UNION INTERNATIONALE DES TÉLÉCOMMUNICATIONS

UIT-T

SECTEUR DE LA NORMALISATION
DES TÉLÉCOMMUNICATIONS
DE L'UIT

Z.100

Annexe F1
(11/2000)

SÉRIE Z: LANGAGES ET ASPECTS GÉNÉRAUX LOGICIELS DES SYSTÈMES DE TÉLÉCOMMUNICATION

Techniques de description formelle – Langage de description et de spécification (SDL)

SDL: Langage de description et de spécification

Annexe F1: Définition formelle du langage SDL: aperçu général

Recommandation UIT-T Z.100 – Annexe F1

RECOMMANDATIONS UIT-T DE LA SÉRIE Z

LANGAGES ET ASPECTS GÉNÉRAUX LOGICIELS DES SYSTÈMES DE TÉLÉCOMMUNICATION

TECHNIQUES DE DESCRIPTION FORMELLE	
Langage de description et de spécification (SDL)	**Z.100–Z.109**
Application des techniques de description formelle	Z.110–Z.119
Diagrammes des séquences de messages	Z.120–Z.129
LANGAGES DE PROGRAMMATION	
CHILL: le langage de programmation de l'UIT-T	Z.200–Z.209
LANGAGE HOMME-MACHINE	
Principes généraux	Z.300–Z.309
Syntaxe de base et procédures de dialogue	Z.310–Z.319
LHM étendu pour terminaux à écrans de visualisation	Z.320–Z.329
Spécification de l'interface homme-machine	Z.330–Z.399
QUALITÉ DES LOGICIELS DE TÉLÉCOMMUNICATION	Z.400–Z.499
MÉTHODES DE VALIDATION ET D'ESSAI	Z.500–Z.599

Pour plus de détails, voir la Liste des Recommandations de l'UIT-T.

Recommandation UIT-T Z.100

SDL: Langage de description et de spécification

ANNEXE F1

Définition formelle du langage SDL: aperçu général

Résumé

La présente annexe contient la motivation, donne un aperçu général de la structure de sémantique et présente une introduction au formalisme d'automate abstrait à états finis (ASM, *abstract state machine*) qui est utilisé pour définir la sémantique SDL.

Source

L'Annexe F1 de la Recommandation Z.100 de l'UIT-T, revisée par la Commission d'études 10 (2001-2004) de l'UIT-T, a été approuvée le 24 novembre 2000 selon la procédure définie dans la Résolution 1 de l'AMNT.

AVANT-PROPOS

L'UIT (Union internationale des télécommunications) est une institution spécialisée des Nations Unies dans le domaine des télécommunications. L'UIT-T (Secteur de la normalisation des télécommunications) est un organe permanent de l'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.

L'Assemblée mondiale de normalisation des télécommunications (AMNT), qui se réunit tous les quatre ans, détermine les thèmes d'étude à traiter par les Commissions d'études de l'UIT-T, lesquelles élaborent en retour des Recommandations sur ces thèmes.

L'approbation des Recommandations par les Membres de l'UIT-T s'effectue selon la procédure définie dans la Résolution 1 de l'AMNT.

Dans certains secteurs des technologies de l'information qui correspondent à la sphère de compétence de l'UIT-T, les normes nécessaires se préparent en collaboration avec l'ISO et la CEI.

NOTE

Dans la présente Recommandation, l'expression "Administration" est utilisée pour désigner de façon abrégée aussi bien une administration de télécommunications qu'une exploitation reconnue.

DROITS DE PROPRIÉTÉ INTELLECTUELLE

L'UIT attire l'attention sur la possibilité que l'application ou la mise en œuvre de la présente Recommandation puisse donner lieu à l'utilisation d'un droit de propriété intellectuelle. L'UIT ne prend pas position en ce qui concerne l'existence, la validité ou l'applicabilité des droits de propriété intellectuelle, qu'ils soient revendiqués par un Membre de l'UIT ou par une tierce partie étrangère à la procédure d'élaboration des Recommandations.

A la date d'approbation de la présente Recommandation, l'UIT n'avait pas été avisée de l'existence d'une propriété intellectuelle protégée par des brevets à acquérir pour mettre en œuvre la présente Recommandation. Toutefois, comme il ne s'agit peut-être pas de renseignements les plus récents, il est vivement recommandé aux responsables de la mise en œuvre de consulter la base de données des brevets du TSB.

TABLE DES MATIÈRES

Recommandation UIT-T Z.100

SDL: Langage de description et de spécification

ANNEXE F1

Définition formelle du langage SDL: aperçu général

1 Préface

La présente définition formelle du langage de description et de spécification (SDL, *specification and description language*) complète en langage précis la définition donnée dans le corps de la Recommandation. Elle pourra être utilisée par ceux qui ont besoin d'une définition très précise du SDL, comme les conservateurs du langage SDL, les concepteurs d'utilitaires SDL et les utilisateurs du langage SDL.

Cette définition formelle se compose de trois annexes:

Annexe F1 Cette annexe contient la motivation, donne un aperçu général de la structure de sémantique et présente une introduction au formalisme d'automate abstrait à états finis (ASM, *abstract state machine*) qui est utilisé pour définir la sémantique SDL.

Annexe F2 Cette annexe décrit les contraintes sémantiques statiques et les transformations désignées par les paragraphes de la Rec. UIT-T Z.100 relatifs à la modélisation.

Annexe F3 Cette annexe définit la sémantique dynamique du langage SDL.

1.1 Motivation

Les spécifications en langage naturel sont ambiguës, c'est-à-dire qu'on peut en donner plusieurs interprétations. Une spécification est formelle si sa signification (ou sa sémantique) est univoque. Des langages particuliers, relevant des techniques de description formelle (FDT, *formal description techniques*) ont été mis au point à cette fin. Les techniques FDT se distinguent des langages formels en général par le fait qu'elles ont une syntaxe formelle *et* une sémantique formelle, ce qui diffère de la plupart des langages formels comme Java ou C++, qui n'ont qu'une syntaxe formelle.

La sémantique formelle d'un langage se définit en termes de formalisme mathématique sous-jacent, par exemple par schéma abstrait (axiome) ou concret (opération). Le choix d'un formalisme approprié est influencé par l'expressivité de la technique FDT employée ainsi que par les objectifs autres que la non-ambiguïté sémantique. La sémantique formelle définit, pour chaque spécification, un modèle mathématique approprié qui en saisit la signification précisément et complètement.

Les Annexes F1, F2 et F3 définissent formellement la sémantique du langage SDL. Toute contradiction entre le corps du texte de la Rec. UIT-T Z.100 et ces annexes dénote une erreur qu'il convient de corriger, aucun des deux textes n'ayant, dans ce cas, la préséance sur l'autre.

1.2 Objectifs principaux

L'un des objectifs principaux d'une sémantique SDL formelle est son intelligibilité, qui est un préalable au bien-fondé, à l'acceptation et à la maintenabilité. L'intelligibilité se fonde sur la construction de formalismes et de notations mathématiques bien établis, sur une étroite correspondance entre la technique de spécification et la sémantique à formaliser, et sur une documentation concise et bien structurée.

La maintenabilité est un autre objectif important parce que le langage SDL est une norme technique évolutive. En dehors des extensions linguistiques qui sont incorporées dans la présente Recommandation, d'autres caractéristiques linguistiques, comme l'expressivité en temps réel, sont à l'étude. Le formalisme mathématique doit donc être suffisamment riche et flexible pour que la sémantique formelle puisse être adaptée et étendue au prix d'un effort raisonnable.

Le langage SDL peut être considéré comme une technique de modélisation permettant de spécifier des systèmes répartis et concurrents. En d'autres termes, une spécification SDL définit explicitement un ensemble de calculs automatiques nécessitant une sémantique opérationnelle afin d'obtenir une corrélation étroite de ces calculs avec la spécification et donc d'améliorer l'intelligibilité de celle-ci. Par ailleurs, une sémantique opérationnelle se prête naturellement à l'exécutabilité compte tenu de la disponibilité d'outils, qui constitue un autre objectif explicite.

1.3 Références

– UIT-T Z.100 (1999), *SDL: langage de description et de spécification*.

1.4 Bibliographie

[1] http://www.uni-paderborn.de/cs/asm/ et http://www.eecs.umich.edu/gasm/.

[2] ESCHBACH (R.), GLÄSSER (U.), GOTZHEIN (R.), PRINZ (A.): On the Formal Semantics of SDL-2000: A Compilation Approach Based on an Abstract SDL Machine. In *Abstract State Machines: Theory and Applications*. Y. Gurevich, P.W. Kutter, M. Odersky and L. Thiele, editors, vol. 1912 of LNCS, Springer-Verlag, 2000.

[3] GUREVICH (Y.): Evolving Algebra 1993: Lipari Guide. In *Specification and Validation Methods*, E. Börger, editor, pages 9-36, Oxford University Press, 1995.

[4] GUREVICH (Y.): ASM Guide 97, *CSE Technical Report CSE-TR-336-97*, EECS Department, University of Michigan-Ann Arbor, 1997.

2 Aperçu général de la sémantique

Afin de définir la sémantique formelle du langage SDL, la définition de celui-ci est décomposée en plusieurs parties comme suit:

- la grammaire;
- les conditions de validité structurelle;
- les règles de transformation;
- la sémantique dynamique.

Le point de départ pour la définition de la sémantique formelle du langage SDL est une spécification SDL syntaxiquement correcte, représentée par un arbre syntaxique abstrait (AST, *abstract syntax tree*).

Les trois premières parties de la sémantique formelle sont collectivement désignées par le terme sémantique statique (*static semantics*) ou aspects statiques (*static aspects*) dans le contexte du langage SDL (voir Figure F1-1).

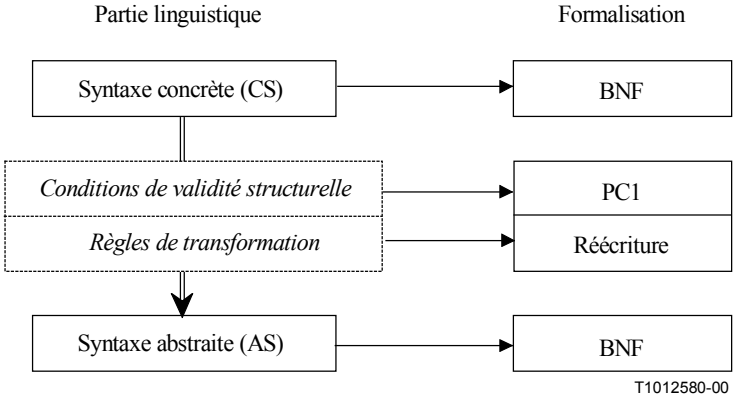


Figure F1-1/Z.100 – Aspects statiques du langage SDL

La grammaire (*grammar*) définit l'ensemble des spécifications SDL syntaxiquement correctes. La Rec. UIT-T Z.100 définit formellement, au moyen du formalisme de Backus-Naur (BNF), une grammaire textuelle concrète, une grammaire graphique concrète et une grammaire abstraite avec des extensions permettant de représenter les créations linguistiques graphiques. La grammaire abstraite se déduit des grammaires concrètes par suppression des détails inapplicables comme les séparateurs et les règles lexicales, ainsi que par l'application de règles de transformation (voir ci-dessous).

Les conditions de validité structurelle (*well-formedness conditions*) définissent les spécifications qui sont correctes en termes grammaticaux et en termes d'informations contextuelles, par exemple quant aux noms qu'il est permis d'utiliser à un endroit donné ou quant à la sorte de valeurs qu'il est permis d'attribuer à une variable. Les conditions de validité structurelle sont définies en termes de calcul de prédicat du premier ordre (PC1, *first order predicate calculus*).

Par ailleurs, certaines créations linguistiques apparaissant dans les grammaires concrètes sont remplacées par d'autres éléments linguistiques dans la grammaire abstraite au moyen de règles de transformation (*transformation rules*) afin de conserver un ensemble de concepts élémentaires réduit. Ces transformations sont décrites dans les paragraphes relatifs aux modèles de la Rec. UIT-T Z.100. Elles sont formellement exprimées par des règles de réécriture.

La sémantique dynamique (*dynamic semantics*) n'est applicable qu'à des spécifications SDL syntaxiquement correctes qui satisfont aux conditions de validité structurelle. La sémantique dynamique définit l'ensemble des calculs associés à une spécification.

2.1 Grammaire

La grammaire (*grammar*) du langage SDL est formalisée au moyen d'une représentation grammaticale en formalisme de Backus-Naur (BNF). La grammaire Z.100 est cependant conçue de façon à être une grammaire de présentation, c'est-à-dire qu'elle n'est pas destinée à produire automatiquement un analyseur. Par ailleurs, certaines restrictions, qui finalement garantissent l'unicité de la sémantique, ne peuvent pas être exprimées en formalisme BNF mais ont été par contre indiquées dans le texte. La grammaire est donc définie au moyen du formalisme BNF et d'un peu de texte (essentiellement pour les règles de préséance). La traduction de la représentation SDL concrète en mode texte vers la représentation en syntaxe abstraite de la Rec. UIT-T Z.100 (appelée AS1) s'effectue en deux temps. Le premier pour

passer de la représentation SDL concrète en mode texte à la syntaxe abstraite AS0. Cette étape n'est pas définie formellement mais elle est déduite de la correspondance entre les deux grammaires, qui est presque univoque, et elle supprime les détails inapplicables comme les séparateurs et les règles lexicales. La deuxième étape, de traduction de syntaxe AS0 en syntaxe AS1, est formellement prise en charge par un ensemble de règles de transformation (voir Annexe F2).

2.2 Conditions de validité structurelle

Les conditions de validité structurelle (*well-formedness conditions*) définissent des contraintes additionnelles qu'une spécification doit respecter. Ces contraintes ne peuvent pas être exprimées en formalisme BNF mais sont statiques, c'est-à-dire qu'elles peuvent être définies et vérifiées indépendamment de la définition sémantique dynamique (voir Annexe F2). Une spécification SDL n'est *valide* que si, et seulement si, elle répond aux règles syntaxiques et aux conditions statiques du langage SDL. En fait, les conditions de validité structurelle se rapportent à la syntaxe concrète mais n'y ont pas été déclarées parce qu'elle ne sont pas exprimables dans une grammaire sans contexte.

Il existe 5 sortes principales de conditions de validité structurelle:

- règles de portée/visibilité (*scope/visibility rules*): la définition d'une entité introduit un identificateur qui peut servir de référence à cette entité. Seuls des identificateurs visibles peuvent être utilisés. Les règles de portée/visibilité sont appliquées afin de déterminer si la définition correspondante d'un identificateur est visible ou non;
- règles de désambiguïsation (disambiguation rules): un nom peut parfois se rapporter à plusieurs identificateurs. Des règles sont appliquées afin de découvrir l'identificateur correct;
- règles de cohérence de type de données (data type consistency rules): ces règles garantissent qu'aucune opération ne sera appliquée dynamiquement à des opérandes qui ne correspondent pas à ses types d'argument. Plus précisément, le type de données d'un paramètre réel doit toujours être compatible avec celui du paramètre formel correspondant. Le type de données d'une expression doit toujours être compatible avec celui de la variable à laquelle cette expression est attribuée;
- règles spéciales (special rules): certaines règles sont applicables à des entités spécifiques. Par exemple, un bloc peut contenir des blocs locaux ou un graphe;
- règles syntaxiques normales (plain syntax rules): certaines règles sont applicables au bien-fondé de la syntaxe concrète et n'ont pas de contrepartie dans la syntaxe abstraite. Par exemple, les noms se trouvant au début et à la fin d'une définition doivent correspondre.

2.3 Règles de transformation

Dans le cas d'un langage possédant une syntaxe enrichie, il importe d'identifier les concepts fondamentaux qui correspondent aux intentions du concepteur de ce langage. D'autres réalisations linguistiques, qui sont introduites par commodité mais qui n'ajoutent pas à l'expressivité du langage (comme les notations abrégées), peuvent être remplacées à l'aide de ces concepts fondamentaux. Etant donné que les remplacements, qui sont décrits par des règles de transformation, peuvent être formalisés, il suffit de définir la sémantique dynamique pour les seuls concepts fondamentaux, ce qui en augmente la concision et l'intelligibilité.

Pour la définition sémantique formelle, le choix de concepts fondamentaux "corrects" est crucial. Si le nombre de concepts fondamentaux est trop grand, la sémantique dynamique sera moins concise et moins intelligible. Si les concepts fondamentaux sont trop peu nombreux ou erronés, les transformations tendent à être très complexes. L'orientation vers les objets a été introduite en 1992 dans le domaine d'application du langage SDL mais la définition sémantique (formelle) et la syntaxe abstraite ont continué à se fonder sur des instances sans faire intervenir la notion de classe: des règles de transformation très lourdes en ont résulté. Ce défaut a été corrigé dans la présente définition sémantique formelle.

La Rec. UIT-T Z.100 prescrit la transformation de spécifications SDL par une séquence d'étapes de transformation (*transformation steps*), dont chacune se compose d'un ensemble de transformations élémentaires comme indiqué dans les paragraphes relatifs au modèle et détermine la façon de traiter une classe spéciale de notations abrégées. Le résultat d'une étape donnée est utilisé comme entrée de l'étape suivante.

Pour formaliser les règles de transformation de la Rec. UIT-T Z.100, des règles de réécriture sont utilisées. Une transformation unique est réalisée par l'application d'une règle de réécriture à la spécification concrète, ce qui revient finalement à remplacer des parties de la spécification par d'autres parties comme défini par la règle (voir Annexe F2).

2.4 Sémantique dynamique

La sémantique dynamique (*dynamic semantics*) (Annexe F3) se compose des parties suivantes (voir Figure F1-2):

- l'automate abstrait SDL (SAM[CP__26426], SDL abstract machine), qui est défini au moyen d'un automate abstrait à états finis (ASM). La définition de l'automate SAM est subdivisée en trois parties correspondant à la syntaxe abstraite:
  1) concepts de flux de signaux de base (signaux, temporisations, exceptions, portes, canaux);
  2) divers types d'agent d'automate ASM (modélisant les agents SDL correspondants);
  3) primitives de comportement (instructions d'automate SAM).
- la compilation, qui est définie comme étant une fonction appliquée dans l'arbre de syntaxe abstraite d'une spécification SDL. La compilation aboutit à des ensembles de primitives de comportement qui modélisent les actions des agents SDL.
- l'initialisation, qui définit un état préinitial d'un système et plusieurs programmes d'initialisation. L'état initial de système est donc atteint par la création de l'agent de système SDL et par l'activation de cet agent dans l'état préinitial. L'initialisation déploie de façon récursive la structure statique du système, créant ainsi de nouveaux agents SDL comme spécifié. En fait, le même processus peut être lancé dans la phase ultérieure d'exécution, à chaque création d'agents SDL. De ce point de vue, l'initialisation décrit seulement l'instanciation de l'agent du système SDL.
- la sémantique des données, qui est séparée du reste de la sémantique par une interface. L'utilisation d'une interface est intentionnelle à ce stade car elle permet de changer le modèle de données si, pour un domaine donné, un autre modèle de données est plus approprié que le modèle incorporé. Par ailleurs, le modèle incorporé peut également être modifié de cette façon sans affecter le reste de la sémantique.

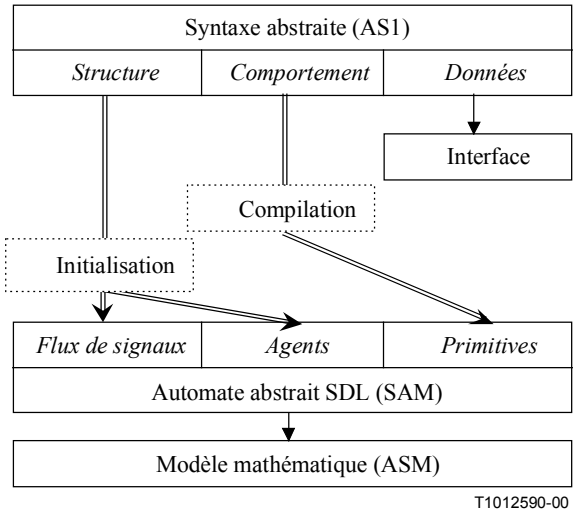


Figure F1-2/Z.100 – Aperçu général de la sémantique dynamique

La sémantique dynamique est formalisée à partir de la syntaxe abstraite (AS1) du langage SDL, de laquelle on déduit un modèle de comportement pour les spécifications SDL. L'approche retenue ici est fondée sur une vue opérationnelle abstraite utilisant le formalisme d'automate ASM comme cadre mathématique sous-jacent afin d'obtenir une définition sémantiquement rigoureuse du modèle d'automate SAM. La compilation définit un compilateur abstrait qui applique les parties comportementales du langage SDL sur un code abstrait (sémantique dénotationnelle). Finalement, l'initialisation décrit une interprétation de l'arbre syntaxique abstrait afin de construire la structure initiale du système (sémantique opérationnelle).

La sémantique dynamique (*dynamic semantics*) associe à chaque spécification SDL un automate ASM particulier, réparti en temps réel, qui se compose logiquement d'un ensemble d'agents autonomes qui coopèrent afin d'effectuer des passes d'automate concurrentes. Le comportement d'un agent est déterminé par un programme d'automate ASM comportant une règle de transition. Collectivement, ces règles définissent l'ensemble des passes possibles de l'automate. Chaque agent possède sa propre vue partielle d'un état global, qui est défini par un ensemble de fonctions et de domaines statiques et dynamiques. Des intersections non vides de vues partielles permettent de modéliser l'interaction entre agents. Le § 3 présente une introduction au modèle d'automate ASM et la notation utilisée dans les Annexes F1, F2 et F3.

3 Automates abstraits à états finis (ASM)

Ce paragraphe explique les notions et concepts de base des automates abstraits à états finis (ASM, *abstract state machines*) ainsi que la notation utilisée dans la présente annexe pour définir le modèle d'automate abstrait SDL (SAM). L'objectif visé ici est d'assurer une compréhension intuitive du formalisme. On trouvera dans les références [3] et [4] une définition rigoureuse des fondements mathématiques de l'automate ASM et dans la référence [2] un examen de la motivation et de l'opportunité du choix du cadre sémantique utilisé dans le présent contexte. D'autres références sur les données associées aux automates ASM peuvent être consultées dans les pages ASM du web [1].

Le modèle d'automate ASM utilisé pour définir la sémantique dynamique du langage SDL est expliqué en plusieurs étapes. La première traite du modèle d'automate ASM de base (*basic ASM model*) à agent unique (§ 3.1). Ce modèle est ensuite étendu aux systèmes à agents multiples (*multi-agent systems*) (§ 3.2). L'on aborde ensuite les systèmes ouverts (*open systems*), c'est-à-dire interagissant avec un environnement qu'ils ne peuvent pas régir, en ajoutant la notion de monde extérieur (*external world*) (§ 3.3). Finalement, le modèle est étendu par introduction d'une notion de comportement en temps réel (*real-time behaviour*) (§ 3.4). Un modèle d'automate ASM pour système unique est peu à peu élaboré pour illustrer ces étapes. Le modèle final d'automate ASM de ce système est résumé au § 3.5. Une notation additionnelle, permettant de définir la sémantique dynamique du langage SDL, est exposée au § 3.6.

Exemple (description informelle):

Afin d'illustrer le modèle d'automate ASM, l'on définit un simple système de gestion de ressource (RMS, *resource, management system*), composé d'un groupe de n >1 agents (*group of n >1 agents*) en concurrence pour l'obtention d'une ressource (*resource*) (par exemple un dispositif ou un service). Ce système est caractérisé comme suit de manière informelle:

- un ensemble de m jetons (*tokens*), $m < n$, utilisés pour accorder un accès exclusif (*exclusive*) ou non exclusif (partagé) (*non-exclusive (shared)*) à la ressource;
- selon que le mode d'accès recherché est exclusif ou partagé, un agent doit posséder tous les jetons ou un seul jeton, selon le cas, avant de pouvoir accéder à la ressource;
- un agent est au repos (*idle*) lorsqu'il n'est pas en concurrence pour une ressource; il est en attente (*waiting*) lorsqu'il essaie d'obtenir l'accès à la ressource ou occupé (*busy*) bien que possédant le droit d'accéder à la ressource;
- une fois qu'un agent est en attente, il reste dans cet état jusqu'à ce qu'il obtienne l'accès à la ressource;
- un agent occupé libère la ressource lorsqu'il n'a plus besoin de celle-ci, comme indiqué par un état d'arrêt (*stop condition*) qui est défini de façon externe pour cet agent. Lors de la libération de la ressource, tous les jetons détenus par l'agent sont restitués;
- les états d'arrêt ne sont indiqués que lorsqu'un agent est occupé. Il s'agit d'une contrainte d'intégrité (*integrity constraint*) appliquée au comportement du monde extérieur;
- initialement, les agents sont au repos et tous les jetons sont disponibles.

Le système sera défini étape par étape au fur et à mesure du déroulement des explications du modèle d'automate ASM, à partir du modèle ASM de base avec agent unique. Le modèle ASM final de ce système est résumé au § 3.5.

3.1 Automate ASM à agent unique

Un automate abstrait à états finis (*Abstract State Machine*) M est défini dans un vocabulaire (*vocabulary*) V donné par ses états (*states*) S, par ses états initiaux (*initial states*) $S_0 \subseteq S$ et par son programme (*program*) P. Ces éléments seront expliqués dans les paragraphes suivants.

3.1.1 Vocabulaire

Le vocabulaire (ou la signature) V correspond à un ensemble fini de noms de fonction (*function names*), de noms de prédicat (*predicate names*) et de noms de domaine (*domain names*), chacun ayant une arité fixe. Les noms contenus dans le vocabulaire V sont classés en basiques (*basic*) ou dérivés (*derived*) puis raffinés en statiques (*static*) ou dynamiques (*dynamic*) (voir Figure F1-3). La signification associée à ces classifications sera expliquée dans les paragraphes suivants.

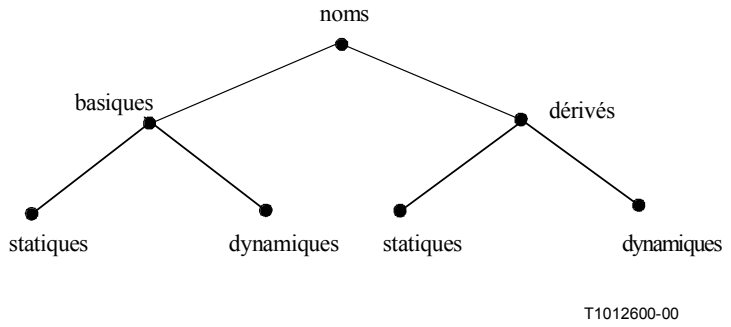


Figure F1-3/Z.100 – Classification des noms d'automate ASM

Le vocabulaire *V* est déclaré lors de la définition d'un automate ASM, sauf pour un sous-ensemble de noms prédéfinis (*predefined names*). Ce sous-ensemble contient, par exemple, le signe d'égalité, les noms de prédicat à arité nulle *True, False*, le nom de fonction à arité nulle *undefined,* les noms de domaine *BOOLEAN, NAT* et *REAL*, ainsi que les noms des fonctions normales fréquemment utilisées (comme les opérateurs booléens $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$ et les opérateurs d'ensembles $\subseteq$, $\cup$, $\cap$, $\in$, $\notin$, etc.). Les noms prédéfinis sont énumérés au § 3.6.

Pour déclarer (*declare*) des noms lors de la définition d'un automate ASM concret, l'on utilisera les conventions nationales suivantes:

- les noms de domaine (*domain names*) sont écrits en lettres capitales (comme dans *AGENT*) sauf pour indiquer un non-terminal de la grammaire abstraite. Ici, les noms de domaine sont écrits comme les non-terminaux, c'est-à-dire en italiques et entre parenthèses et avec la première lettre en majuscule. Un nom de domaine *D* est déclaré par *domain D*;
- les noms de fonction (*function names*) sont écrits en italiques sans majuscule à la première lettre (comme dans *mode*). Un nom de fonction *f* est déclaré par f: $D_1 \times D_2 \times ... \times D_n \rightarrow D_0$, où n est la parité de f, et où $D_0, D_1, D_2, ..., D_n$ sont des noms de domaine;
- les noms de prédicat (*predicate names*) sont également écrits en italiques, comme dans *Available*). Un nom de prédicat *P* est déclaré par P: $D_1 \times D_2 \times ... \times D_n \rightarrow$ *BOOLEAN*;
- les noms statiques basiques (*basic static names*) sont qualifiés par le mot clé **static** lorsqu'ils sont déclarés (voir Figure F1-3);
- les noms dynamiques basiques (*basic dynamic names*) sont qualifiés par un des mots clés **controlled, shared** ou **monitored**, lorsqu'ils sont déclarés (comme cela sera expliqué au § 3.3);
- les noms sans mot clé antérieur sont par défaut des noms dérivés (*derived names*) (voir la Figure F1-3).

Exemple (Vocabulaire):

Pour définir un modèle d'automate ASM du système *RMS*, supposons un vocabulaire *V* comportant les noms suivants:

static domain *AGENT*
static domain *TOKEN*
domain *MODE*

shared *mode*: *AGENT* → *MODE*
controlled *owner*: *TOKEN* → *AGENT*
static *ag*: → *AGENT*

Idle: *AGENT* → *BOOLEAN*
Waiting: *AGENT* → *BOOLEAN*
Busy: *AGENT* → *BOOLEAN*
Available: *TOKEN* → *BOOLEAN*

monitored *Stop*: *AGENT* → *BOOLEAN*

Les noms de domaine statiques *AGENT*, *TOKEN* et *MODE* sont introduits afin de représenter l'agent (unique) du système, l'ensemble des jetons, et les différents modes d'accès (exclusif, partagé), respectivement. Les noms de fonction *mode* et *owner* indiquent des fonctions dynamiques. Ils sont utilisés pour présenter le mode d'accès actuel d'un agent et le détenteur actuel d'un jeton, respectivement. Le nom de fonction d'arité nulle *ag* renvoie à une valeur du domaine *AGENT*. *Idle*, *Waiting*, *Busy* et *Available* sont des noms de prédicats dynamiques dérivés. *Stop* indique un prédicat surveillé, ce qui sera expliqué ultérieurement.

3.1.2 Etats

On indique un état $s \in S$ en attribuant une signification, également appelée "interprétation" (*interpretation*), aux noms contenus dans le vocabulaire V dans un ensemble infini qui est appelé "série de base de M" (*base set of M*) auquel on fera référence par le nom de domaine prédéfini X [1]. En d'autres termes, l'on doit associer respectivement à chaque nom de domaine, de fonction et de prédicat du vocabulaire V un domaine, une fonction ou un prédicat de base. L'interprétation des noms dérivés découle de celle des noms basiques. Noter que la série de base est la même pour tous les états de l'automate M. Il est requis que les prédicats *True, False* et *undefined* qualifient des éléments distincts de la série de base. Les opérations prédéfinies ont leur interprétation habituelle.

Il convient de ne pas perdre de vue que les noms sont classés comme étant statiques ou dynamiques. S'ils sont classés comme étant statiques, les noms sont appelés à avoir la même interprétation dans tous les états de l'automate M. Sinon, ils peuvent avoir différentes interprétations dans différents états de M. Les états S de M sont donc indiqués par l'ensemble de toutes les interprétations des noms contenus dans le vocabulaire V pour la série de base de M, qui sont conformes à ces contraintes et à d'autres contraintes explicitement déclarées.

A vrai dire, toutes les fonctions sont totales (*total*) pour la série de base de M. De façon à simuler des fonctions partielles (*partial functions*), les valeurs "indéfinies" des fonctions sont étiquetées par l'élément distinctif *undefined*. Les prédicats ne fournissent qu'une seule des valeurs *True* ou *False*, c'est-à-dire qu'ils ne doivent jamais être partiels.

Chaque état possède un nombre théoriquement infini d'éléments de réserve (*reserve elements*) permettant des extensions dynamiques des domaines (voir § 3.1.6). Par définition, les éléments de réserve d'un état sont tous les éléments de la série de base qui ne sont ni désignés par une fonction ni contenus dans un des domaines.

3.1.3 Noms dérivés

La signification des noms dérivés découle de l'interprétation des noms basiques. Elle est définie en termes de formules (*formulae*) (voir § 3.6). Les noms dérivés peuvent donc être interprétés comme des abréviations. Soit *DerivedName* un nom d'arité n et soit $Formula(v_1,...,v_n)$ une formule du domaine D avec les variables libres $v_1,...,v_n$ des domaines $D_1,...,D_n$, $n \geq 0$. La forme générale d'une définition de nom dérivé (*derived name definition*) est la suivante:

$DerivedNameDefinition ::= DerivedName(v_1:D_1,...,v_n:D_n):D =_{def} Formula(v_1,...,v_n)$

Le domaine résultant D est omis dans le cas d'une définition de domaine dérivé.

Exemple (Définitions):

Les prédicats dérivés ci-après sont définis de façon à se rapporter à l'état d'un agent/jeton dans un état donné:

$MODE =_{def} \{exclusive, shared\}$

$Idle(a:AGENT): BOOLEAN$	$=_{def}$	$a.mode = undefined \wedge \forall t \in TOKEN: t.owner \neq a$
$Waiting(a:AGENT): BOOLEAN$	$=_{def}$	$a.mode \neq undefined \wedge \forall t \in TOKEN: t.owner \neq a$
$Busy(a:AGENT): BOOLEAN$	$=_{def}$	$a.mode \neq undefined \wedge \exists t \in TOKEN: t.owner = a$
$Available(t:TOKEN): BOOLEAN$	$=_{def}$	$t.owner = undefined$

Un agent a est par exemple au repos si la fonction *mode* fournit la valeur *undefined* pour cet agent a et si celui-ci ne détient aucun jeton. Un jeton t est disponible si aucun agent ne le détient.

Afin d'améliorer la lisibilité, l'on utilisera une notation (*notation*) de type "." pour désigner les fonctions et prédicats unaires. Par exemple, la forme *a.mode* sera équivalente de *mode(a)*.

3.1.4 Etats initiaux

L'ensemble des états initiaux (*initial states*) $S_0 \subseteq S$ est défini par des contraintes imposées à des domaines, fonctions et prédicats associés aux noms contenus dans le vocabulaire V. Les contraintes initiales pour les domaines et opérations prédéfinis sont déclarées implicitement. Voir § 3.6. Les contraintes initiales sont de la forme générale suivante:

initially *ClosedFormula*

[1] Formellement, les états d'automate ASM sont des structures du premier ordre triées en logique multisorte (*many-sorted first-order structures*).

Exemple (Etats initiaux):

Les contraintes suivantes définissent l'ensemble des états initiaux du système *RMS*:

initially $AGENT = \{ag\}$
initially $\forall a \in AGENT: a.Idle \wedge \forall t \in TOKEN: t.Available$

La première contrainte définit l'ensemble initial *AGENT* comme consistant en un seul élément *ag*. La seconde contrainte exprime le fait que, initialement l'agent du système *RMS* est à l'état de repos (*a.mode = undefined*) et que tous les jetons sont disponibles (*t.owner = undefined*). Noter qu'aucune contrainte n'est définie pour l'élément *Stop*.

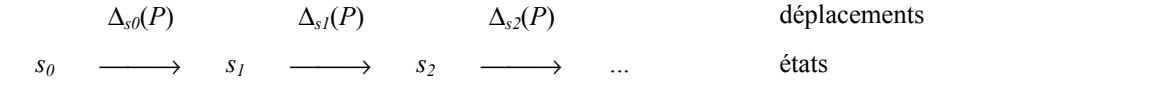
3.1.5 Transitions d'état et passes

L'on se rappelle qu'un état (global) $s \in S$ est donné par une interprétation des noms contenus dans le vocabulaire *V* pour la série de base de *M*. Les transitions d'état peuvent être définies en termes de réinterprétations partielles de domaines, fonctions et prédicats dynamiques. Cela fait apparaître les notions d'emplacement (*location*) comme étant un moyen théorique de faire référence à des parties d'états globaux et à la notion de mise à jour (*update*) comme un moyen de décrire des changements d'état.

Un emplacement d'état *s* d'automate *M* (*location of a state s of M*) est une paire $loc_s = <f, s(x)>$, où *f* est un nom dynamique contenu dans le vocabulaire *V* et où $s(x)$ est une séquence d'éléments de la série de base selon la parité de *f*. Une mise à jour d'état *s* (*update of s*) est une paire $\delta_s = <loc_s, s(y)>$, où $s(y)$ désigne un élément de la série de base en tant que nouvelle valeur à associer à l'emplacement loc_S. Appliquer (*fire*) δ_s signifie transformer *s* en un état *s′* de *M* tel que $f_s(s(x)) = s(y)$, alors que tous les autres emplacements loc'_s of *s*, $loc'_s \neq loc_s$, restent inchangés. En d'autres termes, l'application d'une mise à jour modifie de façon bien définie l'interprétation d'un état.

Le comportement possible d'un automate ASM de base est représenté par un programme (*program*) *P*, qui est défini par une règle de transition (*transition rule*) (voir § 3.1.6 et 3.1.8). Pour chaque état $s \in S$, un programme *P* de *M* définit un ensemble de mise à jour (*update set*) $\Delta_s(P)$ comme étant un ensemble fini de mises à jour de l'état *s*. L'ensemble $\Delta_s(P)$ n'est cohérent (*consistent*) que si et seulement s'il ne contient pas deux mises à jour δ_s, δ'_s telles que $\delta_s = <loc_s, s(y)>$, $\delta'_s = <loc_s, s(y')>$, et $s(y) \neq s(y')$. L'application d'un ensemble de mise à jour cohérent (*firing of a consistent update set*) $\Delta_s(P)$ se trouvant dans l'état *s* implique l'application simultanée de tous ses membres, c'est-à-dire la production (en une seule étape atomique) d'un nouvel état *s′* tel que, pour tous les emplacements $loc_s = <f,s(x)>$ de *s*, $f_{s'}(s(x)) = s(y)$, si $<<f,s(x)>,s(y)> \in \Delta_s(P)$, et sinon $f_{s'}(s(x)) = f_s(s(x))$: ce déblocage est appelé "transition d'état" (*state transition*). L'application d'un ensemble de mise à jour non cohérent[2] reste sans effet, c'est-à-dire que $s' = s$.

Le comportement d'un automage ASM *M* à agent unique est modélisé par des passes (finies ou infinies) de l'automate *M* (*runs of M*) où une passe (*run*) est une séquence de transitions d'état ayant la forme suivante:



de façon que $s_0 \in S_0$ et s_{i+1} soient obtenus à partir de s_i, pour $i \geq 0$, par déblocage de l'ensemble $\Delta_{si}(P)$ pour les s_i, où $\Delta_{si}(P)$ indique un ensemble de mise à jour défini par le programme *P* de *M* pour les s_i (voir § 3.1.8). Un automate ASM est défini comme étant l'ensemble de toutes ses passes. Par la suite, l'on limitera la visibilité aux passes commençant dans un état initial, également appelées "passes normales" (*regular runs*).

3.1.6 Règles de transition

Les règles de transition spécifient des ensembles de mise à jour pour des états d'automate ASM. Des règles complexes sont formées à partir de règles élémentaires faisant appel à divers constructeurs de règle. La forme élémentaire d'une règle de transition est appelée "instruction de mise à jour" (*update instruction*).

- instruction de mise à jour (*update*)

  Rule ::= f(t1,...,tn) := t0 (n ≥ 0)

  Ici, *f* est un nom non statique du vocabulaire *V* désignant une fonction, un prédicat ou un domaine commandé ou partagé. Et $t_0,t_1,...,t_n$ sont les termes du vocabulaire *V* qui désigne respectivement, pour un état *s* donné, l'emplacement $loc = <f,<s(t_1),..., s(t_n)>>$ à modifier ainsi que la nouvelle valeur $s(t_0)$ à attribuer. En d'autres termes,

---

[2] Dans le contexte de la sémantique du langage SDL, un ensemble de mise à jour incohérent indique une erreur dans le modèle sémantique. La sémantique d'automate ASM garantit que de telles erreurs ne détruiront pas la notion d'état.

l'instruction de mise à jour ci-dessus spécifie l'ensemble de mise à jour $\{<<f,<s(t_1),..., s(t_n)>>, s(t_0)>\}$, qui se compose d'une seule mise à jour. Noter que seuls les emplacements correspondant à des noms basiques (non statiques) peuvent apparaître à gauche d'une instruction de mise à jour.

Exemple (Instruction de mise à jour):

Soit *t* une variable désignant un jeton et *ag* un agent.

t.owner := *ag* spécifie l'ensemble de mise à jour {<<*owner*, <*s*(*t*)>>, *s*(*ag*)>}

ag.mode := *undefined* spécifie l'ensemble de mise à jour {<<*mode*, <*s*(*ag*)>>, *s*(*undefined*)>}

La construction de règles de transition complexes à partir d'instructions élémentaires de mise à jour est définie par une suite récursive de constructeurs de règle ASM (*ASM rule constructors*). Dans le cas de l'application du modèle ASM pour définir la sémantique SDL, 6 constructeurs différents sont utilisés. Leur liste est reproduite ci-dessous avec une description informelle de leur signification. Dans cette liste, les notations *Rule*, $Rule_i$ désignent des règles de transition, *g* indique un terme booléen et $v,v_1,...,v_n$ désignent des variables libres dans la série de base de l'automate *M*. La portée d'un constructeur de règle est exprimée par des mots clés appropriés. Elle peut également être indiquée par un renfoncement. Les mots clés de fermeture peuvent être omis si aucun malentendu n'en découle. S'ils sont omis, le constructeur correspondant s'étend autant que possible mais pas sur le paragraphe **where** suivant.

- constructeur d'opération si-alors (*if-then*)

  Rule ::= **if** *g* **then**
  $Rule_1$
  [**else**
  $Rule_2$]
  endif

  L'ensemble de mise à jour spécifié par la règle dans un état donné *s* est défini comme étant l'ensemble de mise à jour de l'élément $Rule_1$ ou $Rule_2$, selon la valeur de *g* dans l'état *s*. Sans la partie facultative **else**, l'ensemble de mise à jour défini par *Rule* est celui de l'élément $Rule_1$ ou l'ensemble de mise à jour vide. L'opérateur **elseif** est parfois utilisé comme abréviation de **else if**.

- constructeur d'opération en parallèle (*do-in-parallel*)

  Rule ::= [**do in-parallel**]
  $Rule_1$
  ...
  $Rule_n$
  [**enddo**]

  L'ensemble de mise à jour défini par la règle *Rule* dans l'état *s* est défini comme étant la réunion logique des ensembles de mise à jour des règles $Rule_1$ à $Rule_n$. En d'autres termes, il n'y a pas de rapport avec l'ordre dans lequel les règles de transition se rapportant au même bloc sont déclarées. Par souci de concision, les mots clés **do in-parallel** et **enddo** peuvent être omis, si aucune confusion n'en découle. Un programme d'automate ASM apparaît donc souvent comme un ensemble de règles plutôt que comme une règle de bloc monolithique.

- constructeur d'opération généralisée (*do-forall*)

  Rule ::= **do forall** *v*: *g*(*v*)
  $Rule_0(v)$
  enddo

  L'effet de cette règle *Rule* est que la règle $Rule_0$ est appliquée simultanément pour tous les éléments *v* de la série de base de *M* pour lesquels la condition booléenne *g*(*v*) est vérifiée dans l'état *s*, où *v* est une variable libre dans la règle $Rule_0$. Plus précisément, l'ensemble $\Delta_s(Rule)$ est la réunion logique de tous les ensembles de mise à jour $\Delta_s(Rule_0(v))$ telle que la condition *g*(*v*) soit vérifiée dans l'état *s*. Compte tenu du fait que les ensembles de mise à jour doivent être finis, la condition *g*(*v*) ne doit être vérifiée que pour un nombre fini de valeurs.

- constructeur d'opération de choix (*choose*)

  Rule ::= **choose** *v: g*(*v*)
  $Rule_0(v)$
  endchoose

  L'effet de cette règle *Rule* est que la règle $Rule_0$ est appliquée pour un certain élément *v* de la série de base de *M* pour lequel la condition *g*(*v*) est vérifiée dans l'état *s*, où *v* est une variable libre dans la règle $Rule_0$. Plus

précisément, l'ensemble $\Delta_s(Rule)$ est un certain ensemble de mise à jour $\Delta_s(Rule_0(v))$ tel que la condition $g(v)$ est vérifiée dans l'état s, ou est l'ensemble de mise à jour vide si aucun élément v n'existe.

- constructeur d'opération d'extension (*extend*)[3]

  Rule ::= **extend** *D* **with** $v_1,...,v_n$
  $Rule_0(v_1,...,v_n)$
  endextend

  L'effet de la règle *Rule*, lorsqu'elle est appliquée dans l'état s, est que n éléments de réserve de l'état s (voir § 3.1.2) sont importés dans le domaine dynamique D (alors qu'ils sont retranchés de la réserve), que les éléments $v_1,...,v_n$ se lient chacun à l'un des éléments importés et enfin que la règle $Rule_0(v_1,...,v_n)$ est appliquée.

  Le constructeur extend peut servir à simuler des définitions d'automate ASM orientées vers les objets, dans laquelle ceux-ci sont créés dynamiquement. Pour chaque objet à créer un élément extrait de la réserve est donc attribué au domaine correspondant et initialisé.

- constructeur d'opération d'expression *let*

  Rule ::= **let** *v* = *expression* **in**
  $Rule_0(v)$
  endlet

  L'effet de la règle *Rule*, lorsqu'elle est appliquée dans un certain état s, est que l'élément v est lié à la valeur de l'expression *expression*, et que la règle $Rule_0$ est appliquée par cette valeur.

Exemple (Règle de transition):

La règle de transition suivante définit le comportement de l'agent *ag* lorsqu'il demande un accès partagé, c'est-à-dire lorsque *ag.mode = shared*. Cette règle applique le constructeur de relation si-alors, le constructeur de choix et une instruction de mise à jour.

if *ag.mode = shared* $\wedge$ *ag.Waiting* **then**
  choose *t*: $t \in$ TOKEN $\wedge$ *t.Available*
    t.owner := *ag*
  endchoose
endif

La signification précise de la règle est donnée par son ensemble de mise à jour par rapport à un état s, qui est soit l'expression {<<*owner*, <$s(t)$>>, $s(ag)$>} pour un certain jeton $s(t)$ disponible dans l'état s à condition que tous les autres prédicats déclarés dans le constructeur si-alors soient vérifiés dans l'état s; sinon par l'ensemble de mise à jour vide.

3.1.7 Abréviations

Les règles peuvent être structurées au moyen d'abréviations (*abbreviations*) composées de macro-instructions (*rule macros*) et de noms dérivés (*derived names*), lesquels peuvent posséder des paramètres. Ces abréviations permettent d'établir des définitions hiérarchiques et de raffiner progressivement des règles complexes, ce qui facilite la compréhension des définitions du modèle d'automate ASM.

Les noms dérivés sont introduits comme expliqué dans les § 3.1.1 et 3.1.3, c'est-à-dire par déclaration et définition; ou, en variante sous forme compacte, par combinaison de la déclaration et de la définition.

- définition de macro-instruction (*rule-macro*)

  Admettons que la règle $Rule_0$ indique une règle de transition avec les variables libres $v_1,...,v_n$ des domaines $D_1,...,D_n$, $n \geq 0$. La forme générale d'une définition de macro-instruction est la suivante:

  RuleMacroDefinition ::= $RuleMacroName(v_1{:}D_1,...,v_n{:}D_n) \equiv$
  $Rule_0(v_1,...,v_n)$

  Les noms des macro-instructions sont, par convention, écrits en petites capitales, avec une lettre initiale majuscule (comme dans SHAREDACCESS).

---

[3] A vrai dire, le constructeur *extend* peut être défini en termes de constructeur d'opération d'importation *import* (mais il n'est pas représenté ici car le constructeur *import* n'est pas utilisé dans la présente annexe).

- Partie *where*

  Par défaut, les macro-instructions (*rule macros*) et les noms dérivés (*derived names*) ont une portée globale. Cependant, leur portée peut aussi être limitée à une règle (*Rule*) de transition particulière, au moyen de la partie *where*.

  Rule ::= $Rule_0$

  where

  (*RuleMacroDefinition* | *DerivedNameDefinition*)$^+$

  endwhere

- constructeur de macro-instruction (*rule-macro*)

  Les macro-instructions sont appliquées comme suit dans les règles de transition:

  Rule ::= *RuleMacroName*(t_1,...,t_n)

  Formellement, les macro-instructions sont des abréviations syntaxiques, c'est-à-dire que chaque apparition d'une macro dans une règle doit être remplacée en mode texte par la définition de macro correspondante (avec remplacement des paramètres formels par des paramètres réels).

Exemple (Macro-instruction):

La règle de transition tirée de l'exemple précédent peut être exprimée au moyen de macro-instructions ou être définie comme étant elle-même une macro-instruction. Ici, la règle SHAREDACCESS est une macro-définition de portée globale qui peut être utilisée à d'autres emplacements de la définition du modèle d'automate ASM. La règle GETTOKEN est une macro-définition paramétrée dont la portée locale est limitée à la règle SHAREDACCESS, avec le paramètre formel *a*. Lorsque la règle GETTOKEN est appliquée dans la définition SHAREDACCESS, le paramètre *a* est remplacé par le paramètre réel *ag*.

SHAREDACCESS ≡
if *ag.mode = shared* ∧ *ag.Waiting* **then**
GETTOKEN(*ag*)
endif
where
GETTOKEN(*a*:*AGENT*) ≡
choose *t*: *t* ∈ *TOKEN* ∧ *t.Available*
t.owner := *a*
endchoose
endwhere

3.1.8 Programmes d'automate ASM à agent unique

Un programme d'automate ASM (*ASM program*) *P* est donné par une règle de transition (*transition rule*) encadrée (ou règle (*rule*) par concision) de la forme suivante:

Rule

Comme cela a déjà été mentionné, les définitions de macro-instruction peuvent avoir une portée locale ou globale. Pour avoir une portée globale, les macrodéfinitions peuvent être appliquées à l'extérieur du programme d'automate ASM et peuvent donc être appliquées également dans le cadre de ce programme.

Dans le modèle d'automate ASM de base, il n'existe qu'un seul programme ASM, qui est associé statiquement à un agent défini implicitement afin d'exécuter ce programme. Dans le paragraphe suivant, l'on permettra de définir plusieurs programmes ASM et de les associer à différents agents qui sont introduits dynamiquement au cours de passes d'automate abstraites.

Exemple (Programme d'automate ASM):

Le programme *P* d'automate ASM du système *RMS* est défini comme suit:

```
do in-parallel
    SHAREDACCESS
    EXCLUSIVEACCESS
    RELEASEACCESS
enddo
where
    SHAREDACCESS ≡
        if ag.mode = shared ∧ ag.Waiting then
            choose t: t ∈ TOKEN ∧ t.Available
                t.owner := ag
            endchoose
        endif
    EXCLUSIVEACCESS ≡
        if ag.mode = exclusive ∧ ∀t ∈ TOKEN: t.Available then
            do forall t: t ∈ TOKEN
                t.owner := ag
            enddo
        endif
    RELEASEACCESS ≡
        if ag.Busy ∧ ag.Stop then
            do in-parallel
                ag.mode := undefined
                do forall t: t ∈ TOKEN ∧ t.owner = ag
                    t.owner := undefined
                enddo
            enddo
        endif
endwhere
```

Le programme d'automate ASM est défini par une règle de transition unique, comme indiqué dans le cadre. Cette règle de transition utilise le constructeur d'opération en parallèle et 3 macro-instructions, ce qui aboutit à une définition de règle hiérarchique.

3.2 Automate ASM à agents multiples

La modélisation mathématique de systèmes concurrents et réactifs nécessite une extension du modèle d'automate ASM de base. Dans le présent paragraphe, l'on expliquera le concept d'automate réparti (*distributed ASM*), qui généralise le modèle ASM de base qui est présenté au § 3.1.

Un automate à états abstraits répartis (*distributed Abstract State Machine*) *M* est défini dans un vocabulaire (*vocabulary*) *V* donné par ses états (*states*) *S*, par ses états initiaux (*initial states*) $S_0 \subseteq S$, par ses agents *A* et par ses programmes (*programs*) *P*. Ces éléments seront développés dans les paragraphes qui suivent, dans la mesure où ils diffèrent du modèle d'automate ASM de base.

3.2.1 Vocabulaire

Le vocabulaire *V* d'un automate ASM à agents multiples *M* contient des noms distinctifs de domaine:

controlled domain *AGENT*

static domain *PROGRAM*

représentant respectivement un ensemble dynamique (*dynamic*) *A* d'agents et un ensemble invariant *P* de programmes ASM. Par ailleurs, le vocabulaire *V* contient un nom distinctif de fonction:

controlled *program*: *AGENT* → *PROGRAM*

ainsi qu'une fonction nullaire spéciale *Self* (voir § 3.2.2).

3.2.2 Agents et passes

Un automate réparti *M* peut avoir un nombre fini quelconque d'agents, ce nombre pouvant varier dynamiquement en fonction de l'état considéré. Le comportement de chaque agent est déterminé par un certain programme de *M*, défini par une règle de transition comme dans le modèle ASM de base. Les agents opèrent concurremment lorsqu'ils exécutent leurs programmes et ils interagissent asynchroniquement au moyen d'emplacements d'état partagés globalement, c'est-à-dire que deux ou plus de deux agents peuvent lire et écrire au même emplacement. Les étapes d'exécution concurrentes du modèle d'automate ASM réparti sont limitées à des opérations indépendantes, dans lesquelles le comportement admissible est défini en termes de passes partiellement ordonnées (*partially ordered runs*) (voir [3]). Intuitivement, cette notion de concurrence permet de formuler une concurrence vraie au lieu d'une approximation de concurrence au moyen d'un modèle d'entrelacement.

Pour attribuer un comportement à un agent de l'automate *M*, la fonction distinctive *program* (voir § 3.2.1) fournit, pour chaque agent *a* de *M*, le programme de *P* que l'agent *a* doit exécuter. La fonction *program* permet donc de définir (ou de redéfinir) dynamiquement le comportement d'agents. Il est donc possible de créer de nouveaux agents au moment de l'exécution. Dans un état *s* donné de *M*, les agents de *M* sont tous les éléments *a* de *s* tels que l'expression *a.program* désigne un comportement (tel que défini par un certain programme de *P*) qu'il y a lieu d'associer à *a*.

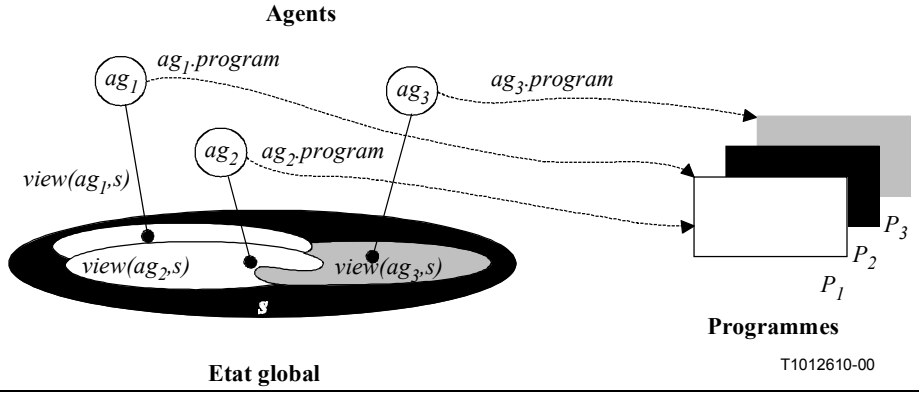
Une fonction nullaire spéciale *Self* sert d'autoréférence (*self reference*) pour identifier l'agent particulier qui appelle la fonction *Self*:

monitored *Self:* $\rightarrow$ *AGENT*

La fonction *Self* possède une interprétation différente pour chaque agent. L'emploi de *Self* comme argument additionnel de fonction permet à chaque agent *a* d'avoir sa propre vue partielle d'un état global donné de l'automate *M* auquel il applique la règle contenue dans le programme *a.program*.

Exemple (Schéma d'un automate ASM réparti):

Dans la figure suivante, un certain automate ASM réparti *M*, se composant de trois agents ag_1, ag_2, et ag_3 est illustré. La fonction *program* associe à chaque agent un des programmes d'automate ASM P_1, P_2, et P_3. Ici, ag_1 et ag_2 sont attribués au même programme. Le programme P_2 n'est actuellement associé à aucun agent. Cela peut cependant changer en cours d'exécution, car *program* est une fonction dynamique. Chaque agent a sa propre vue partielle (*partial view*) d'un état global donné *s* de *M*, auquel il applique la règle de son programme actuel. Dans la figure, cette vue est illustrée par la fonction *view*, qui fournit, pour chaque agent, son état local et son état partagé. En fait, la vue actuelle de chaque agent est déterminée implicitement par la définition du modèle d'automate ASM, y compris les programmes d'automate ASM.



Le modèle sémantique de concurrence sous-jacente dans le modèle ASM réparti définit le comportement en termes de passes partiellement ordonnées. Une passe partiellement ordonnée (*partially ordered run*) représente une certaine classe de passes d'exécution (admissible) de l'automate en limitant le non-déterminisme concernant l'ordre dans lequel les agents individuels peuvent effectuer leurs étapes de calcul, appelées "transferts" (*moves*). Pour éviter que des agents interfèrent les uns avec les autres, il suffit d'ordonner les transferts de différents agents s'ils sont en relation de dépendance causale (comme détaillé ci-dessous).

Passes partiellement ordonnées

Les transferts d'un agent individuel sont ordonnés linéairement tandis que ceux d'agents différents ne doivent être ordonnés que s'ils ne sont pas indépendants (*independent*) les uns des autres. Intuitivement, les transferts indépendants modélisent des actions concurrentes que ne peuvent pas être comparées quant à leur ordre d'exécution. L'acception précise de la notion d'indépendance est impliquée par la condition de cohérence contenue dans la définition formelle des passes partiellement ordonnées (adoptée à partir de [3]).

Une passe ρ d'un automate ASM réparti *M* est indiquée par une triple notation (Λ,*A*,σ) répondant aux quatre conditions suivantes:

1) Λ est un ensemble partiellement ordonné de transferts dont chaque élément n'a qu'un nombre fini de prédécesseurs;
2) *A* est une fonction appliquée sur l'ensemble Λ afin d'associer des agents à des transferts de façon que les transferts d'un agent unique quelconque de *M* soient ordonnés linéairement;
3) σ attribue un état de *M* à chaque segment initial *Y* de Λ, où σ(*Y*) est le résultat de l'exécution de tous les transferts dans le segment *Y*; si celui-ci est vide, alors $\sigma(Y) \in S_0$;
4) si *y* est un élément maximal contenu dans un segment initial *Y* de Λ et que $Z = Y - \{ y \}$, alors *A*(*y*) est un agent dans σ(*Z*) et σ(*Y*) est déduit de σ(*Z*) par application de *A*(*y*) sur σ(*Z*) (condition de cohérence) (*coherence condition*).

Implications

Des passes partiellement ordonnées possèdent certaines propriétés ou caractéristiques que l'on peut exprimer en termes de linéarisations (*linearisations*) d'ensembles partiellement ordonnés. La linéarisation d'un ensemble partiellement ordonné Λ est un ensemble partiellement ordonné Λ' ayant les mêmes éléments de façon que si $y < z$ dans Λ alors $y < z$ dans Λ'. En conséquence, le modèle sémantique de concurrence, impliqué par la notion de passe partiellement ordonnée, peut être caractérisé plus précisément comme suit [3]:

- Toutes les linéarisations du même segment initial fini d'une passe de *M* ont le même état final.
- Une propriété n'est vérifiée dans chaque état possible d'une passe ρ de *M* ρ que si et seulement si elle est vérifiée dans chaque état possible de chaque linéarisation de cette passe ρ.

3.2.3 Programmes d'automate ASM à agents multiples

Un automate réparti *ASM M* possède un ensemble fini *P* de programmes. Chaque programme $p \in P$ est donné par un nom de programme (*program name*) et par une règle de transition (*transition rule*) (ou par une règle (*rule*) pour plus de concision). Le nom de programme désigne de manière unique un programme *p* contenu dans *P* et est représenté par une fonction statique unaire[4]. Les programmes sont déclarés sous la forme suivante:

PROGRAMME D'AUTOMATE ASM:

Rule

Les noms de programme sont, par convention, reliés par des traits d'union et écrits en petites capitales, avec une majuscule à la première lettre (comme dans RESOURCE-MANAGEMENT-PROGRAM).

Par défaut, la contrainte implicite suivante s'applique:

initially PROGRAM = {PROGRAM_1,...,PROGRAM_n}

où PROGRAM_1,...,PROGRAM_n sont les noms des programmes qui sont définis dans le modèle d'automate ASM.

[4] A vrai dire, les noms de programme de l'automate *M* sont représentés par un ensemble distinctif d'éléments extraits de l'ensemble de base.

Exemple (Programme d'automate ASM):

Le programme d'automate ASM réparti du système *RMS* définit un programme unique comme suit:

RESOURCE-MANAGEMENT-PROGRAM:

```
do in-parallel
    SHAREDACCESS
    EXCLUSIVEACCESS
    RELEASEACCESS
enddo
where
        SHAREDACCESS ≡
                if Self.mode = shared ∧ Self.Waiting then
                    choose t: t ∈ TOKEN ∧ t.Available
                        t.owner := Self
                    endchoose
                endif
        EXCLUSIVEACCESS ≡
                if Self.mode = exclusive ∧ ∀t ∈ TOKEN: t.Available then
                    do forall t: t ∈ TOKEN
                        t.owner := Self
                    enddo
                endif
        RELEASEACCESS ≡
                if Self.Busy ∧ Self.Stop then
                    do in-parallel
                        Self.mode := undefined
                        do forall t: t ∈ TOKEN ∧ t.owner = Self
                            t.owner := undefined
                        enddo
                    enddo
                endif
endwhere
```

Le programme de l'automate ASM réparti porte le nom de RESOURCE-MANAGEMENT-PROGRAM. Il est défini comme le précédent programme d'automate ASM à agent unique, avec une seule différence: toutes les occurrences de *ag* ont été remplacées par des appels de la fonction *Self*. Cela permet d'associer le programme à différents agents pendant l'accès à l'état local de ces agents.

3.3 Le monde extérieur

A la suite d'une vue de système ouvert (*open system view*), les interactions entre un système et le monde extérieur, par exemple l'environnement dans lequel le système est intégré, sont modélisées en termes de divers mécanismes d'interface. Concernant la nature réactive des systèmes répartis, il importe d'identifier clairement et de déclarer précisément:

- les conditions préalables imposées au comportement attendu du monde extérieur;
- la façon dont les conditions et événements externes affectent le comportement d'un modèle d'automate ASM.

A cette fin, l'on établit une classification des noms dynamiques (*dynamic*) d'automate ASM en trois catégories nominatives de base, qui élargit la classification des noms représentée sur la Figure F1-3, comme suit:

- noms commandés (*controlled names*)

  Ces domaines, fonctions ou prédicats ne peuvent être modifiés que par des agents du modèle d'automate ASM, conformément aux programmes ASM exécutés. Les noms commandés sont précédés du mot clé **controlled** à leur point de déclaration et sont visibles par l'environnement;

- noms surveillés (*monitored names*)

  Ces domaines, fonctions ou prédicats ne peuvent être modifiés que par l'environnement mais sont visibles par les agents ASM. Un domaine, une fonction ou un prédicat surveillé peut donc modifier ses valeurs d'un état à un autre d'une façon imprévisible, à moins que ces changements ne soient restreints par des contraintes d'intégrité (*integrity constraints*) (voir ci-dessous). Les noms surveillés sont précédés du mot clé **monitored** à leur point de déclaration;

- noms partagés (*shared names*)

  Ces domaines, fonctions ou prédicats sont visibles par l'environnement et peuvent être modifiés par lui ainsi que par les agents ASM. Une contrainte d'intégrité (*integrity constraint*) exercée sur des domaines, fonctions ou prédicats partagés est donc telle qu'aucune perturbation ne doit se produire quant aux emplacements mis à jour en réciprocité. Il est donc nécessaire que l'environnement lui-même joue le rôle d'un agent ASM (ou d'un groupe d'agents ASM). Les noms partagés sont précédés du mot clé **shared** à leur point de déclaration. Voir Figure F1-4.

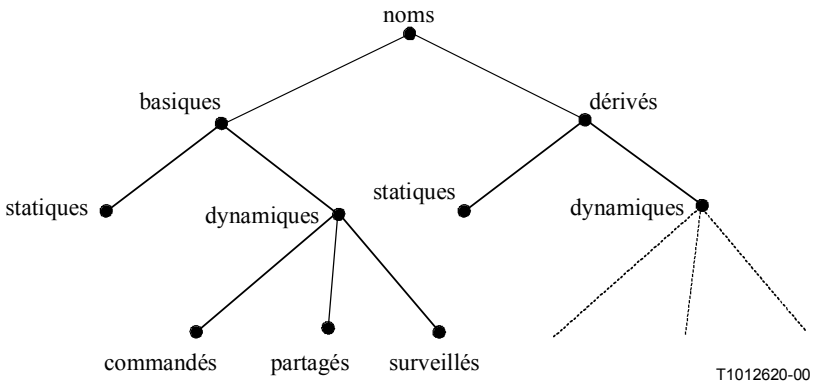


Figure F1-4/Z.100 – Classification étendue des noms d'automate ASM

Exemple (Monde extérieur):

Le vocabulaire *V* du système *RMS* est étendu par une classification des fonctions et prédicats dynamiques:

shared *mode*: $AGENT \rightarrow MODE$
controlled *owner*: $TOKEN \rightarrow AGENT$

monitored *Stop*: $AGENT \rightarrow BOOLEAN$

La fonction *mode*, qui détermine le mode d'accès actuel, est partagée. Elle peut être affectée par des opérations de type "set" commandées de l'extérieur, la faisant passer à l'une des valeurs *exclusive* ou *shared*. Par ailleurs, cette fonction est réinitialisée sur le plan interne lorsque la ressource est libérée (voir § 3.2.3).

Le prédicat *Stop* représente une demande externe d'arrêt, telle qu'une interruption. Il est donc surveillé.

En général, l'influence de l'environnement sur le système, au moyen de noms partagés et surveillés, est totalement imprévisible. Cependant, les conditions préalables imposées au comportement attendu de l'environnement peuvent être exprimées par la déclaration de contraintes d'intégrité (*integrity constraints*) qui doivent être vérifiées dans tous (*all*) les états et passes de *M*. Noter que les contraintes d'intégrité expriment simplement des conditions préalables dans le comportement de l'environnement mais *non* les propriétés que le système est censé posséder.

Les contraintes d'intégrité sont exprimées sous la forme suivante:

IntegrityConstraint ::= **constraint** *ClosedFormula*

Exemple (Contraintes d'intégrité):

La contrainte d'intégrité suivante déclare que les demandes d'arrêt ne sont produites que pour les agents occupés:

constraint $\forall a \in AGENT: (a.Stop \Rightarrow a.Busy)$

3.4 Comportement en temps réel

En introduisant la notion de temps réel (*real time*) et en imposant aux passes des contraintes supplémentaires, l'on obtient une classe spécialisée d'automates ASM, appelés "automates ASM répartis en temps réel" (*distributed real-time ASM*) avec des agents qui exécutent des actions instantanées (*instantaneous*) dans un temps continu (*continuous*). Cela signifie pratiquement que les agents appliquent leurs règles au moment où ils sont activés.

Afin d'incorporer le comportement en temps réel dans le modèle sous-jacent d'exécution de l'automate ASM, l'on introduit une fonction à valeurs réelles surveillée nullaire *currentTime*. Intuitivement, la fonction *currentTime* fait

référence au temps physique. En tant que contrainte d'intégrité sur la nature du temps physique, l'on suppose que la fonction *currentTime* modifie ses valeurs de façon monotone en progressant dans les passes de l'automate ASM.

monitored *currentTime*: → *REAL*

Soit un vocabulaire donné V contenant le domaine *REAL* (mais non la fonction *currentTime*) et soit V^+ l'extension de V avec le symbole de la fonction *currentTime*. La visibilité est limitée aux états V^+ où la fonction *currentTime* a pour valeur un nombre réel. L'on peut alors définir une passe R du modèle d'automate résultant comme étant une application de l'intervalle $[0,\infty)$ aux états du vocabulaire V^+ qui répondent à l'exigence de discontinuité (*discreteness requirement*) suivante, où $\sigma(t)$ indique la réduction[5] de la passe $R(t)$ au vocabulaire V:

1) pour tout $t \geq 0$, la fonction *currentTime* prend la valeur t à l'état $R(t)$;

2) pour tout $\tau > 0$, il existe une séquence finie $0 = t_0 < t_1 < \ldots < t_n = \tau$ telle que si $t_i < \alpha < \beta < t_{i+1}$ on a $\sigma(\alpha) = \sigma(\beta)$.

L'exploitation de la propriété de discontinuité permet d'obtenir effectivement une représentation finie (ou histoire (*history*)) de chaque (sous-)passe finie par déduction à partir des états non considérés comme significatifs en ce sens qu'ils peuvent apporter des informations pertinentes à une description de comportement. L'on peut en particulier se contenter de négliger tous les états qui sont identiques à l'état qui les précède sauf que la valeur de la fonction *currentTime* augmente. De la définition précédente d'une passe, l'on déduit que seuls restent les états dont le nombre est fini.

3.5 Exemple: le système *RMS*

Dans ce paragraphe, l'on assemblera les éléments définissant le modèle d'automate ASM du système *RMS* pour obtenir leur version finale. Afin de faciliter les consultations, l'on répétera la description informelle.

Description informelle

Afin d'illustrer le modèle d'automate ASM, l'on définit un simple système de gestion de ressource (RMS, *resource, management system*), composé d'un groupe de $n > 1$ agents (*group of agents*) en concurrence pour l'obtention d'une ressource (*resource*) (par exemple un dispositif ou un service). Ce système est caractérisé comme suit de manière informelle:

- un ensemble de m jetons (*tokens*), où $m < n$, utilisés pour accorder un accès exclusif (*exclusive*) ou non exclusif (partagé) (*non-exclusive (shared)*) à la ressource;
- selon que le mode d'accès recherché est exclusif ou partagé, un agent doit posséder tous les jetons ou un seul jeton, selon le cas, avant de pouvoir accéder à la ressource;
- un agent est au repos (*idle*) lorsqu'il n'est pas en concurrence pour une ressource; il est en attente (*waiting*) lorsqu'il essaie d'obtenir l'accès à la ressource ou occupé (*busy*) bien que possédant le droit d'accéder à la ressource;
- une fois qu'un agent est en attente, il reste dans cet état jusqu'à ce qu'il obtienne l'accès à la ressource;
- un agent occupé libère la ressource lorsqu'il n'a plus besoin de celle-ci, comme indiqué par un état d'arrêt (*stop condition*) qui est défini de façon externe pour cet agent. Lors de la libération de la ressource, tous les jetons détenus par l'agent sont restitués;
- les états d'arrêt ne sont indiqués que lorsqu'un agent est occupé;
- initialement, les agents sont au repos et tous les jetons sont disponibles.

Vocabulaire

static domain *TOKEN*

shared *mode*: *AGENT* → *MODE*

controlled *owner*: *TOKEN* → *AGENT*

monitored *Stop*: *AGENT* → *BOOLEAN*

---

[5] C'est-à-dire que, pour une valeur t donnée, l'on obtient $\sigma(t)$ à partir de l'état $R(t)$ sans tenir compte de l'interprétation du nom de fonction *currentTime*.

Noms dérivés

MODE $=_{def}$ {*exclusive*, *shared*}

Idle(*a*:*AGENT*): *BOOLEAN* $=_{def}$ $a.mode = undefined \wedge \forall t \in TOKEN: t.owner \neq a$

Waiting(*a*:*AGENT*): *BOOLEAN* $=_{def}$ $a.mode \neq undefined \wedge \forall t \in TOKEN: t.owner \neq a$

Busy(*a*:*AGENT*): *BOOLEAN* $=_{def}$ $a.mode \neq undefined \wedge \exists t \in TOKEN: t.owner = a$

Available(*t*:*TOKEN*): *BOOLEAN* $=_{def}$ $t.owner = undefined$

Contraintes d'intégrité

constraint $\forall a \in AGENT: (a.Stop \Rightarrow a.Busy)$

Contraintes initiales

initially $|AGENT| > 1$

initially $|TOKEN| < |AGENT|$

initially $\forall a \in AGENT: a.program$ = RESOURCE-MANAGEMENT-PROGRAM

initially $\forall a \in AGENT: a.Idle \wedge \forall t \in TOKEN: t.Available$

Programmes d'automate ASM

RESOURCE-MANAGEMENT-PROGRAM:

```
do in-parallel
        SHAREDACCESS
        EXCLUSIVEACCESS
        RELEASEACCESS
enddo
where
        SHAREDACCESS ≡
                if Self.mode = shared ∧ Self.Waiting then
                    choose t: t ∈ TOKEN ∧ t.Available
                        t.owner := Self
                    endchoose
                endif
        EXCLUSIVEACCESS ≡
                if Self.mode = exclusive ∧ ∀t ∈ TOKEN: t.Available then
                    do forall  t: t ∈ TOKEN
                        t.owner := Self
                    enddo
                endif
        RELEASEACCESS ≡
                if Self.Stop then
                    Self.mode := undefined
                    do forall t: t ∈ TOKEN ∧ t.owner = Self
                        t.owner := undefined
                    enddo
                endif
endwhere
```

3.6 Noms prédéfinis

Pour définir un modèle d'automate ASM, en particulier le modèle ASM représentant la sémantique du langage SDL, une prédéfinition est donnée à certains noms et à leur interprétation prévue. Ces noms sont groupés et énumérés ci-après (où *D* désigne la catégorie syntaxique des domaines). Pour les opérateurs préfixes, infixes et suffixes, un caractère de soulignement ("_") sert à indiquer la position de leur argument. Par ailleurs, la préséance des opérateurs est indiquée par l'expression prec(n), où n est un nombre. Les nombres plus élevés indiquent un lien plus étroit. Les opérateurs monadiques ont un lien plus étroit que les opérateurs binaires, lesquels sont associatifs à gauche.

Domaines propres aux automates ASM	
static domain *X*	Ensemble de base ASM (métadomaine)
static domain *BOOLEAN*	Valeurs booléennes
static domain *NAT*	Valeurs d'entier
static domain *REAL*	Valeurs de réel
shared domain *AGENT*	Agents ASM
static domain *PROGRAM*	Programmes d'automate ASM
static domain*TOKEN*	Jetons syntaxiques (chaînes de caractères)
_ *	Constructeur de domaine: séquence de
_ +	Constructeur de domaine: séquence non vide de
_ -set	Constructeur de domaine: ensemble de
_ × _ prec(7)	Constructeur de domaine: nuplet
_ ∪ _ prec(6)	Constructeur de domaine: réunion logique

Fonctions propres aux automates ASM	
static *undefined*: → *X*	Indicateur de valeurs indéfinies
monitored *Self*: → *AGENT*	Autoréférence pour agents ASM
controlled *program*: *AGENT* → *PROGRAM*	Programme d'un agent ASM
monitored *currentTime*: → *REAL*	Temps actuel du système

Fonctions et prédicats booléens	
static *True*: → *BOOLEAN*	Littéral prédéfini
static *False*: → *BOOLEAN*	Littéral prédéfini
_ = _ prec(4)	Egalité
_ ≠ _ prec(4)	Inégalité
_ ∧ _ prec(3)	ET logique
_ ∨ _ prec(2)	OU logique
_ ⇒ _ prec(1)	Implication
_ ⇔ _ prec(1)	Equivalence logique
¬ _	Négation
∃*x* ∈ *D*: *P*(*x*) prec(0)	Quantification existentielle (au moins 1 élément)
∃!*x* ∈ *D*: *P*(*x*) prec(0)	Quantification existentielle unique (exactement 1 élément)
∀*x* ∈ *D*: *P*(*x*) prec(0)	Quantification universelle

Termes	
X	Application de fonction nullaire
$f(t_1,..., t_n)$	Application de fonction avec n expressions d'argument
if *Formula* **then** *Term* **else** *Term* **endif**	Expression conditionnelle; l'on utilise de nouveau **elseif** au lieu de **else if**
s-_(_)	Fonction de sélection de nuplet (voir la rubrique Nuplets ci-dessous)
mk-_(...)	Construction de nuplets (voir la rubrique Nuplets ci-dessous)
inv-_(...)	Inverse d'une fonction ou d'une relation
	inv-*Fun*(*x*) =$_{\text{def}}$ *take*({ *a* ∈ *D*: *Fun*(*a*) =*x* })

Fonctions et relations appliquées à des entiers	
_ > _, _ ≥ _, _ < _ , _ ≤ _ prec(4)	Comparateurs
_ + _ prec(6)	Addition
_ - _ prec(6)	Soustraction
_ * _ prec(7)	Multiplication
_ / _ prec(7)	Division
0, *1*, ...	Littéraux d'entier

Fonctions appliquées à des séquences	
static *empty*: $\rightarrow D^*$	Séquence vide
static *head*: $D^* \rightarrow D$	Début de la séquence (*undefined* si vide)
static *tail*: $D^* \rightarrow D^*$	Fin de la séquence (*undefined* si vide)
static *last*: $D^* \rightarrow D$	Dernier élément d'une séquence (*undefined* si vide)
static *length*: $D^* \rightarrow$ *NAT*	Longueur d'une séquence
static < >: $D^n \rightarrow D^*$	Constructeur de séquence; les arguments sont énumérés entre parenthèses, séparés par des virgules
_ ^ _ prec(6)	Concaténation de séquences
toSet: $D^* \rightarrow D$-**set**	Conversion des éléments d'une séquence en un ensemble
_ [_]	Accès à un élément d'une liste; l'index entre parenthèses doit être du type *NAT*
_ **in** _ prec(4)	Elément de?
< <*result*> | <*var*> **in** <*seq*> : <*cond*> >	Compréhension d'une séquence; opère comme un filtre sur <seq>, c'est-à-dire maintient l'ordre des éléments
< <*var*> **in** <*seq*> : <*cond*> > $=_{def}$ < <*var*> | <*var*> **in** <*seq*> : <*cond*> >	Compréhension de séquence abrégée
< <*result*> | <*var*> **in** <*seq*> > $=_{def}$ < <*result*> | <*var*> **in** <*seq*> : *True* >	Compréhension de séquence abrégée

Fonctions appliquées à des ensembles	
_ ∪ _ prec(6)	Réunion d'ensembles
_ ∩ _ prec(7)	Intersection
_ \ _ prec(6)	Soustraction d'ensemble
_ ∈ _ prec(4)	Elément de?
_ ∉ _ prec(4)	Non élément de?
_ ⊆ _ prec(4)	Sous-ensemble de?
_ ⊂ _ prec(4)	Sous-ensemble propre de?
| _ |	Taille d'un ensemble
U _	Grande réunion: réunion de tous les ensembles inclus dans l'ensemble d'arguments
∅	Ensemble vide
static { }: $D^n \rightarrow D$-**set**	Constructeur d'ensemble; liste d'arguments séparés par des virgules, entre parenthèses
take: *D*-**set** $\rightarrow D$	Sélecteur d'élément arbitraire dans l'ensemble ou valeur *undefined* d'un ensemble vide
_ .. _ prec(5)	Etendue d'entiers de la première à la deuxième valeur. Ensemble vide si la deuxième expression est plus petite que la première
{ <*result*> | <*var*> ∈ <*set*> : <*cond*> }	Compréhension d'un ensemble; agit comme filtre sur <set>
{ <*var*> ∈ <*set*> : <*cond*> } $=_{def}$ { <*var*> | <*var*> ∈ <*set*>: <*cond*> }	Compréhension d'ensemble abrégée
{ <*result*> | <*var*> ∈ <*set*> } $=_{def}$ { <*result*> | <*var*> ∈ <*set*>: *True* }	Compréhension d'ensemble abrégée

Formes et expressions de cas

Les formes permettent d'accéder facilement à la structure de valeurs. Les formes suivantes sont offertes:

- Variables: une variable correspond à une valeur quelconque. Cependant, si la variable est déjà liée, elle ne correspond qu'à elle-même.
- Variables anonymes: les variables anonymes sont indiquées par un astérisque "*". Elles sont une forme abrégée pour introduire une variable inutilisée.
- Constructeur: un constructeur est donné par son nom et par ses arguments, qui sont de nouveau des formes. Il correspond à toute valeur construite par son intermédiaire avec les arguments correspondant à leur forme propre.
- Forme nommée: la notation Variable = Forme introduit un nom pour (la valeur correspondant à) la forme.

Les formes sont utilisées pour décrire des fonctions applicables à l'arbre syntaxique. Les noms de non-terminaux de la grammaire servent de fonctions de constructeur.

Une expression de cas sert à déterminer une valeur qui dépend de la correspondance avec une forme.

CaseExpression ::=**case** *Term* **of**
| *Pattern*$_1$: *Term*$_1$
| *Pattern*$_2$: *Term*$_2$
...
[**otherwise** *Term*$_0$]
endcase

Si la valeur de l'élément *Term* correspond à au moins une forme *Pattern*$_i$, le résultat de l'expression de cas est donné par *Term*$_i$. Si aucune forme ne correspond, le résultat est *Term*$_0$ (si présent). Sinon, le résultat a la valeur *undefined*.

Domaines de réunion

Les domaines de réunion contiennent simplement les valeurs de leurs domaines constituants.

$D =_{\text{def}} D_1 \cup D_2$

Nuplets

Pour tout domaine de nuplet déclaré, plusieurs fonctions implicites de constructeur et de sélecteur sont définies. Une définition

$D =_{\text{def}} D_1 \times D_2^* \times D_3$-**set** $\times D_1 \times (D_1 \cup D_2)$

définit également les fonctions suivantes:

mk-*D*: $D_1 \times D_2^* \times D_3$-**set** $\times D_1 \times (D_1 \cup D_2) \rightarrow D$
s-*D*$_1$: $D \rightarrow D_1$
s-*D*$_2$-**seq**: $D \rightarrow D_2^*$
s-*D*$_3$-**set**: $D \rightarrow D_3$-**set**
s2-*D*$_1$: $D \rightarrow D_1$
s-implicit: $D \rightarrow (D_1 \cup D_2)$

Lorsque le nuplet contient plus d'une seule fois le même domaine, des fonctions de sélection semblables à **s2-***D*$_1$ sont définies. Pour une réunion, la fonction spéciale de sélection **s-implicit** est définie.

Règles syntaxiques abstraites

Les règles syntaxiques abstraites extraites de la définition linguistique sont directement traduites en notation d'automate ASM au moyen de certaines conventions qui seront expliquées par des exemples. Fondamentalement, une règle syntaxique abstraite peut être interprétée comme déclarant un ou plusieurs domaines (de nuplet) et comme des fonctions définissantes pour construire et sélectionner des valeurs des domaines constituants. Les nœuds syntaxiques ont cependant une identité, ce qui n'est pas le cas des nuplets ordinaires. Il existe des règles syntaxiques pour introduire des constructeurs nommés ainsi que des réunions nommées ou non nommées. Les règles d'introduction de constructeurs se composent de symboles terminaux et non-terminaux; elles ont la forme suivante:

Symbol :: *Symbol*$_1$ *Symbol*$_2^+$ *Symbol*$_3$-**set** [*Symbol*$_4$]

ce qui peut se traduire par:

Symbol-aux =$_{\text{def}}$ *Symbol*$_1 \times$ *Symbol*$_2^* \times$ *Symbol*$_3$-**set** $\times$ *Symbol*$_4$
controlled domain *Symbol*

controlled *contents-Symbol*: *Symbol* $\rightarrow$	*Symbol-aux*
s-*Symbol*$_1$(*x*: *Symbol*): *Symbol*	=$_{\text{def}}$ **s-***Symbol*$_1$(*x.contents-Symbol*)
s-*Symbol*$_2$-**seq**(*x*: *Symbol*): *Symbol*$_2^*$	=$_{\text{def}}$ **s-***Symbol*$_2$-**seq**(*x.contents-Symbol*)
s-*Symbol*$_3$-**set**(*x*: *Symbol*): *Symbol*$_3$-**set**	=$_{\text{def}}$ **s-***Symbol*$_3$-**set**(*x.contents-Symbol*)
s-*Symbol*$_4$ (*x*: *Symbol*): *Symbol*$_4$	=$_{\text{def}}$ **s-***Symbol*$_4$(*x.contents-Symbol*)

Il existe par ailleurs une abréviation **mk-***Symbol* qui revient à créer un nouvel objet du domaine *Symbol* au moyen de la primitive **extend** et à rendre la valeur *contents-Symbol* de l'objet nouvellement créé égale au résultat de l'opération M**k-***Symbol-aux*. Noter que cette sorte d'abréviation n'est pas une fonction mais en réalité un élément de règle. Elle ne doit donc être utilisée qu'à l'intérieur de règles. Le fait que le symbole facultatif *Symbol*$_4$ ne soit pas présent est exprimé dans le modèle d'automate ASM en laissant la valeur correspondante indéfinie (*undefined*).

Une séquence vide de symboles (constructeur non fractionné) est notée par ( ).

L'égalité de valeurs syntaxiques est toujours une égalité structurelle, c'est-à-dire que c'est le contenu des symboles qui est comparé et non pas les symboles eux-mêmes.

Les règles syntaxiques introduisant des réunions nommées, c'est-à-dire des synonymes, ont la forme suivante:

Symbol = *Symbol*$_1$ | *Symbol*$_2$ | ... | *Symbol*$_n$ (n ≥ 1)

qui se traduit par:

$Symbol =_{def} Symbol_1 \cup Symbol_2 \cup ... \cup Symbol_n$

Noter que, comme *Symbol* est un domaine de réunion (*union*), l'expansion fournit une définition de domaine mais pas de fonctions **mk-** ou **s-**.

Dans certains cas, il n'est pas nécessaire de faire référence à des synonymes. Ici, des réunions non nommées peuvent être introduites par

$Symbol :: Symbol_1 \{ Symbol_{21} | ... | Symbol_{2n} \}$

au lieu d'introduire des synonymes:

$Symbol :: Symbol_1\ Symbol_2$

$Symbol_2 = Symbol_{21} | ... | Symbol_{2n}$

Pour chaque mot clé SDL **KEYWORD**, il existe un domaine de mot clé associé *Keyword* ne comportant qu'une seule valeur:

static domain *Keyword*

Il est requis que tous les domaines de mot clé s'excluent mutuellement.

Compte tenu de la grammaire abstraite, il existe un domaine dérivé appelé *DefinitionAS1*, qui se compose de tous les domaines de symboles syntaxiques abstraits comme suit:

$DefinitionAS1 =_{def} Symbol_1 \cup Symbol_2 \cup ... \cup Symbol_n$

où $Symbol_1$,$Symbol_2$,...,$Symbol_n$ est la liste de tous (*all*) les symboles terminaux et non terminaux de la grammaire abstraite.

Il existe un domaine *DefinitionAS0* similaire pour la grammaire concrète (AS0).

Les fonctions **s-** peuvent servir à naviguer vers le bas d'un arbre syntaxique abstrait donné. Pour naviguer vers le haut, deux fonctions ascendantes doivent être définies:

controlled *parentAS1*: *DefinitionAS1* → *DefinitionAS1*
controlled *parentAS0*: *DefinitionAS0* → *DefinitionAS0*

Par ailleurs, deux fonctions sont définies pour trouver l'ascendant d'un descendant particulier:

parentAS0ofKind(*from*: *DefinitionAS0*, *x*: *DefinitionAS0***-set**): *DefinitionAS0* =$_{def}$
if *from* = *undefined* **then** *undefined*
elseif *from* ∈ *x* **then** *from*
else *parentAS0ofKind*(*from*.parentAS0, *x*)
endif
parentAS1ofKind(*from*: *DefinitionAS1*, *x*: *DefinitionAS1***-set**):
DefinitionAS1 =$_{def}$
if *from* = *undefined* **then** *undefined*
elseif *from* ∈ *x* **then** *from*
else *parentAS1ofKind*(*from*.*parentAS1*, *x*)
endif

Les fonctions *isAncestorAS1* et *isAncestorAS0* déterminent si le premier nœud est un ancêtre du second.

isAncestorAS1(*n*: *DefinitionAS1* ,*n'*::*DefinitionAS1*:*BOOLEAN* =$_{def}$
n = *n'*.*parentAS1* ∨ *isAncestorAS1*(*n*, *n'*.*parentAS1*)

isAncestorAS0(*n*: *DefinitionAS0* ,*n'*: *DefinitionAS0*): *BOOLEAN* =$_{def}$
n = *n'*. *parentAS0* ∨ *isAncestor AS0*(*n*, *n'*.*parentAS0*)

Le nœud sommital de l'arbre de syntaxe abstraite ou concrète actuel est indiqué par les fonctions nullaires suivantes:

controlled*rootNodeAS1*: → *DefinitionAS1*
controlled*rootNodeAS0*: → *DefinitionAS0*

L'arbre de syntaxe abstraite peut être modifié au moyen de la fonction dérivée suivante:

replaceInSyntaxTree: *DefinitionAS0* × *DefinitionAS0* × *DefinitionAS0* → *DefinitionAS0*

Le premier paramètre de la fonction est l'ancien sous-arbre, le deuxième est le nouveau sous-arbre et le troisième est l'ancien arbre. La fonction donne le nouvel arbre, dans lequel tous les anciens sous-arbres sont remplacés par le nouveau sous-arbre.

SÉRIES DES RECOMMANDATIONS UIT-T

Série A	Organisation du travail de l'UIT-T
Série B	Moyens d'expression: définitions, symboles, classification
Série C	Statistiques générales des télécommunications
Série D	Principes généraux de tarification
Série E	Exploitation générale du réseau, service téléphonique, exploitation des services et facteurs humains
Série F	Services de télécommunication non téléphoniques
Série G	Systèmes et supports de transmission, systèmes et réseaux numériques
Série H	Systèmes audiovisuels et multimédias
Série I	Réseau numérique à intégration de services
Série J	Réseaux câblés et transmission des signaux radiophoniques, télévisuels et autres signaux multimédias
Série K	Protection contre les perturbations
Série L	Construction, installation et protection des câbles et autres éléments des installations extérieures
Série M	RGT et maintenance des réseaux: systèmes de transmission, circuits téléphoniques, télégraphie, télécopie et circuits loués internationaux
Série N	Maintenance: circuits internationaux de transmission radiophonique et télévisuelle
Série O	Spécifications des appareils de mesure
Série P	Qualité de transmission téléphonique, installations téléphoniques et réseaux locaux
Série Q	Commutation et signalisation
Série R	Transmission télégraphique
Série S	Equipements terminaux de télégraphie
Série T	Terminaux des services télématiques
Série U	Commutation télégraphique
Série V	Communications de données sur le réseau téléphonique
Série X	Réseaux de données et communication entre systèmes ouverts
Série Y	Infrastructure mondiale de l'information et protocole Internet
Série Z	**Langages et aspects généraux logiciels des systèmes de télécommunication**

Genève, 2002