UNION INTERNATIONALE DES TÉLÉCOMMUNICATIONS

UIT-T Z.100 – Annexe F.1

SECTEUR DE LA NORMALISATION
DES TÉLÉCOMMUNICATIONS
DE L'UIT

(03/93)

LANGAGES DE PROGRAMMATION

DÉFINITION FORMELLE DU LANGAGE DE DESCRIPTION ET DE SPÉCIFICATION

Recommandation UIT-T Z.100 – Annexe F.1

(Antérieurement «Recommandation du CCITT»)

AVANT-PROPOS

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

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

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

_____

NOTES

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

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

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

TABLE DES MATIÈRES

Page

1 Préface 1

2 Motivations 1

2.1 Le métalangage 1

3 Technique de modélisation 2

3.1 Sémantique statique 3

3.2 Sémantique dynamique 4

3.3 Exemple 5

3.4 Structure physique de la définition formelle 7

4 Comment utiliser la définition formelle 8

4.1 Les utilisateurs du SDL 8

4.2 Les réalisateurs 8

5 Introduction au Meta-IV 9

5.1 Structure générale 9

5.2 Définitions de fonctions 9

5.3 Définitions de variables 10

5.4 Domaines 11

5.4.1 Synonymes 12

5.4.2 Arborescences non nommées 13

5.4.3 Constructions de branchement 14

5.4.4 Domaines élémentaires 15

5.4.5 Domaines d'ensembles 18

5.4.6 Domaines de listes 19

5.4.7 Domaines de mise en correspondance 20

5.4.8 Domaines Pid 22

5.4.9 Domaines de référence 23

5.4.10 Domaines optionnels 24

5.5 Les constructions let et def 24

5.6 Quantification 25

5.7 Enoncés auxiliaires 26

5.8 Différences avec la notation utilisée dans la définition formelle du CHILL 27

5.9 Exemple: Spécification du «Demon game» en Meta-IV 27

Références 28

DÉFINITION FORMELLE DU LANGAGE DE DESCRIPTION ET DE SPÉCIFICATION

(Helsinki, 1993)

1 Préface

La présente définition formelle du SDL décrit une définition de langage qui complète celle donnée dans le texte de la Recommandation. La présente annexe est établie à l'intention de ceux qui ont besoin d'une définition très précise et détaillée du SDL, comme les responsables de la maintenance du langage SDL et les concepteurs des outils SDL.

La définition formelle est contenue dans trois volumes:

Annexe F.1 (présent volume)

Motivations, structure générale et modalités relatives à l'utilisation de la définition formelle; description de la notation utilisée.

Annexe F.2 Définition des propriétés statiques du SDL.

Annexe F.3 Définition des propriétés dynamiques du SDL.

2 Motivations

De manière générale, les langages naturels sont **ambigus** et **incomplets** en ce sens que certaines phrases peuvent donner lieu à plus d'une interprétation, que le lecteur soit une machine ou un homme.

Une définition ou une spécification est **formelle** quand sa signification (sémantique) est sans équivoque et complète. Etant donné que les langages naturels ne peuvent être utilisés à cette fin, des langages particuliers, dits **langages de spécification** (comme le SDL et LOTOS), ont été élaborés. Un langage de mise en œuvre comme le CHILL ou le PASCAL pourrait également servir de langage de spécification (un compilateur, par exemple, spécifie formellement la sémantique d'un autre langage), mais il est souvent indispensable de séparer les détails de la mise en œuvre, qui sont sans rapport pour la compréhension, de la sémantique d'une spécification.

Les langages formels particulièrement bien adaptés à la définition des langages sont connus sous le nom de métalangages. Ainsi, la Backus Naur form (BNF) est un métalangage qui convient tout particulièrement à la définition formelle de la syntaxe des langages de programmation.

Malgré leur ambiguïté, les langages naturels sont généralement plus faciles à lire, pour l'homme, que les langages formels et ils peuvent plus aisément exprimer un raisonnement en donnant un cadre dans lequel la spécification formelle peut être comprise. C'est pourquoi on présente souvent une définition en langage naturel en même temps qu'une définition en langage de spécification formelle.

La présente annexe constitue une définition formelle du SDL. Si l'on constatait une incohérence entre des propriétés quelconques d'une notion du SDL, telles qu'elles sont définies dans la présente Recommandation, et la Recommandation Z.100, et si cette notion se trouve définie de façon cohérente dans la présente Recommandation, cette dernière prévaudra et il faudra corriger la présente définition formelle.

2.1 Le métalangage

Le métalangage utilisé dans la présente définition formelle est le Meta-IV [1]. Ce langage a été choisi pour les raisons suivantes:

- il s'appuie sur une théorie mathématique très solide qui a fait l'objet de recherches approfondies;
- il offre des moyens très pratiques et puissants pour les manipulations d'objets;
- il comporte une notation «de type programmation», ce qui signifie qu'il s'adresse à des programmeurs et à des réalisateurs;
- il est en passe d'être normalisé au sein de la Communauté européenne;

– il est largement cité dans des livres, des rapports et des revues scientifiques et il a été utilisé dans le manuel du CCITT consacré à la définition formelle du CHILL [2], qui contient aussi un résumé de la notation en Meta-IV;

– on dispose d'outils pour le Meta-IV qui permettent la vérification de la syntaxe, l'analyse de la visibilité, la production de documents, les renvois, etc.

On trouvera en 5 une introduction informelle aux chapitres du Meta-IV utilisés dans la définition formelle. Une définition complète est donnée dans l'ouvrage [1].

3 Technique de modélisation

Si l'on cherche à définir le sens de «sémantique du SDL», il convient (en théorie) de scinder la définition du langage en plusieurs parties:

– définition des règles syntaxiques;

– définition des règles de sémantique statique (dites conditions de bonne formation: quels noms, par exemple, est-il permis d'utiliser à un endroit donné, quels types de valeurs est-il permis d'affecter à des variables, etc.;

– définition de la sémantique des constructions du langage au moment de l'interprétation (sémantique dynamique).

Il est inutile d'incorporer les règles syntaxiques dans la définition formelle puisque les règles et les diagrammes de syntaxe BNF figurant dans la présente Recommandation servent déjà de définitions formelles des règles syntaxiques; en d'autres termes, les contributions à la définition formelle constituent une spécification du SDL syntaxiquement correcte. Ces contributions sont représentées par une syntaxe abstraite fondée sur l'arborescence de la syntaxe textuelle concrète (règles de la BNF), des détails inutiles (par exemple, les séparateurs) et des règles lexicales étant supprimés. En conséquence, la présente syntaxe abstraite n'est pas celle des Recommandations de la série Z.100, qui elle, constitue une abstraction du concept de modèle SDL.

Ainsi, la règle de production de la syntaxe abstraite:

1 *Transition* :: *Actstmt** [*Termstmt*]

signifie qu'une *Transition* est composée d'une liste non vide d'*Act*ion *stat*ements et d'un *Term*inator *stat*ement facultatif (on trouve aussi les caractères en italique dans la règle de production). L'ensemble complet des règles de production (dites définitions de domaines) qui définissent la syntaxe SDL sous une forme abstraite s'appelle AS_0. Dans une certaine mesure, ces règles définissent la syntaxe du langage de façon plus élémentaire que les règles syntaxiques décrites dans la présente Recommandation, étant donné que la syntaxe textuelle concrète définie dans cette dernière donne de nombreux renseignements sémantiques (le contexte y est important), contraitement à AS_0. Il est à noter que AS_0 est une abstraction de la syntaxe textuelle concrète. La syntaxe graphique concrète n'a pas été utilisée pour des raisons d'économie de temps et de place, et non en raison d'éventuelles difficultés dans l'exécution des travaux.

Dans la présente Recommandation, par exemple, une liste de signaux est définie comme suit:

<signal list> ::= <signal list item> {, <signal list item>}
<signal list item> ::= <signal identifier> | (<signal list identifier>) | <timer identifier>

alors que les définitions correspondantes de AS_0 sont les suivantes:

2 *Signallist* :: *Signallistitem*$^+$
3 *Signallistitem* = *Id* | *Signallistid*

Une *Signallist* est une liste de *Signallistitems*. Un *Signallistitem* est soit un identificateur, soit un identificateur de liste de signaux. Contrairement à la production BNF dépendant du contexte <éléments de signaux>, AS_0 n'établit aucune distinction entre un identificateur de signaux et un identificateur de temporisateur car tous deux sont des identificateurs, du point de vue syntaxique, par opposition aux listes de signaux qui se différencient par l'utilisation de parenthèses.

Des spécifications SDL syntaxiquement correctes constituent le point de départ de la définition formelle (FD). Les objectifs de la définition formelle sont les suivants:

– définition des conditions de bonne formation pour les spécifications SDL. Cette question fait l'objet de l'Annexe F.2 (Sémantique statique);

– définition des propriétés dynamiques pour les spécifications SDL. Cette question fait l'objet de l'Annexe F.3 (Sémantique dynamique).

Les étapes sont décrites à la Figure 1 et le résultat des travaux sur la sémantique statique (à savoir AS_1) est exposé ci-après:

La transposition de la syntaxe textuelle concrète AS_0 n'est pas définie formellement, mais elle est dérivée de la correspondance des noms dans les deux syntaxes, comme indiqué précédemment pour Signallist.

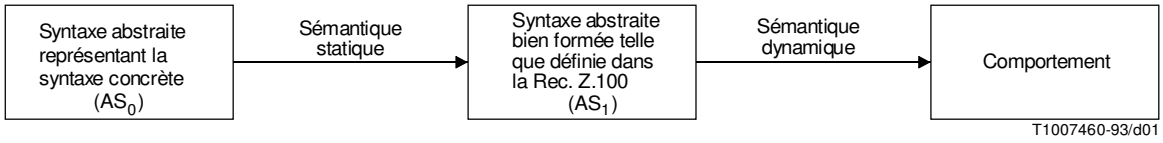

FIGURE 1/Z.100

Objectifs de la sémantique statique et de la sémantique dynamique

3.1 Sémantique statique

La présente Recommandation définit la sémantique dynamique des différentes constructions en termes de syntaxe abstraite. Des paragraphes communs (*Sémantique, Grammaire textuelle concrète, Grammaire graphique concrète et Modèle*) définissent les règles syntaxiques concrètes, indiquent les conditions adéquates de bonne formation et relient les règles syntaxiques concrètes aux autres règles de syntaxe concrète (dans le *Modèle*) et à la grammaire abstraite définies dans la présente Recommandation. La définition de la grammaire abstraite dans la présente Recommandation utilise le Meta-IV (dans les paragraphes communs relatifs à la Grammaire abstraite). La même syntaxe abstraite est utilisée dans la définition formelle, où elle est appelée AS_1. La syntaxe AS_1 est donnée dans l'Annexe F.3; elle est identique à la syntaxe abstraite de la grammaire abstraite définie dans la présente Recommandation.

Après avoir défini les conditions de bonne formation, la sémantique statique doit donc déterminer comment la représentation AS_0 d'une spécification est transposée en représentation AS_1, c'est-à-dire que, dans le cas d'une représentation AS_0, une représentation AS_1 est renvoyée par la sémantique statique si la représentation AS_0 est bien formée. On peut dire que la sémantique statique est «un compilateur abstrait» dans lequel la représentation AS_0 est le langage source et la représentation AS_1 le langage objet.

Outre AS_0 et AS_1, la sémantique statique utilise certains domaines de service internes dits domaines sémantiques, qui conservent l'information demandée à n'importe quel endroit sur une entité donnée. Par exemple, lorsqu'une définition de processus est transformée, l'information relative à ses paramètres formels est conservée dans les domaines sémantiques et peut être extraite lors de la transformation de l'action de demande de création. On aurait pu utiliser à cette fin les domaines AS_0, étant donné que les domaines sémantiques sont de toute façon déduits de AS_0, mais la représentation arborescente est inutile quand il est nécessaire d'obtenir l'information d'une certaine entité (par exemple, une définition de processus) apparaissent à un endroit quelconque de l'arborescence. C'est pourquoi les domaines sémantiques sont généralement des tables de modélisation de mise en correspondance.

Les domaines sémantiques contiennent, par exemple, une mise en correspondance (expliquée plus avant dans 5.4.7) d'identificateurs dans un descripteur renfermant des informations sur les identificateurs:

4 *Descriptordict* = *Qual* $\overrightarrow{\mathbb{m}}$ *Descr*

où *Qual* est la représentation de l'identificateur utilisée au niveau interne dans la définition formelle et *Descr* un descripteur quelconque. Le descripteur peut être, par exemple, un descripteur de processus:

5 *Descr* = *ProcessD* | …

6 *ProcessD* :: *ParameterD* Validinputset Localinputset*

Cela signifie qu'un *Process Des*criptor contient une liste de *Parameter Des*criptors, des informations relatives à l'ensemble de signaux *Valid input set* ainsi qu'à l'ensemble *Local input* signal *set*. Les définitions de ces trois (sous-)descripteurs ne sont pas données ici.

La transformation proprement dite s'effectue au moyen d'un ensemble de fonctions Meta-IV utilisant les trois domaines AS_0, AS_1 et les domaines sémantiques.

3.2 Sémantique dynamique

La sémantique dynamique a pour objet de définir le comportement d'une spécification SDL sous la forme AS_1.

La sémantique dynamique se subdivise en trois grandes parties:

- modèle applicable au système sous-jacent (la machine SDL abstraite);
- interprétation des graphes de processus;
- transformation de AS_1 en une représentation plus adéquate, c'est-à-dire qu'on construit une mise en correspondance (un domaine sémantique) contenant l'information demandée pendant l'interprétation, à savoir, par exemple, la sorte de variable, les trajets de communication éventuels entre processus, les catégories d'équivalence pour les types, etc. La mise en correspondance ou, pour être plus exact, le domaine de la mise en correspondance, s'appelle *Entity-dict*.

La simultanéité dans les sémantiques dynamiques du SDL est modélisée en utilisant des **métaprocessus**; c'est obtenu en exécutant simultanément des métaprocessus dans le modèle Meta-IV et des processus SDL.

On a recours à six types différents de métaprocessus:

- *system*

  Traite l'acheminement du signal entre les ensembles d'instances de processus SDL et la génération de valeurs uniques de PId.

- *path*

  Traite le retard non déterministe des canaux (à retard).

- *view*

  Mémorise toutes les variables révélées.

- *timer*

  Mémorise le temps actuel et gère les débordements de temporisation.

- *process-set-admin*

  Traite tous les signaux entrants et établit les demandes et gère les autres métaprocessus nécessaires à l'interprétation de l'ensemble des instances d'un processus SDL. Pour chaque ensemble d'instances d'un processus SDL, il y a exactement une instance de *processus-set-admin*.

- *input-port*

  Traite la mise en file d'attente des signaux dans une instance de processus SDL. Pour chaque instance de processus SDL, il y a exactement une instance *d'input-port*.

- *sdl-process*

  Interprète le comportement du corps d'une instance de processus SDL. Pour chaque instance de processus SDL, il y a exactement une instance de *sdl-process*.

  Si le processus SDL est décomposé en services, *sdl-process* gère les métaprocessus nécessaires à l'interprétation des services.

- *sdl-service*

  Interprète le comportement (du corps) d'un service SDL. Pour chaque instance de service SDL, il y a exactement une instance de *sdl-service*.

Dans la plupart des cas, il n'existe pas de données partagées entre métaprocessus; ils interagissent en transmettant des valeurs acheminées par des instances (objets) de **domaines de communication** (correspondant au concept de signaux SDL). L'unique exception réside dans les instances de *sdl-services* qui peuvent accéder et modifier les variables Meta-IV appartenant à leur instance de *sdl-process* gérant.

Les domaines de communication sont définis de la même manière que les autres domaines; ainsi, des objets du domaine de communication *Input-Signal* sont dirigés vers une instance *sdl-process* depuis son instance associée *input-port*. Le domaine de communication est défini comme suit:

7 *Input-Signal* :: $\textit{Signal-Identifier}_1$ *Value-List Sender-Value*

Les instances de l'*Input-Signal* véhiculent l'identificateur du signal SDL qui est envoyé, la liste des valeurs transmises par le signal SDL ainsi que la valeur PId de l'expéditeur.

Le «système d'interaction du métaprocessus» est représenté dans son intégralité aux Figures 2 et 3. Les métaprocessus nécessaires pour l'interprétation d'un ensemble d'instances de processus SDL sont indiqués sur la Figure 2 en tant que *sdl-process-set* et leur détail est donné à la Figure 3. Le mécanisme de communication est synchronisé et la notation est connue comme CSP (voir [3] et [4]) (Communicating Sequential Processes).

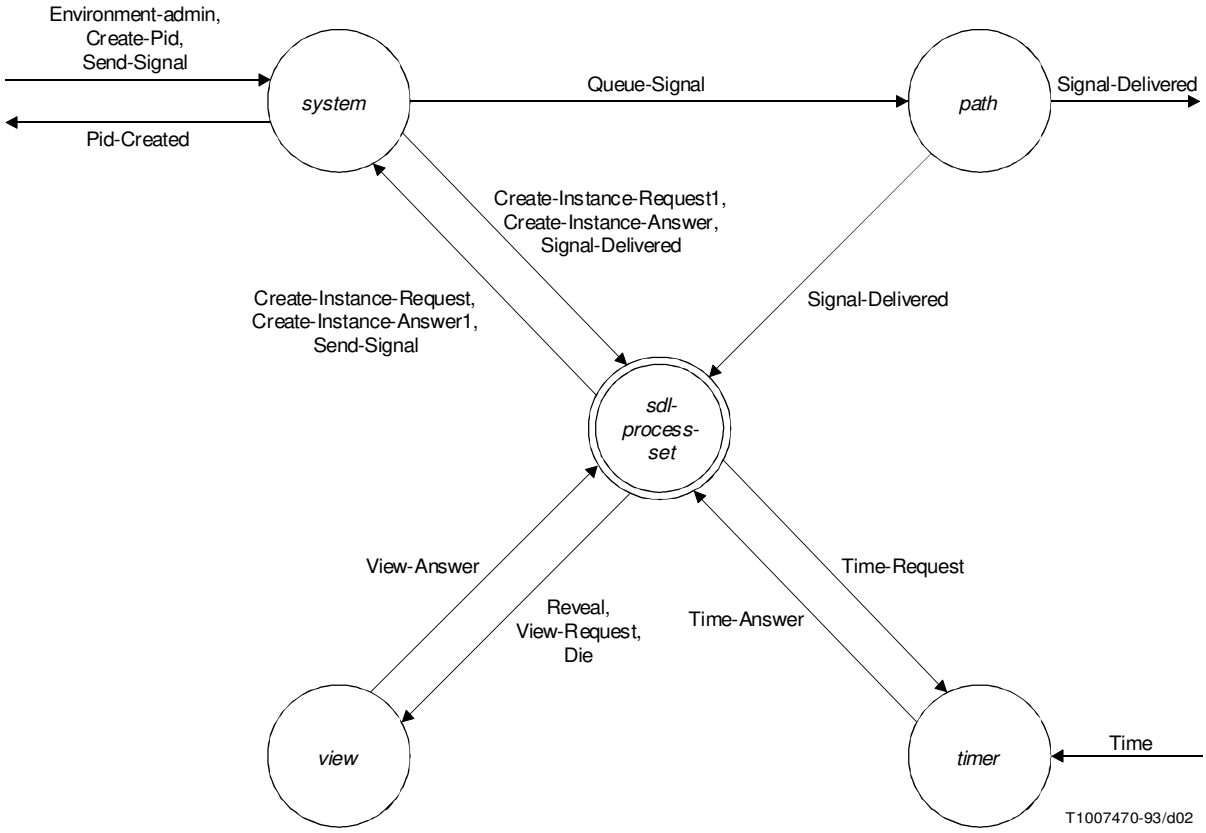

FIGURE 2/Z.100

Schéma de communication global

3.3 *Exemple*

La Figure 4 illustre la communication entre métaprocessus dans la définition formelle pour le processus-SDL (partiel) décrit ci-dessous, lorsqu'un signal («b») arrive de l'environnement et que le processus réagit en renvoyant un signal («a») à l'environnement.

```
…
state S;
    input b;
    output a;
…
```

La communication est illustrée de manière informelle au moyen d'un graphique de séquence de messages. Path(1) et Path(2) représentent deux instances du processus-path, correspondant au trajet de l'environnement à l'ensemble d'instances de processus-SDL [Path(1)] et inversement [Path(2)].

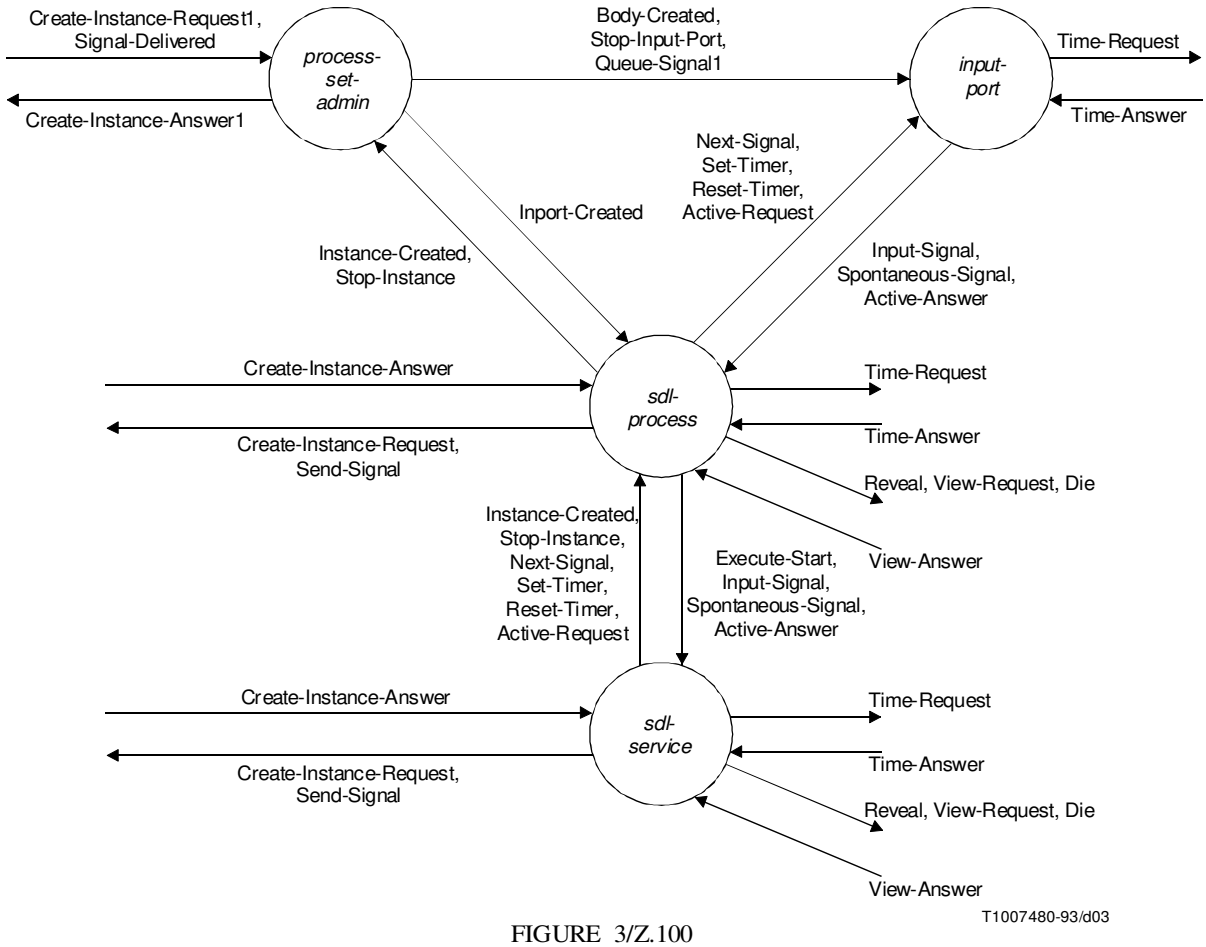

FIGURE 3/Z.100

Schéma de communication pour l'ensemble d'instances de processus LDS

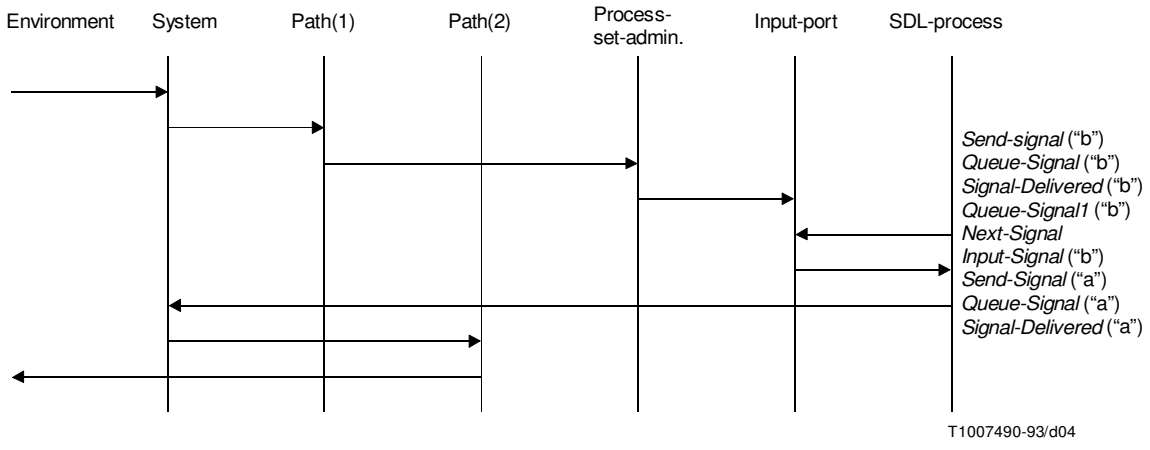

FIGURE 4/Z.100

Exemple de communication entre métaprocessus

3.4 Structure physique de la définition formelle

La sémantique statique (Annexe F.2) se subdivise en trois grandes parties:

1) Définition de domaines pour AS_0.
2) Définition de domaines pour les domaines sémantiques.
3) Fonctions du Meta-IV consistant à vérifier les conditions de bonne formation et à déterminer comment AS_0 est transformé en AS_1.

L'Annexe F.2 contient également des renvois aux noms de fonctions et aux noms de domaines du Meta-IV (tous deux définissant l'occurrence et les occurrences appliquées), ainsi qu'un renvoi à l'application des conditions de bonne formation.

La sémantique dynamique (Annexe F.3) se subdivise en cinq paragraphes principaux:

1) Résumé des domaines de la syntaxe abstraite (AS_1).
2) Définitions de domaines pour les domaines de communication.
3) Définitions de domaines pour les domaines sémantiques (*Entity-dict*).
4) Définitions de métaprocessus et fonctions associées pour le modèle du système sous-jacent.
5) Définitions du métaprocessus et fonctions associées pour l'interprétation du processus-SDL et du service SDL.
6) Création du domaine interne *Entity-dict*. *Entity-dict* étant utilisé par les processus-SDL, sa création précède donc l'interprétation de processus-SDL et du service SDL.
7) Quelques fonctions auxiliaires simples d'utilisation générale relatives aux domaines de la syntaxe abstraite.

Comme l'Annexe F.2, l'Annexe F.3 contient un certain nombre d'index relatifs aux noms de domaines, aux noms de fonctions, aux noms de métaprocessus, aux conditions d'erreur, etc.

Au premier abord, le nombre de pages (notamment dans l'Annexe F.2) pourra paraître impressionnant. Toutefois, plus de la moitié du document contient des annotations relatives aux définitions de domaines, de fonctions et de processus.

La définition d'une fonction et d'un processus est présentée comme suit:

1) La définition de la fonction ou du processus est tout d'abord spécifiée par:
   a) un en-tête qui définit le nom du processus ou de la fonction et les noms de ses paramètres formels;
   b) son corps (algorithme);
   c) une *clause de type* spécifiant le type (domaine) des paramètres formels et celui du résultat (le cas échéant).
2) Viennent ensuite les annotations classées par éléments (en anglais courant) qui sont associées à la définition du processus ou de la fonction:

Objective	Explique le but de la fonction ou du processus;
Parameters	Explique le but de chaque paramètre formel de la fonction ou du processus;
Result	Explique l'objet renvoyé (s'il en existe);
Algorithm	Explique, ligne par ligne, l'algorithme utilisé dans la fonction ou dans le processus.

Exemple:

La fonction la plus à l'extérieur *definition-of-SDL* de l'Annexe F.2, qui relie la sémantique statique (*transform-system*) et la sémantique dynamique (en déclenchant le métaprocessus *system*), est la suivante:

definition-of-SDL(extparms, systemtext, predeftext) ≜

```
1  (let (systemdef, predef) = construct-AS0 (systemtext, predeftext) in
2  let (as1, auxinf) = transform-system (systemdef, predefsorts, extparms) in
3  if as1 = nil then
4    undefined
5   else
6    (let subsetcut = select-consistent-subset (as1, extparms) in
7     start system (as1, subsetcut, auxinf)))
```

type: *External-Information* Sys_0 $Datadef_0^+$ ⇒

Objective Définit les propriétés du SDL.

Parameters

extparms Certains *External-Information* (voir 2.3/Annexe F.2).

systemtext La séquence d'éléments lexicaux (c'est-à-dire les caractères) représentant la spécification SDL.

predeftext La séquence d'éléments lexicaux (c'est-à-dire les caractères) représentant les données prédéfinies.

Algorithm

Ligne 1 Transforme le texte de la spécification SDL (*systemtext*) et le texte relatif aux données prédéfinies (*prédeftext*) en leurs représentations AS_0.

Ligne 2 Transpose le système en syntaxe abstraite (sous la forme AS_1).

Ligne 3 Si des erreurs statiques apparaissent (c'est-à-dire si aucune représentation AS_1 n'a pu être dérivée), le comportement n'est pas défini.

Ligne 4 Si aucune erreur statique n'apparaît à ce moment.

Ligne 6 Choisit l'ensemble de *Block-identifier*$_1$s désignant le sous-ensemble cohérent.

Ligne 7 Crée une instance de système, à savoir un processus Meta-IV ayant le même comportement que celui du système sous-jacent.

4 Comment utiliser la définition formelle

4.1 Les utilisateurs du SDL

La définition formelle n'est pas destinée à servir de manuel de référence sur le SDL à l'intention des utilisateurs. Les nouveaux utilisateurs du SDL pourront trouver dans les Directives pour les usagers un aperçu des concepts du langage qui leur convient (ainsi que leur explication), la présente Recommandation proprement dite servant de manuel de référence sur le SDL, mais cette dernière pourrait se révéler insuffisante dans certains cas. Par exemple:

– s'il manque certaines propriétés (comme des conditions statiques prévues), ou si des propriétés énoncées en contredisent d'autres, ou

– si la signification exacte de certaines propriétés énoncées est difficile à comprendre, ou

– si certaines propriétés sont difficiles à trouver (en raison de l'absence de renvois dans la présente Recommandation), ou

– si l'utilisateur veut approfondir ses connaissances sur des questions plus complexes comme la machine SDL abstraite, la résolution par le contexte, le mécanisme d'héritage, la question de savoir quand et comment choisir un sous-ensemble cohérent, etc.

En pareils cas, la définition formelle pourra être un document d'appui utile. Il va de soi que l'utilisateur doit tout d'abord se familiariser avec la structure de la définition formelle, comprendre comment sont organisées les fonctions et quelles sont les utilisations des domaines. En outre, il est nécessaire de posséder un certain nombre de connaissances sur la notation en Meta-IV, mais dans la mesure où les fonctions sont largement annotées, on pourra éventuellement lire le Meta-IV après avoir pris connaissance de l'introduction relative (voir 5 ci-dessous), en lisant les fonctions en parallèle avec les annotations. Les utilisateurs qui consultent la définition formelle pourront avoir intérêt à utiliser la table des matières et les renvois.

4.2 Les réalisateurs

Comme indiqué précédemment, la méthode Meta-IV permet aux réalisateurs de trouver automatiquement une mise en œuvre (par exemple, un analyseur statique, un simulateur, etc.) d'après la spécification Meta-IV. Pour ce qui est du SDL, il est possible de dériver un analyseur statique de l'annexe F.2 et un simulateur de l'Annexe F.3. Il est conseillé d'utiliser la représentation AS_1 (engendrée par l'analyseur statique) comme base de la simulation, car il n'y a pas d'informations de contexte pour les identificateurs en AS_0 (ils ne sont généralement pas qualifiés sous cette forme), et il peut être difficile de dériver la sémantique dynamique d'une spécification en forme AS_0 en raison des nombreuses abréviations du SDL (en particulier pour des concepts comme les types de données).

Il faut signaler que la dérivation vers une mise en œuvre est systématique, mais pas mécanique.

Il convient d'examiner les points suivants:

– Il faut trouver des types de données appropriés pour représenter les types de données idéaux (domaines) en Meta-IV (par exemple, mises en correspondance, listes et ensembles utilisés en AS_0, en AS_1 ainsi que dans les domaines sémantiques).

– La méthode algébrique initiale sous-entend que la définition formelle manipule des objets infinis. AS_1 contient également des objets infiniment grands. Il est donc nécessaire de modifier légèrement et de restreindre l'utilisation des types de données, ou encore d'employer une technique d'abstraction permettant de coder ces objets.

5 Introduction au Meta-IV

Le présent article contient une introduction informelle au Meta-IV et à la méthode selon laquelle le Meta-IV a été utilisé dans la définition formelle, c'est-à-dire que le Meta-IV est expliqué selon les termes de la définition formelle (dont l'abréviation est FD), et qu'en conséquence seules sont expliquées les parties du Meta-IV ayant été utilisées dans la FD.

5.1 Structure générale

La FD se compose:

– D'un ensemble de définitions de fonction et de processus définissant la sémantique du SDL. Les processus (dits processeurs dans le Meta-IV et dans la FD) servent à modéliser la simultanéité et ne sont donc utilisés que dans la sémantique dynamique. Les définitions de processeurs du point de vue syntaxique, ressemblent aux définitions de fonction (exception faite du mot clé processor qui suit le nom du processeur); la description du concept de fonction que l'on trouvera ci-après s'applique donc aussi aux processeurs.

– D'un ensemble de définitions de domaines définissant le type des objets manipulé par les fonctions. Des termes désignant certains groupes de définitions de domaine sont introduits afin d'être classés en un ordre logique. Les domaines AS_0 désignent la représentation de la syntaxe concrète, les domaines AS_1 décrivent la syntaxe abstraite du SDL et l'ensemble des domaines *Dict* et *Entity-dict* se rapportent aux «domaines de service» internes (domaines sémantiques) de la sémantique dynamique et de la sémantique statique respectivement. Dans le présent paragraphe, nous emploierons souvent le mot «valeur» comme synonyme d'objet et le mot «type» comme synonyme de domaine.

Les définitions peuvent être spécifiées dans n'importe quel ordre et les noms qui y sont introduits peuvent être utilisés avant d'être définis textuellement.

5.2 Définitions de fonctions

Une définition de fonction comprend trois parties:

1) L'en-tête, commençant par le nom de la fonction et suivi d'une ou deux listes de paramètres formels, chacune placée entre parenthèses. La division des paramètres en deux listes n'a pas de signification formelle. Souvent certains paramètres sont indiqués dans une liste de paramètres distincte (deuxième) s'ils ne sont pas indispensables à l'évaluation; c'est le cas, par exemple, des domaines sémantiques qui sont fréquemment utilisés par les fonctions et qui sont insérés dans une liste de paramètres distincte.

2) Le corps de la fonction, qui peut être soit une expression soit une suite d'énoncés. Il n'est pas nécessaire qu'une fonction produise un résultat quelconque (voir ci-dessous).

3) La clause de type, qui spécifie le type des paramètres formels et le type du résultat. Le type de la première liste de paramètres est tout d'abord spécifié, vient ensuite le type de la deuxième liste de paramètres (s'il y en a une) séparée de la première par une flèche ($\rightarrow$ ou $\Rightarrow$), puis une autre flèche et enfin le résultat.

Exemple:

f(a, b) (d) $\triangleq$

1 /∗ *expression* ∗/

type: *DomX DomY* $\rightarrow$ *DomZ* $\rightarrow$ *DomW*

Dans cet exemple:

f désigne le nom de la fonction,

a, *b*, *d* sont des paramètres formels. *a* et *b* figurent dans la première liste de paramètres formels et *d* dans la deuxième liste. Le type de *a* est *DomX*, celui de *b* est *DomY* et celui de *d* est *DomZ*. Le type du résultat est *DomW*. Les domaines *DomX*, *DomY*, *DomZ* et *DomW* doivent être définis dans certaines définitions de domaines.

Si les paramètres formels ou le résultat ne sont pas utilisés conformément à la clause de type, il y aura une erreur dans la spécification Meta-IV. Dans l'exemple ci-dessus, le texte informel Meta-IV (placé entre les signes /**/) est utilisé pour décrire une certaine expression Meta-IV qui n'a pas été mentionné pour des raisons d'économie de place. Le texte informel Meta-IV est analogue au SDL et est largement utilisé dans les exemples du présent article.

On fait normalement une distinction entre les fonctions **applicatives** et les fonctions **impératives**. Les premières sont des fonctions qui ne se rapportent pas à des parties de l'état global (variables), c'est-à-dire que leur résultat dépend seulement de la valeur des paramètres effectifs appliqués. Le corps d'une fonction applicable est restreint de manière à être une expression, car les énoncés imposent un certain changement d'état. Ces fonctions doivent toujours fournir un résultat. Les fonctions impératives sont des fonctions qui concernent ou même qui modifient l'état global (fonctions avec effets de bord). Si une fonction est impérative, elle doit être reflétée dans la clause de type au moyen de $\rightarrow$, et non de $\Rightarrow$, au moment de la spécification du résultat. Ainsi,

f(a, b) (d) $\triangleq$

1 */∗ expression referring to the global state or sequence of statements ∗/*

type: *DomX DomY* $\rightarrow$ *DomZ* $\Rightarrow$ *DomW*

Dans la FD, la sémantique statique et la création du domaine interne *Entity-dict* de la sémantique dynamique sont des fonctions applicatives.

5.3 Définitions de variables

Les variables globales sont définies au niveau le plus à l'extérieur dans les définitions de processeur. Elles sont visibles pour toutes les fonctions qu'utilise le processeur afin de définir la variable, même si les fonctions sont normalement définies en dehors des définitions de processeur. Cependant, une fonction utilisée par un ou plusieurs processeurs ne peut avoir accès à des variables. Quand il existe plusieurs instances d'un processeur donné, il existe aussi plusieurs instances de variables définies par le processeur (il n'y a pas de variables partagées).

Les définitions de variables sont introduites au moyen de la spécification du mot clé **dc1** suivi d'une liste de noms de variables et, au besoin, d'une expression initiale terminée par le type de la variable.

Exemple:

dcl v1 := 5 **type** *Intg*;
dcl v2 **type** *DomD*;

Nous avons défini ici deux variables v1 et v2; v1 est du type entier et initialisé à 5; v2 est du type *DomD*. Notons que des variables peuvent toujours être distinguées d'autres noms, du point de vue syntaxique, car elles n'apparaissent pas en italique. Il existe une autre syntaxe possible de définition de variables:

dcl v1 := 5 **type** *Intg*;
v2 **type** *DomD*;

On a accès à la valeur associée aux variables au moyen de l'opérateur de contenu qui est le mot clé **c**.

Exemple:

f ( ) $\triangleq$

1 cv1 + cv2

type: ( ) $\Rightarrow$ *Intg*

5.4 Domaines

On définit généralement les domaines au début d'un document. Il est possible de distinguer syntaxiquement des noms de domaines d'autres noms, étant donné que la première lettre est une majuscule. Un domaine se définit par la spécification du nom du domaine suivi du symbole «::» (ou de «=» dans le cas d'un nom de synonyme, comme indiqué dans 5.4.1), puis d'une expression de domaine indiquant ses propriétés (voir 1.5.1 à titre d'introduction à la notation des domaines).

Exemple:

8 *Output-node*$_1$ :: *Signal-identifier*$_1$
[Expression$_1$]*
[*Signal-destination*$_1$]
Direct-via$_1$

Cet exemple est tiré de la syntaxe abstraite du SDL (par souci de clarté, tous les noms de AS$_1$ ont le suffixe «$_1$» dans la FD. Il définit une **arborescence nommée**, à savoir un type de données analogue à un enregistrement où le nom du type d'enregistrement est Output-node$_1$ et où ses champs sont du type *Signal-identifier*$_1$, [*Expression*$_1$]*, [*Signal-destination*$_1$] et *Direct-via*$_1$.

Pour les arborescences nommées, l'opérateur le plus important est le **mk**- (make) qui sert à composer et à décomposer des objets d'arborescence (c'est-à-dire des valeurs d'enregistrement).

Ainsi, si un nom *sigid* désigne un objet du domaine *Signal-identifier*$_1$, un nom *exprlist* désigne un objet du domaine [*Expression*$_1$]*, un nom *dest* désigne un objet du type [*Signal-destination*$_1$] et un nom *via* désigne un objet du domaine *Direct-via*$_1$; un objet du domaine *Output-node*$_1$ est alors construit en écrivant:

mk-*Output-node*$_1$ (*sigid, exprlist, dest, via*)

qui peut être utilisé dans l'expression Meta-IV. Signalons que l'ordre dans lequel les arguments sont spécifiés dans l'opérateur **mk-** est important. Cela s'applique également aux appels de fonction.

De la même façon, si nous avons un objet appelé *outputnode* de domaine *Output-node*$_1$, et si nous voulons avoir accès aux champs, nous pouvons donner des noms aux champs en décomposant cet objet (on a choisi ici les mêmes noms que ceux indiqués ci-dessus):

let mk-*Output-node*$_1$ (*sigid, exprlist, dest, via*) = *outputnode* **in**
/∗ some expression using the fields ∗/

Au moyen de la construction **let**, nous avons introduit des noms pour indiquer les champs dans l'objet *outputnode*. C'est cette construction qu'on utilise généralement pour introduire des noms d'objets (et non uniquement en combinaison avec l'opérateur **mk-**). La construction **let** est expliquée plus avant dans 5.5.

Si certains champs ne sont pas utilisés dans l'expression, nous pouvons omettre les noms correspondants dans la décomposition. Par exemple, si *sigid* n'est pas utilisé dans l'expression, nous pouvons écrire:

let mk-*Output-node*$_1$ (,*exprlist*, *dest*, *via*) = *outputnode* **in**
/ some expression using exprlist and dest */*

Si nous voulons uniquement utiliser dans l'expression le *Signal-Identifier*$_1$, nous pouvons utiliser la variante de l'opérateur de séléction du champ **s-**:

let *sigid* = **s-***Signal-Identifier*$_1$ (*outputnode*) **in**
/ some expression using sigid */*

L'opérateur de sélection du champ ne peut être utilisé que si le champ peut être déterminé de manière univoque par l'indication du nom du domaine.

Nous pouvons choisir de décomposer (c'est-à-dire d'introduire des noms pour les éléments contenus) les paramètres formels dans l'en-tête de la fonction, et non dans le corps, si nous estimons que la lecture s'en trouve facilitée. Ainsi:

int-create-node (**mk**-*Create-request-node*$_1$ (*prid, exprl*))(*dict*) $\triangleq$

1 */∗ body of int-create-node ∗/*

type: *Create-request-node*$_1$ $\rightarrow$ *Entity-dict* $\Rightarrow$

est équivalent à:

int-create-node (*createnode*) (*dict*) ≙

```
1  (let mk-Create-request-node1 (prid, exprl) = createnode in
2  /* body of int-create-node */ )
```

type: *Create-request-node*$_1$ → *Entity-dict* ⇒

NOTE - Dans cet exemple, nous avons aussi une deuxième liste de paramètres contenant le paramètre formel *dict* du domaine *Entity-dict*.

5.4.1 Synonymes

Il n'est possible d'utiliser l'opérateur de sélection du champ que si le champ est représenté par un nom dans la définition de domaine. Si nous voulons par exemple utiliser l'opérateur de sélection sur le deuxième champ d'objets du domaine *Output-node*$_1$, nous devons définir *Output-node*$_1$ quelque peu différemment:

```
9   Output-node1          :: Signal-identifier1
                             Valuelist
                             [Signal-destination1]
                             Direct-Via1
10  Valuelist             =  [Expression1]*
```

Cet *Output-node*$_1$, est exactement le même domaine que le *Output-node*$_1$ défini précédemment, la seule différence étant que nous avons donné un nom au deuxième champ; autrement dit, nous avons défini un synonyme ou une abréviation pour l'expression du domaine [*Expression*$_1$]* (le symbole «=» est utilisé dans la définition des synonymes). Il y a souvent d'autres raisons à la définition de synonymes: c'est le cas lorsque la même expression de domaine est utilisée à plusieurs endroits, ou pour plus de visibilité. Ainsi, dans la syntaxe abstraite du SDL, *Channel-name*$_1$, *Block-name*$_1$, *Process-name*$_1$, etc., sont tous des synonymes du domaine *Name*$_1$, mais ils donnent au lecteur des informations sur les objets représentés par les divers *Name*$_1$s appartenant à certaines catégories d'entités. Un autre cas typique est celui où nous sommes en présence d'une longue liste de variantes. Par exemple, la syntaxe abstraite pour *Expression*$_1$ est:

```
11  Expression1           =  Ground-expression1 |
                             Active-expression1 |
12  Active-expression1    =  Variable-access1 |
                             Conditional-expression1 |
                             Operator-application1 |
                             Imperative-operator1 |
                             Error-expression1 |
13  Imperative-operator1  =  Now-expression1 |
                             Pid-expression1 |
                             View-expression1 |
                             Timer-active-expression1 |
                             Anyvalue-expression1 |
14  Pid-expression1       =  Self-expression1 |
                             Parent-expression1 |
                             Offspring-expression1 |
                             Sender-expression1 |
```

qui reflète mieux le regroupement des différents types d'expressions que

```
15  Expression1           =  Ground-expression1 |
                             Variable-access1 |
                             Conditional-expression1 |
                             Operator-application1 |
                             Now-expression1 |
                             Self-expression1 |
                             Parent-expression1 |
                             Offspring-expression1 |
                             Sender-expression1 |
                             View-expression1 |
                             Timer-active-expression1 |
                             Anyvalue-expression1 |
                             Error-expression1 |
```

5.4.2 Arborescences non nommées

Dans certains cas, il est inutile de nommer une définition arborescente. Les arborescences non nommées sont largement utilisées dans la FD, mais elles sont anonymes car bien souvent, il n'y a pas lieu de les définir explicitement.

Exemple:

Dans la sémantique dynamique, la première ligne de la définition de *Entity-dict* est la suivante:

16 *Entity-dict* = (*Qualifier*$_1$ **TYPE**) $\vec{m}$ *TypeDD*

Cela signifie que *Entity-dict* contient une correspondance en provenance des deux domaines *Qualifier*$_1$ et **TYPE** vers un descripteur *TypeDD*. Ces deux domaines constituent une arborescence non nommée. Si nous devions utiliser une arborescence nommée, nous devrions reformuler la définition comme suit:

17 *Entity-dict* = *Pair* $\vec{m}$ *TypeDD*
18 *Pair* :: *Qualifier*$_1$ **TYPE**

Exemple:

Dans la sémantique dynamique, *Reachability* est défini comme suit:

19 *Reachability* = *Reachability-endp Signal-identifier*$_1$-**set** *Path*

Nous avons défini ici un synonyme pour une arborescence non nommée contenant trois champs:

1) Le premier est le champ du domaine *Reachability-endp.*
2) Le deuxième comprend un ensemble d'identificateurs de signaux.
3) Le troisième est un champ du domaine *Path.*

Comme cela est indiqué, dans les définitions de domaine, les parenthèses servent à la fois à définir des arborescences non nommées et à regrouper des variantes.

Exemple:

Dans la sémantique dynamique, la fonction *make-procedures-formal-parameters* se définit comme suit:

make-procedure-formal-parameters(*parml, level*) $\triangleq$

1 /∗ *The body, which is not shown here* ∗/

type: *Procedure-formal-parameters*$_1$* *Qualifier*$_1$ → *FormparmDD** *Entity-dict*

Cette fonction renvoie deux objets *FormparmDD** et *Entity-dict*, ce qui signifie qu'elle retourne en réalité une arborescence non nommée constituée par deux objets.

L'opérateur **mk-** ne peut pas être utilisé sur des arborescences non nommées. La composition et la décomposition de ces arborescences s'obtient en mettant les champs entre parenthèses.

Exemple:

Composition d'un objet *Reachability-endp*, désigne un *Process-Identifier*$_1$, *b* un ensemble d'identificateurs de signaux et *d* un *Path.*

(a, b, d)

Si, pour faciliter la lecture, nous voulons désigner l'objet par un nom (il est plus simple d'avoir un nom qu'(*a*, *b*, *d*) surtout si (*a*, *b*, *d*) est utilisé plusieurs fois dans une expression), nous pouvons à nouveau faire appel à la construction **let**, c'est-à-dire que l'expression:

/∗ *some expression using* "(a, b, d)" ∗/

est équivalente à

(**let** *reach* = (*a, b, d*) **in**
/∗ *some expression using "reach"* ∗/)

La construction **let** sert aussi à décomposer des objets d'arborescences non nommées. Par exemple, la décomposition d'un objet *Reachability* nommé *reach* où, pour une raison quelconque, nous n'utilisons pas l'ensemble d'identificateurs de signaux est:

let (*a, ,d*) = *reach* **in**
/∗ *some expression using a and d* ∗/

Lorsque nous appelons une fonction, nous décomposons habituellement les arborescences non nommées qui résultent de l'appel de fonction, c'est-à-dire que

let *(parmddl, fdict) = make-procedure-formal-parameters(…, …)* **in**
/ some expression using the function results parmddl and fdict */*

est équivalent à

let *parminf = make-procedure-formal-parameters(…, …)* **in**
let *(parmddl, fdict) = parminf* **in**
/ some expression using the function results parmddl and fdict */*

5.4.3 Constructions de branchement

Dans certains cas, il doit être possible de distinguer un certain nombre d'objets d'arborescences les uns des autres. Ainsi, des objets du synonyme précédemment défini *Imperative-operator*$_1$ sont soit une *Now-expression*$_1$, soit une *Pid-expression*$_1$, soit une *View-expression*$_1$, etc. Si nous sommes en présence d'un *Imperative-operator*$_1$, nous devons d'abord déterminer le type de *Imperative-operator*$_1$ avant de pouvoir l'évaluer. A cet effet, nous pouvons employer le cas expression/énoncé. Ainsi, la fonction permettant d'évaluer les expressions SDL impératives peut s'exprimer comme suit:

eval-imperative-expression(*expr*) ≙

```
1   cases expr:
2   (mk-Now-expression1( )
3     → eval-now-expression( ),
4   mk-View-expression1(vid, pidexpr)
5     → eval-view-expression(vid, pidexpr)
6   mk-Timer-active-expression1(tid, actlist)
7     → eval-timer-expression,(tid, actlist)
8   mk-Anyvalue-expression1(sortref)
9     → eval-anyvalue-expression(sortref),
10  T → eval-pid-expression(expr))
```

type: *Imperative-operator*$_1$ ⇒

Signalons que le branchement se fait sur le type de *Imperative-operator*$_1$, et non sur la valeur réelle des champs de l'arborescence. **T** indique une clause «sinon» qui intervient ici étant donné que la variante finale, dans *Imperative-operator*$_1$ (*Pid-expression*$_1$) est un synonyme représentant quatre nouvelles variantes que nous ne voulons pas différencier ici. L'évaluation de ces variantes est renvoyée à *eval-pid-expression*.

Pour ce faire, on peut aussi utiliser l'opérateur booléen **is-**, qui renvoie vrai si l'objet fourni comme argument appartient à un certain domaine, soit:

eval-imperative-expression(*expr*) ≙

```
1   if is-Now-expression1(expr) then
2     eval-now-expression( )
3    else
4     if is-View-expression1(expr) then
5       eval-view-expression(s-Variable-identifier1(expr),s-Expression1(expr))
6      else
7       if is-Timer-active-expression1(expr) then
8         (let mk-Timer-active-expression1(tid, actlist) = expr in
9          eval-timer-expression(tid, actlist))
10       else
11        if is-Anyvalue-expression1(expr) then
12          (let mk-Anyvalue-expression1(sortref) = expr in
13           eval-anyvalue-expression(sortref))
14         else
15          eval-pid-expression(expr)
```

type: *Imperative-operator*$_1$ ⇒

Il convient de noter que l'accès aux champs par décomposition (ligne 8) et l'accès aux champs au moyen de l'opérateur de sélection du champ (ligne 5) sont tous deux illustrés ici.

Comme dans la plupart des autres langages de programmation et de spécification, il faut que les variantes soient «constantes» dans le cas expression/énoncé (comme elles le sont en cas de branchement sur le type d'arborescence); cela signifie que si les variantes sont de nature dynamique (variables ou paramètres formels), il faut utiliser la construction si-lors-sinon. Il existe néanmoins une autre notation pour la construction si-alors-sinon, c'est la construction dite Mc-arthy, qui convient mieux lorsqu'il y a beaucoup de variantes.

eval-imperative-expression(*expr*) ≙

```
1   (is-Now-expression1(expr)
2     → eval-now-expression( ),
3   is-View-expression1(expr)
4     → (let mk-View-expression1(vid, pidexpr) = expr in
5         eval-view-expression(vid, pidexpr)),
6   is-Timer-expression1(expr)
7     → (let mk-Timer-expression1(tid, actlist) =expr in
8         eval-timer-expression(tid, actlist)),
9   is-Anyvalue-expression1(expr)
10    → (let mk-Anyvalue-expression1(sortref) = expr in
11        eval-anyvalue-expression(sortref)),
12  T → eval-pid-expression(expr))
```

type: *Imperative-operator*$_1$ ⇒

Notons que certains noms de fonctions FD commencent aussi par «is-». Ces cas peuvent facilement être distingués de l'opérateur «**is-**» car ils ne sont pas en caractères gras.

5.4.4 Domaines élémentaires

Le Meta-IV fournit un certain nombre de domaines élémentaires prédéfinis. Leur notation et les opérateurs qui leur sont associés sont définis ci-après.

5.4.4.1 Booléen

Le nom *Bool* du Meta-IV décrit le domaine des valeurs vraies, à savoir l'ensemble {**true**,**false**}.

Opérateurs booléens:

Notation	Type			Opération
¬	*Bool*	→	*Bool*	négation
∧	*Bool*	→	*Bool*	et
∨	*Bool*	→	*Bool*	ou
⊃	*Bool*	→	*Bool*	implique
=	*Bool Bool*	→	*Bool*	égal
≠	*Bool Bool*	→	*Bool*	différent

Exemple:

En expressions Meta-IV, les propriétés des opérateurs Bool ¬, ∧, ∨ et ⊃ peuvent être représentées comme suit:

¬*a* = (**if** *a* **then false else true**)
a ∨ *b* = (**if** *a* **then true else** *b*)
a ∧ *b* = (**if** *a* **then** *b* **else false**)
a ⊃ *b* = (**if** ***a*** **then** *b* **else true**)

5.4.4.2 Entier

Trois noms de domaines sont prédéfinis pour les valeurs entières:

- Le nom *Intg* désigne le domaine de toutes les valeurs entières, à savoir l'ensemble {... −2,−1,0,1,2,...}.
- Le nom N_0 désigne le domaine des valeurs entières non négatives, à savoir l'ensemble {0,1,2,...}.
- Le nom N_1 désigne le domaine des valeurs entières positives, à savoir l'ensemble {1,2,...}.

Opérateurs entiers:

Notation	Type			Opération
−	*Intg*	→	*Intg*	négation
−	*Intg Intg*	→	*Intg*	soustraction
+	*Intg Intg*	→	*Intg*	additon
*	*Intg Intg*	→	*Intg*	multiplication
/	*Intg Intg*	→	*Intg*	division d'entiers
mod	N_0 N_1	→	N_0	module
=	*Intg Intg*	→	*Bool*	égal
≠	*Intg Intg*	→	*Bool*	différent
<	*Intg Intg*	→	*Bool*	inférieur à
≤	*Intg Intg*	→	*Bool*	inférieur ou égal à
>	*Intg Intg*	→	*Bool*	supérieur à
≥	*Intg Intg*	→	*Bool*	supérieur ou égal à

5.4.4.3 Caractère

Le nom *Char* du Meta-IV désigne le domaine des valeurs de caractères ASCII. Pour les caractères d'imprimerie, il existe des représentations d'objets qui sont placés entre guillemets, par exemple, “a”, “Z”, “ ”.

Opérateurs caractères:

Notation	Type			Opération
=	*Char Char*	→	*Bool*	égal
≠	*Char Char*	→	*Bool*	différent
<	*Char Char*	→	*Bool*	inférieur à
≤	*Char Char*	→	*Bool*	inférieur ou égal à
>	*Char Char*	→	*Bool*	supérieur à
≥	*Char Char*	→	*Bool*	supérieur ou égal à

Les opérateurs relationnels sont appliqués aux valeurs numériques ASCII associées.

Pour faciliter la lecture, les objets du domaine $Char^+$ peuvent être représentés par une séquence de caractères mis entre guillements, soit “abc” équivaut à ⟨“a”, “b”, “c”⟩ (voir 5.4.6).

5.4.4.4 Mot clé

Le nom *Quot* du Meta-IV désigne le domaine des mots clés. Il s'agit d'objets élémentaires distincts, représentés comme une séquence en caractères gras, de lettres majuscules et de chiffres, par exemple, **ENVIRONMENT**, **REVERSE**.

Opérateurs mots clés:

Notation	Type			Opération
=	*Quot Quot*	→	*Bool*	égal
≠	*Quot Quot*	→	*Bool*	différent

Par opposition aux autres domaines, on peut trouver des objets de *Quot* dans des définitions de domaines lorsque seuls certains objets de *Quot* sont possibles dans le contexte donné. Ainsi, dans la syntaxe abstraite de la présente Recommandation, *Originating-block*$_1$ est défini de la façon suivante:

1 *Originating-block*$_1$ = *Block-identifier*$_1$ | **ENVIRONMENT**

On aurait encore pu définir *Originating-block*$_1$ au moyen de *Quot:*

2 *Originating-block*$_1$ = *Block-identifier*$_1$ | *Quot*

cependant, l'utilisation d'**ENVIRONMENT** dans la définition de domaine est plus précise, puisque cet objet est la seule valeur *Quot* possible dans le contexte en question.

5.4.4.5 Token

Le nom *Token* du Meta-IV désigne le domaine des marques. On peut dire que ce domaine est constitué par un ensemble potentiellement infini d'objets élémentaires distincts pour lesquels aucune représentation n'est nécessaire.

Opérateurs marques:

Notation	Type			Opération
=	*Token Token*	→	*Bool*	égal
≠	*Token Token*	→	*Bool*	différent

Exemple:

Dans la syntaxe abstraite de la présente Recommandation, *Name*$_1$ est défini comme suit:

1 *Name*$_1$ :: *Token*

La seule propriété requise pour *Name*$_1$ au cours de l'interprétation est l'égalité. Un *Name*$_1$ est donc constitué par une valeur *Token* (l'orthographe réelle des noms est sans importance).

5.4.4.6 Ellipse

Le domaine d'ellipse (représenté par …) décrit une construction non spécifiée. Il est utilisé dans des définitions de domaines ou dans des expressions:

- chaque fois que le domaine ou l'expression réel est sans importance pour la sémantique, ou
- chaque fois que l'élaboration du domaine ou de l'expression dépasse le cadre de la spécification.

Exemple:

Dans la syntaxe abstraite de la présente Recommandation *Informal-text*$_1$ est défini comme suit:

1 *Informal-text*$_1$ :: …

Informal-text$_1$ ne peut être interprété au moyen du Meta-IV. Il contient donc un autre objet non spécifié.

5.4.5 Domaines d'ensembles

Un domaine d'ensembles se construit en ajoutant le suffixe du mot clé **-set** (le tiret est important) au domaine d'élément. Exemple:

2 *State-node*$_1$ :: *State-name*$_1$
Save-signalset$_1$
Input-node$_1$**-set**
Spontaneous-transition$_1$,**-set**

3 *Save-signalset* :: *Signal-Identifier*$_1$**-set**

signifie que les objets du domaine *State-node*$_1$ se composent d'un nom d'état, d'un ensemble de signaux de mise en réserve qui contient un ensemble d'identificateurs de signaux, d'un ensemble de nœuds d'entrée et d'un ensemble de transitions spontanées. Les valeurs d'ensemble peuvent être construites à l'aide d'un constructeur d'ensemble explicite, qui est une liste d'expressions entre accolades, soit:

{1,3,5,1}

indique un objet du domaine *Intg***-set** et contient les trois valeurs *Intg* 1,3,5. Une forme plus courante est le constructeur d'ensemble dit implicite, dans lequel l'ensemble comprend tous les éléments remplissant une certaine condition (prédicat), soit:

$\{i \in Intg \mid 0 \leq i \leq 5 \wedge i \textbf{ mod } \textbf{2} = 0\}$

définit l'ensemble

{0, 1, 2, 3, 4, 5, 6, 8, 10, 12, 14, 16, …}

qui désigne l'ensemble des valeurs situées à gauche de la barre verticale (qu'on peut qualifier par une valeur ou par un domaine) pour lesquelles l'expression située à droite de la barre verticale est vérifiée.

L'ensemble vide est représenté par { }.

Dans l'explication ci-après de la sémantique des opérateurs sur ensembles, *s* désigne l'ensemble {1,3,5}:

∈ Opérateur d'appartenance
Vérifie si un élément donné du domaine d'élément est contenu dans un ensemble, c'est-à-dire $1 \in s \equiv$ **true** et $2 \in s \equiv$ **false**.

∉ Vérifie si un élément donné du domaine d'élément n'est pas contenu dans un ensemble, c'est-à-dire $1 \notin s \equiv$ **false** et $2 \notin s \equiv$ **true**.

∪ Opérateur d'union
Unit deux ensembles, c'est-à-dire $\{2,3\} \cup s \equiv \{1,2,3,5\}$ et $s \cup s \equiv s$.

∩ Opérateur d'intersection. Renvoie l'intersection de deux ensembles, c'est-à-dire $\{2,3\} \cap s \equiv \{3\}$ et $\{\} \cap s \equiv \{\ \}$.

\ Opérateur de complément
Exclut un ensemble donné de valeurs d'un ensemble, c'est-à-dire $s \setminus \{1,2\} \equiv \{3,5\}$ et $\{1,2\} \setminus s \equiv \{2\}$.

⊂ Opérateur de sous-ensemble strict
Vérifie si les éléments d'un ensemble donné sont contenus dans un ensemble, c'est-à-dire $\{1,5\} \subset s \equiv$ **true**, $s \subset \{1,5\} \equiv$ **false** et $s \subset s \equiv$ **false**.

⊆ Opérateur de sous-ensemble
Vérifie si les éléments d'un ensemble donné appartiennent ou sont égaux à un ensemble, c'est-à-dire $\{1,5\} \subseteq s \equiv$ **true**, $s \subseteq \{1,5\} \equiv$ **false** et $s \subseteq s \equiv$ **true**.

card Opérateur de cardinalité
Renvoie le nombre d'éléments d'un ensemble, c'est-à-dire **card** $s \equiv 3$ et **card** $\{\ \} \equiv 0$.

union Opérateur d'union distributive

L'argument est un ensemble d'ensembles et le résultat est l'union de tous les ensembles contenus dans cet argument, c'est-à-dire **union** $\{s_1\{5,6\},\{1,5,8\}\} \equiv s \cup \{5,6\} \cup \{1,5,8\} \equiv \{1,3,5,6,8\}$.

=, ≠ Vérifie l'égalité ou l'inégalité des ensembles.

Exemple:

En expressions Meta-IV, on peut représenter les propriétés des opérateurs d'ensembles $\notin$, $\cup$, $\cap$, $\subset$, $\subseteq$, **card** et **union** de la façon suivante:

$element \notin s1 = (\neg(element \in s1))$

$s1 \cup s2 = \{element \mid element \in s1 \vee element \in s2\}$

$s1 \cap s2 = \{element \mid element \in s1 \wedge element \in s2\}$

$s1 \setminus s2 = \{element \mid element \in s1 \wedge element \notin s2\}$

$s1 \subset s2 = (\forall element \in s1)\ (element \in s2) \wedge s1 \neq s2$

$s1 \subseteq s2 = (\forall element \in s1)\ (element \in s2)$

card s1 = (**if** $s1 = \{\ \}$
then 0
else (**let** $element \in s1$ **in**
$1 +$ **card** $(s1 \setminus \{element\ \})$))

union $s1 = \{element \mid \exists set \in s1)\ (element \in set)\}$

Les quantificateurs ($\forall$ et $\exists$) sont expliqués dans 5.6.

5.4.6 Domaines de listes

Une liste ou un domaine de multiplets se construit en ajoutant le suffixe «*» au domaine d'élément dans le cas d'une liste éventuellement vide et, sinon, en ajoutant le suffixe «$^+$».

Exemple:

4 *Signal-definition*$_1$ :: *Signal-name*$_1$
Sort-reference-identifier$_1$*

Cette définition de domaine indique qu'une définition de signal est constituée par un nom de signal et par une liste éventuellement vide d'identificateurs de sorte.

On peut construire une valeur de liste à l'aide d'un constructeur de multiplets explicites. C'est une liste d'expressions mise entre crochets obliques, par exemple:

⟨11,12,11,13,14⟩

désigne un objet du domaine *Intg*$^+$ (*Intg**) et contient 5 éléments ordonnés.

La liste vide est représentée par ⟨ ⟩.

Il existe également des constructeurs de listes implicites analogues à ceux utilisés pour les ensembles. Par exemple, dans la fonction int-output-node de la sémantique dynamique, nous construisons un multiplet (*vall*) qui contient les valeurs de tous les paramètres réels (*exprl*) dans un nœud de sortie:

let $vall = \langle eval\text{-}expression\ (exprl\ [i])(dict) \mid 1 \leq i \leq$ **len** $exprl\rangle$ **in**

correspond à une énumération explicite de tous les éléments de la liste:

let $vall = \langle eval\text{-}expression(exprl\ [1])(dict),$
$eval\text{-}expression(exprl\ [2])(dict),$
$eval\text{-}expression(exprl\ [3])(dict),$
$\ldots\rangle$ **in**

Notons que les crochets de multiplets (⟨ et ⟩) ont une forme différente de celle des opérateurs de relations < et >.

Dans l'explication ci-après de la sémantique des opérateurs sur listes, *l* désigne la liste ⟨11,12,11,13,14⟩:

hd Renvoie le premier élément (l'en-tête d'une liste), soit **hd** *l* ≡ 11. L'argument associé à **hd** ne doit pas être une liste vide (⟨ ⟩).

tl Renvoie la liste dans laquelle le premier élément a été supprimé (renvoie la queue), soit **tl** *l* ≡ ⟨12,11,13,14⟩.

[*i*] Renvoie le nombre d'éléments *i* dans une liste, soit *l*[3] ≡ 11 et *l*[5] ≡ 14. La valeur d'index ne doit pas être inférieure à 1 ou supérieure à la longueur de la liste.

len Renvoie la longueur d'une liste, soit **len** *l* ≡ 5.

elems Renvoie l'ensemble comprenant les éléments contenus dans une liste, soit **elems** *l* ≡ {11,12,13,14}.

ind Renvoie l'ensemble des objets entiers qui sont les valeurs d'index pour une liste, soit **ind** *l* ≡ {1,2,3,4,5}.

⁀ Concatène deux listes, soit *l* ⁀ ⟨0,1⟩ ≡ ⟨11,12,11,13,14,0,1⟩

conc Concatène toutes les listes qui sont des éléments de la liste fournie comme argument, soit conc ⟨⟨0,7⟩,*l*,⟨9⟩⟩ ≡ ⟨0,7,11,12,11,13,14,9⟩.

=, ≠ Vérifie l'égalité ou l'inégalité des listes.

Exemple:

En expressions Meta-IV, les propriétés des opérateurs de liste **hd**, **tl**, **ind**, **elems** et **conc** peuvent être représentées comme suit:

hd ***l*** = (*if* *l* = ⟨ ⟩ **then undefined else** ***l*** [1])
tl ***l*** = ⟨*l* [***i***] | 2 ≤ ***i*** ≤ **len** ***l***⟩
ind ***l*** = {***i*** | 1 ≤ ***i*** ≤ **len** ***l***}
elems ***l*** = {***l*** [***i***] | ***i*** ∈ **ind** ***l***}
conc ***l*** = (**if** ***l*** = ⟨ ⟩ **then** ⟨ ⟩ **else hd** ***l*** ⁀ **conc tl** ***l***)

5.4.7 Domaines de mise en correspondance

On construit un domaine de mise en correspondance (c'est-à-dire un tableau) en spécifiant le domaine des objets d'entrée, suivi de l'opérateur $\overrightarrow{m}$ puis du domaine des objets contenus dans la mise en correspondance (valeurs d'intervalle).

Exemple:

5 *Entity-dict* = (*Identifier*$_1$ **PROCESS**) $\overrightarrow{m}$ *ProcessDD* ∪
(*Identifier*$_1$ **SERVICE**) $\overrightarrow{m}$ *ServiceDD* ∪
ENVIRONMENT $\overrightarrow{m}$ *Reachabilities* ∪
EXPIREDF $\overrightarrow{m}$ *Is-expiredF* ∪
PIDSORT $\overrightarrow{m}$ *Sort-identifier*$_1$ ∪
NULLVALUE $\overrightarrow{m}$ *Value* ∪
TRUEVALUE $\overrightarrow{m}$ *Value* ∪
FALSEVALUE $\overrightarrow{m}$ *Value* ∪

Afin d'améliorer la vue générale de cet exemple, on a uniquement donné ici une partie de la définition de *Entity-dict.* On trouvera dans la sémantique dynamique la définition complète de la mise en correspondance *Entity-dict.* Il y est montré comment l'opérateur $\overrightarrow{m}$ est utilisé et il ressort également que des mises en correspondances composites peuvent être construites à l'aide de l'opérateur ∪ de fusion de domaine, c'est-à-dire, lorsqu'on a une mise en correspondance du domaine *Entity-dict:*

- nous recherchons la mise en correspondance en appliquant un objet de l'arborescence non nommée (*Identifier*$_1$ **PROCESS**) et le résultat est un objet du domaine *Process DD*; ou
- nous recherchons la mise en correspondance en appliquant un objet de l'arborescence non nommée (*Identifier*$_1$ **SERVICE**) et le résultat est un objet du domaine *Service DD*; *ou*
- nous appliquons la valeur *Quot* **ENVIRONMENT** et le résultat est un objet du domaine *Reachabilities*; ou
- nous appliquons la valeur *Quot* **EXPIREDF** et le résultat est un objet du domaine *Is-expiredF*; ou
- nous appliquons la valeur *Quot* **PIDSORT** et le résultat est un objet du domaine *Sort-Identifier*$_1$; ou

– nous appliquons la valeur *Quot* **NULLVALUE** et le résultat est un objet du domaine $Value_1$; ou

– nous appliquons la valeur *Quot* **TRUEVALUE** et le résultat est un objet du domaine $Value_1$; ou

– nous appliquons la valeur *Quot* **FALSEVALUE** et le résultat est un objet du domaine $Value_1$.

Nous ne pouvons appliquer une valeur que si elle a été placée auparavant dans l'objet de mise en correspondance, par opposition aux fonctions où la correspondance entre valeurs d'argument et valeurs de résultat est fixée et définie au moment de la définition de la fonction.

Les valeurs de mise en correspondance peuvent être construites à l'aide d'un constructeur de mise en correspondance explicite, qui est une liste de paires de valeurs d'entrée et de valeurs d'intervalle entre crochets, soit:

[1 ↦ **D**,
2 ↦ **AA**,
4 ↦ **BB**,
9 ↦ **ABC**,
5 ↦ **XYZ**]

indique une valeur de mise en correspondance du domaine *Intg* $\overrightarrow{m}$ *Quot*.

On peut aussi construire des mises en correspondance implicites. Ainsi, la mise en correspondance implicite:

$[a \mapsto b \mid a \in N_1 \wedge a * a = b]$

est équivalente à la mise en correspondance infinie

[1 ↦ 1,
2 ↦ 4,
3 ↦ 9,
… ↦ …]

Dans l'explication suivante de la sémantique des opérateurs sur mises en correspondance, *m* désigne la première des mises en correspondance spécifiée de façon explicite ci-dessus:

m(*entryvalue*) Renvoie une valeur issue d'une mise en correspondance, c'est-à-dire que *m* (1) ≡ **D** et *m* (9) ≡ **ABC**.

\+ Recouvre une mise en correspondance par une autre mise en correspondance. Cet opérateur n'est pas un opérateur commutatif, c'est-à-dire que

m + [0 ↦ **XX**, 1 ↦ **B**] ≡

[0 ↦ **XX**,1 ↦ **B**,2 ↦ **AA**,4 ↦ **BB**,9 ↦ **ABC**,5 ↦ **XYZ**]

alors que

[0 ↦ **XX**,1 ↦ **B**] + *m* ≡

[0 ↦ **XX**,1 ↦ **D**,2 ↦ **AA**,4 ↦ **BB**,9 ↦ **ABC**,5 ↦ **XYZ**]

Exclut un ensemble donné de valeurs d'entrée d'une mise en correspondance, c'est-à-dire que *m* \ {1,2,3}≡

[4 ↦ **BB**,9 ↦ **ABC**,5 ↦ **XYZ**]

dom Renvoie l'ensemble qui contient exactement les valeurs d'entrée apparaissant dans une mise en correspondance donnée, soit:

dom *m* ≡ {1,2,4,5,9}

rng Renvoie l'ensemble qui contient exactement les valeurs d'intervalle contenues dans une mise en correspondance donnée, soit:

rng *m* ≡ {**D**,**AA**,**BB**,**ABC**,**XYZ**}

=, ≠ Vérifie l'égalité et l'inégalité de deux mises en correspondance.

merge Renvoie, à partir de l'ensemble de mises en correspondance donné, la mise en correspondance construite par la fusion de toutes celles contenues dans l'ensemble, soit:

{*m*,[0 ↦ **WE**],[10 ↦ **D**] }≡

[0 ↦ **WE**,10 ↦ **D**,1 ↦ **D**, 2 ↦ **AA**,4 ↦ **BB**,9 ↦ **ABC**,5 ↦ **XYZ**]

Si certaines des mises en correspondance de l'ensemble ont des entrées qui se recouvrent, une valeur arbitraire est choisie parmi les valeurs possibles.

La mise en correspondance vide est représentée par [] (deux crochets très rapprochés l'un de l'autre).

Exemple:

En expressions Meta-IV, les propriétés des opérateurs de mise en correspondance \ , + et **merge** peuvent s'illustrer comme suit:

$m1 \setminus s = [a \rightarrow b \mid a \in \textbf{dom}\ m1 \setminus s \wedge m1(a) = b]$
$m1 + m2 = [a \rightarrow b \mid (a \in \textbf{dom}\ m2 \wedge m2(a) = b) \vee (a \in \text{dom}\ m1 \setminus \text{dom}\ m2 \wedge m1(a) = b)]$
merge $m1$ = (if $m1 = \{\}$
then []
else (**let** *element* $\in$ $m1$ **in**
element + **merge** $m1 \setminus \{element\}$))

5.4.8 Domaines Pid

Un domaine Pid (correspondant à la sorte Pid en SDL) est construit au moyen du symbole Π. Il peut facultativement être qualifié par le type de processeur afin d'indiquer la nature des valeurs Pid désignées par le domaine. Exemple:

6 *Import-Create* :: Π(*input-port*)

Le domaine *Discard-Signals* (défini dans la sémantique dynamique) contient des objets Pid qualifiés par le type de processeur *input-port*. Il ne faut pas confondre les valeurs Meta-IV Pid avec les valeurs Pid en SDL qui, en SDL, sont des *Ground-term*$_1$s. Cela signifie que dans la sémantique dynamique, le domaine des valeurs Pid SDL est défini comme suit:

7 *Pid-Value* = *Value*
8 *Value* = *Ground-term*$_1$

Les valeurs Meta-IV Pid sont créées au moment de l'application de l'énoncé/expression de **départ**, ce qui correspond à l'action de demande de création en SDL. Par exemple, l'opération de la création, par le processeur *system* d'une instance de processeur *timer* à l'aide du paramètre effectif *timerf*, s'exprime comme suit:

Exemple:

start *timer* (*timeinf*)(*dict*)

Lorsque la construction de départ est prise comme une expression, elle crée une instance de processeur et renvoie la valeur Pid Meta-IV de cette instance (correspondant à la valeur **offspring** en SDL). Ainsi, quand le processeur *process-set-admin* déclenche son processeur input-port:

start *input-port*(*offspring, dict* (**EXPIREDF**), *delayf*, **self**)

Une instance du processeur *input-port* est créée et la valeur Meta-IV Pid résultante est utilisée par le *process-set-admin* pour identifier l'accès d'entrée *input-port*. Les paramètres *offspring*, *dict* (**EXPIRED**) *delayf* et **self** sont donnés à l'instance créée. Tout comme dans le LDS, une instance peut accéder à sa propre valeur de Pid en utilisant l'expression **self**.

La communication est assurée par les primitives de communication synchrone **input** et **output**. Dans la construction output, nous pouvons choisir de communiquer soit avec une instance de processeur spécifique, soit avec une instance non spécifiée d'un type de processeur spécifique.

Exemple:

output mk-*Some-tree* (*somevalue, someothervalue*, …) **to** *p*

où *p* désigne une valeur Pid ou encore le nom d'un type de processeur. Les valeurs envoyées par le processeur sont généralement renfermées dans un objet d'arborescence nommée (appartenant à un certain domaine de communication) et ces arborescences peuvent être rendues équivalentes au concept de signal en SDL. Ainsi, *Some-tree* peut être considérée comme étant un signal.

Dans la construction input, nous spécifions à la fois l'objet de communication que nous voulons recevoir et l'action à entreprendre une fois l'objet reçu. Nous pouvons par ailleurs spécifier un nom qui, après réception de l'objet, indique la valeur Pid du processeur d'émission (correspondant à **sender** en SDL) ou restreint les expéditeurs éventuels, c'est-à-dire:

input mk-*Some-tree(a, b, d)* **from** *p*
⇒ */* some statements or an expression */*

Après réception de *Some-tree*, *a*, *b* et *d* désigneront les valeurs véhiculées par *Some-tree* et *p* peut avoir trois interprétations:

- Si *p* est un nom de type de processeur, l'entrée doit être reçue d'une instance de ce type de processeur particulier.
- Si *p* est un nom qui n'est pas encore défini, cette occurrence est celle qui définit le nom et qui est visible dans l'expression ou les énoncés suivant la clause d'entrée. Elle désigne la valeur Meta-IV Pid de l'expéditeur.
- Si *p* est une expression, il doit être du type Π et l'entrée sera reçue de l'instance de processeur indiquée par l'expression.

Si une ou plusieurs entrées peuvent être reçues, un certain nombre de constructions d'entrée séparées par des virgules sont spécifiées, et ce nombre est placé entre accolades:

{**input mk**-*Some-tree(a, b, d)* **from** *p*

⇒ */* some statements or an expression */,*

input mk-*Some-other-tree(a, b, d)* **from** *p*

⇒ */* some statements or an expression */* }

Dans certains cas, il arrive que nous voulions spécifier qu'une sortie ou une entrée doivent être faites, ce qui dépend tout d'abord de la communication qu'il est possible d'assurer (cette opération est impossible en SDL car la communication y est asynchrone). Des constructions de sortie sont alors insérées dans l'ensemble des événements de communications, soit:

{**input mk**-*Some-tree*(*a, b, d*) **from** *p*

⇒ */* some statements or an expression */,*

input mk-*Some-other-tree*(*a, b, d*) **from** *p*

⇒ */* some statements or an expression */,*

output mk-*Something*(*/* expression */, /* expression */*) to *pi* }

Si la communication doit être répétée, il est fréquent d'utiliser la construction de **cycle** conjointement avec l'entrée et la sortie:

cycle {**input mk**-*Some-tree*(*a, b, d*) **from** *p*

⇒ */* some statements or an expression */,*

input mk-*Some-other-tree*(*a, b, d*) **from** *p*

⇒ */* some statements or an expression */,*

output mk-*Something*(*/* expression */, /* expression */*) **to** *pi* }

Cela signifie qu'après un événement de communication, l'instance de processeur accomplira l'action appropriée et commencera à attendre qu'un nouvel événement se produise.

5.4.9 Domaines de référence

Quand une variable Meta-IV est déclarée par:

dcl v **type** *Intg*;

une position de mise en mémoire Meta-IV est attribuée et la variable (v) désignera une référence à la position. Une fois qu'on a accès au contenu de la position, l'opérateur **c** (opérateur de contenu) est utilisé comme indiqué précédemment. Si la variable est utilisée sans l'opérateur de contenu, le résultat obtenu est une valeur du domaine **ref**, c'est-à-dire une référence à la position de mise en mémoire. On spécifie les domaines **ref** par le mot clé **ref**, suivi du domaine approprié. Exemple:

9 *VarDD* :: *Variable-identifier*$_1$ *Sort-reference-identifier*$_1$
[*Ground-expression*$_1$] [**REVEALED**] **ref** *Stg*

Le descripteur de paramètre *IN*/*OUT* pour les procédures contient une référence au domaine *Stg*. Le descripteur *VarDD* est défini dans la sémantique dynamique et est décrit plus avant dans les annotations associées.

5.4.10 Domaines optionnels

Les crochets, qui sont largement utilisés dans les définitions de domaines, désignent la notion d'option.

Exemple:

10 *Signal-definition*$_1$ :: *Signal-name*$_1$
Sort-reference-identifier$_1$*
[*Signal-refinement*$_1$]

signifie que parmi les objets de l'arborescence *Signal-definition*$_1$, l'objet du domaine *Signal-refinement* peut ou non être présent. S'il ne l'est pas, le champ contiendra la valeur de type-moins zéro.

Exemple:

(**let mk**-*Signal-definition*$_1$(*name, sort, refinement*) = /* *some Signal-definition*$_1$*, object* */ **in**
if *refinement* = **nil then**
/* *some actions* */

else
(**let mk**-*Signal-refinement*$_1$ (...) = *refinement* **in**
/* *some other actions using the signal refinement* */))

5.5 Les constructions let et def

Comme nous l'avons vu plus haut, la construction **let** peut servir à composer et à décomposer des objets. Elle est plus couramment utilisée dans les cas où il s'agit de désigner un objet particulier (il s'agit souvent simplement d'éviter des expressions trop complexes et illisibles). Dans la construction **let**, les noms placés à gauche du signe égal sont les occurrences de définition (à l'exception des noms de domaines qui doivent toujours être définis quelque part dans une définition de domaine). Un nom introduit peut aussi être utilisé à droite du signal égal (le nom est alors défini récursivement) et dans l'expression suivant la construction **let**. Dans l'exemple ci-dessous, *name*$_1$ est visible (c'est-à-dire qu'il peut être utilisé) dans */*expression1*/*, */*expression2*/*, */*expression3*/* et */*expression4*/*, *name2* est visible dans */*expression2*/*, */*expression3*/* et */*expression4*/* et *name3* est visible dans */*expression3*/* et */*expression4*/*. Afin de restreindre la visibilité des noms introduits par un **let**, la construction **let** est placée entre parenthèses. Dans l'exemple ci-dessus, un affinage de signal forme une expression et commence par une parenthèse gauche puisqu'on emploie une construction **let**.

Il existe deux moyens de spécifier une séquence de **let**:

let *name*1 = /* *expression1* */ **in**
let *name*2 = /* *expression2* */ **in**
let *name*3 = /* *expression3* */ **in**
/* *expression4* */

ou

let *name*1 = /* *expression1* */,
*name*2 = /* *expression2* */,
*name*3 = /* *expression3* */ **in**
/* *expression4* */

Dans la FD, on utilise généralement la première forme, avec trois **let**, quand l'ordre est important, c'est-à-dire si /**expression2**/ utilise *name1* et si /**expression3**/ utilise *name2*, alors que la deuxième forme est employée quand les différents **let** sont indépendants.

Il existe plusieurs formes différentes de construction **let**. Nous avons déjà montré comment les utiliser pour décomposer des objets. On peut aussi utiliser les formes suivantes:

let *name* ∈ *setorname*1 **in**
/* *some expression using name* */
let *name* **be s.t.** /* *condition using name* */ **in**
/* *some expression using name* */
let *name* ∈ *setorname*2 **be s.t**. /* *condition using name* */ **in**
/* *some expression using name* */
let *name* (*parameters*) = /* *function body* */ **in**
/* *some expression applying name* */

La première indique qu'on extrait une valeur arbitraire appartenant à l'ensemble ou au domaine désigné par *setorname1* et qu'on représente la valeur au moyen de *name*.

La deuxième forme indique qu'on construit une valeur, *name* étant tel que la condition spécifiée est vérifiée pour la valeur.

La troisième forme est une combinaison des deux précédentes, comportant les mêmes restrictions. Si une telle valeur n'existe pas, la spécification est erronée.

La quatrième forme indique qu'on construit une fonction locale (appelée *name*) comportant certains paramètres formels (*parameters*) et un corps.

Exemple:

Définir la racine carrée de 3:

let $r \in$ *Real* **be s.t.** $r > 0 \wedge r * r =$ **3 in**

Exemple:

Définir la fonction factorielle dans laquelle *n* est le paramètre formel:

let *fact* (*n*) = **if** $n < 0$ **then error else if** $n = 0$ **then** 1 **else** $n *$ *fact* $(n - 1)$ **in**

Lorsqu'on définit un nom pour un objet construit par référence à l'état global (c'est-à-dire si le nom est défini du point de vue d'une expression impérative), on utilise la notation **def** en lieu et place de la notation **let**. Ainsi, le mot clé **let** est remplacé par le mot clé **def**, le symbole égal par deux points et le mot clé **in** par un point-virgule (étant donné que la construction **def** est employée dans un contexte d'énoncé; voir 5.7). Si par exemple nous voulons désigner une valeur d'instance de processeur de création par un nom, nous écrirons:

(**def** *pid* : **start** *input-port*(*somevalue*);
/ some statements using the pid value *)*

ou encore si nous voulons décomposer le résultat d'une fonction impérative, nous écrirons:

(**def mk**-*Some-tree*(*a, b*)*: some-imperative-function*(...);
/* some statements using a and b */)

Il existe aussi une version **def** de la construction "be such that"

(**def** $r \in$ *Real* **s.t.** $r > 0 \wedge r * r =$ **c**v1;
*/*some statements using r *)*

où **def** est employé car nous utilisons une variable (v1) dans l'évaluation de *r*, soit: définir une valeur *Real r* telle que le carré de *r* soit égal au contenu de la variable v1.

Il faut signaler que les noms introduits dans **let** et **def** ne sont pas des variables, mais des noms représentant une valeur spécifique auxquels il n'est pas permis d'affecter une valeur nouvelle.

5.6 Quantification

Le Meta-IV contient en outre des quantificateurs mathématiques: le quantificateur **universel**, représenté par le symbole $\forall$, le quantificateur existentiel, représenté par le symbole $\exists$ et le quantificateur **unique**, représenté par le symbole $\exists!$. Il est possible de les utiliser dans des expressions **quantifiées** qui renvoient la valeur booléenne vrai si une condition spécifiée (**prédicat**) pour un objet est satisfaite.

Exemple:

identifiers-defined-on-system-level(*p*) $\triangleq$

1 $(\forall$**mk**-$Identifier_1(q,) \in p)$ (**len** $q = 1$)

type: $Identifier_1$-**set** $\rightarrow$ *Bool*

Cette fonction renvoie vrai si, et seulement si, pour tous les identificateurs ($Identifier_1$) de l'ensemble *p*, il est vérifié que la longueur du qualificateur (*q*) est égale à 1 (la seconde paire de parenthèses entoure l'expression de prédicat).

Exemple:

one-identifier-defined-on-system-level(p) $\triangleq$

1 ($\exists$**mk**-*Identifier*$_1$(q,) $\in p$) (**len** $q = 1$)

type: *Identifier*$_1$-**set** $\rightarrow$ *Bool*

Cette fonction renvoie vrai si, et seulement si, il existe au moins un identificateur (*Identifier*$_1$) dans l'ensemble p, pour lequel la longueur du qualificateur (q) est égale à 1.

Exemple:

exactly-one-identifier-defined-on-system-level(p) $\triangleq$

1 ($\exists!$**mk**-*Identifier*$_1$(q,) $\in$ p) (**len** $q = 1$)

type: *Identifier*$_1$-**set** $\rightarrow$ *Bool*

Cette fonction renvoie vrai si, et seulement si, il existe exactement un (*Identifier*$_1$) dans l'ensemble p, pour lequel la longueur du qualificateur (q) est égale à 1.

Nous pouvons, à titre de variante, choisir de décomposer l'identificateur dans l'expression de prédicat, et non dans la quantification, soit:

identifiers-defined-on-system-level(p) $\triangleq$

1 ($\forall p' \in p$)
2 ((**let mk**-*Identifier*$_1$(q,) = p' **in**
3 **len** $q = 1$))

type: *Identifier*$_1$-**set** $\rightarrow$ *Bool*

NOTE – L'apostrophe et le tiret sont des caractères permis dans les noms du Meta-IV.

5.7 Enoncés auxiliaires

– Enoncé d'identité

Le mot clé **I** désigne un énoncé vide, à savoir un énoncé qui ne fait rien.

– Enoncé/expression indéfinis

Le mot clé **undefined** indique qu'aucune sémantique ne peut être fournie.

– Enoncé retour

Le mot clé **return** suivi d'une expression met fin à l'élaboration d'une fonction impérative, le résultat étant l'expression donnée.

– Enoncé/expression d'erreur

Le mot clé **error** indique une erreur SDL dynamique dans la FD.

– Enoncé d'affectation

Comme en SDL. L'opérateur de contenu (c) n'est pas utilisé lors de l'affectation à des variables.

– Enoncé For et while

Concept identique (bien connu) à celui du CHILL. Les énoncés à répéter sont mis entre parenthèses.

– Enoncé/expression Trap et exit

Piège (gère) des exits provoqués par un énoncé/expression exit. Si on fournit un argument à l'énoncé exit, il est piégé uniquement dans le cas où l'expression donnée correspond à la valeur donnée dans l'énoncé trap exit. Une version spéciale du mécanisme trap exit (la construction **tixe**) a été utilisée dans les fonctions *int-process-graph* et *int-procedure-graph*. On trouvera une explication de la construction **tixe** dans les annotations associées.

5.8 Différences avec la notation utilisée dans la définition formelle du CHILL

– Dans la définition formelle du CHILL, les noms de domaines prédéfinis sont constitués par des lettres majuscules en caractères gras (par exemple, **BOOL**, **INTG**) et les noms désignant des domaines sémantiques peuvent se composer uniquement de lettres majuscules.

  Dans la définition formelle du SDL, tous les noms de domaines sont en italique, la première lettre étant en majuscule, et ils comportent au moins une lettre minuscule.

– Dans la définition formelle du CHILL, tous les objets sont finis.

  Dans celle du SDL, il se peut que les objets soient infinis. La sémantique de certains des opérateurs n'est pas bien définie quand ils s'appliquent à de tels objets; ainsi, des opérateurs comme la cardinalité et l'égalité n'ont pas été utilisés sur des objets potentiellement infinis.

  Par ailleurs, une constante particulière *infinite* a été utilisée dans le *transform-process* de l'Annexe F.2 pour représenter le «nombre d'instances illimité» dans AS_1.

– Dans la définition formelle du SDL, la notation en Meta-IV a été élargie de manière à inclure le domaine élémentaire *Char* et les objets de chaînes de caractères (voir 5.4.4.3).

– Dans le processeur path de l'Annexe F.3, on a également fait appel à une notion dite de «garde de sortie». Cette notion est décrite dans les annotations associées au processeur *Path* ainsi que dans l'ouvrage [4].

5.9 Exemple: Spécification du «Demon game» en Meta-IV

La procédure exposée ci-après montre comment le Meta-IV peut servir à définir la sémantique du «Demon game». Pour de plus amples détails sur ce jeu, voir 2.9/Z.100.

Communication *demon* → *monitor* and *monitor* → *game*

11 *Bump* :: ( )

Communication *user* → *monitor*

12 *Newgame* :: ( )

Communication *game* → *monitor*

13 *Gameover* :: Π

Communication *monitor* → *game*

14 *Gameoverack* :: ( )

Communication *game* → *user*

15 *Gameid* :: ( )
16 *Win* :: ( )
17 *Lose* :: ( )
18 *Score* :: *Intg*

Communication *user* → *game*

19 *Probe* :: ( )
20 *Result* :: ( )
21 *Endgame* :: ( )

int-demon-game( ) ≙

1 **start** *monitor* ( )

type: ( ) ⇒ ( )

monitor **processor** ( ) ≙

```
1   (dcl userset := { } type Π-set,
2         gameset := { } type Π-set;
3   cycle (input mk-Newgame ( ) from sender)
4               ⇒ if sender ∉ c userset then
5                     (def offspring : start game (sender);
6                     gameset := c gameset ∪ {offspring};
7                     userset := c userset ∪ {sender})
8                 else
9                  I,
10          input mk-Gameover (player) from sender
11              ⇒ (gameset := c gameset \ {sender};
12                 userset := c userset \ {player};
13                 output mk-Gameoverack( ) to sender),
14          input mk-Bump( ) from demon
15              ⇒ for all pid ∈ gameset do
16                output mk-Bump( ) to pid))
```

type: ( ) ⇒

game **processor** *(player)* ≙

```
1   (dcl count := 0 type Intg;
2   dcl even := true type Bool;
3   output mk-Gameid( ) to player;
4   cycle (input mk-Probe( ) from user
5               ⇒ if c even
6                   then (output mk-Win( ) to player;
7                         count := c count + 1)
8                   else (output mk-Lose( ) to player;
9                         count := c count – 1),
10          input mk-Result( ) from user
11            ⇒ output mk-Score(count) to player,
12          input mk-Endgame( ) from user
13            ⇒ (output mk-Gameover(player) to monitor;
14               input-mk-Gameoverack( ) from monitor
15                 ⇒ stop),
16          input-mk-Bump( ) from monitor
17            ⇒ even := ¬c even))
```

type: Π ⇒ ( )

Références

[1] BJØRNER (D.) et JONES (C. B.): Formal specification and software development, *Prentice-Hall Publ.*, 1982.

[2] Manuel du CCITT *Définition formelle du CHILL*, UIT, Genève,1981.

[3] FOLKJAER (P.) et BJØRNER (D.): A formal model of a generalized CSP-like language, *IFIP 8th World Computer Conference*, Proceedings, North Holland Publ. 1980.

[4] HOARE (C. A. R.): Communicating Sequential Processes, *Prentice-Hall*, 1985.