur qui contient cette couleur primaire ne fait pas de diff´erence. De plus, l'ordre dans lequel les primaires sont m´elang´es est sans importance. Cela peut | tre ´enonc´e dans les ´equations 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 ´equations 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 repr´esentant une classe d’´equivalence arbitraire (ou la valeur associ´ee à cette classe d’´equivalence). Cela peut | tre tr`es utile dans la situation ci-dessus; toutes les ´equations susmentionn´ees et celles qui sont indiqu´ees par etc. peuvent | tre d´ecrites 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 ´equations ci-dessus, il y a chevauchement mais cela ne pose pas de probl`emes pour autant que les ´equations ne se contredisent pas mutuellement.

Les ´equations susmentionn´ees cr´eent 7 classes d’´equivalence dans l’ensemble de termes de la sorte Colour, de sorte qu’avec ces ´equations, il y a sept valeurs de couleurs. Les termes suivants sont dans les classes d’´equivalence diff´erentes:

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 ´equivalents à l’un des termes ci-dessus.

Dans les exemples d’´equations comportant la construction FOR-ALL, appel´ees ´equations explicitement quantifi´ees, l’information que p1 et p2 sont des identificateurs de valeur de sorte Primary _colour est redondante; l’argument de l’op´erateur Take et le premier argument de l’op´erateur Mix ne peuvent | tre de la sorte Primary _colour. En g´en´eral, les ´equations explicitement quantifi´ees deviennent plus lisibles mais il est possible d’omettre la quantification si la sorte des identificateurs de valeur peut
| tre d´eduite du contexte. Dans ce cas, on dit que l’´equation est implicitement quantifi´ee.

Exemple:

Les ´equations 4 et 5 ci-dessus sont les m | mes 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 esp`eces particuli`eres d’´equation, introduites parce que dans des situations pratiques, de nombreuses ´equations se rapportent à des bool´eens. Dans ce cas, les ´equations tendent à prendre la forme . | | == True (vrai), c’est-`a-dire qu’elles indiquent qu’un terme donn´e est ´equivalent à True.

Exemple:

Admettons qu’un op´erateur soit d´efini pour une couleur type: Contains: Colour, Primary _colour-> Boolean; ce qui doit donner la r´eponse True (vrai) si la couleur primaire est contenue dans la couleur et False (faux) dans le cas contraire. Voici un exemple des ´equations 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 <<== True>> de ces ´equations peut | tre omise et les r´esultats sont appel´es axiomes. Des axiomes peuvent se reconna | tre à l’absence du symbole d’´equivalence ==; ils d´esignent des termes qui sont ´equivalents à la valeur True (vrai) de la sorte bool´eenne.

La construction de la seconde ´equation peut para | tre quelque peu forc´ee. Une meilleure mani`ere d’´ecrire ces ´equations est indiqu´ee apr`es l’introduction de certaines constructions utiles.

D.6.1.4.3 *Equations conditionnelles*

Les ´equations conditionnelles constituent un moyen d’´ecrire des ´equations qui ne sont valables que dans certaines conditions. Ces conditions sont d´esign´ees par la m | me syntaxe que les ´equations non conditionnelles et sont s´epar´ees par un symbole ==> de l’´equation qui est valable si la condition est remplie.

Exemple:

L’exemple type d’une ´equation conditionnelle est la d´efinition de la division en type r´eel o`u:

FOR ALL x, z IN Real

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

indique que, si la condition <<z n’est pas ´egale à 0>> est valable, la division par z suivie de la multiplication par z donne la valeur originale. Cette ´equation conditionnelle n’indique rien quant à ce qui devrait arriver si une valeur de la sorte Real ´etait divis´ee par 0. Si l’on veut sp´ecifier ce qui se passe en cas de division par z´ero, il faudrait cr´eer une ´equation conditionnelle de la forme suivante:

FOR ALL x, z IN Real

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

En pareil cas, cependant, il est recommand´e de placer un <<terme conditionnel>> du c | t´e droit, pour faciliter la lecture. Dans le cas ci-dessus l’´equation 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 compl´ementaires concernant les ´equations et les axiomes*

Les deux sections qui suivent traitent de certaines difficult´es que l’on peut rencontrer lorsque des op´erateurs donnent des r´esultats appartenant à une sorte d´ej`a d´efinie. Le § D.6.1.5.3 explique la notion d’erreur en tant que terme d’une ´equation.

D.6.1.5.1 *Coh´erence de la hi´erarchie*

En un point quelconque d’une sp´ecification en LDS, il existe une seule et unique d´efinition de type de donn´ees. Cette d´efinition de type de donn´ees contient les sortes, op´erateurs et ´equations pr´ed´efinis et l’ensemble des sortes, op´erateurs et ´equations d´efinis par l’utilisateur dans les d´efinitions de type partielles visibles en ce point. (C’est la raison pour laquelle un texte NEWTYPE . | | ENDNEWTYPE est appel´e d´efinition de type *partielle* .)

Il en r´esulte certaines cons´equences pour les d´efinitions de type aux niveaux inf´erieurs. Cette influence sur le type pourrait | tre peu souhaitable. A titre d’exemple, on pourrait sp´ecifier de fa,con erron´ee que deux termes sont ´equivalents, les rendant ainsi ´equivalents alors qu’ils ne sont pas dans une port´ee englobante.

Il n’est pas admis de donner des ´equations telles que:

a) les valeurs d'une sorte qui sont diff´erentes dans une port´ee à un niveau sup´erieur soient rendues ´equivalentes;

b) de nouvelles valeurs soient ajout´ees à une sorte d´efinie dans une port´ee de niveau sup´erieur.

Cela signifie par exemple que, dans un bloc au niveau du syst`eme, des d´efinitions de type partielles sp´ecifi´ees par l'utilisateur contenant un op´erateur ayant un r´esultat pr´ed´efini doivent rapporter tous les termes produits par cet op´erateur à des valeurs de cette sorte de r´esultat.

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 sp´ecifier que si les r´esultats de l'op´erateur Fact sont les m | mes, les arguments sont alors les m | mes. (A noter que => est l'implication bool´eenne; cela a peu de relation avec le signe d'´equation conditionnellle ==>). Ainsi, par accident, les valeurs sont unifi´ees. Des ´equations des exemples pr´ec´edents on peut d´eduire que Fact(0) = Fact(1) et cette derni`ere ´equation indique que 0 et 1 sont des notations diff´erentes de la m | me valeur. Sur la base de cette derni`ere ´equation, on peut prouver que les nombres d'´el´ements de la sorte Integer sont r´eduits à un.

Avec l’aide d’une ´equation conditionnelle, on peut indiquer que, pourvu que n et m ne soient pas ´egaux à 0, le m | me r´esultat de l’op´erateur Fact sur n et m implique n = m. En Led:

FOR ALL n, m IN Integer

(n/=0, m/=0 ==> (Fact(n) = Fact(m)) => (n = m))

A noter que cette derni`ere ´equation n’ajoute rien à la s´emantique des nombres entiers; c’est un th´eor`eme qui peut | tre d´eduit des autres ´equations. Par ailleurs, l’adjonction d’une ´equation prouvable ne pr´esente pas d’inconv´enient.

b) Admettons que l’on d´ecouvre la n´ecessit´e d’un op´erateur pour les factorielles lors de la sp´ecification d’un type donn´e. Dans la d´efinition de type partielle de ce type, l’op´erateur Fact est introduit:

Fact: Integer -> Integer;

et les ´equations suivantes sont donn´ees pour d´efinir cet op´erateur:

Fact(0) == 1;

FOR ALL n IN Integer

(n > 0 ==> Fact(n) == n * Fact(n—1))

Ces ´equations ne d´efinissent 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 cons´equence, 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 d´efinition correcte de fact.

D.6.1.5.2 *Egalit´e et in´egalit´e*

Dans chaque type, les op´erateurs d’´egalit´e et d’in´egalit´e sont implicites. Ainsi, si une d´efinition de type partielle introduit une sorte S, il y a alors des d´efinitions implicites d’op´erateur:

"=": S, S -> Boolean;

"/=": S, S -> Boolean;

(*Remarque* — Les guillemets sp´ecifient que = et/= sont utilis´es comme op´erateurs infixes.)

L’op´erateur d’´egalit´e pr´esente les propri´et´es pr´evisibles:

a = a,

a = b => b = a,

a = b AND b = c => a = c,

a = b => a == b,

a = b => op(a) = op(b) for all operators op.

Ces propri´et´es ne sont pas ´ecrites en syntaxe LDS et ne doivent pas
| tre ´enonc´ees dans des axiomes ou des ´equations car elles sont implicites. La valeur bool´eenne obtenue lorsque cet op´erateur est appliqu´e est True (vrai) si les termes de la partie gauche et de la partie droite sont de m | me classe d’´equivalence; sinon la valeur obtenue est False (faux). S’il n’est pas explicitement sp´ecifi´e, que la valeur est True (vrai) ou False (faux), la sp´ecification est incompl`ete.

Pour l’op´erateur d’in´egalit´e, c’est par une ´equation en LDS que l’on peut le mieux expliquer la s´emantique:

FOR ALL a, b IN S

(a/= b == Not(a = b))

Il n’y a pas de diff´erence entre l’´egalit´e et l’´equivalence. Deux termes qui sont ´equivalents d´esignent la m | me valeur et l’op´erateur d’´egalit´e entre eux donne le r´esultat True (vrai).

D.6.1.5.3 *Erreur*

On a jug´e n´ecessaire, pour les exemples qui pr´ec`edent, de sp´ecifier que l’application de l’op´erateur à certaines valeurs est consid´er´ee comme une erreur. Le LDS a un moyen de le sp´ecifier formellement: l’´el´ement ERROR. L’erreur devrait servir à exprimer:

<<l’application de cet op´erateur à cette valeur n’est pas admise et lorsqu’il se pr´esente, le comportement futur est ind´efini>>.

Dans la syntaxe concr`ete, cela est indiqu´e par le terme Error!, qui ne peut | tre utilis´e comme argument d’un op´erateur.

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 donnée 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'emboîtage 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 ´equations ont une quantification implicite. La premi`ere ´equation indique que la suppression d'un ´el´ement de l'ensemble vide donne l'ensemble vide pour r´esultat. La seconde ´equation indique que la suppression apr`es insertion du m | me ´el´ement donne pour r´esultat l'ensemble tel qu'il ´etait avant l'insertion (a condition que l'ensemble ne contienne pas l'´el´ement); sinon l'ordre d'insertion et de suppression peut | tre interchangeable. La troisi`eme ´equation indique que l'´el´ement vide ne contient pas d'´el´ements. La quatri`eme ´equation indique qu'un ´el´ement se trouve dans un ensemble s'il est le dernier ´el´ement ajout´e ou s'il se trouvait dans l'ensemble avant l'adjonction du dernier ´el´ement. La derni`ere ´equation indique que l'ordre d'addition des ´el´ements ne fait aucune diff´erence.

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 ´egalement besoin d'un PId _set, d'un Subscriber _set (ensemble _d'abonn´es) et d'un Exchange _name _set (ensemble _de _noms _de _central) dans la sp´ecification, personne ne sera surpris qu'il contienne tous les op´erateurs Add (ajouter), Delete (supprimer) et Is _in (est _dans) et un litt´eral pour l'ensemble vide. Les ´equations donn´ees pour ces op´erateurs facilitent la g´en´eralisation à d'autres ensembles.

Tel est le cas où la notion de g´en´erateur se r´ev`ele utile; le texte commun peut | tre donn´e une fois et | tre utilis´e plusieurs fois. La figure D-6.2.2 pr´esente le g´en´erateur. (A noter que les noms de sortes formels sont introduits par le mot cl´e 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 diff´erents noms à l'ensemble entier vide et aux ensembles vides dans d'autres types, on fait ´egalement de ce litt´eral un param`etre formel.

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

Avec ce g´en´erateur, 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`etres formels de g´en´erateur (le texte entre parenth`eses apr`es le nom de g´en´erateur) sont supprim´es;

c) Set, Item et empty _set sont remplac´es dans tout le g´en´erateur par Int _set, Integer et empty _int _set, respectivement.

Ainsi, il n’y a en fait aucune diff´erence 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´eer le type à 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´esent´es par un type introduisant la sorte Subscr, l’ensemble d’abonn´es peut | tre cr´e´e à 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´en´erateur utilisant une sorte, un op´erateur, un litt´eral et une constante formels. Il d´ecrit une rang´ee d'´el´ements ayant une longueur maximale max _length. La sorte comprend un litt´eral d´esignant la rang´ee vide et les op´erateurs pour insertion et suppression d'´el´ements dans une rang´ee ou de celle-ci, l'encha | nement de rang´ees, le choix d'une sous-rang´ee et la d´eterminationde la longueur d'une rang´ee. Ce dernier op´erateur est rendu formel ce qui permet de la nommer à 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´erateur formel Length (longueur) et le litt´eral Empty (vide) sont donn´es une fois de plus dans le corps du g´en´erateur parce qu'ils sont renomm´es lors de leur instantiation. Dans le cas de l'op´erateur, les arguments et la sorte de r´esultat ne sont donn´es que dans le corps. Le g´en´erateur Row (rang´ee) peut servir à 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´eritage*

L'h´eritage constitue un moyen d'´etablir toutes les valeurs de la sorte dite parente, certains ou tous les op´erateurs du type parent et toutes les ´equations du type parent. Pour les litt´eraux et les op´erateurs, il existe une possibilit´e 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

  NEWTYPE Colour

devient

  GENERATOR Colour (TYPE Primary _colour)

2) le mot-clé ENDNEWTYPE 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:

```
NEWTYPE Paint _primary
  LITERALS Red, Yellow, Blue
ENDNEWTYPE Paint _primary;
```

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

```
NEWTYPE Light _colours Colour (Light _primary) ENDNEWTYPE;

NEWTYPE Paint _colours Colour (Paint _primary) ENDNEWTYPE;
```

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:

```
NEWTYPE Light
  INHERITS Light _colours
  OPERATORS (Beam=Take, Mix, Contains)
  ADDING
    LITERALS White
    AXIOMS
      White == Mix (Red, Mix (Yellow, Beam (Blue)))
ENDNEWTYPE 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 ajouté indique que ce littéral ajouté 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):

NEWTYPE Palette

INHERITS Paint _colours

OPERATORS (Paint _Take, Mix, Contains)

ENDNEWTYPE Palette;

b) Admettons que l’on veuille ´etendrel’ensemble de types entiers (sorte Int _set), introduit dans la section pr´ec´edente, par un op´erateur qui trouve le plus petit entier de l’ensemble. Tout d’abord, il faut se demander si cet op´erateur peut | tre introduit dans la d´efinition de g´en´erateur pour le rendre disponible à tous les ensembles et autres choses.

S’il est vrai que cela peut | tre fait, cela limiterait l’´el´ement à la d´efinition de > et <. Cela ne convient pas à tous les ´el´ements (PId par exemple) et il peut | tre pr´ef´erable de cr´eer un newtype ayant la sorte New _int _set comportant un op´erateur 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 donn´e que l’adjonction apr`es une instantiation de g´en´erateur est relativement courante, le texte commen,cant par ADDING et se terminant juste avant le ENDNEWTYPE peut | tre donn´e à l’int´erieur de l’instantiation de g´en´erateur. On en trouvera un exemple au § 5.4.1.12 de la Recommandation.

D.6.3 *Observations relatives aux ´equations*

Lorsque l’on introduit un nouveau type de donn´ees, il est essentiel d’introduire suffisamment d’´equations. Dans les paragraphes qui suivent, trois observations sont pr´esent´ees concernant les ´equations qui en faciliteront l’´etablissement.

D.6.3.1 *Conditions g´en´erales*

De quelque mani`ere que l’on construise les ´equations, il faut tenir compte des faits suivants:

a) chaque op´erateur appara | t au moins une fois dans l’ensemble des ´equations (sauf pour les cas <<patho-logiques>>);

b) tous les ´enonc´es vrais peuvent | tre tir´es des ´equations. Ils sont soit indiqu´es comme des axiomes, soit d´eduits par substitution de termes ´equivalents dans les ´equations;

c) aucune incoh´erence doit | tre d´ecel´ee, c’est-`a-dire que l’on ne peut d´eduire des ´equations que Vrai = Faux.

Une proc´edure permettant de trouver des ´equations peut | tre exprim´ee en LDS informel, comme indiqu´e dans la figure D-6.3.1.

D.6.3.2 *Application de fonctions aux constructeurs*

D’une mani`ere g´en´erale, l’ensemble d’op´erateurs poss`ede un sous-ensemble d’op´erateurs appel´es <<constructeurs>> et <<fonctions>>. Les constructeurs peuvent servir à g´en´erer toutes les valeurs (classes d’´equivalence) de la sorte. Dans cette approche, les litt´eraux sont consid´er´es comme des op´erateurs sans argument.

Exemples:

a) le type bool´een a ses litt´eraux pour constructeurs;

b) le type naturel a le litt´eral 0 et l’op´erateur Next comme constructeurs; un naturel quelconque peut |tre construit seulement avec 0 et Next;

c) le g´en´erateur pour les ensembles a le litt´eral ensemble _vide et l’op´erateur _additif comme constructeurs; un ensemble quelconque ne peut |tre construit qu’en utilisant Empty _set (ensemble _vide) et Add (ajouter);

d) le type entier peut |tre construit au moyen des litt´eraux 0 et 1, des op´erateurs + 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 pr´esente section mais les petits ensembles sont g´en´eralement les meilleurs.

Ensuite, les fonctions sont trait´ees une par une. Pour chaque argument d’une fonction, seuls sont ´enum´er´es tous les termes possibles compos´es de constructeurs. Pour ´eviter le probl`eme que posent les nombres infinis de termes, il faut appliquer la quantification.

Exemples:

a) Pour les nombres naturels, cette liste peut |tre r´eduite `a:

0

Next (n) oὺ n est tout nombre naturel.

b) Pour les ensembles, la liste ´eventuelle peut |tre:

empty _set

Add (s,i) oὺ s est tout ensemble et i tout ´el´ement (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

Add (empty _set,i) où i est un élément (item) quelconque,

Add (Add (s,i),j) où s est un ensemble quelconque et i, j, un élément quelconque.

Apr`es la cr´eation de cette liste, on obtient les parties de gauche des ´equations en appliquant chaque fonction à une combinaison d’arguments tir´es de la liste. Les identificateurs de valeurs de diff´erents arguments re,coivent des noms diff´erents. La proc´edure indiqu´ee ci-dessus pour les fonctions peut | tre appliqu´ee aux constructeurs; dans ce cas, elle donne les relations entre des termes où les constructeurs sont utilis´es dans diff´erents ordres.

Exemples:

a) Pour l’op´erateur de multiplication de nombres naturels portant la signature

<<*>>:Natural, Natural -> Natural

Cette proc´edure donne la partie de gauche des ´equations (incompl`etes) suivantes. L’utilisateur devra ajouter la partie de droite.

Next (n) 0*0 == . | | ;

Next (n) 0*Next (n) == . | | ;

Next (n)*0 == . | | ;

Next (n)*Next (m) == . | | ;

b) Pour les op´erateurs Is _in (est _dans) et Delete (supprimer, dans le g´en´erateur Set (voir le § D.6.2.1), cette approche est d´ej`a appliqu´ee.

c) Pour la sorte Colour, les constructeurs sont Take et Mix. Un op´erateur analogue à Contains dans le § D.6.1.4.2 doit | tre d´efini 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 donn´e que l’on avait annonc´e au § D.6.1.4.2 que les ´equations compl`etes seraient donn´ees pour cet op´erateur:

Contains(Take(p),q) ==p=q;

Contains(Mix(p,c),q) ==(p=q) OR Contains(c,q);

Cette proc´edure de construction peut donner plus d’´equations que cela n’est n´ecessaire, mais elle est tr`es s | re. Dans l’exemple susmentionn´e de multiplication des nombres naturels, il est vraisemblable que la propri´et´e de commutativit´e de la multiplication sera indiqu´ee et qu’en cons´equence, seule la derni`ere (ou la seconde) ´equation des trois premi`eres sera n´ecessaire.

La proc´edure d´ecrite dans cette section peut | tre appliqu´ee en combinaison avec la proc´edure d´ecrite dans la section pr´ec´edente, où elle est utile pour la t | che <<Penser _à _un _´enonc´e>>.

D.6.3.3 *Sp´ecification d’ensemble d’essai*

On peut aussi consid´erer les ´equations du point de vue de la mise en oeuvre. Si les op´erateurs sont mis en oeuvre sous la forme de fonctions dans un langage de programmation, les ´equations montrent comment ces fonctions doivent | tre test´ees.

Il convient d’´evaluer les expressions correspondant à la partie de gauche d’une ´equation, de faire de m | me pour la partie droite de cette ´equation et de voir si elles sont ´equivalentes. C’est la construction FOR ALL qui pourrait poser certains probl`emes. Ceux-ci peuvent souvent
| tre r´esolus de mani`ere pragmatique:

Au lieu de tif 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 proc´eder ainsi dans la plupart des cas.

Consid´erer des ´equations comme n´ecessaires à la mise en oeuvre peut | tre utile pour ce qui est de la t | che <<Penser _à _un _´enonc´e>> dans la proc´edure du § D.6.3.1.

D.6.4 *Caract´eristiques*

La pr´esente section d´ecrit certains dispositifs du LDS qui sont rarement n´ecessaires dont on peut pratiquement se passer, mais qui rendent quelquefois la t | che plus facile.

D.6.4.1 *Op´erateurs cach´es*

Il arrive que l'ensemble des ´equations puisse | tre simplifi´e ou rendu plus lisible gr | ce à l'introduction d'un op´erateur suppl´ementaire, mais cet op´erateur ne devrait pas | tre utilis´e dans les processus. Cela signifie que l'op´erateur est visible de l'int´erieur mais cach´e en dehors de la d´efinition de type.

On peut atteindre ce r´esultat en d´efinissant un <<type cach´e>>, c'est-`a-dire un type que l'utilisateur ne doit pas employer. A partir de ce type cach´e, l'utilisateur peut h´eriter de tous les op´erateurs auxquels il peut avoir acc`es; c'est le type h´erit´e qui doit | tre utilis´e. On peut s'assurer qu'il est correctement employ´e en examinant toutes les d´eclarations de variables (aucune variable de sorte introduite par le type cach´e ne doit appara | tre).

Ce qui caract´erise les op´erateurs cach´es, c'est qu'ils peuvent
| tre atteints par une restriction de leur visibilit´e aux seules ´equations. On peut le faire en pla,cant un point d'exclamation apr`es l'op´erateur.

Exemple:

La mani`ere courante de faire un ensemble d'un ´el´ement à partir du g´en´erateur Set est la suivante:

Add(empty _set,x)

et c'est ainsi que doit le faire chaque utilisateur. Dans les ´equations, le sp´ecificateur peut utiliser un op´erateur sp´ecial, par exemple:

Mk _set!:Item->Set;

d´efini par l'´equation:

Mk _set!(itm) == Add(empty _set,itm);

qui peut | tre utilis´e dans des d´efinitions de type partiel mais non dans le corps de processus en LDS.

D.6.4.2 *Relations d'ordre*

Lorsqu'il faut sp´ecifier une relation d'ordre sur les ´el´ements d'une sorte, cela signifie en g´en´eral qu'il faut d´efinir quatre op´erateurs (<,<=,>,>=) et les propri´et´es math´ematiques ordinaires (transitivit´e, etc.). S'il y a de nombreux litt´eraux, il faut ´egalement donner de nombreuses ´equations. Par exemple, le type de donn´ees pr´ed´efinies Character est d´efini de cette mani`ere.

Le LDS comporte une caract´eristique qui permet d'abr´eger ces d´efinitions de type longues, peu lisibles et ennuyeuses: c'est l'abr´eviation ORDERING (relation d'ordre).

ORDERING est donn´e dans la liste d'op´erateurs, de pr´ef´erence au d´ebut ou à la fin de celle-ci. Cela permet d'introduire des op´erateurs de relation d'ordre et les ´equations normales. Lorsque ORDERING est sp´ecifi´e, il faut donner les litt´eraux, 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 (H´eritage), on soulignait qu'il fallait s'assurer que les litt´eraux ´etaient ajout´es à une sorte h´erit´ee. Il convient d'en indiquer ici la raison.

Admettons que l’on d´esire l’extension suivante de la sorte Even _decimal _digit:

```
NEWTYPE 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
ENDNEWTYPE Decimal _digit;
```

Les axiomes donn´es ici ne peuvent | tre 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 serrait le suivant:

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

D.6.4.3 *Sortes avec champs*

Comme indiqu´e au § 5.4.1.10 de la Recommandation, on peut d´efinir une sorte structur´ee sans constructions sp´eciales, mais les sortes structur´ees sont à la fois courantes et utiles, ce qui justifie certaines constructions suppl´ementaires dans le langage.

Une sorte structur´ee devrait | tre utilis´ee lorsqu'une valeur d'objet est form´ee par l'association de valeurs de plusieurs sortes. Chaque valeur de cette association est caract´eris´ee par un nom, appel´e 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 structur´ee, certains op´erateurs sont d´efinis implicitement:

a) l'op´erateur constructeur, <<(.<<avant, et>>.)>> apr`es les valeurs de champ;

b) les op´erateurs de s´election de champ, les variables de la sorte structur´ee suivies par un ! et le nom de champ, ou suivies par le nom de champ entre parenth`eses. Il ne faut pas confondre la variable suivie d'un ! avec l'op´erateur cach´e (§ 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 index´ees*

Une sorte index´ee est une sorte pour laquelle le type a pour nom d'op´erateur Extract! (extraction). Dans les types de donn´ees pr´ed´efinies, le g´en´erateur Array est un tel type. Array est l'un des exemples les plus courants de type index´e.

Pour l'op´erateur cach´e Extract!, il existe une syntaxe concr`ete sp´eciale qui doit | tre appliqu´ee en dehors des d´efinitions de type.

On peut penser que le type Index dans le g´en´erateur pr´ed´efini Array doit | tre 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 | tre utilis´ee comme Index.

Exemple:

```
NEWTYPE Subsc _data _base
  Array (Name _key, Subscriber)
ENDNEWTYPE Subsc _data _base;
```

Les sortes Name _key et Subscriber sont celles qui ont ´et´e d´efinies dans la section pr´ec´edente. Supposons qu'il existe une proc´edure Bill comportant un param`etre de sorte Subscriber et que cette proc´edure soit d´efinie dans un processus qui comporte aussi une variable Sub _db de la sorte Subsc _data _base. Dans ce processus, l'appel suivant pourrait appara | tre.

CALL Bill (Sub _db) (. 'P.M.', 'Downingstreet'10. 'Londres'.)));

D.6.4.5 *Valeur par d´efaut de variables*

Comme indiqu´e dans la section concernant la d´eclaration de variables (§ D.3.10.1), il est possible d'affecter des valeurs à une variable imm´ediatement apr`es la d´eclaration.Cependant, certains types ont une valeur qui sera (presque) toujours la valeur initiale d'une variable. Il existe une caract´eristiquequi permet d'´eviter d'´ecrire la valeur initiale pour chaque d´eclaration: la clause DEFAULT.

A titre d'exemple, on peut consid´erer l'ensemble. Il est tr`es probable que presque toutes les variables, de tout ensemble imaginaire, seront initialis´ees avec empty _set.

La notation:

DEFAULT empty _set

apr`es la liste d'´equations indique que chaque variable de chaque instantiation de ce g´en´erateur sera initialis´ee 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 par s | r que la valeur initiale de toutes les variables d'une sorte soit la m | me, il ne faut pas utiliser la clause DEFAULT, sinon il est difficile d'´eviter des surprises.

D.6.4.6 *Op´erateurs actifs*

Les utilisateurs qui connaissent la Recommandation Z.104 de 1984 concernant le LDS pourraient se demander ce qui est arriv´e aux op´erateurs dits actifs. En fait, cette caract´eristique a ´et´e supprim´ee, pour les raisons suivantes:

a) elle n'est pas n´ecessaire car les op´erateurs courants et les proc´edures et/ou macros offrent la m | me capacit´e 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