LesSystème CESAR : description, spécification et analyse des applications réparties

Jean-Pierre Queille

▶ **To cite this version:**

Jean-Pierre Queille. LesSystème CESAR : description, spécification et analyse des applications réparties. Modélisation et simulation. Université Joseph-Fourier - Grenoble I, 1982. Français. NNT : . tel-00305292

HAL Id: tel-00305292
https://theses.hal.science/tel-00305292

Submitted on 23 Jul 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THESE

présentée à

l' Université Scientifique et Médicale de Grenoble

pour obtenir le grade de
DOCTEUR-INGENIEUR
«Informatique»

par

QUEILLE Jean-Pierre

LE SYSTEME CESAR: DESCRIPTION, SPECIFICATION ET ANALYSE DES APPLICATIONS REPARTIES.

Thèse soutenue le 15 Juin 1982 devant la commission d'examen.

L. BOLLIET	Président
G. BERRY	
Ph. JORRAND	
C. MERAUD	Examinateurs
J.M. PITIE	
G. ROUCAIROL	
J. SIFAKIS	
J.P. VERJUS	

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

année scolaire 1980-1981

Président de l'Université : M. J.J. PAYAN

MEMBRES DU CORPS ENSEIGNANT DE L'U.S.M.G.

PROFESSEURS DE 1ère CLASSE

Mlle	AGNIUS DELORD Claudine	Biophysique
	ALARY Josette	Chimie analytique
MM.	AMBLARD Pierre	Clinique dermatologie
	AMBROISE THOMAS Pierre	Parasitologie
	ARNAUD Paul	Chimie
	ARVIEU Robert	Physique nucléaire
	AUBERT Guy	Physique
	AYANT Yves	Physique approfondie
Mme	BARBIER Marie-Jeanne	Electrochimie
MM.	BARBIER Jean-Claude	Physique expérimentale
	BARBIER Reynold	Géologie
	BARJON Robert	Physique nucléaire
	BARNOUD Fernand	Biosynthèse de la cellulose
	BARRA Jean-René	Statistiques
	BARRIE Joseph	Clinique chirurgicale A
	BEAUDOING André	Clinique pédiatrie et puériculture
	BELORISKY Elie	Physique
	BENZAKEN Claude	Mathématiques appliquées
Mme	BERIEL Hélène	Pharmacodynamie
M.	BERNARD Alain	Mathématiques pures
Mme	BERTRANDIAS Françoise	Mathématiques pures
MM.	BERTRANDIAS Jean-Paul	Mathématiques pures
	BEZES Henri	Clinique chirurgicale & traumatologie
	BILLET Jean	Géographie
	BONNET Jean-Louis	Clinique ophtalmologique
	BONNET EYMARD Joseph	Clinique Hépato-gastro-entérologie
Mme	BONNIER Jane-Marie	Chimie générale
MM.	BOUCHERLE André	Chimie et toxicologie
	BOUCHET Yves	Anatomie
	BOUCHEZ Robert	Physique nucléaire
	BRAVARD Yves	Géographie

.../...

MM.	BUTEL Jean	Orthopédie
	CABANEL Guy	Clinique rhumatologie et hydrologie
	CARLIER Georges	Biologie végétale
	CAU Gabriel	Médecine légale et toxicologie
	CAUQUIS Georges	Chimie organique
	CHARACHON Robert	Clinique O.R.L.
	CHATEAU Robert	Clinique neurologique
	CHIBON Pierre	Biologie animale
	COEUR André	Chimie analytique et bromotologique
	COUDERC Pierre	Anatomie pathologique
	CRABBE Pierre	C.E.R.M.O.
	DAUMAS Max	Géographie
	DEBELMAS Jacques	Géologie générale
	DEGRANGE Charles	Zoologie
	DELOBEL Claude	M.I.A.G.
	DELORMAS Pierre	Pneumo-phtisiologique
	DENIS Bernard	Clinique cardiologique
	DEPORTES Charles	Chimie minérale
	DESRE Pierre	Electrochimie
	DODU Jacques	Mécanique appliquée IUT 1
	DOLIQUE Jean-Michel	Physique des plasmas
	DUCROS Pierre	Cristallographie
	FONTAINE Jean-Marc	Mathématiques pures
	GAGNAIRE Didier	Chimie physique
	GASTINEL Noël	Analyse numérique
	GAVEND Jean-Michel	Pharmacologie
	GEINDRE Michel	Electro-radiologie
	GERBER Robert	Mathématiques pures
	GERMAIN Jean-Pierre	Mécanique
	GIRAUD Pierre	Géologie
	JANIN Bernard	Géographie
	JEANNIN Charles	Pharmacie galénique
	JOLY Jean-René	Mathématiques pures
	KAHANE André	Physique
	KAHANE Josette	Physique
	KLEIN Joseph	Mathématiques pures
	KOSZUL Jean-Louis	Mathématiques pures
	LACAZE Albert	Hermodynamique
	LACHARME Jean	Biologie cellulaire
	LAJZEROWICZ Joseph	Physique

../..

Mme	LAJZEROWICZ Jeannine	Physique
MM.	LATREILLE René	Chirurgie thoracique
	LATURAZE Jean	Biochimie pharmaceutiques
	LAURENT Pierre	Mathématiques appliquées
	LE NOC Pierre	Bactériologie virologie
	LLIBOUTRY Louis	Géophysique
	LOISEAUX Jean-Marie	Sciences nucléaires
	LOUP Jean	Géographie
	LUU DUC Cuong	Chimie générale et minérale
	MALINAS Yves	Clinique obstétricale
Mlle	MARIOTTE Anne-Marie	Pharmacognostie
MM.	MAYNARD Roger	Physique du solide
	MAZARE Yves	Clinique médicale A
	MICHEL Robert	Minéralogie et pétrographie
	MICOUD Max	Clinique maladies infectieuses
	MOURIQUAND Claude	Histologie
	NEGRE Robert	Mécanique IUT 1
	MOZIERES Philippe	Spectrométrie physique
	OMONT Alain	Astrophysique
	OZENDA Paul	Botanique
	PAYAN Jean-Jacques	Mathématiques pures
	PEBAY PEYROULA Jean-Claude	Physique
	PERRET Jean	Sémeiologie médicale (neurologie)
	PERRIER Guy	Géophysique
	PIERRARD Jean-Marie	Mécanique
	RACHAIL Michel	Clinique médicale B
	RASSAT André	Chimie systématique
	RENARD Michel	Thermodynamique
Mme	RENAUDET Jacqueline	Bactériologie
M.	REVOL Michel	Urologie
Mme	RINAUDO Marguerite	Chimie CERMAV
MM.	DE ROUGEMONT Jacques	Neuro-chirurgie
	SARRAZIN Roger	Clinique chirurgicale B
Mme	SEIGLE MURANDI Françoise	Botanique et cryptogamie
MM.	SENGEL Philippe	Biologie animale
	SIBILLE Robert	Construction mécanique IUT 1
	SOUTIF Michel	Physique
	TANCHE Maurice	Physiologie
	VAILLANT François	Zoologie
	VALENTIN Jacques	Physique nucléaire

.../...

MM.	VAN CUTSEM Bernard	Mathématiques appliquées
	VAUQUOIS Bernard	Mathématiques appliquées
	VERAIN Alice	Pharmacie galénique
	VERAIN André	Biophysique
	VIGNAIS Pierre	Biochimie médicale

PROFESSEURS DE 2ème CLASSE

MM.	ARNAUD Yves	Chimie IUT 1
	AURIAULT Jean-Louis	Mécanique IUT 1
	BEGUIN Claude	Chimie organique
	BOITET Christian	Mathématiques appliquées
	BOUTHINON Michel	E.E.A. IUT 1
	BRUGEL Lucien	Energétique IUT 1
	BUISSON Roger	Physique IUT 1
	CASTAING Bernard	Physique
	CHARDON Michel	Géographie
	CHEHIKIAN Alain	E.E.A. IUT 1
	COHEN Henri	Mathématiques pures
	COHENADDAD Jean-Pierre	Physique
	COLIN DE VERDIERE Yves	Mathématiques pures
	CONTE René	Physique IUT 1
	CYROT Michel	Physique du solide
	DEPASSEL Roger	Mécanique des fluides
	DOUCE Roland	Physiologie végétale
	DUFRESNOY Alain	Mathématiques pures
	GASPARD François	Physique
	GAUTRON René	Chimie
	GIDON Maurice	Géologie
	GIGNOUX Claude	Sciences nucléaires
	GLENAT René	Chimie organique
	GOSSE Jean-Pierre	E.E.A. IUT 1
	GROS Yves	Physique IUT 1
	GUITTON Jacques	Chimie
	HACQUES Gérard	Mathématiques appliquées
	HERBIN Jacky	Géographie
	HICTER Pierre	Chimie
	IDELMAN Simon	Physiologie animale
	JOSELEAU Jean-Paul	Biochimie
	JULLIEN Pierre	Mathématiques appliquées
	KERCKOVE Claude	Géologie

../...

MM.	KRAKOWIACK Sacha	Mathématiques appliquées
	KUHN Gérard	Physique IUT 1
	KUPKA Yvon	Mathématiques pures
	LUNA Domingo	Mathématiques pures
	MACHE Régis	Physiologie végétale
	MARECHAL Jean	Mécanique
	MICHOULIER Jean	Physique IUT 1
Mme	MINIER Colette	Physique IUT 1
MM.	NEMOZ Alain	Thermodynamique
	NOUGARET Marcel	Automatique IUT 1
	OUDET Bruno	Mathématiques appliquées
	PEFFEN René	Métallurgie IUT 1
	PELMONT Jean	Biochimie
	PERRAUD Robert	Chimie IUT 1
	PERRIAUX Jean-Jacques	Géologie minéralogie
	PERRIN Claude	Sciences nucléaires
	PFISTER Jean-Claude	Physique du solide
	PIERRE Jean-Louis	Chimie organique
Mlle	PIERY Yvette	Physiologie animale
MM.	RAYNAUD Hervé	Mathématiques appliquées
	RICHARD Lucien	Biologie végétale
	ROBERT Gilles	Mathématiques pures
	ROBERT Jean-Bernard	Chimie physique
	ROSSI André	Physiologie végétale
	SAKAROVITCH Michel	Mathématiques appliquées
	SARROT REYNAUD Jean	Géologie
	SAXOD Raymond	Biologie animale
Mme	SOUTIF Jeanne	Physique
MM.	STUTZ Pierre	Mécanique
	VIALON Pierre	Géologie
	VIDAL Michel	Chimie organique
	VIVIAN Robert	Géographie

CHARGES D'ENSEIGNEMENT PHARMACIE

MM.	ROCHAS Jacques	Hygiène et hydrologie
	DEMENGE Pierre	Pharmacodynamie

PROFESSEURS SANS CHAIRE (médecine)

M.	BARGE Michel	Neuro-chirurgie

.../...

MM.	BOST Michel	Pédiatrie
	BOUCHARLAT Jacques	Psychiatrie
	CHAMBAZ Edmond	Biochimie (hormonologie)
	CHAMPETIER Jean	Anatomie
	COLOMB Maurice	Biochimie
	COULOMB Max	Radiologie
Mme	ETERRADOSSI Jacqueline	Physiologie
MM.	FAURE Jacques	Médecine légale
	GROULADE Joseph	Biochimie A
	HOLLARD Daniel	Hématologie
	HUGONOT Robert	Gérontologie
	JALBERT Pierre	Histologie
	MAGNIN Robert	Hygiène
	PHELIP Xavier	Rhumatologie
	REYMOND Jean-Charles	Chirurgie générale
	STIEGLITZ Paul	Anesthésiologie
	VROUSOS Constantin	Radiothérapie

MAITRES DE CONFERENCES AGREGES (médecine)

MM.	BACHELOT Yvan	Endocrinologie
	BENABID Alim Louis	Médecine et chirurgie
	BERNARD Pierre	Gynécologie obstétrique
	CONTAMIN Charles	Chirurgie thoracique
	CORDONNIER Daniel	Néphrologie
	CROUZET Guy	Radiologie
	DEBRU Jean-Luc	Médecine interne
	DYON Jean-François	Chirurgie infantile
	FAURE Claude	Anatomie et organogènèse
	FAURE Gilbert	Urologie
	FLOYRAC Roger	Biophysique
	FOURNET Jacques	Hépato-gastro-entérologie
	GAUTIER Robert	Chirurgie générale
	GIRARDET Pierre	Anesthésiologie
	GUIDICELLI Henri	Chirurgie générale
	GUIGNIER Michel	Thérapeutique (réanimation)
	JUNIEN-LAVILLAUROY Claude	Clinique O.R.L.
	KOLODIE Lucien	Hématologie biologique
	MALLION Jean-Michel	Médecine du travail
	MASSOT Christian	Médecine interne
	MOUILLON Michel	Ophtalmologie

.../

MM.	PARAMELLE Bernard	Pneumologie
	RACINET Claude	Gynécologie-Obstétrique
	RAMBAUD Pierre	Pédiatrie
	RAPHAEL Bernard	Stomatologie
	SCHAEFER René	Cancérologie
	SEIGNEURIN Jean-Marie	Bactériologie-virologie
	SOTTO Jean-Jacques	Hématologie
	STOEBNER Pierre	Anatomie-pathologique

REMERCIEMENTS

Je tiens à remercier :

- Ph. JORRAND, Maître de Recherches au CNRS, pour m'avoir permis de participer aux travaux de son équipe depuis 1978, et pour avoir favorisé le développement du projet CESAR au sein de celle-ci ;

- J. SIFAKIS, Chargé de Recherches au CNRS, pour tout le temps qu'il a investi dans ce projet, qui sans lui n'aurait jamais vu le jour. Nombre des résultats exposés ici sont le fruit d'une étroite collaboration avec lui ;

- L. BOLLIET, Professeur à l'Université de Grenoble, pour m'avoir fait le grand honneur de présider le Jury de cette thèse, et G. ROUCAIROL, Professeur à l'Université de Paris VI, pour avoir bien voulu juger ce travail ;

- G. BERRY, Maître de Recherches à l'Ecole des Mines, J.M. PITIE, Ingénieur au CNET de Lannion, et J.P. VERJUS, Professeur à l'Université de Rennes, pour avoir accepté de participer au Jury de cette même thèse.

Je remercie également toutes les personnes qui ont contribué, à des titres divers, au projet CESAR : J.P. CHARBONNIER, S. GRAF, F. MARTINET, J.Ch. MARTY, M. ROZIER, J.Ph. SCHWARTZ, J. VOIRON, ainsi que F. GUERREIRO de l'Universidade Nova de Lisboa.

Je remercie aussi C. MERAUD, Ingénieur à la SAGEM, pour les longues discussions que j'ai eu avec lui dans lesquelles j'ai pu confronter mon point de vue d'universitaire avec la réalité des problèmes industriels, et qui a bien voulu participer également au Jury.

Je remercie enfin D. IGLESIAS et son équipe pour le soin apporté à l'impression de cette thèse.

LE SYSTEME CESAR : DESCRIPTION, SPECIFICATION ET ANALYSE DES APPLICATIONS REPARTIES.

SOMMAIRE DETAILLE

Chapitre I

INTRODUCTION
PRESENTATION DU SYSTEME CESAR

1. Objectifs
2. Le Système CESAR
3. Tour d'horizon

1 - OBJECTIFS

L'objet de cette thèse est de proposer un système d'aide à la conception et à la validation d'architectures (logicielles et/ou matérielles) d'applications réparties. Ce système, dénommé CESAR (Conception, Evaluation et Spécification des Applications Réparties), tente de répondre dans une certaine mesure aux besoins exprimés par les concepteurs d'applications réparties, de façon de plus en plus pressante à mesure que la complexité des applications informatiques croit.

Par application répartie nous entendons en fait toute application mono- ou multi-tâches, que celles-ci soient logicielles ou réalisées par des organes matériels, que le parallélisme entre les tâches soit réel ou simulé (multi-processeurs ou mono-processeur multi-programmé), qu'il y ait ou non répartition physique ou géographique (réseaux à petite ou grande échelle). Les applications réparties peuvent être assez arbitrairement regroupées en deux grandes classes, suivant qu'elles sont sujettes ou non à des contraintes temporelles critiques. Les premières sont connues sous la dénomination d'applications "temps-réel" (le principal domaine en est le contrôle de processus industriel).

Les besoins exprimés à l'heure actuelle par les concepteurs d'applications réparties convergent généralement vers un système d'aide à la conception et à l'intégration permettant :

- de décrire les différentes parties (ou modules) de l'application à concevoir ou intégrer,

- d'archiver les descriptions de ces modules en vue de leur ré-utilisation ultérieure,

- de définir l'architecture de l'application en cours de conception ou d'intégration à l'aide de ces modules,

- d'évaluer la conformité de cette architecture à une spécification de son comportement souhaité (ou d'en comparer plusieurs).

En réponse à ces besoins, le cahier des charges établi pour le système CESAR est de fournir :

- d'une part, un langage de haut-niveau pour la description des applications réparties :
  * structuré, de façon à permettre une description modulaire et la définition d'une architecture hiérarchisée ;

* typé, de façon à assurer la cohérence des objets manipulés par les différents modules de l'application ; ces objets ne sont d'ailleurs pas nécessairement d'un type informatique (entier, booléen, etc.) ;

* autorisant une description indépendante au maximun des contraintes d'implémentation matérielles (due à la configuration physique des ressources), logicielles (due à l'organisation du système d'exploitation) ou syntaxiques (due au langage de programmation devant être utilisé) ; en particulier, l'absence de non-déterminisme étant en général une telle contrainte, le langage doit offrir la possibilité de décrire des situations non-déterministes ;

* permettant d'ajuster la finesse de la description selon le niveau de détail où l'on se place (donc adapté aussi bien à une démarche de conception descendante qu'à une synthèse ascendante à partir d'éléments pré-existants) ; le langage doit donc permettre de définir et de manipuler des objets et des traitements non-spécifiés (non décrits), que l'on pourra raffiner par la suite ;

* utilisable tant pour la description de modules logiciels que pour celle de "boites noires" matérielles ; le langage doit donc permettre une description algorithmique classique, tout en offrant la possibilité de décrire le comportement d'un automate réagissant à des évènements extérieurs (au moyen de "commandes gardées" comportant des échanges) ;

- d'autre part, un langage de spécification du fonctionnement requis, sous la forme d'un ensemble de propriétés devant être vérifiées ; une étude des besoins tend à montrer que ces propriétés sont de quatres types :

* séquencement correct des synchronisations et des échanges entre les différentes tâches de l'application,

* relations entre les données manipulées devant être vérifiées,

* séquencement correct des actions exécutées,

* contraintes temporelles devant être respectées ;

- enfin un système d'analyse permettant de comparer la description et la spécification fournies, c.a.d. de vérifier que les propriétés requises sont garanties par l'architecture décrite. Pour ceci le système CESAR doit construire automatiquement à partir de la description fournie, un modèle aux propriétés mathématiquement bien connues, sur lequel il pourra procéder à une telle analyse.

Le système CESAR que nous proposons ici ne répond pas à un des objectifs du cahier des charges : il ne permet pas de spécifier et de vérifier des contraintes temporelles (exprimées en termes d'intervalles de temps). Toutefois, cela n'exclut pas son utilisation pour les applications

temps-réel, dans lesquelles, en effet, un grand nombre de contraintes qui sont considérées être des contraintes temporelles sont en fait des contraintes de séquencement sur des actions, contraintes que le système CESAR permet d'exprimer et de vérifier (un exemple où des contraintes de périodicité sont ainsi transformées en contraintes de séquencement est traité au chapitre II-10.1).

Le système CESAR permet ainsi au concepteur d'une application répartie d'évaluer l'architecture qu'il propose, en ce sens qu'il peut savoir quelles sont les propriétés qu'elle vérifie parmi l'ensemble des propriétés qu'il a spécifié. Il peut également comparer deux architectures possibles pour une même application, problème qui présente deux aspects :

- On peut désirer savoir laquelle des deux architectures satisfait le mieux un ensemble de propriétés, certaines absolument nécessaires et d'autres seulement souhaitées. Dans ce cas le système CESAR fournit des éléments de réponse au concepteur en lui disant quelles sont les propriétés vérifiées par chacune des architectures proposées. Si une des architectures proposées ne vérifie pas une des propriétés absolument nécessaires, le concepteur est placé devant un choix, relativement facile, entre une architecture qui "marche" et une qui "ne marche pas". Dans le cas contraire, les éléments fournis par le système CESAR permettent au concepteur de peser soigneusement les avantages et les inconvénients de chacune des deux architectures proposées. Il est toutefois à noter que sa tâche est rendue plus malaisée par le fait que les propriétés pouvant être exprimées sur deux descriptions différentes ne sont pas toujours équivalentes, ni même comparables (l'exemple du protocole du bit alterné traité tout au long de cette thèse le montre clairement).

- On peut désirer choisir entre deux architectures satisfaisant le même ensemble de propriétés de bon fonctionnement (approximativement, compte tenu de la remarque précédente). Le choix repose alors sur des critères quantitatifs (performances, fiabilité) ou sur des critères qualitatifs (souplesse, facilité d'utilisation, transportabilité, etc.) que le système CESAR ne considère pas. Certains sont d'ailleurs plus ou moins difficiles à évaluer ou subjectifs (facilité et coût de réalisation, politique de l'entreprise sur les produits nouveaux, etc.). Malgré ces considérations, le modèle utilisé par le système CESAR pour l'analyse des propriétés peut à notre avis être étendu pour prendre en compte des données numériques sur les temps d'exécution des différents traitements, qui peuvent être exploitées pour une évaluation des performances. Il en est de même pour certains calculs élémentaires de fiabilité qui peuvent être effectués sur le modèle si la description

est complétée de données numériques sur la fiabilité des différentes tâches de l'application. Ainsi étendu, le modèle généré par le système CESAR à partir de la description d'une application regrouperait la quasi-totalité des données (objectives) nécessaires au concepteur.

2 - LE SYSTEME CESAR

Le système CESAR tel qu'il est décrit dans cette section (et dans les autres chapitres de la thèse) est en fait une proposition pour un tel système. Différents programmes ont été écrits, pour valider la démarche adoptée et tester la faisabilité des différents algorithmes qu'elle implique, mais un système CESAR totalement intégré n'a pas été réalisé à l'heure actuelle. (Nous ferons le point sur ce qui a été fait et ce qui ne l'a pas été au cours du bilan qui figure dans la conclusion).

2.1 - Le langage de description du système CESAR

Le langage de description du système CESAR, syntaxiquement proche du langage ADA [ADA 80], permet de représenter une application répartie par un ensemble de tâches communiquant par échanges de messages.

Les données (internes aux tâches ou échangées) sont typées. Outre les types standard (entiers, booléens, énumérés) et les constructeurs usuels de types (tableaux, structures), l'utilisateur peut définir des types (dits "non-spécifiés") sans leur associer une implémentation en termes de ces types standard. De même, il peut définir des constantes et manipuler des expressions dont la valeur n'est pas spécifiée.

Les structures syntaxiques usuelles (IF, CASE, WHILE, FOR, dans leur syntaxe ADA) sont étendues de façon à permettre un choix non-déterministe si les conditions dont elles dépendent sont non-spécifiées. En outre, le langage de description du système CESAR (nous dirons souvent le langage CESAR) offre les possibilités suivantes :

- affectation simultanée (vectorielle) de plusieurs variables, pour éviter les contraintes de séquentialité entre affectations, présentes dans les langages de programmation classiques ;

- instructions d'échanges (notées ! et ?) inspirées du langage CSP [HOA 78] ;

- instruction EITHER permettant le choix non-déterministe entre commandes gardées [DIJ 75] comportant éventuellement des échanges (comme dans CSP, mais les deux sens d'échange sont traités identiquement).

Contrairement au langage CSP, les valeurs échangées sont identifiées par des "noms de variables échangées" (analogues aux noms de "portes" de [MM 79], mais les processus partenaires ne le sont pas. Chaque variable échangée peut avoir plusieurs producteurs possibles et plusieurs comsommateurs possibles. Les échanges s'effectuent par rendez-vous selon deux modes possibles :

- mode "diffusion" : le rendez-vous pour l'échange d'une variable a lieu entre l'un quelconque de ses producteurs et tous ses consommateurs,

- mode "anonyme" : le rendez-vous a lieu entre l'un quelconque de ses producteurs et l'un quelconque de ses consommateurs.

Les traitements déclarés (regroupés en tâches ou procédures) peuvent ne pas être décrits (ils sont alors également dits "non-spécifiés").

Une application est décrite de façon hiérarchisée selon une structure arborescente dont les feuilles sont des tâches "élémentaires", qui sont les unités séquentielles d'exécution de l'application (ce sont les seules dont le corps est composé d'instructions). Les autres noeuds sont des tâches "composées", qui permettent de regrouper plusieurs tâches (élémentaires ou composées) s'exécutant en parallèle. Ceci permet, outre une structuration fonctionnelle, de limiter la portée des échanges, en les effectuant au sein de tâches composées (ceci correspond à la notion assez fréquente d'échanges "privés" à un sous-système de l'application).

Le langage CESAR permet également de "typer" une tâche (pour l'utiliser ultérieurement comme "modèle" de tâche, éventuellement paramétré), et de regrouper les tâches et les variables échangées en tableaux.

Le langage de description est présenté de façon plus détaillé dans le deuxième chapitre de cette thèse.

2.2 - Le_langage_de_spécification_du_système_CESAR

Chaque propriété du fonctionnement (propriétés comportementales) est exprimée par une formule d'une logique utilisant des opérateurs "temporels", c.a.d. des opérateurs prenant en compte l'évolution du système décrit.

Les variables propositionnelles de cette logique représentent des prédicats sur l'état courant de l'application décrite et sont de deux types :
- des relations sur les données,
- des états du contrôle des différentes tâches (par rapport aux actions qu'elles peuvent exécuter).

Les opérateurs temporels permettent d'exprimer :

- l'atteignabilité d'un prédicat selon diverses modalités (possible, inévitable ...), éventuellement sous certaines conditions,

- le fait qu'un prédicat reste toujours vrai, ou peut rester toujours vrai au cours de l'évolution du système, éventuellement sous certaines conditions.

La liaison entre la description fournie et les formules de spécification est faite grâce aux variables propositionnelles, où figurent des noms de tâches, d'actions ou de données définies dans la description.

L'existence du non-déterminisme soulève le problème de l'équitabilité (en anglais "fairness") du modèle, problème dû à la possibilité de résoudre les conflits de façon à privilégier certaines actions par rapport à d'autres. Dans une telle situation, certaines actions toujours possibles peuvent ne jamais être exécutées. Ce problème a été traité en détail dans [QS 82c], dont nous n'avons présenté ici que les résultats nécessaires dans le cas du système CESAR.

Le langage de spécification du système CESAR est présenté de façon plus détaillé dans les parties 1 à 4 du troisième chapitre de cette thèse.

2.3 - Organisation_du_système_CESAR

La confrontation de la description d'une application avec ses propriétés exprimées par des formules de spécification s'effectue en deux étapes :

- le programme de description est d'abord traduit en un modèle conservant la totalité de l'information utilisable pour l'analyse,

- chaque formule est ensuite évaluée dans le modèle ainsi généré.

L'organisation du système CESAR est représenté par la figure 1.

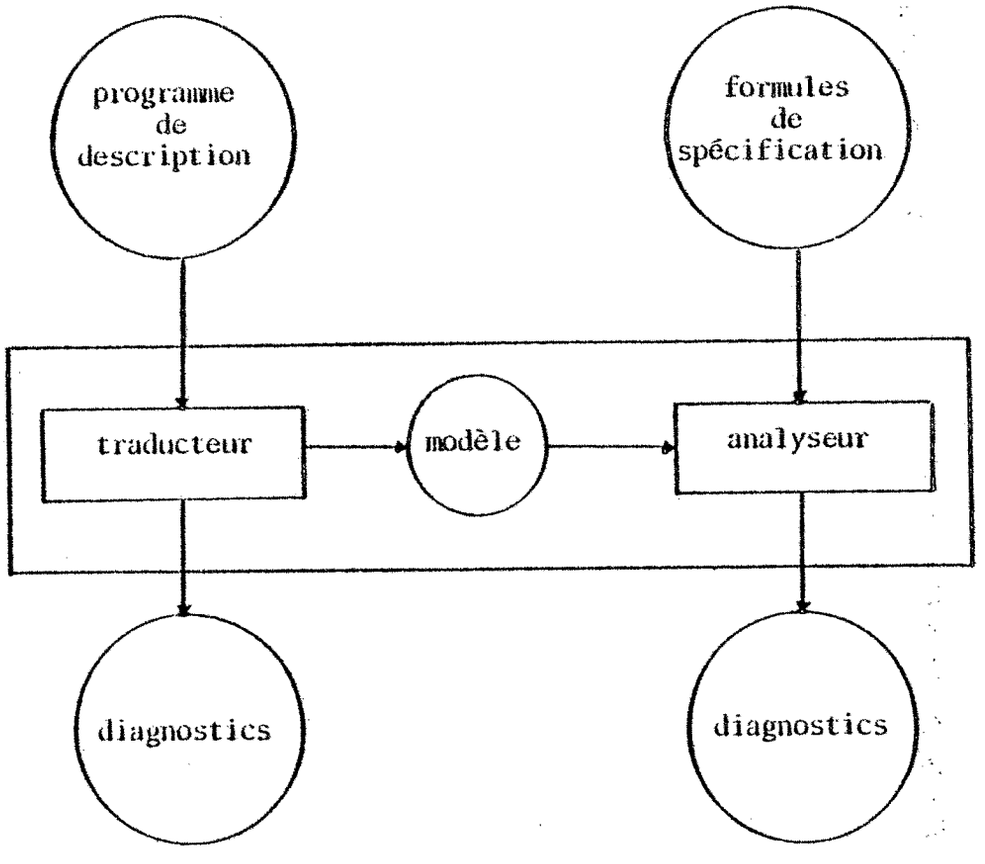


Figure 1 - Organisation du système CESAR

Le modèle retenu est le réseau de Petri interprété, c.a.d. un réseau de Petri muni d'un ensemble de variables, qui sont testées et modifiées par des commandes gardées associées aux transitions. Les raisons objectives de ce choix peuvent se classer en deux catégories (les raisons subjectives sont essentiellement nos travaux antérieurs sur les réseaux de

Petri, qui nous ont familiarisé avec lui et nous en ont fait apprécier les nombreuses possibilités) :

- d'une part : a) les réseaux de Petri permettent de représenter des systèmes parallèles en conservant la décomposition initiale du système en processus (chaque processus étant caractérisé par un ensemble de places) ; b) ils permettent d'exprimer toutes sortes de synchronisations, la plus primitive d'entre elles (qui permet d'exprimer toutes les autres) étant le rendez-vous, représenté par une transition commune à plusieurs processus ; c) ils permettent de représenter naturellement (un peu à la façon des organigrammes) toutes sortes de structures de composition entre instructions (séquentialité, alternative, etc.) ; d) enfin, en leur adjoignant une interprétation sous forme de commandes gardées, on peut conserver la totalité des informations fournies dans le programme de description, le réseau de Petri en représentant en quelque sorte la "partie contrôle". Toutes ces raisons concernent l'expressibilité du modèle, qui doit être élevée afin de ne pas devenir une de ces contraintes d'implémentation que nous avons cherché à éliminer en définissant le langage de description.

- d'autre part : a) la méthode d'analyse des propriétés comportementales des systèmes de transitions (par calcul itératif de points fixes de transformateurs de prédicats [SIF 79]) est directement applicable aux réseaux de Petri interprétés, sans traduction supplémentaire ; b) le réseau de Petri fournit un "squelette du contrôle", qui peut être exploité pour l'analyse car permettant de caractériser les points de contrôle significatifs de chaque processus sans adjonction de variables auxiliaires ; c) de plus, des méthodes plus classiques d'analyse des réseaux de Petri (la réduction [BER 78] par exemple) peuvent être adaptées aux réseaux de Petri interprétés ; d) il est possible d'utiliser les réseaux de Petri temporisés ([SIF 77]) ou stochastiques ([FN 82]) pour l'analyse de performances ou de fiabilité, et qu'il semble donc intéressant d'avoir un modèle proche de ces derniers (ils ont en commun le réseau de Petri), si l'on veut se réserver la possibilité ultérieure d'aborder ces domaines ; e) enfin, les réseaux de Petri interprétés sont aisément simulables, ce qui n'est pas en soi une méthode d'analyse, mais peut-être considéré comme une possibilité supplémentaire agréable, sinon utile. Cette deuxième liste de raisons concerne les possibilités d'analyse du modèle, dont il est bon, quand il s'agit de validation, qu'elles soient nombreuses.

Ajoutons à toutes ces raisons que l'une des critiques que l'on fait usuellement aux réseaux de Petri, à savoir qu'ils perdent toute lisibilité lorsque leur taille devient importante, ne tient pas ici, puisque nous n'utilisons le réseau de Petri que comme modèle intermédiaire que

l'utilisateur final du système CESAR n'a jamais en principe à considérer (bien que nous pensons qu'il est souvent intéressant de connaître le squelette du contrôle de l'application décrite).

La traduction d'un programme de description en un réseau de Petri interprété est menée en deux phases : la traduction de chaque tâche élémentaire, puis la construction des réseaux représentant les tâches composées (et en définitive le système décrit) à partir des réseaux ainsi générés. La première phase relève de la compilation classique (bien que ne générant pas à proprement parler du code exécutable). Elle utilise une grammaire de "briques" de réseaux de Petri interprétés, dont chaque règle représente un sous-réseau associé à une instruction composée du langage de description.

La deuxième phase utilise une opération de composition entre réseaux (analogue en quelque sorte à une édition de liens), fondée sur la fusion des transitions dont l'interprétation comporte des échanges de façon à représenter les rendez-vous.

Les différentes opérations impliquées pour la génération du réseau de Petri interpété représentant un programme de description sont décrites dans le chapitre III de cette thèse.

L'analyse repose sur l'évaluation (pour chaque formuule de spécification) des opérateurs temporels comme points fixes de certains transformateurs de prédicats, chaque variable propositionnelle étant représentée par un prédicat particulier sur l'espace d'états du système de transitions associé au réseau de Petri interprété. La méthode de calcul et les résultats théoriques sur lesquels elle s'appuie sont brièvement présentés dans la cinquième partie du chapitre IV de cette thèse.

3 - IQUB_D'UQBIZQU

Il n'est pas dans notre intention de faire ici un tour d'horizon complet des travaux touchant de près ou de loin à l'un ou l'autre des domaines abordés par le projet CESAR. C'est pourquoi le lecteur ne trouvera ici mention que des travaux qui nous ont directement influencé, ou dont l'approche est similaire.

Les Réseaux de Petri ont été utilisés pour l'expression de la sémantique des primitives de synchronisation (sémaphores et leurs extensions [COO 76] [SZL 77] [QUE 78b], expressions de chemins [LC 74] [LC 75]), puis pour la représentation de types de contrôles parallèles [KOT 76], avant d'être utilisés pour la représentation de la sémantique des langages (langage de type ALGOL [QUE 78a], langages de description de systèmes [JKM 79] [JK 82], et enfin le langage CSP [CMPS 80]).

Le modèle réseau de Petri interprété que nous utilisons est dérivé de celui de [KEL 76], dont les réseaux de Petri à prédicats définis dans [GLT 79] et utilisés par [BT 82] ou [JEN 81] sous le nom de réseaux de Petri colorés, peuvent être considérés comme des variantes.

Pour la définition du langage de description du système CESAR, nous avons été inspiré par "Distributed Processes" [BH 78] et surtout par "Communicating Sequential Processes" [HOA 78] pour les concepts relatifs au parallélisme, et par le langage ADA [ADA 80] pour les autres concepts et l'aspect syntaxique.

La méthode de traduction des tâches élémentaires en réseaux de Petri interprétés est une extension du travail que nous avions fait dans [QUE 78a]. L'opération de composition entre ces réseaux présente des analogies avec celles de [MIL 78], [GUE 81] ou [JOR 82].

La forme du langage de spécification est celle de la logique temporelle. Les travaux actuels utilisant la logique temporelle se classent en deux groupes suivant l'hypothèse qu'ils adoptent pour la représentation du temps et le modèle pour la logique que ce choix induit :

- si l'on adopte l'hypothèse du temps linéaire ("linear time"), le modèle induit est un ensemble de séquences de calcul. Cette approche a été initialement adoptée par [PNU 77] [GPSS 80] ;
- si l'on adopte l'hypothèse du temps arborescent ("branching time"), le modèle induit est un système de transitions (ensemble d'états muni d'une relation représentant les évolutions possibles du système). Cette approche permet de différencier les modalités "Il est possible que" et "Il est inévitable que" [LAM 80]. Elle a

été adoptée par [ABR 79] [CE 81] [BMP 81].

La méthode employée pour l'analyse des spécifications est une concrétisation de la théorie de la preuve des "systèmes conditions-actions" ("Vector Replacement Systems" de [KEL 72]) établie dans [SIF 79] [SIF 80] [SIF 82]. Elle repose sur la caractérisation en termes de points fixes des opérateurs de la logique temporelle utilisée [EC 80] [QS 82b] [QS 82c].

On notera enfin le travail récent de [GDS 82] qui utilise également le modèle Réseaux de Petri Interprétés et la logique temporelle avec une méthode de preuve inspirée des tableaux sémantiques [HC 68].

Chapitre II

LE LANGAGE DE DESCRIPTION

1. Introduction
2. Eléments lexicaux
3. Déclarations
4. Expressions
5. Instructions
6. Procédures
7. Tâches
8. Packages
9. Unités de compilation
10. Deux exemples
11. Conclusion

1 - INTRODUCTION

L'objet de ce chapitre est de présenter le langage de description du système CESAR. Ce chapitre est un résumé très informel du "Rapport de Définition du Langage de Description du Système CESAR" [QUE 82], dont il garde grossièrement l'organisation en sections. Ce langage est un langage de description algorithmique de systèmes distribués en termes d'un ensemble de tâches communiquant par échanges de messages.

Ce langage présente des analogies syntaxiques avec le langage de programmation récemment défini ADA [ADA 80]. Cette ressemblance de forme ne doit pas cacher au lecteur une différence de fond essentielle : ADA est un langage de programmation, son objet final est d'être exécuté par une machine, alors que le langage CESAR est un langage de description de systèmes (qui n'est d'ailleurs pas a priori limité aux processus informatiques). En CESAR il ne s'agit pas de donner une séquence d'instructions en vue de réaliser un calcul, mais de décrire ce qui peut se passer afin d'analyser le comportement du système décrit. Si le langage de description a une forme algorithmique, comme c'est le cas pour CESAR, la différence entre ces deux types de langages est parfois difficile à mettre en évidence. Les exemples qui suivent précisent notre démarche.

* ADA (et la plupart des langages de programmation traditionnels) impose de séquentialiser toutes les instructions (d'une même tâche), y compris les affectations indépendantes, de la forme

X := X + 1 ; Y := 0 ; ...

alors qu'il n'y a aucune raison d'exécuter ces affectations dans cet ordre, on pourrait choisir l'ordre inverse, ou faire les choses simultanément. Cette contrainte est purement une contrainte d'implémentation. Nous avons donc introduit l'affectation vectorielle ou "simultanée", notée

X := X + 1 , Y := 0 , ...

qui est interprétée de la façon suivante : d'abord évaluer (sans effet de bord) les membres droits dans un ordre quelconque, ou simultanément, puis exécuter les affectations (assignation des valeurs calculées aux variables) dans un ordre quelconque, ou simultanément. Ceci permet d'éviter une sur-spécification du système décrit, en réservant les contraintes de séquentialité aux cas où elles sont absolument nécessaires.

* Pour décrire tout ce qui peut se passer dans un processus (qui n'est pas nécessairement informatique) en faisant abstraction de détails relevant de l'implémentation, il apparaît nécessaire de pouvoir décrire le non-déterminisme exprimant le fait que plusieurs choses peuvent se produire.

En ADA, l'instruction SELECT offre un certain non-déterminisme limité au cas des échanges entre tâches. L'instruction EITHER que nous avons introduite est, d'une certaine façon, une généralisation de cette instruction. (Sa sémantique, définie en termes de Réseaux de Petri Interprétés, en est sensiblement différente.)

* Pour écrire un programme il est nécessaire de connaître les types des objets manipulés, en termes des types standard (entier, booléen, ...) et des constructeurs usuels de types (tableaux, structures, ...), afin d'en définir l'implémentation. Par contre, pour décrire, il est certes nécessaire de typer tous les objets, mais sans forcément leur associer une sémantique en termes des types informatiques standard. De plus, si on ne se limite pas au domaine purement informatique, il apparaît des objets dont la nature est totalement différente (par exemple des outils dans un atelier) et dont la description en termes de types informatiques est parfois acrobatique. Nous avons donc introduit la notion de type non-spécifié pour pouvoir typer de tels objets. Nous avons également introduit la notion de valeur et de constante non-spécifiées, qui peuvent représenter une valeur d'un type non spécifié, ou une valeur d'un type spécifié, mais qu'il n'est pas nécessaire de préciser. Dans le même ordre d'idée, nous avons introduit les notions de procédure et de tâche non-spécifiées, pour représenter des traitements non informatiques, ou que l'on ne désire pas décrire.

Outre l'axe DESCRIPTION, il nous a fallu prendre en considération l'axe ANALYSE, qui nous a imposé certaines contraintes. En particulier la méthode d'analyse employée (cf chapitres ultérieurs) utilise un modèle statique. Aussi tous les traits dynamiques (à l'exception de ceux aisément simulables) ont été abandonnés :

- le langage CESAR ne possède pas d'allocation dynamique ; le constructeur de types RECORD existe toutefois pour permettre de décrire des variables structurées.

- les procédures sont considérées comme des "macros", c.a.d. que leur corps est inséré à chaque appel, la récursivité (directe ou non) étant évidemment interdite.

- la structuration de l'application décrite en tâches est statique. Elle est vue comme un arbre, dont les feuilles sont des tâches élémentaires et les autres noeuds des tâches composées ; seules les tâches élémentaires ont un corps représenté par une séquence d'instructions, les tâches composées étant la simple "mise en parallèle" des tâches les composant.

Toutefois, si l'axe ANALYSE est un axe privilégié du système CESAR, le langage n'a pas été conçu uniquement dans ce but, et certaines facilités de description ont été introduites bien que l'analyse ne sache pas les exploiter, ou les exploite imparfaitement. C'est le cas par exemple des valeurs non spécifiées, dans la plupart des cas. De même, les types entiers ne sont pas nécessairement bornés dans le langage de description, bien que l'analyse ne puisse exploiter que les types entiers bornés.

D'autres modifications à la syntaxe d'ADA, et à sa sémantique, ont été effectuées. Nous n'avons pas conservé le mécanisme de rendez-vous d'ADA par appel-réponse, et nous lui avons préféré un rendez-vous élémentaire très proche de celui existant dans CSP [HOA 78], qui était mieux adapté au modèle utilisé pour l'analyse, tout en permettant de réaliser (par plusieurs échanges) des modes d'interaction entre tâches plus complexes. (La sémantique exacte du rendez-vous du langage CESAR est définie par l'opération de composition entre réseaux de Petri interprétés introduite dans le chapitre suivant). Considérant les applications envisagées, nous avons jugé que le rendez-vous "à deux" ne suffisait pas et nous avons introduit le rendez-vous avec diffusion, où un émetteur communique simultanément l'information qu'il produit à tous les consommateurs de celle-ci. Nous avons également introduit un rendez-vous anonyme dans lequel l'un quelconque des consommateurs de la variable échangée la reçoit, et lui seulement. Par souci de propreté, nous avons fait nôtre le principe d'étanchéité entre tâches interdisant les variables communes entre celles-ci, toutes les interactions devant se faire par échanges de messages. D'autres modifications ont été faites encore, soit par souci de généralité, soit pour des raisons de simplicité (par exemple l'abandon de la généricité).

Malgré cette longue liste (pourtant non exhaustive) de transformations et sévices que nous avons fait subir à ADA, son étude nous a permis de confronter les idées (naïves) que nous avions initialement à la réalité (beaucoup plus complexe) d'un langage véritable.

L'ensemble des règles de grammaire est donné en annexe. Les notations employées pour leur écriture y sont précisées. Dans le corps de ce chapitre ne figurent que les règles correspondant aux instructions, qui seront nécessaires à notre propos (cf chapitre suivant).

2 - ELEMENTS LEXICAUX

2.1 - Identificateurs

Un identificateur est une suite de lettres ou chiffres, commençant par une lettre. Le symbole "_" est autorisé et significatif.

Les minuscules sont équivalentes aux majuscules, dans les identificateurs et les mots-clés.

2.2 - Nombres

Tous les nombres manipulés sont des entiers. Ils peuvent être représentés en base 10, 2 ou 16. Les nombres en base 10 peuvent avoir un exposant (puissances de 10 positives uniquement), dénoté par la lettre "E". Les nombres en base 2 ou 16 sont notés entre dièses, précédés de l'indication de la base. Ils ne peuvent pas avoir d'exposant.

Les caractères "_" sont autorisés (pour améliorer la lisibilité des nombres) et sont non significatifs.

2.3 - Commentaires

Un commentaire est introduit par le symbole "--". Le reste de la ligne est alors ignoré.

Certaines commandes au compilateur peuvent être insérées dans le programme de description. Elles commencent par "--$" et sont considérées comme des commentaires du point de vue du langage.

3 - DECLARATIONS

Une déclaration est utilisée pour définir un objet et lui donner un nom (identificateur).

Il y a sept types de déclarations :
- déclaration de constantes (cf 3.1 p. II-5),
- déclaration de variables (cf 3.2 p. II-6),
- déclaration de type (cf 3.3 p. II-6),
- déclaration de procédure (cf 6 p. II-22),
- déclaration de tâche (cf 7 p. II-25),
- déclaration de variables échangées (cf 7.3 p. II-26),
- déclaration de package (cf 8 p. II-35).

Suivant l'environnement, certaines déclarations sont autorisées ou non. On peut distinguer trois environnements types :

* partie déclarative de bloc, procédure ou tâche élémentaire où sont autorisées déclarations de constantes, de variables, de types, et de procédures ;

* partie déclarative de tâche composée où sont autorisées déclarations de constantes, de types, de procédures, de tâches, et de variables échangées ;

* partie déclarative de package où sont autorisées déclarations de constantes, de types, de procédures, et de tâches.

Une déclaration de package forme à elle seule une unité de compilation (cf 9 p. II-37).

3.1 - Déclaration de constantes

Une constante est un objet qui a une valeur donnée, d'un type donné, non modifiable.

Le nom, le type (par défaut INTEGER) et la valeur d'une constante sont définis par sa déclaration.

Sa valeur doit être statiquement évaluable c.a.d. ne dépendre que de valeurs littérales et de constantes déjà définies.

Une constante peut être déclarée sans fournir sa valeur, elle est alors dite non-spécifiée. Une constante non-spécifiée peut être utilisée pour représenter une valeur donnée d'un type non spécifié (cf 3.3.6 p. II-10), ou une valeur donnée d'un type spécifié, qu'il n'est pas nécessaire

de préciser.

Exemples :

```
LIMIT : CONSTANT := 10_000 ;
LOW_LIMIT : CONSTANT := LIMIT / 10 ;
NULL_KEY : CONSTANT KEY ;      -- non spécifiée du type KEY
```

3.2 - Déclaration de variables

Une variable est un objet qui peut avoir une valeur, d'un type donné, modifiable par affectation.

Le nom, le type et la valeur initiale d'une variable sont définis par sa déclaration.

Sa valeur initiale doit être statiquement évaluable. Si la valeur initiale n'est pas fournie, celle-ci est considérée comme non spécifiée.

Les variables ne peuvent être déclarées que dans une partie déclarative d'une tâche élémentaire, d'un bloc ou d'une procédure. On assure ainsi que les différentes tâches du système décrit n'ont pas de variables communes.

Exemples :

```
COUNT, SUM : INTEGER := 0 ;
COLOR_TABLE : ARRAY (1..N) OF COLOR ;  -- non initialisée
```

3.3 - Déclaration de type

Un type caractérise un ensemble de valeurs possibles pour une constante, une variable, un paramètre ou une variable échangée.

Il y a six sortes de types :
- type énuméré,
- type entier,
- type tableau,
- type structure,
- type dérivé,
- type non spécifié.

Un type peut être déclaré directement (sans référence à un autre type) ou par dérivation (restriction de l'ensemble des valeurs possibles d'un type déjà défini, le type source). Dans ce dernier cas l'obligation de faire référence à un type précédemment défini interdit les définitions récursives

(directes ou mutuelles).

Une déclaration de variable ou de constante peut également définir un type (type tableau ou type dérivé) sans lui donner explicitement un nom. Il est appelé type anonyme.

3.3.1 - Type_énuméré

Un type énuméré est un ensemble ordonné de valeurs distinctes représentées par des identificateurs.

Les valeurs d'un type énuméré sont ordonnées par l'ordre de leur énumération dans la déclaration du type.

Un même identificateur peut être utilisé pour désigner une valeur dans des types énumérés distincts, auquel cas il est dit multidéfini.

Il est possible de déclarer un type énuméré dérivé d'un autre type énuméré en restreignant l'ensemble des valeurs possibles au moyen d'une contrainte d'appartenance définissant un intervalle de valeurs possibles (de bornes statiquement évaluables) à l'intérieur du type source.

Le type prédéfini BOOLEAN est équivalent à la définition suivante :

```
TYPE BOOLEAN IS (FALSE, TRUE) ;
```

Exemples :

```
TYPE DAY IS (MON, TUE, WED, THU, FRI, SAT, SUN) ;
TYPE WEEKDAY IS DAY RANGE MON..FRI ;   -- dérivation

TYPE COLOR IS (WHITE, RED, YELLOW, GREEN, BLUE, BROWN,
     BLACK) ;
TYPE LIGHT IS (RED, AMBER, GREEN) ;     -- RED multidéfini
TYPE RAINBOW IS COLOR RANGE RED..BLUE ;
     -- dérivation, RED est du type COLOR
```

3.3.2 - Type_entier

Un type entier est un intervalle (non nécessairement borné) de valeurs entières consécutives signées.

Il est également possible de dériver un type entier d'un autre type entier au moyen d'une contrainte d'appartenance définissant un intervalle à l'intérieur du type source.

Les types prédéfinis INTEGER et NATURAL sont équivalents aux définitions suivantes :

```
TYPE INTEGER IS RANGE <..> ;
TYPE NATURAL IS RANGE 0..> ;
```

La présence du symbole "<" ou ">" indique que la borne correspondante est infinie.

Exemples :

```
TYPE COL_NUM IS RANGE 1..72 ;
TYPE ZONE IS COL_NUM RANGE 1..6 ;        -- dérivation
```

3.3.3 - Type_tableau

Un tableau est un objet formé d'un ensemble d'objets du même type, appelés éléments de tableau, auquel on accède par indexation.

Le nom du type tableau, le nombre, le type et les bornes de ses index, et le type des éléments de tableau sont définis par sa déclaration. Les index d'un tableau doivent être de types discrets (énumérés ou entiers).

Un type tableau peut être déclaré avec certains de ses index spécifiés (c.a.d. que les bornes de l'index sont précisées) et d'autres non (seul le type de l'index est connu et son nom est suivi de "RANGE <>"). Toutefois cela ne signifie pas qu'il y a des tableaux à bornes variables : une variable ou une constante déclarée d'un type tableau doit avoir tous ses index spécifiés au moment de sa déclaration. Cette contrainte ne s'applique pas aux paramètres. Les index spécifiés doivent avoir leurs bornes finies et statiquement évaluables.

Il est possible de dériver un type tableau d'un autre type tableau dont tous les index ne sont pas spécifiés en imposant une contrainte d'index à ce dernier, qui spécifie certains des index qui ne l'étaient pas dans le type source.

Exemples :

```
TYPE TABLE IS ARRAY (1..10) OF INTEGER ;
     -- index spécifié explicitement
TYPE MATRIX IS ARRAY (INTEGER RANGE <>, INTEGER RANGE <>)
     OF INTEGER ;  -- index non spécifiés
-- on peut spécifier l'index soit par dérivation :
TYPE SQUARE IS MATRIX (1..10, 1..10) ;
-- soit à la déclaration d'une variable :
BOARD : MATRIX (1..8, 1..8) ;
```

```
-- on peut également utiliser des types énumérés :
TYPE SCHEDULE IS ARRAY (DAY) OF BOOLEAN ;
     -- index spécifié par le type DAY dans sa totalité
TYPE GEN_SCHEDULE IS ARRAY (DAY RANGE <>) OF BOOLEAN ;
     -- index non spécifié inclus dans le type DAY
TYPE SUB_SCHEDULE IS GEN_SCHEDULE (MON..THU) ;
     -- index spécifié par dérivation
```

3.3.4 - Type_structure

Une structure est un objet formé d'un ensemble d'objets de types différents, chacun étant accessible par son nom (composants de la structure, ou "champs").

Le nom du type structure, le nom et le type de chacun de ses champs sont définis par sa déclaration.

La structure vide est dénotée par le mot-clé NULL.

Une structure peut avoir des discriminants qui sont des paramètres pouvant être utilisés comme constantes à l'intérieur de celle-ci, pour définir le type ou la taille de certains de ses champs. Les discriminants sont nécessairement d'un type discret (énuméré ou entier).

Il est possible de déclarer un type dérivé d'un type à discriminants au moyen d'une contrainte de discriminant, qui fixe la valeur de certains des discriminants (non déjà fixés dans le type source). Les expressions figurant dans une contrainte de discriminant doivent être statiquement évaluables.

Pour déclarer une variable ou une constante d'un type à discriminants il est nécessaire de fixer la valeur de chaque discriminant. Cette contrainte ne s'applique pas aux paramètres.

Une fois fixée, la valeur d'un discriminant ne peut être changée.

Exemples :

```
TYPE DATE IS
RECORD
  DAY : INTEGER RANGE 1..31 ;
  MONTH : MONTH_NAME ;
  YEAR : INTEGER RANGE 0..4000 ;
END RECORD ;
TODAY : DATE := (21, JUL, 1981) ;
     -- déclaration d'une variable initialisée du type DATE
```

```
-- avec la déclaration suivante :
TYPE STRING IS ARRAY (NATURAL RANGE <>) OF CHARACTER ;
     -- tableau à index non spécifié
-- on peut définir la structure :
TYPE BUFFER (SIZE : INTEGER RANGE 0..MAX_SIZE) IS
     -- SIZE est un discriminant, MAX_SIZE une constante
RECORD
  POS : INTEGER RANGE 0..MAX_SIZE ;
  VALUE : STRING (1..SIZE) ;
        -- utilisation du discriminant pour
        -- spécifier l'index du champ
END RECORD ;
-- on fixera le discriminant soit par dérivation :
TYPE BUFFER_10 IS BUFFER (10) ;
-- soit à la déclaration d'une variable :
BUFF : BUFFER (30) ;
```

Note :

Par rapport à ADA, il faut remarquer que les discriminants sont des paramètres statiques de la structure qui ne sont pas modifiables une fois fixés.

3.3.5 - Déclaration de type dérivé

Un type dérivé est un type déclaré à partir d'un autre type, par restriction de l'ensemble de ses valeurs.

On peut dériver :

- un type énuméré ou entier, au moyen d'une contrainte d'appartenance restreignant l'intervalle des valeurs possibles ;
- un type tableau dont tous les index ne sont pas spécifiés, au moyen d'une contrainte d'index en spécifiant certains ;
- un type structure à discriminants, dont tous les discriminants ne sont pas fixés, au moyen d'une contrainte de discriminant en fixant certains ;
- un type non spécifié (avec ou sans discriminant).

Exemples :

Voir les sections correspondant à chacun de ces cas.

3.3.6 - Déclaration de type non spécifié

Un type non-spécifié est un type pour lequel l'ensemble des valeurs n'est pas précisé. Il peut avoir des discriminants.

Les types non spécifiés sont utilisés pour représenter des données pour lesquelles il n'est pas nécessaire de définir précisément leur représentation en termes des types standard.

Les valeurs d'un type non spécifié ne peuvent être que des constantes ou des valeurs non spécifiées (cf 3.1 p. II-5 et 4.2.1 p. II-12).

Les types non spécifiés à discriminants sont soumis aux mêmes contraintes que les structures à discriminants (cf 3.3.4 p. II-9).

Exemples :

```
TYPE CHARACTER ;
TYPE MESSAGE ;
TYPE SIGNAL IS MESSAGE ;        -- dérivation directe
TYPE COEFFICIENT (DIGITS : INTEGER) ;   -- discriminant
TYPE SHORT_COEFFICIENT IS COEFFICIENT (5) ;
     -- dérivation par contrainte de discriminant
```

4 - EXPRESSIONS

4.1 - Opérateurs

Il y a six types d'opérateurs (donnés dans l'ordre des priorités croissantes) :
- opérateurs logiques : et (AND), ou (OR), ou exclusif (XOR),
- opérateurs de comparaison : égal (=), différent (/=), inférieur (<), inférieur ou égal (<=), supérieur (>), supérieur ou égal (>=),
- opérateurs additifs : addition (+), soustraction (-),
- opérateurs unaires : identité (+), changement de signe (-), négation (NOT),
- opérateurs multiplicatifs : multiplication (*), division entière (/), modulo (MOD),
- opérateur d'élévation à la puissance entière (**).

A priorité égale, les expressions sont évaluées de gauche à droite. Un ordre d'évaluation quelconque peut être imposé par un emploi judicieux de parenthèses (convention usuelle). La fonction valeur absolue (ABS) est également disponible.

Pour tous les opérateurs, lorsque l'un des opérandes à un résultat non spécifié, il en est de même du résultat de l'opération, sauf si l'autre opérande à une valeur absorbante (par exemple FALSE pour AND, TRUE pour OR, 0 pour *, etc.).

4.2 - Primaires des expressions

4.2.1 - Littéraux

Les littéraux figurant dans les expressions peuvent être :
- des nombres,
- des identificateurs représentant des valeurs énumérées,
- des valeurs non spécifiées.

Une valeur non spécifiée (d'un type quelconque) est représentée par une chaîne de caractères entre doubles quotes. Elle peut être qualifiée (cf 4.5 p. II-14) pour en préciser le type. Elle peut être utilisée pour représenter une valeur d'un objet de type non spécifié, ou une valeur d'un type spécifié mais qu'il n'a pas été jugé nécessaire de préciser.

Exemples :

```
123_456  -- un littéral entier en base 10
2#1111_1010#  -- un littéral entier en base 2
INTEGER'("good value")  -- valeur non spécifiée qualifiée
```

4.2.2 - Variables et constantes

Les variables et les constantes sont désignées par leur identificateur. On peut accéder à un élément de tableau par indexation. On peut sélectionner un champ ou un discriminant d'une structure en employant son nom préfixé par le nom de la structure. Les deux techniques (indexation et sélection) peuvent être imbriquées autant de fois que nécessaire.

4.2.3 - Attributs

Il y a cinq attributs, notés FIRST, LAST, LENGTH, SUCC et PRED.

Les attributs FIRST, LAST et LENGTH appliqués à un nom de type énuméré ou de type entier s'emploient sans argument et ont les résultats suivants :
- FIRST : première valeur du type considéré,
- LAST : dernière valeur du type considéré,
- LENGTH : nombre de valeurs possibles du type considéré.

Les attributs FIRST, LAST et LENGTH appliqués à un type tableau doivent avoir un argument entre parenthèses statiquement évaluable à résultat entier, qui désigne la dimension concernée. Ils ont les résultats suivants :
- FIRST : première valeur d'indice de la dimension considérée,
- LAST : dernière valeur d'indice de la dimension considérée,
- LENGTH : nombre d'éléments de la dimension considérée.

Les attributs SUCC et PRED appliqués à un type discret sont employés nécessairement avec un argument à résultat de ce type. Ils fournissent la valeur suivante (SUCC) ou précédente (PRED) dans le type considéré.

Les attributs appliqués à une variable ou une constante ont le même effet que s'ils étaient appliqués à son type.

Exemples :

```
COLOR'FIRST = WHITE
COLOR'LAST = BLACK
```

```
RAINBOW'FIRST = RED
RAINBOW'LAST = BLUE
COLOR'SUCC (BLUE) = BROWN
RAINBOW'SUCC (BLUE)  : erreur statiquement détectée
```

4.3 - Expressions composées

Une expression composée est une liste entre parenthèses d'expressions séparées par des virgules, représentant une valeur possible :

* pour une structure : l'expression composée doit comporter une valeur par champ de la structure ;
* pour un tableau unidimensionnel : l'expression composée doit comporter une valeur par élément du tableau.

Un tableau à n dimensions peut être représenté par un tableau unidimensionnel de tableaux à (n-1) dimensions (voir exemple), il est donc possible de représenter une valeur d'un tableau multidimensionnel par une expression composée munie d'un parenthésage adéquat.

Exemples :

```
(4, JUL, 1776)                  -- structure
(1, 2, 3, 4, 5, 6, 7)           -- tableau unidimensionnel
((1, 1), (2, 2), (3, 3))        -- tableau à 2 dimensions 3 x 2
-- c.a.d. dont le premier index a une LENGTH de 3,
-- et le deuxième de 2.
```

4.4 - Expression convertie

Une conversion est possible :

- entre types entiers,
- entre types énumérés compatibles (c.a.d. dérivés l'un de l'autre, ou tous deux d'un même troisième),
- entre tableaux ou structures de mêmes dimensions dont les éléments en correspondance sont convertibles.

S'il y a un risque de débordement au cours de la conversion, une vérification sera effectuée lors de l'analyse.

Exemple :

```
-- la conversion de l'expression E vers le type T est notée
        T (E)
```

4.5 - Expression_qualifiée

La qualification d'une expression (simple ou composée) est utilisée pour indiquer le type qu'a cette expression. Il ne s'agit pas d'une conversion, l'expression qualifiée doit être effectivement de ce type, ou pouvoir l'être (en cas d'ambiguité non levée par le contexte).

La qualification peut être utilisée à titre de clarification d'écriture ; elle est obligatoire pour les littéraux énumérés multidéfinis lorsque l'ambiguité ne peut être levée par le contexte ; elles est également utile pour les expressions composées, et les valeurs non-spécifiées.

Exemple_:

```
COLOR'BLUE          -- la valeur BLUE du type COLOR
RAINBOW'BLUE        -- la valeur BLUE du type RAINBOW.
```

5 - INSTRUCTIONS

Une instruction définit une action effectuée par une tâche du système décrit.

Il y a huit types d'instructions :
- instruction vide (NULL - cf 5.1 p. II-16),
- affectation (et/ou échange) (":=" - cf 5.2 p. II-16 et 7.4 p. II-27),
- instruction conditionnelle (IF - cf 5.3 p. II-17),
- instruction cas (CASE - cf 5.4 p. II-18),
- instruction itérative (LOOP - cf 5.5 p. II-19),
- instruction choix non-déterministe (EITHER - cf 5.6 p. II-20),
- appel de procédure (cf 6.4 p. II-24),
- instructions spéciales (EXIT, RETURN et TERMINATE - cf 5.5 p. II-19, 6.3 p. II-23 et 7.5 p. II-28).

Les instructions sont regroupées en séquences d'instructions (les instructions d'une séquence sont exécutées une à une successivement jusqu'à la fin de la séquence ou jusqu'à ce qu'une instruction EXIT, RETURN ou TERMINATE soit exécutée).

Un bloc (séquence d'instructions éventuellement précédée de déclarations ayant uniquement le bloc pour portée) peut également figurer partout où peut figurer une instruction (cf 5.7 p. II-21).

Les instructions peuvent être étiquetées (au moyen d'un identificateur entre les symboles "<<" et ">>"), ce qui permet de donner des noms à certaines actions du programme de description, qui pourront être utilisés pour la spécification des propriétés du système décrit, et l'analyse de ces spécifications.

5.1 - Instruction_vide

<instruction vide> ::= [<étiquette>]* NULL ;

Cette instruction n'a aucun effet.

5.2 - Affectation

L'affectation modifie simultanément la valeur d'un ensemble de variables. Un échange (cf 7.4 p. II-27) peut être exécuté simultanément à une affectation.

```
<affectation> ::= [ <étiquette> ]* [ <échange> , ]
          <affectation simple> [, <affectation simple>]* ;  |
          [ <étiquette> ]* <échange> ;

<affectation simple> ::= <nom de variable>
          [ , <nom de variable> ]* := <expression>
```

Les noms de variables en partie gauche de toutes les affectations simultanées doivent désigner des variables distinctes, ou des éléments distincts d'un même tableau (si les indices ne sont pas statiquement évaluables un contrôle sera réalisé au cours de l'analyse), ou des champs distincts d'une même structure.

L'exécution simultanée d'un ensemble d'affectations simples doit être considérée comme une affectation vectorielle, c.a.d. toutes les parties droites sont évaluées dans un ordre quelconque, puis chaque affectation est effectuée, dans un ordre quelconque.

Exemple :

```
T(N) := ELT, N := N+1 ;
      -- la valeur de l'indice est l'ancienne valeur de N
```

5.3 - Instruction conditionnelle

L'instruction conditionnelle permet de choisir une séquence d'instructions à exécuter selon la valeur d'un ensemble de conditions, évaluées séquentiellement.

```
<instruction if> ::= [ <étiquette> ]*
          IF <condition> THEN <séquence>
          [ ELSIF <condition> THEN  <séquence> ]*
          [ ELSE <séquence> ] END [IF] ;
```

Pour exécuter une instruction conditionnelle, les différentes conditions sont évaluées dans l'ordre où elles sont rencontrées jusqu'à ce que l'une ayant la valeur TRUE soit rencontrée. On exécute alors la séquence d'instructions correspondante. Si aucune condition n'est vraie, la séquence suivant le mot-clé ELSE est exécutée (s'il y en a une ; l'absence de clause ELSE est équivalente à la clause "ELSE NULL ;").

Lorsqu'une condition figurant dans une instruction conditionnelle a un résultat non spécifié, le choix entre l'exécution de la séquence d'instructions correspondante et le passage à la condition suivante est non déterministe.

Exemple :

```
IF TODAY.MONTH = DEC AND TODAY.DAY = 31 THEN
  TODAY := (1, JAN, TODAY.YEAR + 1) ;
END ;
```

5.4 - Instruction cas

L'instruction cas permet de choisir une séquence d'instructions à exécuter selon la valeur d'une expression d'un type discret.

```
<instruction case> ::= [ <étiquette> ]*
          CASE <expression> IS
          [ WHEN <choix> [ | <choix> ]* => <séquence> ]*
          [ WHEN OTHERS => <séquence> ]
          END [CASE] ;

<choix> ::= <expression> |
          <intervalle discret>
```

Les différents choix sont évalués et la séquence d'instructions correspondant au choix vérifié est exécutée. Un choix peut être :

- une expression statiquement évaluable d'un type discret (le choix est vérifié si l'expression testée lui est égale),
- un intervalle discret de bornes statiquement évaluables (le choix est vérifié si l'expression testée lui appartient).

Les choix doivent être disjoints. Un seul choix peut donc être vérifié par l'expression testée.
Si aucun choix n'est vérifié, la séquence correspondant à la clause OTHERS est exécutée (si elle est présente ; l'absence de clause OTHERS est équivalente à la clause "WHEN OTHERS => NULL ;").

Si l'expression testée a un résultat non spécifié, le choix de la séquence à exécuter est non déterministe.

Exemple :

```
TYPE INFO = (ELEVATION, AZIMUTH, DISTANCE, ... ) ;
SENSOR : INFO ;
...
CASE SENSOR IS
  WHEN ELEVATION => RECORD_ELEVATION (SENSOR_VALUE) ;
  WHEN AZIMUTH => RECORD_AZIMUTH (SENSOR_VALUE) ;
  WHEN DISTANCE => RECORD_DISTANCE (SENSOR_VALUE) ;
END ;
```

5.5 - Instruction_itérative_et_instruction_EXIT

Une instruction itérative permet de répéter l'exécution d'une séquence d'instructions un certain nombre de fois. L'instruction EXIT permet d'interrompre une instruction itérative.

```
<instruction loop> ::= [<étiquette>]* [<clause itérative>]
          <corps de loop>

<corps de loop> ::= LOOP <séquence> END [LOOP] ;

<clause itérative> ::= WHILE <condition> |
          FOR <identificateur> IN [ REVERSE ]
          <contrainte d'appartenance>

<exit> ::= [ <étiquette> ]* EXIT [ WHEN <condition> ] ;
```

En l'absence de clause itérative, la séquence d'instruction est exécutée et répétée tant qu'une instruction EXIT n'est pas exécutée (ou une instruction RETURN ou TERMINATE). L'instruction EXIT ne peut figurer qu'à l'intérieur d'une instruction itérative. Lorsqu'une instruction EXIT est exécutée, la condition après le mot-clé WHEN est évaluée ; si celle-ci est vraie, la boucle LOOP immédiatement englobante est abandonnée ; l'exécution se poursuit alors en séquence immédiatement après le mot-clé END de la boucle.

L'absence de condition dans une instruction EXIT est traitée comme "WHEN TRUE". Si la condition d'une instruction EXIT a un résultat non spécifié le choix entre sortir de la boucle LOOP ou ne pas en sortir est non déterministe.

La clause WHILE signifie que la séquence est exécutée tant que la condition indiquée est vraie. Cette condition est testée à chaque itération avant d'exécuter la séquence.

La condition peut avoir un résultat non spécifié, ce qui introduit à chaque itération un choix non déterministe.

La clause FOR signifie que la séquence est exécutée une fois pour chaque valeur de l'intervalle indiqué. L'identificateur suivant le mot-clé FOR désigne une variable déclarée par la clause FOR elle-même, prenant successivement les différentes valeurs de l'intervalle dans l'ordre croissant (ou décroissant si le mot-clé REVERSE est présent). Les bornes de l'intervalle n'ont pas à être statiquement évaluables. La clause FOR déclare également une autre variable (non accessible à l'utilisateur) mémorisant la borne à atteindre.

A l'intérieur de l'instruction itérative, la variable déclarée par la clause FOR ne peut pas être modifiée par affectation, ou passée en paramètre à une procédure qui modifie ce paramètre.

La borne de départ (borne inférieure dans le cas d'un parcours dans l'ordre croissant, ou borne supérieure dans le cas d'un parcours dans l'ordre décroissant) doit être finie et avoir un résultat spécifié. Par contre la borne à atteindre peut être infinie (ce qui introduit une itération sans fin), ou avoir un résultat non spécifié (ce qui introduit un choix non déterministe à chaque itération).

Exemple :

```
FOR J IN BUFFER'FIRST..BUFFER'LAST LOOP
  IF BUFFER(J) /= SPACE THEN
    PUT(BUFFER(J)) ;
  END IF ;
END LOOP ;
```

5.6 - Instruction choix non-déterministe

L'instruction EITHER offre la possibilité de sélectionner une séquence d'instructions à exécuter d'après les valeurs d'un ensemble de conditions, celles-ci étant toutes évaluées avant le choix (non-déterministe si plusieurs conditions sont vraies) de la séquence à exécuter.

```
<instruction either> ::= [ <étiquette> ]* EITHER
          [ WHEN <condition> => ] <séquence>
          [ OR [ WHEN <condition> => ] <séquence> ]*
          [ ELSE <séquence> ] END [EITHER] ;
```

L'absence de condition est traitée comme la condition TRUE. Le mot-clé ELSE (s'il est présent) remplace une condition égale à l'intersection des négations des autres conditions.

Les conditions peuvent avoir un résultat non spécifié, elles sont alors traitées comme la condition TRUE, et leur négation figurant dans la clause ELSE éventuelle également.

Une séquence d'une instruction EITHER est exécutable si la condition qui la précède est vraie, et si la première instruction de la séquence est elle-même exécutable, ce qui est toujours le cas sauf s'il s'agit d'une affectation comportant un échange, qui peut être exécutable ou non (voir en 7.7 p. II-32 pour l'exécutabilité des échanges entre taches).

Si plusieurs séquences sont exécutables un choix non-déterministe sera fait entre elles et une seule sera exécutée. Si toutes les conditions sont fausses il y blocage (définitif). S'il y a des conditions vraies, mais que les séquences correspondantes ne sont pas exécutables, il y aura attente jusqu'à ce que l'une au moins le devienne (cette attente peut éventuellement ne pas avoir de fin).

Exemple :

```
EITHER
  WHEN NOT BUSY => BUSY:=TRUE ;
OR                 BUSY:=FALSE;
END EITHER ;
```

Note :

La sémantique de l'instruction EITHER est analogue à celle de l'instruction WHEN de "Distributed Processes" [BH 78] et non à celle de l'instruction alternative de CSP [HOA 77]. En effet, si aucune condition n'est vraie, il n'y a pas d'erreur (au sens d'exception provoquant un déroutement) mais une attente infinie (blocage). La méthode d'analyse employée permet de détecter les blocages.

5.7 - Bloc

L'ouverture d'un bloc permet de faire précéder une séquence d'instructions de déclarations qui n'ont que cette séquence pour portée, et sont initialisés à chaque entrée dans le bloc.

```
<bloc> ::= [ DECLARE <partie déclarative de bloc> ]
           [ <étiquette> ]* BEGIN <séquence> END ;
```

La partie déclarative d'un bloc peut contenir des déclarations de variables, mais pas de tâches ou de variables échangées.

Exemple :

```
DECLARE
  I : INTEGER RANGE 1..MAX_ELEM := 1 ;
BEGIN
  ...
END ;
```

6 - PROCEDURES

Une procédure est un traitement séquentiel (admettant des paramètres) pouvant être activé au moyen d'un appel de procédure.

Une procédure est définie par une déclaration de procédure formée :
- d'un en-tête introduit par le mot-clé PROCEDURE, avec une partie formelle,
- d'une partie déclarative,
- d'un corps.

Si seul l'en-tête de la procédure est fourni, la procédure est dite non spécifiée (elle peut être utilisée pour représenter des traitements que l'on ne désire pas préciser au niveau de description où l'on se place).

La partie déclarative suit les mêmes règles qu'une partie déclarative de bloc. La partie déclarative définit des objets qui n'ont pour portée que le corps de la procédure, et sont réinitialisés à chaque appel de la procédure.

6.1 - En-tête de procédure

L'en-tête de procédure définit son nom et la liste des paramètres avec leur nature et type.

Il y a trois catégories de paramètres :
- les paramètres échangés (simples), précédés d'un mot-clé INPUT ou OUTPUT (selon le sens des échanges où ils peuvent figurer),
- les tableaux de paramètres échangés (une définition de tableau précédant le mot-clé indiquant le sens de l'échange),
- les paramètres non échangés (sans mot-clé).

Les paramètres formels sont substitués (dans l'environnement statique où figure l'instruction d'appel) par les paramètres effectifs indiqués par celle-ci.

Les variables passées en paramètres (non échangés) à une procédure non spécifiée ont une valeur non spécifiée après l'appel de la procédure.

Seuls les paramètres échangés figurant dans la partie formelle de la procédure peuvent être échangés dans son corps (et non ceux d'une éventuelle procédure ou tâche englobante). Une procédure ayant des paramètres échangés doit nécessairement être spécifiée.

Exemple :

```
PROCEDURE PROC
  (X : INTEGER ;                    -- paramètre non échangé
   Y : INPUT SIGNAL ;               -- paramètre échangé
   Z : ARRAY (1..6) OF OUTPUT SIGNAL ;
       -- tableau de paramètres échangés
   T : INPUT OF ARRAY (1..80) OF CHARACTER )
       -- paramètre échangé d'un type tableau
       -- et non tableau de paramètres échangés !
IS ...
```

6.2 - Corps de procédure

Le corps d'une procédure est formé d'une séquence d'instructions. Il peut être étiqueté.

Exemple :

```
TYPE STACK (SIZE : INTEGER) IS
RECORD
  INDEX : INTEGER RANGE 0..SIZE ;
  SPACE : ARRAY (1..SIZE) OF ELEMENT_TYPE ;
END STACK ;

PROCEDURE PUSH (E : ELEMENT_TYPE ; S : STACK ;
                C : CONDITION) IS
<<PUSH_BODY>> BEGIN
  IF S.INDEX = S.SIZE THEN
     C := STACK_OVERFLOW ;
  ELSE
     S.INDEX := S.INDEX + 1 ;
     S.SPACE (S.INDEX) := E ;
  END ;
END PUSH ;
```

6.3 - Instruction RETURN

L'instruction RETURN termine l'exécution d'une procédure.

<return> ::= [<étiquette>]* RETURN ;

Cette instruction, qui ne peut figurer que dans un corps de procédure, a pour effet d'arrêter l'exécution de celle-ci et de revenir au programme appelant immédiatement après l'instruction d'appel. Une instruction RETURN implicite figure à la fin du corps de la procédure.

6.4 - Appel de procédure

L'appel d'une procédure indique son nom et les valeurs des paramètres effectifs. L'effet d'un appel de procédure est identique à celui qu'a l'insertion du corps de la procédure (sous la forme d'un bloc précédé de la partie déclarative de la procédure) avec substitution (dans l'environnement statique où figure l'instruction d'appel) des paramètres formels par les paramètres effectifs.

<appel de procédure> ::= [<étiquette>]* <nom de procédure>
[(<liste de paramètres effectifs>)] ;

Suivant la nature des paramètres formels, les paramètres effectifs en correspondance peuvent être :
- paramètre non échangé : n'importe quelle expression (éventuellement avec conversion), si la procédure appelée ne modifie pas par affectation le paramètre formel ; sinon, un nom de variable (ou un nom de paramètre non échangé d'une procédure englobante), éventuellement avec conversion,
- paramètre échangé (simple) : un nom de paramètre échangé (de la tâche ou de la procédure immédiatement englobante), dans le même sens, éventuellement avec conversion, ou un élément d'un tableau de paramètres échangés (paramètre de la procédure ou de la tâche immédiatement englobante),
- tableau de paramètres échangés : un nom d'un tableau de paramètres échangés (paramètre de la tâche ou de la procédure immédiatement englobantee).

Les appels récursifs (directs ou non) sont interdits. Ceci est garanti par le fait qu'une procédure doit être déclarée avant d'être appelée.

Exemple :

```
PUSH (ELT, ST1, COND) ;
```

7 - TACHES

Les tâches sont des unités de traitement pouvant s'exécuter en parallèle. Les tâches communiquent entre elles par rendez-vous permettant l'échange de valeurs identifiées (appelées variables échangées). Elles n'ont pas d'autres variables communes. Toutes les tâches composant le système décrit sont activées indépendamment à l'initialisation du système.

Une tâche peut être simple ou appartenir à un type.

7.1 - Déclaration_de_tâche_simple_ou_de_type_de_tâche

Une tâche simple est définie au moyen d'une déclaration de tâche simple, formée :
- d'un en-tête avec une partie formelle,
- d'une partie déclarative,
- d'un corps.

L'en-tête d'une tâche définit son nom et la liste des paramètres qu'elle admet ou qu'elle échange.

Les paramètres d'une tâche peuvent être :
- des paramètres non échangés utilisables comme constantes à l'intérieur de la tâche,
- des paramètres échangés, ou tableaux de paramètres échangés.

Note : Les paramètres non échangés seront obligatoirement substitués par des constantes. Ils ne peuvent en aucun cas être modifiés (et ne peuvent donc pas servir de variables communes). Fin_de_note.

Si seul l'en-tête de la tâche simple est fourni, la tâche est dite non spécifiée. Une tâche non spécifiée est utilisée pour représenter un traitement que l'on ne désire pas décrire. Une tâche non spécifiée est censée pouvoir effectuer tout échange avec les tâches qui veulent communiquer avec elle (les valeurs émises sont des valeurs non spécifiées).

Une tâche simple spécifiée peut être élémentaire ou composée. Une tâche composée est une tâche formée par l'exécution "en parallèle" de plusieurs tâches (élémentaires ou composées). Une tâche élémentaire est une unité séquentielle d'exécution.

Une déclaration de tâche élémentaire est introduite par le mot-clé TASK, alors qu'une déclaration de tâche composée est

introduite par le mot-clé COTASK. Une tâche non spécifiée est considérée comme élémentaire, elle est donc introduite par le mot-clé TASK.

La partie déclarative d'une tâche élémentaire suit les mêmes règles qu'une partie déclarative de bloc. Elle peut donc contenir des variables, mais non des tâches ou des variables échangées. Les objets déclarés dans la partie déclarative d'une tâche élémentaire ont pour portée le corps de la tâche, et sont initialisés à l'initialisation du système.

La partie déclarative d'une tâche composée peut contenir des tâches et des variables échangées, mais non des variables (non échangées). Les objets déclarés dans la partie déclarative d'une tâche composée ont pour portée le corps de celle-ci.

Exemples :

```
TASK TYPE PHILOSOPHE (I : INTEGER ;
     F_DROITE, F_GAUCHE : EXCHANGED BOOLEAN) IS ...
     -- type de tâche élémentaire
COTASK SUB_SYSTEM (X : OUTPUT MSG) IS ...
       -- tâche simple composée
TASK TYPE USER (REQ : OUTPUT REQUEST) ;
     -- type de tâche non-spécifiée
```

7.2 - Déclaration de tâches d'un type

Une déclaration de tâches d'un type permet de déclarer une ou plusieurs tâches (éventuellement un tableau de tâches) en utilisant un type de tâche préalablement défini.

Il est également possible de déclarer des tableaux de tâches, tous les index devant être spécifiés et avoir leurs bornes statiquement évaluables.

Exemple :

```
POOL : ARRAY (1..10) OF KEYBOARD_DRIVER ;
```

7.3 - Déclaration de variables échangées

Une variable échangée permet d'identifier les communications entre tâches. Elle ne peut être déclarée que dans la partie déclarative d'une tâche composée. Sa déclaration définit son nom et son type.

Il est également possible de déclarer des tableaux de variables échangées, tous les index devant avoir leurs bornes spécifiées et statiquement évaluables.

Exemple :

```
REQUEST : ARRAY (1..10) OF EXCHANGED SIGNAL ;
        -- tableau de variables échangées
WAITING_REQUESTS : EXCHANGED ARRAY (1..10) OF SIGNAL ;
        -- variable échangée unique (d'un type tableau)
```

7.4 - Echange

```
<échange> ::= [ <nom de variable> , ]*
            ! <nom de paramètre échangé> := <expression> !
            ! <nom de paramètre échangé> !
            [ <nom de variable> [ , <nom de variable> ]* :* ]
            ? <nom de paramètre échangé>
```

Un échange est une affectation simple dans laquelle la partie gauche (cas d'une émission) ou la partie droite (cas d'une réception) est un paramètre échangé (dont le nom est précédé du symbole "!" ou "?" suivant qu'il s'agit d'une émission ou d'une réception). Dans le cas où la valeur échangée n'est pas significative (synchronisation pure), l'autre membre de l'affectation peut être omis, ainsi que le symbole ":=". Pour une émission cela est équivalent à l'émission d'une valeur non-spécifiée.

Une expression émise ou un paramètre reçu peuvent simultanément être affectés à plusieurs variables internes.

Les paramètres échangés doivent figurer dans la partie formelle de la procédure immédiatement englobante, si l'instruction figure dans la séquence principale d'une procédure, ou dans la partie formelle de la tâche élémentaire immédiatement englobante si l'instruction figure dans la séquence principale d'une tâche élémentaire.

Un paramètre échangé d'un type structure ou tableau ne peut être échangé que "globalement", c.a.d sans sélection de champs et sans indexation. Par contre, les éléments d'un tableau de paramètres échangés doivent être échangés individuellement, l'indexation est donc obligatoire dans ce cas.

Les échanges sont effectués par rendez-vous entre les tâches impliquées (cf 7.7 p. II-32). Les échanges (parfaitement symétriques du point de vue de la synchronisation) ne permettent un transfert d'information que dans un seul sens.

Exemples :

```
!RELEASE ;          -- synchronisation pure
ELEM(N) := ?OUT_ELEM , N := N+1 ;
```

7.5 - Instruction TERMINATE

<terminate> ::= [<étiquette>]* TERMINATE ;

Cette instruction provoque l'arrêt de la tâche qui l'exécute. Si cette instruction figure dans une procédure, c'est la tâche qui a appelé la procédure qui se termine.

7.6 - Corps de tâches

7.6.1 - Corps de tâche élémentaire

Le corps d'une tâche élémentaire est formé d'une séquence d'instructions.

La tâche termine son exécution lorsqu'elle atteint une instruction TERMINATE, ou à la fin de sa séquence. Une tâche ne doit pas nécessairement se terminer (par exemple si elle contient une instruction LOOP sans clause itérative ni instruction EXIT).

Exemples :

```
TASK RESOURCE (SEIZE, RELEASE : INPUT SIGNAL) IS
   BUSY : BOOLEAN := FALSE ;
BEGIN
   LOOP
      EITHER
        WHEN NOT BUSY => ?SEIZE, BUSY := TRUE ;
      OR                 ?RELEASE, BUSY := FALSE ;
      END EITHER ;
   END LOOP ;
END RESOURCE ;
```

```
TASK PROTECTED_ARRAY
     (READ_N, WRITE_N : INPUT SIGNAL ;
     V : OUTPUT ELEM ; E : INPUT ELEM ) IS

-- cet exemple montre comment représenter le mécanisme
-- d'appel-réponse ADA au moyen de deux échanges consécutifs
```

```
    TABLE : ARRAY (INDEX) OF ELEM := NULL_TABLE ;
    N : INDEX ;

BEGIN
  LOOP
    EITHER
       N := ?READ_N ;
       !V := TABLE(N) ;
    OR
       N := ?WRITE_N ;
       TABLE(N) := ?E ;
    END EITHER ;
  END LOOP ;
END PROTECTED_ARRAY ;
```

7.6.2 - Corps de tâche composée

Le corps d'une tâche composée est formé de la liste des noms des tâches qui la composent, avec leurs paramètres effectifs, séparés par le symbole "//". Cette liste est précédée, si nécessaire, par l'indication du mode de rendez-vous (cf 7.7 p. II-32) employé pour chacune des variables ayant plus d'une tâche réceptrice.

Les paramètres effectifs peuvent être :

- pour un paramètre formel non échangé : une expression statiquement évaluable (ou dépendant de paramètres non échangés de la tâche englobante);

- pour un paramètre formel échangé simple :
  * une variable échangée déclarée dans la tâche composée,
  * un élément d'un tableau de variables échangées déclaré dans la tâche composée,
  * un paramètre échangé (non nécessairement dans le même sens, cf 2ème exemple) de la tâche composée,
  * un élément d'un tableau de paramètres échangés (non nécessairement dans le même sens), paramètre de la tâche composée ;

- pour un paramètre formel tableau de paramètres échangés : un tableau de variables échangées déclaré dans la tâche composée ou un tableau de paramètres échangés (non nécessairement dans le même sens) paramètre de celle-ci ;

(le tout éventuellement avec conversion).

Les appels récursifs (directs ou indirects) de tâches sont interdits. Ils sont rendus impossibles par le fait qu'une tâche doit avoir été déclarée avant d'être utilisée dans le

corps d'une tâche composée.

Exemples :

```
COTASK PROD_CON IS

    TYPE CHARACTER ;  -- type non spécifié

    TASK PRODUCER (CHAR : OUTPUT CHARACTER) IS
    BEGIN
       LOOP
          -- production du caractère CHAR
          !CHAR ;
       END LOOP ;
    END PRODUCER ;

    TASK CONSUMER (CHAR : INPUT CHARACTER) IS
    BEGIN
       LOOP
          ?CHAR ;
          -- consommation du caractère CHAR
       END LOOP ;
    END CONSUMER ;

    TASK BUFFER (C_OUT : OUTPUT CHARACTER ;
                 C_IN  : INPUT  CHARACTER) IS

       POOL_SIZE : CONSTANT INTEGER := 100 ;
       POOL : ARRAY (1..POOL_SIZE) OF CHARACTER ;
       COUNT : INTEGER RANGE 0..POOL_SIZE := 0 ;
       IN_INDEX,OUT_INDEX : INTEGER RANGE 1..POOL_SIZE := 1 ;

    BEGIN  -- corps de BUFFER
       LOOP
          EITHER
            WHEN  COUNT<POOL_SIZE => POOL(IN_INDEX) := ?C_IN,
                     IN_INDEX := IN_INDEX MOD POOL_SIZE + 1,
                     COUNT := COUNT + 1 ;
          OR
            WHEN COUNT>0 => !C_OUT := POOL(OUT_INDEX),
                     OUT_INDEX := OUT_INDEX MOD POOL_SIZE + 1,
                     COUNT := COUNT -1 ;
          END EITHER ;
       END LOOP ;
    END BUFFER ;

    C_READ, C_WRITE : EXCHANGED CHARACTER ;

BEGIN  -- corps de PROD_CON
   PRODUCER (C_WRITE) // CONSUMER (C_READ) //
   BUFFER (C_READ, C_WRITE)
END PROD_CON ;
```

```
TYPE MSG ;          -- type non spécifié

COTASK CT (X : OUTPUT MSG) IS

  TASK P (X1 : OUTPUT MSG) IS
  ...     -- production du message
  END P ;

  TASK C (X2 : INPUT MSG) IS
  ...     -- consommation du message
  END C ;

BEGIN  -- corps de CT
  P (X) // C (X)
END CT ;

-- dans cet exemple, le paramètre X sortant de la tâche
-- composée CT est également consommé à l'intérieur de
-- celle-ci par la tâche C.
-- la figure 1 illustre cette situation.
```

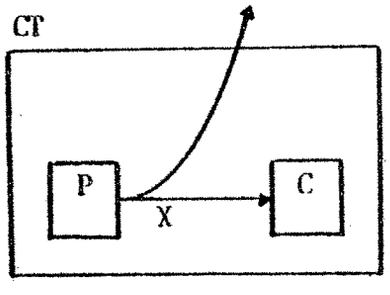


Figure 1

7.7 - Modes_de_rendez-vous

Les échanges entre les tâches composantes d'une tâche composée sont effectués par rendez-vous, c.a.d. que toutes les tâches impliquées dans l'échange doivent être dans un état où l'échange est possible pour que celui-ci puisse être effectué. Un échange possible n'est pas obligatoirement immédiatement exécuté, et peut d'ailleurs être rendu impossible avant que l'échange soit effectivement réalisé, par le fait qu'une de des tâches impliquées sorte de son état de "réceptivité". Lorsque l'échange est effectué, toutes les tâches impliquées effectuent "simultanément" leur instruction

d'échange, ainsi que l'affectation simultanée dont elle fait éventuellement partie, c.a.d. :
- tous les membres droits de toutes les affectations simultanées impliquées par l'échange sont évalués, y compris celui de l'instruction d'émission (qui est unique dans un rendez-vous, cf infra) ;
- les membres droits de la ou des instructions de réception sont remplacés par la valeur en membre droit de l'instruction d'émission ;
- toutes les affectations (y compris celle de la ou des instructions de réception) sont effectuées.

Il y a deux modes de rendez-vous appelés mode "diffusion" et mode "anonyme", désignés respectivement par les mots-clés ALL et ANY. Les modes de rendez-vous sont indiqués pour chacune des variables échangées dans le corps de la tâche composée, avant la liste des tâches composantes.

L'indication du mode pour une variable échangée (ou un paramètre échangé de la tâche composée englobante) entre les tâches composantes indique quel type de rendez-vous est effectué lorsqu'il y a plusieurs tâches recevant cette variable :
- dans le mode ANY : une seule tâche (quelconque dans l'ensemble de celles qui peuvent recevoir la variable) la reçoit,
- dans le mode ALL : toutes les tâches de cet ensemble la reçoivent simultanément (la réception doit donc être simultanément possible dans chacune d'entre elles pour que le rendez-vous puisse avoir lieu).

Une indication de mode est obligatoire pour chaque variable (ou paramètre) que plusieurs tâches reçoivent. Elle est inutile (et non significative) pour les variables (ou paramètres) reçues par une seule tâche.

Une variable échangée (ou un paramètre) peut avoir plusieurs émetteurs possibles. Dans ce cas, il faut et il suffit que l'un quelconque de ceux-ci puisse émettre la variable (ou le paramètre) pour qu'un rendez-vous puisse avoir lieu.

Un rendez-vous a donc lieu :

- pour une variable (ou un paramètre) échangée en mode ANY : entre l'une quelconque des tâches émettrices et l'une quelconque des tâches réceptrices de cette variable (ou ce paramètre),

- pour une variable (ou un paramètre) échangée en mode ALL : entre l'une quelconque des tâches émettrices et toutes les tâches réceptrices simultamément.

Exemple :

```
COTASK SYSTEM IS

   V : EXCHANGED BOOLEAN ;

   COTASK TYPE RECEIVER (Y : INPUT BOOLEAN) IS

      TASK TYPE CONSUMER (Z : INPUT BOOLEAN) IS
      ...

      BEGIN  -- corps de CONSUMER
         ...
         ?Z
         ...
      END CONSUMER ;

      C : ARRAY (1..3) OF CONSUMER ;

   BEGIN  -- corps de RECEIVER
      MODE
         Y : ANY ; -- l'un quelconque des C(i) reçoit Y
      END ;
      C(1)(Y) // C(2)(Y) // C(3)(Y)
   END RECEIVER ;

   R : ARRAY (1..3) OF RECEIVER ;

   TASK EMITTER (X : OUTPUT BOOLEAN) IS
   ...

   BEGIN  -- corps de EMITTER
      ...
      !X
      ...
   END EMITTER ;

BEGIN  -- corps de SYSTEM
   MODE
      V : ALL ;
        -- tous les R(i) reçoivent V simultanément
   END MODE ;
   EMITTER(V) // R(1)(V) // R(2)(V) // R(3)(V)

END SYSTEM ;

-- cet exemple a la structure représentée
-- sur la figure 2
```

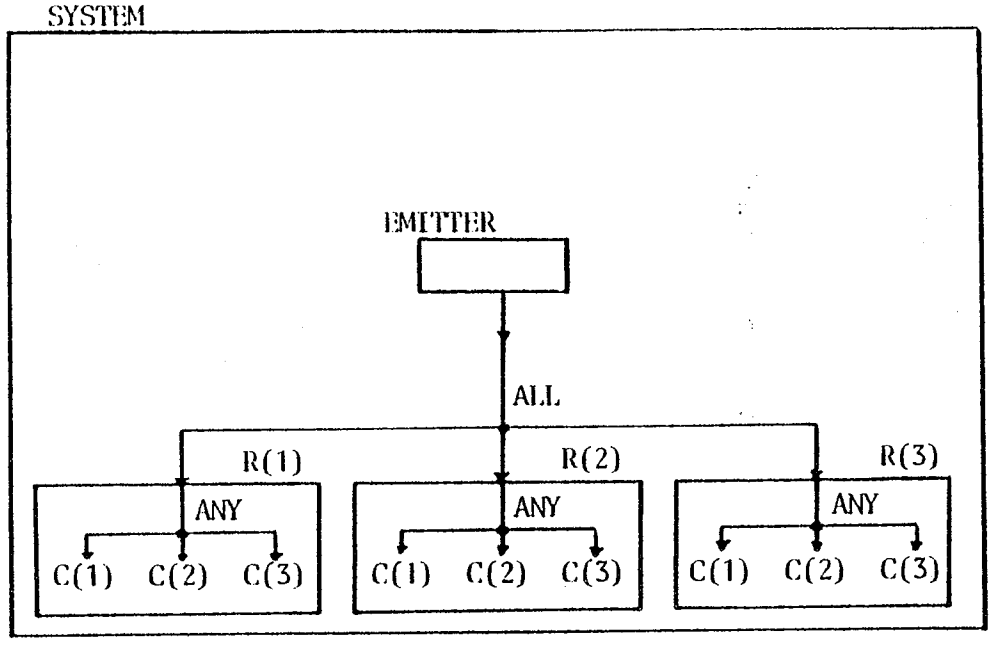


Figure 2

8 - PACKAGES

Un "package" est un ensemble de déclarations (de constantes, de types, de procédures ou de tâches) regroupées en vue de leur utilisation ultérieure.

Un package est formé de deux parties :
- la partie visible du package, définie par sa déclaration,
- le corps du package.

Tous les objets visibles du package (c.a.d. accessibles de l'extérieur du package) sont déclarés dans la partie visible du package. Le corps du package contient les objets non visibles du package utilisés pour implémenter la partie visible, ainsi que les corps des procédures et des tâches de la partie visible (si elles sont spécifiées).

Pour les procédures ou les tâches visibles d'un package, on doit fournir deux déclarations (sauf si elles sont non spécifiées) :
- l'une sans corps dans la partie visible du package,
- l'autre avec corps dans le corps du package.

Les déclarations contenues dans la partie visible du package et la partie déclarative de son corps sont considérées ne former qu'une seule partie déclarative. De ce fait, les objets déclarés dans la partie visible du package sont accessibles au sein de son corps et ne doivent pas être redéclarés à l'intérieur de celui-ci, à l'exception des procédures et tâches spécifiées.

Les objets visibles du package sont accédés à l'extérieur par leur nom, préfixé par le nom du package. Les types standard (BOOLEAN, INTEGER, NATURAL) sont déclarés à l'intérieur du package prédéclaré STANDARD, mais sont dispensés de préfixation. Ce package peut d'ailleurs être configuré différemment selon les besoins de l'utilisateur du système CESAR.

Une déclaration ou un corps de package forme une unité de compilation.

Exemples :

```
PACKAGE CALENDAR IS
  TYPE TIME IS
  RECORD
    YEAR : INTEGER RANGE 1901..2099 ;
    MONTH : INTEGER RANGE 1..12 ;
    DAY : INTEGER RANGE 1..31 ;
  END RECORD ;
  PROCEDURE PLUS (X, Y : TIME ; SUM : TIME) ;
END CALENDAR ;
```

```
PACKAGE RATIONAL_NUMBERS IS
  TYPE RATIONAL IS
  RECORD
    NUMERATOR : INTEGER ;
    DENOMINATOR : INTEGER RANGE 1..> ;
  END;

  PROCEDURE EQUAL (X, Y : RATIONAL ; RESULT : BOOLEAN) ;
  PROCEDURE PLUS (X, Y : RATIONAL ; RESULT : RATIONAL) ;
  PROCEDURE MULT (X, Y : RATIONAL ; RESULT : RATIONAL) ;
END RATIONAL_NUMBERS ;

PACKAGE BODY RATIONAL_NUMBERS IS

  -- procédure utilitaire (non visible)
  PROCEDURE SAME_DENOMINATOR (X, Y : RATIONAL) IS
  BEGIN
  ...  -- reduction au même dénominateur
  END ;

  PROCEDURE EQUAL (X, Y : RATIONAL ; E : BOOLEAN) IS
    U, V : RATIONAL ;   -- variables auxiliaires
  BEGIN
    U := X, V := Y ;
    SAME_DENOMINATOR (U, V) ;
    E := U.NUMERATOR = V.NUMERATOR ;
  END EQUAL ;

  PROCEDURE PLUS (X, Y : RATIONAL ; R : RATIONAL) IS
  ...
  END PLUS ;

  PROCEDURE MULT (X, Y : RATIONAL ; R : RATIONAL) IS
  ...
  END MULT ;

END RATIONAL_NUMBERS ;
```

9 - UNITES_DE_COMPILATION

Un train de compilation est une suite d'unités de compilation, éventuellement précédée de déclarations de types ou de constantes ayant tout le train de compilation pour portée. Une unité de compilation peut être :
- une déclaration de tâche ou de type de tâche,
- une déclaration de procédure,
- une déclaration de package (partie visible),
- un corps de package.

Une clause WITH précédant une unité de compilation permet d'indiquer les packages utilisés pour la compilation de cette unité.

Lorsqu'on utilise la compilation séparée, le système CESAR conserve toutes les unités de compilation précédemment compilées, ce qui permet de les utiliser ultérieurement.

Outre les déclarations et corps de packages, il est possible de compiler séparément un type de tâche ou une procédure.

La commande au compilateur "--$I xxx" permet d'insérer le texte contenu dans le fichier de nom xxx. Ceci permet de regrouper dans un fichier les déclarations de types et de constantes communes aux différentes unités de compilation séparée.

Exemple

* 1ère opération : regrouper les déclarations communes de types et constantes dans un fichier de nom "decl.incl.cesar" par exemple.

* 2ème opération : compiler séparément le type de tâche TA :

```
--$I decl.incl.cesar
TASK TYPE TA (...) IS
...
```

* 3ème opération : procéder de même pour TB :

```
--$I decl.incl.cesar
TASK TYPE TB (...) IS
...
```

* 4ème opération : on peut utiliser TA et TB pour construire la tâche TC :

```
--$I decl.incl.cesar
EXTERNAL TASK TYPE TA, TB ;
  -- le système CESAR sait retrouver les parties formelles
  -- de TA et TB, ce qui en garantit la cohérence.
COTASK TC (...) IS
   A : TA ;
   B : TB ;
   ...
BEGIN
   ... A (...) // B (...) // ...
END TC ;
```

Portée_des_déclarations_et_règles_de_visibilité

Les règles usuelles de portée et de visibilité des objets déclarés dans une structure hiérarchique sont appliquées dans le langage de description de CESAR. La seule exception est le cas des variables et paramètres échangés :
- les instructions d'échanges ne peuvent faire référence qu'à des paramètres échangés de la procédure ou tâche élémentaire immédiatement englobante ;
- les tâches composantes d'une tâche composée ne peuvent utiliser que des paramètres échangés de cette dernière (la tâche composée immédiatement englobante) ou ces variables échangées déclarées dans celle-ci.

En outre, les parties déclaratives de la partie visible d'un package et de son corps sont considérées ne former qu'une seule partie déclarative. De ce fait, les objets déclarés dans la partie visible du package sont accessibles :
- directement, à l'intérieur du corps du package,
- à l'extérieur, par utilisation de la notation préfixée par le nom du package, si le nom du package est mentionné dans une clause WITH précédant l'unité de compilation considérée.

Remarque_:_notion_de_système

Un système est un ensemble de tâches fermé, c.a.d. communicant entre elles mais non avec l'extérieur. Il est usuellement décrit au moyen d'une tâche composée sans partie formelle.

Dans le cas d'un système ouvert (interagissant avec l'extérieur), il est commode de décrire l'environnement par une tâche non spécifiée et de créer une tâche composée représentant le système fermé formé du système ouvert et de son environnement.

10 - DEUX_EXEMPLES

Nous donnons ici deux exemples illustrant l'emploi du langage CESAR pour la description d'applications réparties. Le premier est une version très simplifiée d'un système d'échanges périodiques selon une "trame" prédéfinie sur un bus multiplexé. Il montre comment des contraintes de périodicité sont réalisées au moyen d'une contrainte d'ordonnancement sur les échanges. La solution que nous présentons est réellement utilisée pour des bus multiplexés en avionique ; l'exemple est donc représentatif, si ce n'est sa taille.

Le deuxième est celui du protocole du bit alterné. Nous avons choisi cet exemple, d'une part parce que l'étude des protocoles nous semble être un des domaines d'application du système CESAR ; d'autre part, parce qu'il représente un exemple "réaliste", c.a.d. un assez bon compromis entre un exemple "jouet" (du type "producteur-consommateur") dont la démonstrativité est insuffisante et un exemple "réel" dont le volume trop grand masque les faits importants sous une multitude de détails. De plus cet exemple a été traité plusieurs fois déjà ([SMS 81], [GIR 81], [SIG 81], [MUL 82]), ce qui permet de comparer les approches adoptées.

10.1 - Echanges_périodiques_tramés

Cet exemple très simplifié décrit un système inertiel hypothétique composé de trois tâches dénommées POSITION, DERIVE1 et DERIVE2, qui échangent périodiquement des données.

La tâche POSITION calcule un vecteur de position (X, Y, Z) à partir d'informations qui lui sont personnelles, et des informations de dérive produites par les deux tâches DERIVEi.

Les tâches DERIVEi calculent les vecteurs de dérive (DXi, DYi, DZi). Elles utilisent pour cela le vecteur (X, Y, Z) produit par la tâche POSITION. Elles fournissent également chacune un mot d'état Si, que nous supposerons prendre deux valeurs : OK si tout va bien, KO si elles ne sont plus en état de fournir des informations cohérentes.

Les échanges sont périodiques, leur fréquence étant supposée être :
- pour (X, Y, Z) : 100 Hz,
- pour les (DXi, DYi, DZi) : 50 Hz,
- pour les Si : 25 Hz.

Les différents vecteurs sont supposés transmis par groupes de trois messages consécutifs, correspondant à leurs trois composantes.

Ces trois tâches communiquent via un bus série multiplexé, géré selon une "trame" de 40 ms (activée à la fréquence de 25 Hz) qui donne à chaque tâche l'autorisation d'émettre. Cette trame assure les fréquences relatives des échanges, en minimisant les commutations d'émetteur sur le bus. Elle est divisée en quatre sous-cycles de 10 ms chacun :

* 1er sous-cycle :
  - émission de (X, Y, Z) par la tâche POSITION, puis
  - émission de (DX1, DY1, DZ1) par la tâche DERIVE1, puis
  - émission de S1 par la tâche DERIVE1 ;
* 2ème sous-cycle :
  - émission de (X, Y, Z) par la tâche POSITION, puis
  - émission de (DX2, DY2, DZ2) par la tâche DERIVE2, puis
  - émission de S2 par la tâche DERIVE2 ;
* 3ème sous-cycle :
  - émission de (X, Y, Z) par la tâche POSITION, puis
  - émission de (DX1, DY1, DZ1) par la tâche DERIVE1 ;
* 4ème sous-cycle :
  - émission de (X, Y, Z) par la tâche POSITION, puis
  - émission de (DX2, DY2, DZ2) par la tâche DERIVE2.

Le gestionnaire de la ligne reçoit également les mots d'états S1 et S2, et arrête le système inertiel (à la fin d'un cycle de 40 ms uniquement) si les deux tâches DERIVEi sont hors-service.

Ce système peut être décrit en CESAR de la façon suivante :

```
COTASK SYSTEME_INERTIEL IS

-------------------------------------------------------------
--          Types et variables échangées
-------------------------------------------------------------

TYPE MSG ;    -- informations échangées
TYPE VECTOR IS ARRAY (1..3) OF MSG ;    -- vecteurs
XYZ : EXCHANGED VECTOR ;
DXYZ : ARRAY (1..2) OF EXCHANGED VECTOR ;

TYPE STATUS IS (OK, KO) ;    -- mots d'états
S : ARRAY (1..2) OF STATUS ;

-------------------------------------------------------------
--                  tâche POSITION
-------------------------------------------------------------

TASK POSITION (DXYZ1, DXYZ2 : INPUT VECTOR ;
               S1, S2 : INPUT STATUS ;
               XYZ : OUTPUT VECTOR) IS
```

```
   BEGIN
      LOOP
         EITHER
            -- transmission du vecteur position
            !XYZ ;
         OR
            -- réception des dérives
            ?DXYZ1 ;
         OR
            ?DXYZ2 ;
         OR
            -- réception des mots d'état
            ?S1 ;
         OR
            ?S2 ;
         END EITHER ;
      END LOOP ;
   END POSITION ;

-----------------------------------------------------------
--                Tâches DERIVE1 et DERIVE2
-----------------------------------------------------------

TASK TYPE T_DERIVE (XYZ : INPUT VECTOR ;
                    DXYZ : OUTPUT VECTOR ;
                    S : OUTPUT STATUS) IS

   BEGIN
      LOOP
         EITHER
            -- réception du vecteur position
            ?XYZ ;
         OR
            -- émission de la dérive
            !DXYZ ;
         OR
            -- émission de l'état
            !S ;
         END EITHER ;
      END LOOP ;
   END T_DERIVE ;

DERIVE : ARRAY (1..2) OF T_DERIVE ;
         -- tableau de tâches

-----------------------------------------------------------
--                   contrôle du bus
--
-- cette tâche impose l'ordre des rendez-vous en
-- participant à tous les échanges (en diffusion)
-----------------------------------------------------------
```

```
TASK TRAME (XYZ, DXYZ1, DXYZ2 : INPUT VECTOR ;
            S1, S2 : INPUT STATUS) IS

S_1, S_2 : STATUS := OK ;    -- initialisation

   BEGIN
      -- trame répétée à 25 Hz
      WHILE (S_1 = OK) OR (S_2 = OK) LOOP

         -- 1er sous-cycle
         ?XYZ ; ?DXYZ1 ; S_1 := ?S1 ;

         -- 2ème sous-cycle
         ?XYZ ; ?DXYZ2 ; S_2 := ?S2 ;

         -- 3ème sous-cycle
         ?XYZ ; ?DXYZ1 ;

         -- 4ème sous-cycle
         ?XYZ ; ?DXYZ2 ;

      END LOOP ;
      -- lorsque la trame sort de la boucle, la totalité
      -- du système s'arrête.
   END TRAME ;

-----------------------------------------------------------
--          Corps de SYSTEME_INERTIEL
-----------------------------------------------------------

BEGIN
   MODE
      XYZ, DXYZ, S : ALL ;    -- mode diffusion
   END MODE ;

   POSITION (DXYZ(1), DXYZ(2), S(1), S(2), XYZ) //
   DERIVE(1) (XYZ, DXYZ(1), S(1)) //
   DERIVE(2) (XYZ, DXYZ(2), S(2)) //
   TRAME (XYZ, DXYZ(1), DXYZ(2), S(1), S(2))
END SYSTEME_INERTIEL ;
```

10.2 - Le protocole du bit alterné

Le Protocole du Bit-Alterné, défini dans [BSW 69], a été initialement introduit pour permettre une transmission full-duplex fiable sur une ligne half-duplex. C'est un protocole où l'information de contrôle accompagnant les messages et les acquittements transmis consiste en un seul bit, qui est utilisé pour détecter les pertes (de messages ou d'acquittements). Nous ne nous intéresserons pas aux erreurs de transmission, qui au niveau du protocole peuvent être

assimilées à des pertes de messages (ou d'acquittements). Comme le protocole est totalement symétrique, nous ne considérerons que la transmission des données dans un seul sens (et la transmission des acquittements dans l'autre). Le système décrit sera donc formé de deux processus : un émetteur de messages (nommé SENDER) et un récepteur (nommé RECEIVER). Le moyen de transmission entre l'émetteur et le récepteur sera représenté par deux processus, l'un transmettant les messages (nommé SEND_TO_RECEIVE), l'autre les acquittements (nommé RECEIVE_TO_SEND). Une hypothèse nécessaire au bon fonctionnement du protocole est que le moyen de communication conserve l'ordre des messages et des acquittements émis. Il peut donc être assimilé à une file d'attente ou à un buffer, que nous devrons supposer borné pour éviter les délais de propagation infinis qui ne peuvent être différenciés de la perte effective du message ou de l'acquittement. La taille du buffer n'étant pas significative pour le fonctionnement du protocole, nous fixerons pour simplifier la description celle-ci à 1.

Une description possible en langue naturelle de ce protocole est la suivante. L'émetteur envoie des messages au récepteur, qui lui répond par des acquittements. L'émetteur associe à chaque message qu'il envoie un bit de contrôle, qui prend alternativement les valeurs 0 et 1. Après avoir émis un message, l'émetteur n'envoie pas le message suivant tant qu'il n'a pas reçu l'acquittement correspondant (un acquittement avec le même bit de contrôle que le message émis). Pour parer aux pertes de messages ou d'acquittements, l'émetteur attend un délai fini (mesuré par une horloge arbitraire locale à l'émetteur) et répète alors le même message (sans en changer le bit de contrôle).

Le récepteur se comporte de façon symétrique. Après avoir reçu un message, il envoie un acquittement avec le même bit de contrôle, et attend le prochain message (avec un bit de contrôle complémentaire de celui du message reçu). Si le message attendu n'arrive pas dans un délai fini (mesuré par une horloge arbitraire locale au récepteur), le récepteur renvoie l'acquittement précédent.

Si on suppose que la ligne ne peut pas perdre tous les messages ou tous les acquittements (c.a.d. qu'elle n'est pas définitivement coupée dans un sens ou dans l'autre), ce protocole assure la transmission de tous les messages (et acquittements) après un nombre suffisant de répétitions. Les répétitions de messages qui n'ont pas été réellement perdus ne posent pas de problème, car le protocole garantit que toute suite de messages reçus consécutivement avec le même bit de contrôle est formée uniquement de répétitions du même message (puisque la ligne conserve l'ordre des messages émis, et que l'émetteur change la valeur du bit de contrôle chaque fois qu'un nouveau message est émis). De ce fait le

récepteur peut ignorer tous les messages consécutifs ayant le même bit de contrôle (à l'exception du premier, qui est effectivement un nouveau message). De façon symétrique, l'émetteur peut ignorer les acquittements dupliqués.

Nous donnons ici deux versions différant par la façon de représenter les bits de contrôle transmis avec les messages et les acquittements.

Dans la première version nous les avons représentés par un champ des structures représentant les messages et les acquittements. C'est l'approche la plus naturelle ; elle permet une description assez facile et concise du protocole. Dans cette version, il est évident que les valeurs échangées ont une influence sur le comportement du système décrit.

Pour la deuxième version nous avons différencié par leur type les messages accompagnés d'un bit de contrôle à 0 et ceux accompagnés d'un bit de contrôle à 1 (de même pour les acquittements). Cette approche impose de dédoubler tous les traitements suivant que le bit de contrôle est 0 ou 1, le volume de la description est donc environ deux fois plus important. Toutefois dans cette version la totalité du comportement est décrite par les structures syntaxiques et les rendez-vous, et de ce fait les valeurs échangées ne sont pas significatives.

Cette dialectique de la description en termes de données ou de contrôle se présente dans tous les cas. La première approche permet une description plus facile, mais certaines choses sont cachées du fait de l'influence des données sur le comportement. La deuxième approche met mieux en évidence le comportement, mais présente les inconvénients d'être plus difficile et de nécessiter une description plus volumineuse.

10.2.1 - Première_version

```
COTASK AB_PROTOCOL_V1 IS

-------------------------------------------------------------
--                  Déclarations de types
-------------------------------------------------------------

TYPE DATA ;  -- partie données du message
TYPE PATTERN ;  -- patterns de bits

TYPE MSG IS RECORD
   MESSAGE : DATA ;
   B : BOOLEAN ;  -- bit de contrôle
END MSG ;
```

```
TYPE ACK IS RECORD
   ACKNOWLEDGE : PATTERN ;
   B : BOOLEAN ;  -- bit de contrôle
END ACK ;

-------------------------------------------------------------
--                      EMETTEUR
-------------------------------------------------------------

TASK SENDER (M : OUTPUT MSG ; A : INPUT ACK) IS

   X : DATA ;
   BIT : BOOLEAN := FALSE ;
   RECEIVED_ACK : ACK ;

   BEGIN
      LOOP  -- boucle principale
         <<SEND>> !M := (X, BIT) ;  -- envoi du message
         LOOP  -- boucle d'attente de l'ack
            EITHER
               <<RECEIVE_ACK>> RECEIVED_ACK := ?A ;
               -- réception d'un ack
               IF RECEIVED_ACK.B = BIT THEN
                  -- c'est l'ack attendu
                  -- on passe au message suivant
                  <<ACCEPT_ACK>> BIT := NOT BIT ;
                  EXIT ;  -- sortie de la boucle d'attente
               ELSE
                  <<SKIP_ACK>> NULL ;  -- sinon on saute
               END IF ;
            OR
               <<REPEAT>> !M := (X, BIT) ;
               -- répétition du message
            END EITHER ;
         END LOOP ;  -- fin boucle d'attente
      END LOOP ;  -- fin boucle principale
   END SENDER ;

-------------------------------------------------------------
--                      RECEPTEUR
-------------------------------------------------------------

TASK RECEIVER (M : INPUT MSG ; A : OUTPUT ACK) IS

   BIT : BOOLEAN := FALSE ;
   RECEIVED_MSG : MSG ;
```

```
   BEGIN
      LOOP  -- boucle principale
         LOOP  -- boucle d'attente de message
            EITHER
               <<RECEIVE>> RECEIVED_MSG := ?M ;
               -- réception d'un message
               IF RECEIVED_MSG.B = BIT THEN
                  -- c'est le message attendu
                  -- on sort de la boucle
                  -- pour émettre l'ack
                  <<ACCEPT>> EXIT ;
               ELSE
                  <<SKIP>> NULL ;  -- on saute le message
               END IF ;
            OR
               <<REPEAT_ACK>>
               !A := (PATTERN'("ack"), NOT BIT) ;
               -- répétition de l'ack précédent
            END EITHER ;
         END LOOP ;  -- fin de boucle d'attente
         <<SEND_ACK>>
         !A := (PATTERN'("ack"), BIT), BIT := NOT BIT ;
         -- envoi de l'ack, et changement du bit de contrôle
      END LOOP ;  -- fin boucle principale
   END RECEIVER ;

-----------------------------------------------------------------
--                      LIGNE DE TRANSMISSION
--              décrite par deux tâches indépendantes
-----------------------------------------------------------------

TASK SEND_TO_RECEIVE (M : INPUT MSG ; MM : OUTPUT MSG) IS

   MMM : MSG ;  -- variable tampon

   BEGIN
      LOOP
         <<GET>> MMM := ?M ;
         -- le message émis est échantillonné
         EITHER
            <<TRANSMIT>> !MM := MMM ;
            -- le message est transmis
         OR
            <<LOOSE>> NULL ;  -- le message est perdu
         END EITHER ;
      END LOOP ;
   END SEND_TO_RECEIVE ;

TASK RECEIVE_TO_SEND (A : INPUT ACK ; AA : OUTPUT ACK) IS

   AAA : ACK ;  -- variable tampon
```

```
   BEGIN
      LOOP
         <<GET_ACK>> AAA := ?A ;
         -- l'ack émis est échantillonné
         EITHER
            <<TRANSMIT_ACK>> !AA := AAA ;
            -- l'ack est transmis
         OR
            <<LOOSE_ACK>> NULL ;  -- l'ack est perdu
         END EITHER ;
      END LOOP ;
   END RECEIVE_TO_SEND ;

-----------------------------------------------------------------
--                        Variables échangées
-----------------------------------------------------------------

M1, M2 : EXCHANGED MSG ;
A1, A2 : EXCHANGED ACK ;

-----------------------------------------------------------------
--                          corps de AB_PROTOCOL
-----------------------------------------------------------------

BEGIN
   SENDER (M1, A1) //
   RECEIVER (M2, A2) //
   SEND_TO_RECEIVE (M1, M2) //
   RECEIVE_TO_SEND (A2, A1)
END AB_PROTOCOL_V1 ;
```

10.2.2 - Seconde version

```
COTASK AB_PROTOCOL_V2 IS

-----------------------------------------------------------------
--                      Déclarations de types
-----------------------------------------------------------------

TYPE MSG (B : BOOLEAN) ;
TYPE ACK (B : BOOLEAN) ;
-- le bit de contrôle est un discriminant
-- des types non-spécifiés MSG et ACK

TYPE MSG0 IS MSG (FALSE);
TYPE MSG1 IS MSG (TRUE) ;
TYPE ACK0 IS ACK (FALSE);
TYPE ACK1 IS ACK (TRUE) ;
```

```
-------------------------------------------------------------
--                      EMETTEUR
-------------------------------------------------------------

TASK SENDER (M0 : OUTPUT MSG0 ; M1 : OUTPUT MSG1 ;
             A0 : INPUT ACK0 ; A1 : INPUT ACK1) IS

   BEGIN
      LOOP  -- boucle principale
         -- émission d'un message avec bit à 0
         <<SEND0>> !M0 ;  -- envoi du message
         LOOP  -- boucle d'attente de l'ack
            EITHER
               <<RECEIVE_ACK0>> ?A0 ;
               -- c'est le bon ack
               -- on passe au message suivant
               EXIT ;  -- sortie de la boucle d'attente
            OR
               <<SKIP_ACK1>> ?A1 ;  -- on saute
            OR
               <<REPEAT1>> !M1 ;
               -- répétition du message précédent
            END EITHER ;
         END LOOP ;  -- fin boucle d'attente

         -- émission d'un message avec bit à 1
         <<SEND1>> !M1 ;  -- envoi du message
         LOOP  -- boucle d'attente de l'ack
            EITHER
               <<RECEIVE_ACK1>> ?A1 ;
               -- c'est le bon ack
               -- on passe au message suivant
               EXIT ;  -- sortie de la boucle d'attente
            OR
               <<SKIP_ACK0>> ?A0 ;  -- on saute
            OR
               <<REPEAT0>> !M0 ;
               -- répétition du message précédent
            END EITHER ;
         END LOOP ;  -- fin boucle d'attente
      END LOOP ;  -- fin boucle principale
   END SENDER ;

-------------------------------------------------------------
--                      RECEPTEUR
-------------------------------------------------------------

TASK RECEIVER (M0 : INPUT MSG0 ; M1 : INPUT MSG1 ;
               A0 : OUTPUT ACK0 ; A1 : OUTPUT ACK1) IS
```

```
   BEGIN
      LOOP  -- boucle principale
         -- réception d'un message avec bit à 0
         LOOP  -- boucle d'attente de message
            EITHER
               <<RECEIVE0>> ?M0 ;
               -- c'est le message attendu
               -- on sort de la boucle
               -- pour émettre l'ack
               EXIT ;
            OR
               <<SKIP1>> ?M1 ;  -- on saute
            OR
               <<REPEAT_ACK1>> !A1 ;
               -- on répète l'ack précédent
            END EITHER ;
         END LOOP ;  -- fin de boucle d'attente
         <<SEND_ACK0>> !A0 ;  -- envoi de l'ack

         -- réception d'un message avec bit à 1
         LOOP  -- boucle d'attente de message
            EITHER
               <<RECEIVE1>> ?M1 ;
               -- c'est le message attendu
               -- on sort de la boucle
               -- pour émettre l'ack
               EXIT ;
            OR
               <<SKIP0>> ?M0 ;  -- on saute
            OR
               <<REPEAT_ACK0>> !A0 ;
               -- on répète l'ack précédent
            END EITHER ;
         END LOOP ;  -- fin de boucle d'attente
         <<SEND_ACK1>> !A1 ;  -- envoi de l'ack
      END LOOP ;  -- fin boucle principale
   END RECEIVER ;

-------------------------------------------------------------
--                    LIGNE DE TRANSMISSION
--              décrite par deux tâches indépendantes
-------------------------------------------------------------

TASK SEND_TO_RECEIVE (M0 : INPUT MSG0 ; M1 : INPUT MSG1 ;
                      MM0 : OUTPUT MSG0 ; MM1 : OUTPUT MSG1)
IS
```

```
   BEGIN
      LOOP
         EITHER
            -- transmission d'un message avec bit à 0
            <<GET0>> ?M0 ;
            -- le message émis est échantillonné
            EITHER
               <<TRANSMIT0>> !MM0 ;
               -- le message est transmis
            OR
               <<LOOSE0>> NULL ;  -- le message est perdu
            END EITHER ;
         OR
            -- transmission d'un message avec bit à 1
            <<GET1>> ?M1 ;
            -- le message émis est échantillonné
            EITHER
               <<TRANSMIT1>> !MM1 ;
               -- le message est transmis
            OR
               <<LOOSE1>> NULL ;  -- le message est perdu
            END EITHER ;
         END EITHER ;
      END LOOP ;
   END SEND_TO_RECEIVE ;

TASK RECEIVE_TO_SEND (A0 : INPUT ACK0 ; A1 : INPUT ACK1 ;
                      AA0 : OUTPUT ACK0 ; AA1 : OUTPUT ACK1)
IS

   BEGIN
      LOOP
         EITHER
            -- transmission d'un ack avec bit à 0
            <<GET_ACK0>> ?A0 ;
            -- l'ack émis est échantillonné
            EITHER
               <<TRANSMIT_ACK0>> !AA0 ;
               -- l'ack est transmis
            OR
               <<LOOSE_ACK0>> NULL ;  -- l'ack est perdu
            END EITHER ;
```

```
         OR
            -- transmission d'un ack avec bit à 1
            <<GET_ACK1>> ?A1 ;
            -- l'ack émis est échantillonné
            EITHER
               <<TRANSMIT_ACK1>> !AA1 ;
               -- l'ack est transmis
            OR
               <<LOOSE_ACK1>> NULL ;  -- l'ack est perdu
            END EITHER ;
         END EITHER ;
      END LOOP ;
   END RECEIVE_TO_SEND ;

------------------------------------------------------------------
--                        Variables échangées
------------------------------------------------------------------

M10, M20 : EXCHANGED MSG0 ;
M11, M21 : EXCHANGED MSG1 ;
A10, A20 : EXCHANGED ACK0 ;
A11, A21 : EXCHANGED ACK1 ;

------------------------------------------------------------------
--                          corps de AB_PROTOCOL
------------------------------------------------------------------

BEGIN
   SENDER (M10, M11, A10, A11) //
   RECEIVER (M20, M21, A20, A21) //
   SEND_TO_RECEIVE (M10, M11, M20, M21) //
   RECEIVE_TO_SEND (A20, A21, A10, A11)
END AB_PROTOCOL_V2 ;
```

11 - CONCLUSION

Dans ce chapitre nous avons présenté le langage de description du système CESAR. Un travail important a été effectué pour en faire un "véritable" langage, immédiatement utilisable, et non un langage "jouet", comme celui que nous avions esquissé dans [QUE 81].

La syntaxe concrète est parente de celle d'ADA. Cela est voulu, car cela peut fournir au langage CESAR la possibilité de s'insérer naturellement dans la chaîne des outils de l'environnement de programmation ADA. De plus cette parenté permet à un utilisateur familiarisé avec ADA une absorption plus facile des détails syntaxiques (sans trop de risques de confusion, puisque les concepts nouveaux sont introduits par des mots-clés nouveaux : EITHER ou COTASK par exemple). Cette ressemblance de forme lui donne toutefois une apparence de langage de programmation classique. En fait, nous avons introduit des notions originales (tous les objets non spécifiés : type, constante, valeur littérale, procédure, tâche ; extension non-déterministe des instructions usuelles) qui permettent une description progressive, en faisant abstraction des détails d'implémentation non significatifs pour le comportement étudié.

Ce langage comporte quelques restrictions, dont certaines peuvent sembler peu nécessaires (constantes statiquement évaluables, absence de récursivité, etc.). Cela tient au fait que le langage proposé s'inscrit dans le cadre du projet CESAR, dont l'objectif premier est la vérification des propriétés du comportement du système décrit, et non la description pour elle-même.

Chapitre III

LE MODELE

1. Introduction
2. Définitions
3. Principe d'obtention du modèle
4. Traductions des instructions
5. Traductions des blocs, procédures et tâches élémentaires ou non-spécifiées
6. Traduction des tâches composées
7. Deux exemples
8. Conclusion

1 - INTRODUCTION

L'objet de ce chapitre est de donner la méthode de traduction d'un programme de description représentant une application répartie, décrite en termes de tâches élémentaires et composées échangeant des messages par rendez-vous, en un modèle sur lequel pourra s'effectuer l'analyse des propriétés de bon fonctionnement requises.

Le modèle retenu est le réseau de Petri interprété et étiqueté. Le réseau de Petri représente la structure "syntaxique" de l'application, c.a.d. le contrôle imposé par les instructions composées à l'intérieur d'une tâche élémentaire, le parallélisme entre tâches élémentaires, et les rendez-vous entre elles. L'interprétation associée aux transitions utilise un ensemble de variables qui ne sont autres que celles du programme de description, et représente les tests et les actions faites sur ces variables, et donc l'influence de celles-ci sur le comportement imposé par le réseau. Les étiquettes, également associées aux transitions, permettent d'identifier les transitions représentant les actions étiquetées dans le programme de description.

La traduction des procédures et des tâches élémentaires repose sur l'utilisation d'une grammaire de briques, dont chaque règle permet de produire une brique de réseau de Petri interprété et étiqueté à partir d'une instruction du programme de description. Les appels de procédures sont traduits par l'insertion à leur place du réseau représentant le corps de la procédure (opération de substitution). Le réseau représentant une tâche composée est obtenu par association des différents réseaux représentant les différentes tâches composantes au moyen de l'opération de composition, puis en ramenant les possibilités de communications du réseau ainsi obtenu à celles autorisées par la partie formelle de la tâche composée (opération d'encapsulation).

Les principes de cette traduction ont été présentés pour un langage jouet dans [QUE 81].

2 - DEFINITIONS

On a regroupé ici la quasi totalité des définitions nécessaires à la lecture de la suite du chapitre III. On rappelle d'abord les définitions classiques relatives aux Réseaux de Petri ; on introduit ensuite les notions de réseaux de Petri interprétés, puis communicants. Les réseaux de Petri interprétés que nous définissons ici sont dérivés du modèle de [KEL 76]. Un cas particulier de réseau de Petri, le "graphe de processus" est également traité ; il correspond à la structure particulière des réseaux générés par le traducteur du langage CESAR. On introduit ensuite le réseau de Petri "procédural", modèle qui nous sera pratique pour représenter les appels de procédures figurant dans le programme de description. Enfin on précisera la nature des étiquettes associées aux transitions pour représenter les actions étiquetées de ce même programme.

2.1 - Réseau_de_Petri

Définition :

Un réseau de Petri (ou RdP) est un quadruplet

R = (P, T, alpha, beta)

tel que :

- P est un ensemble d'objets appelés "places",
- T est un ensemble d'objets appelés "transitions",
- alpha est une relation de P vers T appelée "relation d'incidence avant",
- beta est une relation de T vers P appelée "relation d'incidence arrière".

Représentation graphique :

On représente un réseau de Petri par un graphe biparti orienté dont les noeuds sont les places et les transitions représentées respectivement par des cercles et des rectangles (cf figure 1), tel que :

- il y a un arc de la place pi à la transition tj ssi (pi,tj) appartient à alpha,
- il y a un arc de la transition ti à la place pj ssi (ti,pj) appartient à beta.

Figure 1

Exemple :

Soit le réseau R1 = (P1, T1, alpha1, beta1) tel que :
P1 = {p1, p2, p3}
T1 = {t1, t2}
alpha1 = {(p1, t1), (p2, t1), (p3, t2)}
beta1 = {(t1, p3), (t2, p1)}
Il est représenté par la figure 2.

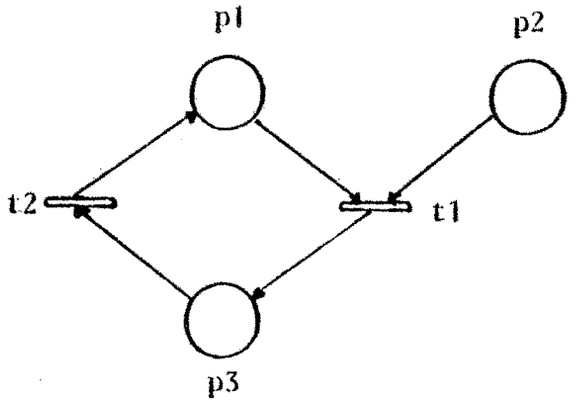


Figure 2

Notations :

Etant donné un RdP R = (P, T, alpha, beta), une transition t0 de T et une place p0 de P, on utilisera les notations suivantes :

- >t0 l'ensemble des places d'entrée de la transition t0, c.a.d. l'ensemble des places p telles que (p, t0) vérifie la relation alpha,

- t0> l'ensemble des places de sortie de la transition t0, c.a.d. l'ensemble des places p telles que (t0, p) vérifie la relation beta ;

- >p0 l'ensemble des transitions d'entrée de la place p0, c.a.d. l'ensemble des transitions t telles que (t, p0) vérifie la relation beta ;

- p0> l'ensemble des transitions de sortie de la place p0, c.a.d. l'ensemble des transitions t telles que (p0, t) vérifie la relation alpha.

Exemple :

Pour le RdP R1 de la figure 2 p. III-4 on a

>t1 = {p1, p2}	>p1 = {t2}
t1> = {p3}	p1> = {t1}

Marquage d'un RdP

Etant donné un RdP R = (P, T, alpha, beta), on définit un marquage M de R comme étant une application de P dans l'ensemble N des entiers naturels.

Représentation graphique :

Sur la représentation graphique, on représentera les marques par des points à l'intérieur des places, une place p contenant M(p) marques.

Transition validée

Etant donné un RdP R = (P, T, alpha, beta), une transition t est dite validée par le marquage M ssi

pour toute place p de >t : M(p) > 0.

Mise à feu d'une transition validée

Etant donné un RdP R = (P, T, alpha, beta) et MM(R) l'ensemble des marquages de R, on définit la mise à feu d'une transition t validée comme une application de MM(R) dans lui-même qui au marquage M fait correspondre le marquage M' tel que :

pour tout p de P :

M'(p) = M(p)-1 si p appartient à >t et non à t>
M(p)+1 si p appartient à t> et non à >t
M(p) sinon.

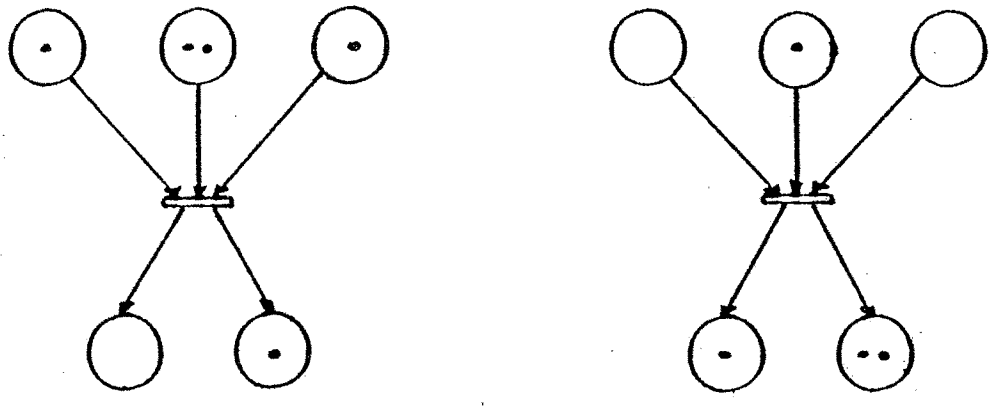
Exemple : (figure 3).

Notations :

Etant donné une transition t et un marquage M de R, on notera :

M-t->

la proposition : "la transition t est validée par M".



avant mise à feu — après mise à feu

Figure 3

Etant donné une transition t, et deux marquages M et M' de R, on notera :

M-t->M'

la proposition : "la mise à feu de t transforme M en M'".

Etant donné une séquence de transitions t1t2...tn, on notera

M-t1t2...tn->M'

la proposition : "il existe une suite de marquages M0, M1, ... Mn telle que :
- M = M0, M' = Mn
- pour tout k de 0 à n-1 : Mk-t(k+1)->M(k+1)" ;

et

M-*->M'

la proposition : "il existe une séquence de transitions t1t2...tn (de longueur n>0) telle que M-t1t2...tn->M', ou M = M'". On appellera M' successeur de M.

Enfin on notera M*> l'ensemble des marquages successeurs de M, c.a.d. l'ensemble des marquages M' tel que M-*->M'.

Fonctionnement d'un RdP

Etant donné un RdP R et un marquage M0 élément de MM(R), on peut faire évoluer le marquage du réseau à partir du marquage initial M0 en appliquant l'algorithme suivant (on notera M le marquage courant) :

1) M := M0 ;
2) construire l'ensemble val(M) des transitions t validées par M, c.a.d. telles que M-t-> ;
3) si val(M) = ∅ le RdP ne peut plus évoluer,
   sinon :
4) choisir arbitrairement une transition t0 dans val(M) et la mettre à feu, c.a.d. faire M := M' tel que M-t0->M' ;
5) reprendre à 2).

On obtient ainsi un "comportement" du réseau R à partir du marquage initial M0, parmi l'ensemble de ses comportements possibles à partir de ce même marquage.

Fusion_de_deux_places

Etant donné un RdP R = (P, T, alpha, beta) et deux places p1 et p2, on appellera fusion des places p1 et p2 l'opération qui transforme le RdP R en en RdP R' = (P', T, alpha', beta') tel que :

- P' = P - (p1, p2) + (p'), où p' est une place n'appartenant pas à P,

- alpha' et beta' coincident avec alpha et beta sur P'-(p') et pour toute transition t :
(p1, t) ou (p2, t) vérifient alpha <=> (p', t) vérifie alpha,
(t, p1) ou (t, p2) vérifient beta <=> (t, p') vérifie beta.

2.2 - Réseau_de_Petri_sauf,_Graphe_d'états_et_Graphe_de processus

Réseau_sauf_pour_un_marquage_initial

Un RdP est sauf pour un marquage initial M0 ssi pour toute place p, pour tout marquage M successeur de M0 :

$$M(p) = 0 \text{ ou } M(p) = 1.$$

Graphe_d'états

Un RdP est un graphe d'états (ou GE) ssi chaque transition a une et une seule place d'entrée, et une et une seule place de sortie, c.a.d.

pour toute transition t : $|{>}t| = |t{>}| = 1$.

Exemple :

Le réseau R1 (cf figure 2 p. III-4) n'est pas un graphe d'états puisque $|{>}t1| = 2$.

Par contre, le réseau G1 de la figure 4 est un graphe d'états.

Propriété_1 :

Un graphe d'état est sauf pour tout marquage initial M0 qui marque au plus une de ses places.

Démonstration : immédiate de la définition d'un graphe d'états et de celle de la mise à feu d'une transition.

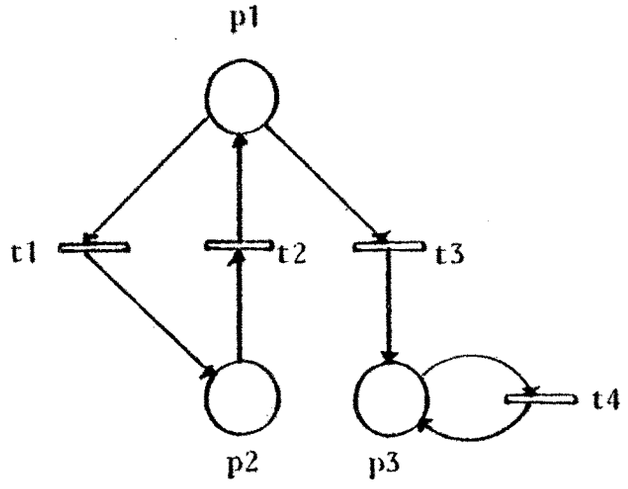


Figure 4

Graphe de processus

Un graphe de processus (ou GP) est un doublet G = (R, (Pi)i) tel que :

- R = (P, T, alpha, beta) est un RdP tel que chaque transition a autant de places d'entrée que de places de sortie, c.a.d.
pour toute transition t : |>t| = |t>| ≥ 1,

- (Pi)i est une partition de l'ensemble des places P (les Pi seront appelés processus de G) telle que :

* chaque transition possède au plus une place d'entrée appartenant à chaque processus, c.a.d.
pour toute transition t, pour tout processus Pj :
|>t ∩ Pj| ≤ 1

* si une transition possède une place d'entrée appartenant à un processus, elle possède également une place de sortie appartenant à ce processus, c.a.d.
pour toute transition t, pour tout processus Pj :
|>t ∩ Pj| = 1 => |t> ∩ Pj| = 1.

Exemple :

Le réseau G2 de la figure 5 est un graphe de processus, les processus étant :
P1={p11,p12,p13} P2={p21,p22,p23} P3={p31,p32,p33}.

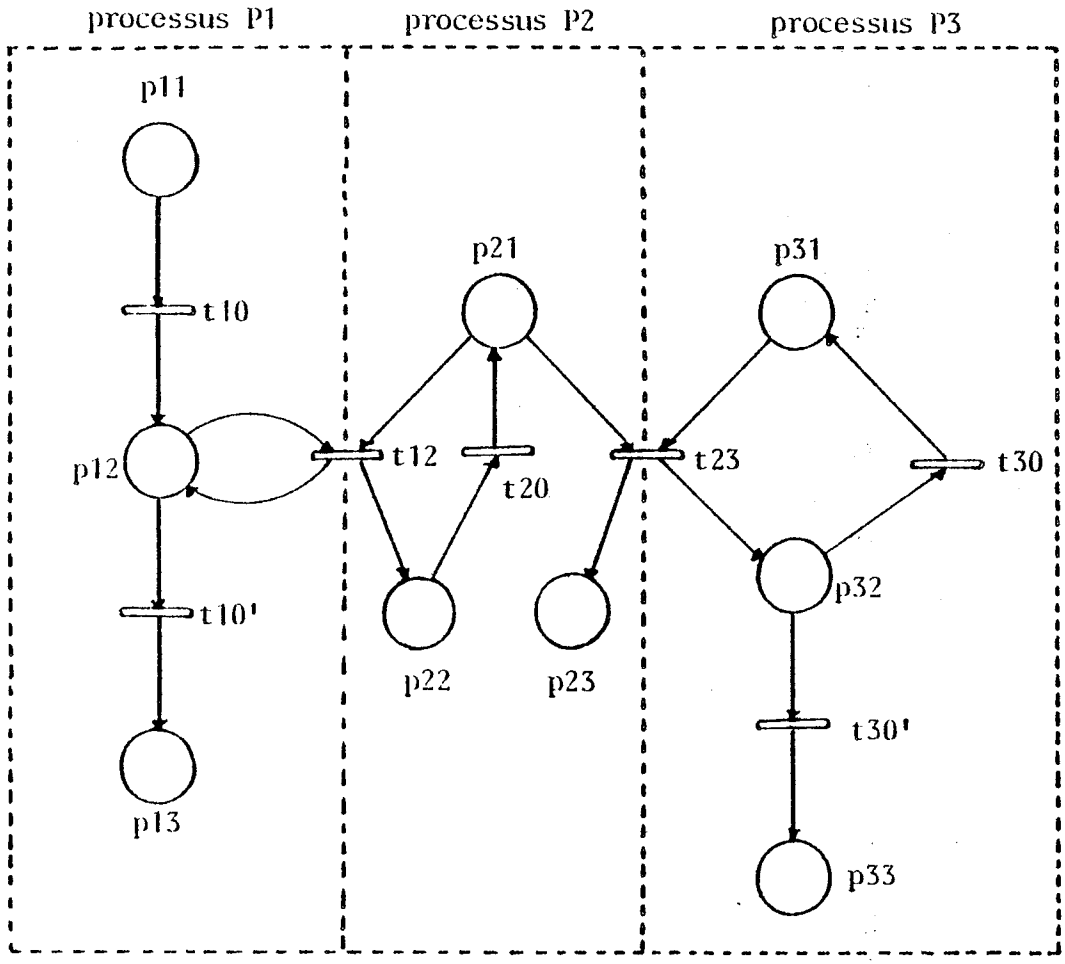


Figure 5

Propriété 2 :

Etant donné un graphe de processus G = (R, (Pi)i), avec R = (P, T, alpha, beta), les RdP Ri = (Pi, Ti, alphai, betai) tels que :

- Ti est l'ensemble des transitions en relation par alpha ou beta avec les places du processus Pi,
- alphai et betai sont les restrictions des relations alpha et beta aux ensembles Pi et Ti,

sont des graphes d'états.

Démonstration : la définition d'un GP implique que
$|>t \cap Pi| = |t> \cap Pi| = 0$ ou 1,
d'où la propriété.

Exemple :

Pour le GP G2 (cf figure 5 p. III-8), les réseaux obtenus à partir des ensembles de places et transitions suivants sont des graphes d'états :
P1 = {p11, p12, p13}, T1 = {t10, t10', t12}
P2 = {p21, p22, p23}, T2 = {t12, t20, t23}
P3 = {p31, p32, p33}, T3 = {t23, t30, t30'}.

Propriété 3 :

Un graphe de processus considéré comme un RdP est sauf pour tout marquage initial M0 marquant au plus une place de chaque processus.

Démonstration : immédiate à partir des propriétés 1 et 2.

Notations :

Etant donné un GP G = (R, (Pi)i) et une transition t, on notera (Pi)[t] l'ensemble des processus tels que

$$Pi \cap {}^{\bullet}t \neq \emptyset.$$

Il sera appelé ensemble des processus participant à la transition t. Remarquons qu'il peut être défini de façon équivalente à partir des places de sortie de t. On notera [t] l'ensemble des indices des processus participant à t.

2.3 - Réseau de Petri Interprété

Définition :

Un Réseau de Petri Interprété (ou RdPI) est un triplet RI = (R, X, I) tel que :
- R = (P, T, alpha, beta) est un RdP,
- X est une variable de domaine de définition D,
- I est une application (appelée interprétation) qui à chaque transition t de R associe un couple (c, a) où
  * c est une condition sur X, c.a.d. une application de D dans l'ensemble (vrai, faux),
  * a est une affectation sur X de la forme X := exp(X) où exp est une application de D dans lui-même.

Représentation graphique :

On représentera un RdPI RI = (R, X, I) au moyen de la représentation graphique usuelle du RdP R, accompagnée d'inscriptions à proximité des transitions de la forme

c -> a

où c est la condition et a l'affectation associées à la transition par l'interprétation.

Notations

On notera true la condition toujours vraie, c.a.d. telle que :

pour tout x de D : true(x) = vrai.

On notera nop l'affectation par la fonction identité, c.a.d. l'affectation X := X.

Etat_d'un_RdPI

Etant donné un RdPI RI = (R, X, I) on appellera état de RI un couple (M,x) où
- M est un marquage de R,
- x est un élément de D (c.a.d. une valeur de X).

On notera Q l'ensemble des états de RI :

Q = MM(R)xD.

Transition_validée_dans_un_RdPI

Etant donné un RdPI RI = (R, X, I), une transition t telle que I(t) = (c, a) est dite validée (dans RI) par un état q = (M, x) de Q ssi :
- la transition t est validée dans R par le marquage M, c.a.d.

pour toute place p de >t : M(p) > 0,

- la condition c associée à t est vérifiée par x, c.a.d.

c(x) = vrai.

Mise_à_feu_d'une_transition_validée_dans_un_RdPI

Etant donné un RdPI RI = (R, X, I) et une transition t validée dans RI par un état q = (M, x) telle que I(t) = (c, a), on appelle mise à feu de t une application de Q dans lui-même telle que à l'état q corresponde l'état q' = (M', x') tel que
- M' soit le marquage obtenu à partir de M par mise à feu de t dans R,
- x' est le résultat de l'affectation a sur x, c.a.d. si a est de la forme X := exp(X) alors x' = exp(x).

Notations :

Comme précédemment, on emploiera les notations suivantes, où q et q' sont des états et t, t1, ... tn des transitions :

q-t->

pour la proposition : "la transition t est validée par q" ;

q-t->q'

pour la proposition : "la mise à feu de t transforme q en q'" ;

$q\text{-}t_1t_2\ldots t_n\text{->}q'$

pour la proposition : "il existe une suite d'états q0, q1, ... qn telle que :
- q = q0, q' = qn
- pour tout k de 0 à n-1 : qk-t(k+1)->q(k+1)" ;

$q\text{-}*\text{->}q'$

pour la proposition : "il existe une séquence de transitions t1t2...tn (n>0) telle que q-t1t2...tn->q' ou q = q'".

Enfin on notera q*> l'ensemble des états successeurs de q, c.a.d. l'ensemble des états q' tel que q-*->q'.

Fonctionnement d'un RdPI

Etant donné un RdPI RI et un état q0 = (M0, x0) de Q, on peut faire évoluer l'état du réseau à partir de l'état initial q0 en appliquant l'algorithme suivant (on notera M le marquage courant):

1) M := M0, X := x0 ;
2) construire l'ensemble val(q) des transitions t validées par q = (X, M), c.a.d. telles que q-t-> ;
3) si val(q) = ∅ le RdPI ne peut plus évoluer,
   sinon :
4) choisir arbitrairement une transition t0 de val(q) et la mettre à feu, c.a.d. faire M := M', X := x' tel que q-t0->q' et q' = (M', x') ;
5) reprendre à 2).

Graphe de processus interprété

Un graphe de processus interprété est un triplet

$$GI = (G, (Xi)i, I)$$

tel que
- G == (R, (Pi)i) est un graphe de processus,
- (Xi)i est un ensemble de variables en bijection avec l'ensemble des processus (Pi)i ; Xi est dite variable interne du processus Pi,
- I est une interprétation associant à chaque transition t de R un couple (c, a) tel que
  * c est une condition sur (Xi)[t],
  * a est une affectation sur (Xi)[t],

c.a.d. seules les variables internes des processus participant à la transition t peuvent figurer dans l'interprétation de t.

Remarque : vecteur de variables

Si la variable X d'un RdPI est une variable vectorielle X = (x1, ... , xn) de domaine de définition D = D1x ... xDn, on dira que X est un "vecteur de variables", et on appellera "variables" les composantes xj de X. On notera également de façon abusive X l'ensemble des variables (x1, ... , xn).

De même, chaque variable interne des processus d'un GP interprété peuvent être des vecteurs de variables. Considérés comme des ensembles de variables, ils doivent être deux à deux disjoints.

Notations :

Dans le cas d'un vecteur de variables X = (x1, ... , xn) on notera

x1 := exp1(X), ... , xn := expn(X)

l'affectation vectorielle X := exp(X), en omettant les composantes de X non modifiées.

2.4 - Réseau de Petri Interprété Communicant

Définition

Un RdPI communicant est un quintuplet

RC = (R, X, Xin, Xout, I)

tel que :

- R est un RdP R = (P, T, alpha, beta),
- X, Xin, Xout sont des vecteurs de variables. Les ensembles Xin et Xout doivent être disjoints de X (mais non nécessairement entre eux) ;
- I est une interprétation associant à chaque transition t un couple (c, a) tel que :
  - c est une condition sur X,
  - a est ume affectation de la forme

    (X, Xout) := exp(X, Xin).

On appellera les variables de X (resp. Xin, Xout) les variables internes (resp. d'entrée, de sortie) de RC.

Variables consommées et variables produites

La notation vectorielle

(X, Xout) := exp (X, Xin)

cache le fait que les variables de Xout ne sont pas nécessairement toutes modifiées (c.a.d. que la fonction exp projetée sur tout ou partie des composantes de Xout peut coincider avec l'identité). On appellera variables produites par une affectation les variables de Xout effectivement modifiées par cette affectation.

De même, on appellera variables consommées par une affectation les variables de Xin dont dépend effectivement le membre droit de l'affectation.

Notations :

On notera les noms des variables produites précédés d'un point d'exclamation (!) et les noms des variables consommées précédés d'un point d'interrogation (?).

Echanges simples

Un RdPI communicant sera dit à échanges simples ssi toutes ses affectations sont de l'une des trois formes suivantes :
- aucune variable de Xout n'est produite et aucune variable de Xin n'est consommée. L'affectation sera dite interne ; elle est de la forme

$$X := exp(X) ;$$

- une seule variable !xo de Xout est produite et aucune variable de Xin n'est consommée. L'affectation sera appelée émission de !xo ; elle est de la forme

$$!xo := exp(X),\ X := exp'(X) ;$$

- aucune variable de Xout n'est produite et une seule variable ?xi de Xin est consommée, et utilisée telle quelle (c.a.d. sans figurer dans une expression) pour affecter certaines composantes de X. L'affectation sera appelée réception de ?xi ; elle est de la forme

$$X1 := ?xi,\ X2 := exp(X),$$

où X1 et X2 sont deux sous-ensembles de X disjoints.

Graphe de processus interprété communicant

Un GP interprété communicant est un quintuplet

$$GC = (G, (Xi)i, Xin, Xout, I)$$

tel que :
- G = (R, (Pi)i) est un graphe de processus,
- (Xi)i est un ensemble de vecteurs de variables en bijection avec l'ensemble des processus de G, deux à deux disjoints,
- Xin et Xout sont deux vecteurs de variables, disjoints avec chaque Xi,
- I est une interprétation associant à chaque transition t un couple (c, a) tel que :
  * c est une condition sur (Xi)[t],
  * a est une affectation de la forme

$$((Xi)[t], Xout) := exp((Xi)[t], Xin).$$

2.5 - Réseau de Petri Procédural

Définition :

Un RdP interprété (éventuellement communicant) procédural est un quadruplet (éventuellement un sextuplet)

RP = (R, X, PR, I) ou RP = (R, X, Xin, Xout, PR, I)

tel que :

- R est un RdP,
- X, (éventuellement Xin, Xout) sont des vecteurs de variables (Xin et Xout étant disjoints de X),
- PR est un ensemble fini appelé vocabulaire des "noms de procédures",
- I est une interprétation associant à chaque transition un couple (c, a) ou bien un couple (c, pr) tels que :
  * c est une condition sur X,
  * a est une affectation de la forme
    X := exp(X) ou (X, Xout) := exp(X, Xin),
  * pr est un nom de procédure.

Représentation_graphique_:

Les transitions dont l'interprétation est un couple (c, pr) où pr est un nom de procédure seront représentées accompagnées d'une inscription de la forme
c -> (pr).

2.6 - Graphes_de_processus_étiquetés_par_des_actions

Ensemble_d'étiquettes_associé_à_un_ensemble_d'actions

Etant donné un ensemble fini V, appelé vocabulaire d'actions, on définit l'ensemble E(V) appelé vocabulaire des étiquettes associées à V comme l'ensemble E(V) = Ve u Vc avec :
Ve = {"v" / v élément de V}
Vc = {"_v", "_v_", "v_" / v élément de V}.

Chemin_dans_un_graphe_de_processus

Etant donné un graphe de processus G = (R, (Pi)i), on définit un chemin de longueur k>1 dans un processus Pj comme une séquence de transitions toutes différentes t1t2...tk de Tj (ensemble des transitions en relation avec des places de Pj) tel qu'il existe une suite de places p1j, ... p(k-1)j de Pj telle que :
pour tout i de 1 à k-1 :
(ti, pij) vérifie beta
et (pij, t(i+1)) vérifie alpha.

Pour un chemin c = t0t1...ti...tk, on dira que :
- c est un chemin à partir de t0, aboutissant à tk,
- c passe par ti, ou ti est sur le chemin c,
- t1, ... t(k-1) sont entre t0 et tk sur le chemin c.

Exemple :

Dans le GP de la figure 5 p. III-8 :
- t10t12t10* est un chemin de P1,
- t12t20t23 est un chemin de P2,
- t23t30 est un chemin de P3.

Graphe de processus étiqueté

Un GP (éventuellement interprété ou communicant) étiqueté est un triplet (G, V, lambda) tel que :
- G est un GP (éventuellement interprété ou communicant),
- V est un vocabulaire d'actions,
- lambda est une application de l'ensemble des transitions de G vers l'ensemble des parties du vocabulaire E(V), appelée fonction d'étiquetage vérifiant les propriétés suivantes :

1) pour toute action v :

1a) ou bien, il n'existe aucune transition étiquetée par "v", "_v", "_v_" ou "v_" ;

1b) ou bien, il existe une transition unique t étiquetée par "v", et aucune transition n'est étiquetée par "_v", "_v_" ou "v_" ; l'action v est dite action élémentaire ;

1c) ou bien, il existe une seule transition t0 étiquetée par "_v" et tous les chemins à partir de t0 sont de la forme t0t1...tf... (ou t0tf...) à l'intérieur d'un même processus (quelconque) et sont tels que :
- lambda(t0) contient "_v" et pas "_v_" ni "v_",
- lambda(tf) contient "v_" et pas "_v" ni "_v_",
- toute transition t' entre t0 et tf (s'il y en a) est telle que lambda(t') contienne "_v_" et pas "_v" ni "v_",
- aucune transition n'appartenant pas à ces chemins n'est étiquetée par "_v", "_v_" ou "v_",
- aucune transition n'est étiquetée par "v" ;

l'action v est dite action composée et on appellera un tel chemin "chemin d'exécution de v".

2) pour toute paire d'actions composées v1 et v2, si deux chemins d'exécution c1 et c2 respectivement de v1 et v2 sont non disjoints, alors l'un est inclus dans l'autre.

Interprétation intuitive

Ces règles peuvent être vues intuitivement comme exprimant :
- pour la règle 1c : la continuité d'une action composée, du point de vue du processus qui l'exécute ; cette action commence en un point unique, mais peut évoluer et se terminer selon plusieurs chemins ;
- pour la règle 2 : le non chevauchement d'actions composées, ce qui n'interdit pas leur imbrication.

3 - PRINCIPE D'OBTENTION DU MODELE

3.1 - Principe général d'obtention du modèle

Une application décrite en langage CESAR sera représentée par un GP interprété et étiqueté (muni d'un ou plusieurs états initiaux) généré automatiquement à partir du programme de description. La partie réseau représente la partie contrôle de l'application, c.a.d. le comportement imposé par la décomposition en tâches, les rendez-vous entre elles, et les structures syntaxiques des instructions composées, alors que l'interprétation représente l'influence des variables (internes ou échangées) sur ce comportement. Les étiquettes des transitions permettent de faire la liaison avec les étiquettes du programme de description. Elles seront utilisées pour l'analyse des propriétés de comportement du système décrit. L'état initial représente l'état initial du système décrit (contrôle et valeurs initiales des variables).

La traduction des tâches et des procédures du programme de description présente cinq aspects suivant l'objet traité :

- pour une procédure (spécifiée) : le corps de la procédure est traduit en un GE interprété (éventuellement communicant) étiqueté (procédural si la procédure appelle elle-même d'autres procédures) au moyen d'une grammaire de briques (cf 3.4 p. III-22, 4 p. III-34 et 5 p. III-55) qui associe à chaque instruction du langage un schéma de réseau représentant sa sémantique. Si on obtient un GE procédural, une opération de substitution (cf 3.5 p. III-28) des procédures appelées permet de le transformer en un GE non procédural ;

- pour une procédure non spécifiée : un GE interprété particulier est généré pour représenter son corps (il n'est pas communicant puisqu'une procédure non spécifiée ne peut pas avoir de paramètres échangés) ;

- pour une tâche élémentaire : le corps de la tâche élémentaire est traduit en un GE interprété (en général communicant) étiqueté, au moyen de la grammaire de brique ; si on obtient un réseau procédural, celui-ci est transformé au moyen de l'opération de substitution des procédures appelées par la tâche ; ce réseau est muni d'un ou plusieurs états initiaux possibles ;

- pour une tâche non spécifiée : un GE interprété (en général communicant) particulier (muni d'un état initial) est généré selon la partie formelle de la tâche non spécifiée ;

- pour une tâche composée : les différents réseaux (munis de leurs états initiaux) représentant les tâches composantes sont associés entre eux au moyen d'une opération de composition (cf 3.6 p. III-30 et 6.2 p. III-67) représentant la sémantique du rendez-vous, afin d'obtenir un GP communicant et étiqueté unique représentant la tâche composée, puis une opération d'encapsulation (cf 3.7 p. III-33 et 6.3 p. III-82) réduit ses possibilités d'échanges pour les ramener à celles autorisées par sa partie formelle. Si la tâche composée est un système fermé, cette opération transforme le GP communicant en un GP interprété (non communicant) représentant le système. Le réseau ainsi obtenu est muni d'un ou plusieurs états initiaux possibles.

3.2 - Signification des différents éléments du modèle

3.2.1 - Le réseau de Petri

Le réseau de Petri (GE ou GP) représente le contrôle imposé par la structuration en tâches élémentaires et composées, leurs rendez-vous, et les structures syntaxiques des instructions composées.

Pour une procédure ou une tâche élémentaire, le réseau généré au moyen de la grammaire de briques est un graphe d'états. L'opération de substitution des procédures qui remplace des transitions par des GE conserve cette propriété.

L'opération de composition des tâches élémentaires associe les GE les représentant par fusion de transitions ; on obtient ainsi un graphe de processus. Appliquée à des tâches composées, l'opération de composition conserve cette propriété.

L'opération d'encapsulation ne modifie le GP sur lequel elle s'applique que par suppression de certaines transitions, ce qui conserve aussi cette propriété.

Pour un GE représentant une procédure, trois places sont distinguées :
- une place pe qui représente le point d'entrée de la procédure ;
- une place ps qui représente le point de sortie normale de la procédure ;
- une place pt qui représente le point de sortie de la procédure par terminaison de la tâche appelante à l'intérieur de la procédure.

Pour un GE représentant une tâche élémentaire ou non-spécifiée, deux places sont distinguées :
-une place pi (initialement marquée) qui représente l'état initial de la tâche ;
-une place pf qui représente l'état final de la tâche.

Lors de la substitution des procédures à l'intérieur d'une autre procédure, toutes les places pt des procédures substituées sont fusionnées entre elles et avec la place pt de la procédure appelante. Lors de la substitution des procédures à l'intérieur d'une tâche élémentaire, toutes les places pt des procédures substituées sont fusionnées entre elles et avec la place pf de la tâche.

3.2.2 - Les variables

Les variables de l'interprétation représentent les variables significatives (internes ou échangées) du programme de description. Par variable significative, on entend toute variable dont la valeur est toujours connue de façon à pouvoir évaluer son influence sur le comportement de l'application décrite. Les variables non-significatives sont éliminées (cf 3.3 p. III-21) de façon à n'exclure aucun des comportements qu'elles pourraient provoquer. Ceci se traduit par une augmentation du non-déterminisme, donc une augmentation de l'ensemble des comportements possibles, pouvant introduire des comportements qui n'existaient pas avant l'élimination des variables non-significatives.

Toutes les variables des vecteurs X, Xin et Xout devant avoir des noms différents, les noms des variables du programme CESAR sont transformés de façon à obtenir de "noms absolus", obtenus en préfixant l'identificateur de la variable par le nom (lui-même absolu) de l'environnement de déclaration (tâche, procédure ou bloc), un nom fictif étant donné à chaque bloc comportant des déclarations de variables pour les différencier. Comme il n'y a pas d'allocation dynamique de variables dans notre modèle, à chaque variable déclarée dans un bloc ou procédure est associée une variable unique, qui est initialisée à chaque entrée du bloc ou de la procédure. Toutefois, si la procédure est appelée par plusieurs tâches, une variable est allouée par tâche appelante de façon à ne pas enfreindre le principe d'étanchéité (pas de variables partagées).

Lors de la génération du réseau associé à une procédure ou une tâche élémentaire, les paramètres sont considérés comme des variables, qui sont ensuite substituées par les paramètres effectifs lors de la substitution de cette procédure, ou de la composition de cette tâche avec une ou plusieurs autres.

3.2.3 - L'interprétation

L'interprétation associe aux transitions des couples (c, a) ou (c, pr), où c est une condition, a une affectation et pr un nom de procédure.

Les conditions représentent les conditions sur les variables significatives permettant d'effectuer le choix des instructions à exécuter au sein des instructions composées. Les conditions portant sur des variables non-significatives sont simplifiées (cf 3.3 p. III-21).

Les affectations représentent les modifications des variables significatives, ainsi que les échanges. Les modifications des variables non-significatives sont supprimées.

Les noms de procédures représentent les appels de procédures ; ils sont utilisés pour l'opération de substitution des procédures, et disparaissent au cours de cette opération.

3.2.4 - L'étiquetage des transitions

A chaque étiquette e introduite dans le programme de description est associé un nom absolu ea obtenu par préfixation avec le nom (absolu) de l'environnement où elle figure. L'ensemble V des actions est l'ensemble de ces noms absolus. L'ensemble des étiquettes est le vocabulaire E(V) (cf 2.6 p. III-15) des étiquettes "ea", "_ea", "_ea_" et "ea_", pour chaque ea de V, ces étiquettes étant associées aux transitions de telle façon que :

- l'étiquette "ea" est associée à une transition réalisant l'action ea (en une seule transition) ;
- l'étiquette "_ea" est associée à une transition marquant le début de l'action ea ;
- l'étiquette "ea_" est associée à une transition marquant la fin de l'action ea ;
- l'étiquette "_ea_" est associée à toutes les transitions entre les transitions étiquetées par "_ea" et "ea_" sur les chemins d'exécution de ea.

L'existence des instructions EXIT, RETURN et TERMINATE permet de terminer "brutalement" l'exécution d'une action étiquetée, sans passer par sa sortie "syntaxique". Les transitions représentant ces actions seront donc étiquetées (outre les étiquettes qu'elles peuvent porter du fait de leur étiquetage propre) par un ensemble d'étiquettes de la forme "x_", où x parcourt l'ensemble des actions commencées mais non terminées depuis le dernier mot-clé LOOP (pour EXIT), ou

depuis l'entrée de la procédure (pour RETURN ou TERMINATE à l'intérieur d'une procédure), ou depuis le début de la tâche (pour TERMINATE dans le corps d'une tâche élémentaire). De plus, lors de la substitution d'une procédure comportant une instruction TERMINATE, la transition représentant cette instruction sera en outre étiquetée par toutes les étiquettes "y_" où y parcourt l'ensemble des actions commencées mais non terminées depuis l'entrée de la procédure appelante, ou depuis le début de la tâche, si l'appel figure dans sa séquence principale.

3.3 - Elimination des variables non significatives

Les variables de type non spécifié, ou d'un type spécifié mais pouvant avoir une valeur non spécifiée, sont considérées comme non significatives. Lors de la traduction du programme de description, on procède à une élimination des variables non significatives, en éliminant :

- les variables de type non-spécifié, ou les tableaux dont le type des éléments est non spécifié, ou les champs de type non spécifié des structures (en conservant les autres),
- les variables de types entiers non bornés, (ces types sont en fait non-spécifiés du point de vue de l'analyse, puisque leurs bornes ne sont pas connues),
- les variables affectées par une expression à résultat non spécifié, ou passées en paramètres à une procédure non spécifiée,
- les variables des procédures ou des blocs déclarées sans valeur initiale, parce qu'il est nécessaire (si l'on désire respecter les règles usuelles de portée des variables) de réinitialiser les variables déclarées dans les blocs ou procédures à chaque entrée dans le bloc ou la procédure,
- les variables affectées par une expression dépendant d'une variable précédemment éliminée.

Lorsque les variables non-significatives sont éliminées les affectations de l'interprétation où elles figurent en partie gauche sont supprimées (transformées en $\Omega\Omega\Omega$).

Dans le même temps, les conditions où figurent des variables à éliminer sont simplifiées dans la partie condition de l'interprétation. Chaque condition c dans ce cas est simplifiée en remplaçant par TRUE chacun des termes de la condition à résultat non spécifié (ou FALSE s'il est précédé de l'opérateur NOT), de façon à obtenir une condition c' telle que

$$c \Rightarrow c'$$

ce qui garantit qu'aucun des comportements possibles avant simplification n'est exclu par celle-ci.

Exemple

En notant tns les termes à résultats non spécifiés, les simplifications suivantes sont effectuées :

c1 AND tns --> c1
c1 OR tns --> true
(c1 AND NOT tns) OR (c2 AND (c3 OR tns)) --> c1 OR c2.

3.4 - Principe de la génération au moyen d'une grammaire de briques

La grammaire d'un langage est généralement définie comme un quadruplet GL = (VT, VN, S, R) où :
- VT et VN sont des ensembles finis non vides et disjoints d'objets appelés respectivement vocabulaire terminal et vocabulaire non terminal,
- S est un élément de VN appelé axiome,
- R est un ensemble de règles, une règle étant un couple (N, sigma) où N est un élément de VN et sigma un élément du monoide libre engendré par (VT u VN), c.a.d. une chaine finie (ou vide) d'éléments de VT ou VN.

Dans la pratique, on considère les règles de la grammaire comme des règles de réécriture des symboles non-terminaux en chaînes d'éléments terminaux et non-terminaux. On appelle production terminale une chaîne d'éléments terminaux obtenue par application d'un nombre fini de règles à partir d'un symbole non-terminal (en général l'axiome, sauf indication contraire).

Nous allons de façon analogue définir une "grammaire de briques" GB dont les productions terminales seront des réseaux interprétés (communicants ou procéduraux si nécessaires) étiquetés. Une grammaire de briques sera définie en relation avec la grammaire GL d'un langage L (la grammaire du langage CESAR, telle qu'elle figure en annexe, dans le cas pratique qui nous intéresse).

Vocabulaire terminal de la grammaire de briques

Toute production terminale de la grammaire sera un réseau de Petri, c.a.d. un graphe orienté sur les éléments terminaux, qui seront les places et les transitions, muni d'une interprétation et d'une fonction d'étiquetage.

Vocabulaire non-terminal et semi-terminal de la grammaire de briques

Pour la construction du réseau on introduit les symboles non-terminaux suivants (et on étend les fonctions d'interprétation et d'étiquetage):

- les "briques ouvertes" dont l'interprétation est une production terminale de GL, appelé contenu de la brique ; on étend la fonction d'étiquetage aux briques ouvertes ;
- les "briques fermées", dont l'interprétation (le contenu) est également une production terminale de GL ; les briques fermées ne sont pas étiquetées.

On introduit de plus les symboles suivants (dits "semi-terminaux") :
- les "demi-transitions supérieures" dont l'interprétation est une condition seule,
- les "demi-transitions inférieures" dont l'interprétation est une action seule, ou un nom de procédure.

La fonction d'étiquetage est également étendue aux demi-transitions inférieures, mais non aux demi-transitions supérieures.

Notations

Dans un graphe comportant des 1/2-transitions ou des briques ouvertes, on utilisera les notations >t et t> où t est une 1/2-transition ou une brique ouverte pour désigner les ensembles des objets de ce graphe tels qu'il existe un arc dans ce graphe d'eux à t (resp. de t à eux).

Représentation graphique

On représentera (cf figure 1) une demi-transition supérieure par un rectangle ouvert en bas, accompagné d'une inscription de la forme

c ->

où c est la condition associée à la 1/2-transition par l'interprétation.

De même, on représentera une demi-transition inférieure par un rectangle ouvert en haut, accompagné d'une inscription de la forme

-> a ou -> (pr)

où a est l'action ou pr le nom de procédure associé à la 1/2-transition par l'interprétation.

Enfin, on représentera une brique par un rectangle plus grand en pointillé contenant le texte xxx (où xxx est le contenu de la brique associé par l'interprétation), avec si la brique est ouverte, en haut la représentation d'une 1/2-transition inférieure et en bas la représentation d'une 1/2-transition supérieure (sans interprétation).

Règles de la grammaire de briques

Une règle est un doublet (N, gamma) où :
- N est une brique (ouverte ou fermée),

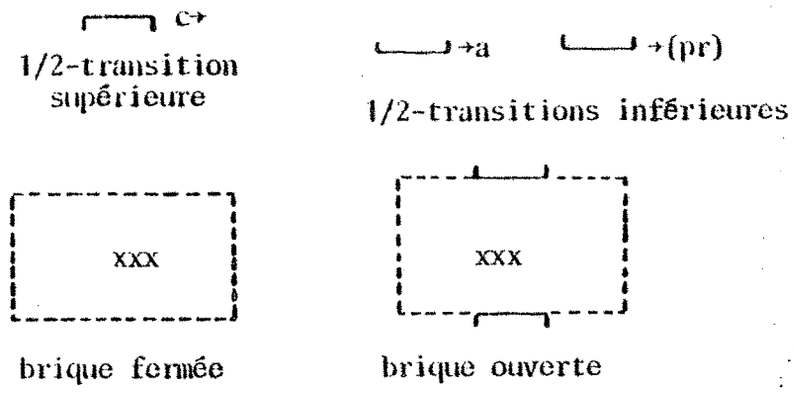


Figure 1

- gamma est un graphe orienté pouvant comporter des places, des transitions, des briques ouvertes et des demi-transitions, tel qu'il puisse exister un arc :
  * entre une place et une transition ou une 1/2-transition supérieure,
  * entre une transition ou une 1/2-transition inférieure et une place,
  * entre une 1/2-transition supérieure ou une brique ouverte et une 1/2-transition inférieure ou une brique ouverte.

Une règle précise également les valeurs des fonctions d'interprétation et d'étiquetage pour les divers éléments de gamma en fonction de leurs valeurs pour N et de son contenu.

Une règle (N, gamma) doit de plus être bien formée, c.a.d. vérifier les propriétés suivantes :
- si N est une brique fermée, toute 1/2-transition inférieure ou brique ouverte t doit être telle que :

$$|{>}t| = 1,$$

et toute 1/2-transition supérieure ou brique ouverte t' doit être telle que :

$$|t'{>}| = 1,$$

- si N est une brique ouverte, gamma doit posséder une et une seule 1/2-transition inférieure ou brique ouverte t0 telle que >t0 soit vide, les autres étant telles que :

$$t \neq t0 \Rightarrow |{>}t| = 1,$$

et une et une seule 1/2-transition supérieure ou brique t0' telle que t0'> soit vide, les autres étant telles que :

$$t' \neq t0' \Rightarrow |t'{>}| = 1.$$

Représentation graphique

Pour la représentation graphique d'un graphe membre droit d'une règle, on notera par une flèche les arcs entre places et transitions ou demi-transitions ou inversement. Par contre on notera par la simple juxtaposition verticale (cf

figure 2) l'arc entre une 1/2-transition supérieure et une brique, ou entre une brique et une 1/2-transition inférieure, entre deux briques, ou entre deux demi-transitions (on parlera de 1/2-transitions ou de briques en correspondance).

Dans la suite, les règles seront toujours données sous forme graphique.

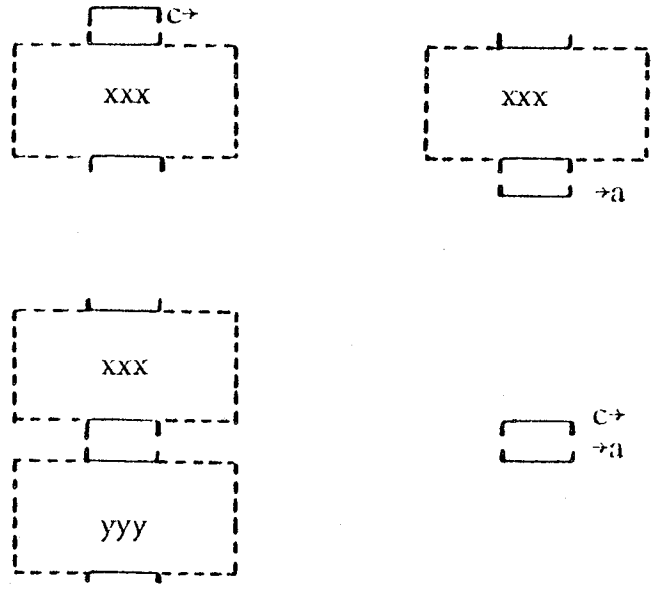


Figure 2

Exemple

Si c est une condition et s une séquence d'instructions dans le langage de description CESAR, la règle représentée par la figure 3 apparaît dans la grammaire de briques associée (E est un ensemble d'étiquettes).

Règle d'agglomération

Si on considère les règles de grammaire comme des règles de réécriture d'un graphe, les contraintes présidant à la construction d'une règle bien formée permettent d'assurer que, par application successive des règles à partir d'une brique fermée, on peut finalement mettre en correspondance des couples

(1/2-transition supérieure, 1/2-transition inférieure),

là où dans la règle initiale il y avait une correspondance

(brique ouverte, 1/2-transition inférieure),
(1/2-transition supérieure, brique ouverte),

ou (brique ouverte, brique ouverte).

Toutefois ceci ne nous permet d'obtenir que des productions

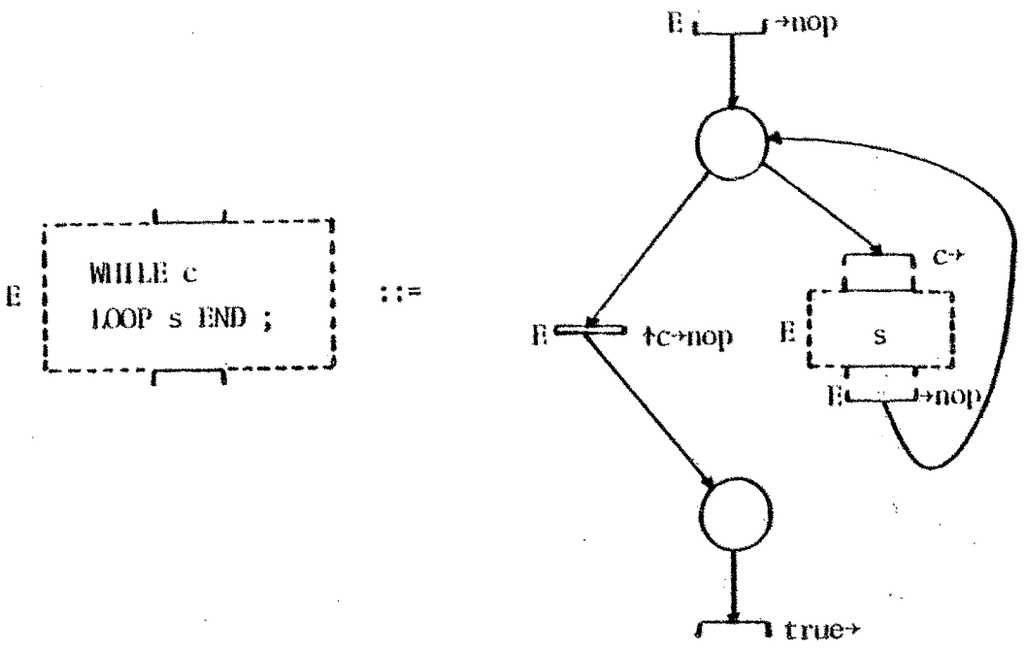


Figure 3

semi-terminales, puisque les 1/2-transitions sont des symboles semi-terminaux.

La grammaire de brique sera donc complétée de la règle de réécriture suivante (appelée règle d'agglomération) permettant d'éliminer les 1/2-transitions en correspondance en les fusionnant pour obtenir des transitions. Cette règle est exprimée de façon graphique par la figure 4, où c est une condition, a une affectation, E un ensemble d'étiquettes.

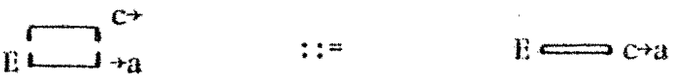


Figure 4 - Règle d'agglomération

Cette règle ne fait pas formellement partie de la grammaire de briques. C'est une règle de réécriture suplémentaire à appliquer lorsqu'on désire utiliser la grammaire de briques comme un ensemble de règles de réécriture de graphes pour obtenir une production terminale ne possédant que des places et des transitions.

Axiome_de_la_grammaire_de_briques

L'axiome d'une grammaire de briques est une brique fermée.

Dans le cas pratique qui nous intéresse, l'axiome sera choisi en fonction de l'objectif de génération. Pour générer le réseau représentant une procédure on prendra pour axiome la brique fermée de contenu le texte CESAR définissant cette procédure. De même, pour une tâche élémentaire ou non-spécifiée, l'axiome sera la brique fermée de contenu le texte de cette tâche.

Pour une procédure ou une tâche élémentaire déclarée dans un package, c'est la déclaration fournie à l'intérieur du corps du package qui sera utilisée (sauf si la procédure ou la tâche est non spécifiée).

Méta-grammaire

Les règles d'une grammaire de briques GB sont définies en termes de productions terminales d'une grammaire GL d'un langage L. Pour ne pas avoir à la faire pour chaque production terminale possible de GL, on utilisera une méta-grammaire.

On appellera méta-grammaire de briques associée à GL une grammaire de briques dont la fonction d'interprétation associe aux briques des chaînes de symboles terminaux et non-terminaux (productions non-terminales) de la grammaire GL, les symboles non-terminaux identiques pouvant être indiciés de façon à les différencier. Les règles d'une méta-grammaire seront appelées méta-règles. Une règle de la grammaire de briques peut être obtenue à partir d'une méta-règle par substitution uniforme (dans les deux membres) des symboles non-terminaux de même indice par des productions terminales à partir d'eux dans la grammaire GL.

Une méta-grammaire a pour méta-axiome une brique fermée de contenu un symbole non-terminal de GL (dans le cas de CESAR le symbole non-terminal <déclaration de procédure>, <déclaration de tâche élémentaire> ou <déclaration de tâche non spécifiée>, suivant les cas).

La grammaire de briques associée à une procédure ou une tâche élémentaire ou non spécifiée sera obtenue en prenant pour axiome la brique fermée de contenu le texte de cette procédure ou tâche élémentaire ou non-spécifiée, et en en construisant les règles (par substitution uniforme dans les méta-règles) au fur et à mesure de la dérivation de façon à éliminer les symboles non terminaux.

Le réseau représentant la procédure ou la tâche élémentaire ou non-spécifiée sera la production terminale de cette grammaire de briques.

Ce réseau est unique, car la grammaire du langage CESAR et la méta-grammaire de briques proposée sont telles qu'au vu d'un symbole non-terminal et de son contenu, une seule méta-règle puisse être utilisée de façon à générer une règle (elle-même unique) qui permette d'éliminer ce symbole non-terminal.

La forme des méta-règles fournies garantit que le RdP généré est un GE (à l'exception de certaines "branches mortes" qui peuvent être éliminées).

La méta-grammaire est définie aux paragraphes "Traduction des instructions" (cf 4 p. III-34) et "Traduction des blocs, procédures et tâches élémentaires ou non spécifiées" (cf 5 p. III-55).

3.5 - Substitution d'une procédure

Etant donné un GE procédural G1 représentant une procédure ou une tâche élémentaire, tel que la procédure pr soit appelé par celui-ci, et le GE interprété (ou communicant) G2 représentant cette même procédure, l'opération de substitution de pr dans G1 consiste à transformer G1 en G1' en remplaçant chaque transition t0 de G1 comportant l'appel de pr par une transition te suivi du GE G2, selon la règle exprimée graphiquement par la figure 5 (la notation p2/ps signifie que ces deux places sont fusionnées).

Simultanément, la place pt de G2 est fusionnée avec la place pt de G1 (si G1 représente une procédure) ou avec la place pf de G1 (si G1 représente une tâche élémentaire).

La substitution de toutes les procédures appelées par G1 le transforme en un GE interprété (ou communicant) sans appel de procédure.

En ce qui concerne l'interprétation des transitions, les paramètres formels figurant dans l'interprétation de G2 sont substitués par les paramètres effectifs fournis lors de l'appel, ce qui peut avoir pour effet secondaire de provoquer l'élimination supplémentaire de certaines variables (selon les règles définies en 3.3 p. III-21). Cette sustitution est également effectuée dans les vecteurs de variables X2, X2in et X2out de G2. Le vecteur de variables X1' de G1' est l'union des vecteurs de variables X1 de G1 et X2 de G2 après substitution. (Ces deux vecteurs sont généralement non disjoints, puisque X2 est formé non seulement des variables déclarées dans la procédure substituée, mais également des variables de l'environnement où elle est déclarée). Les vecteurs X1in' et X1out' de G1' sont identiques aux vecteurs

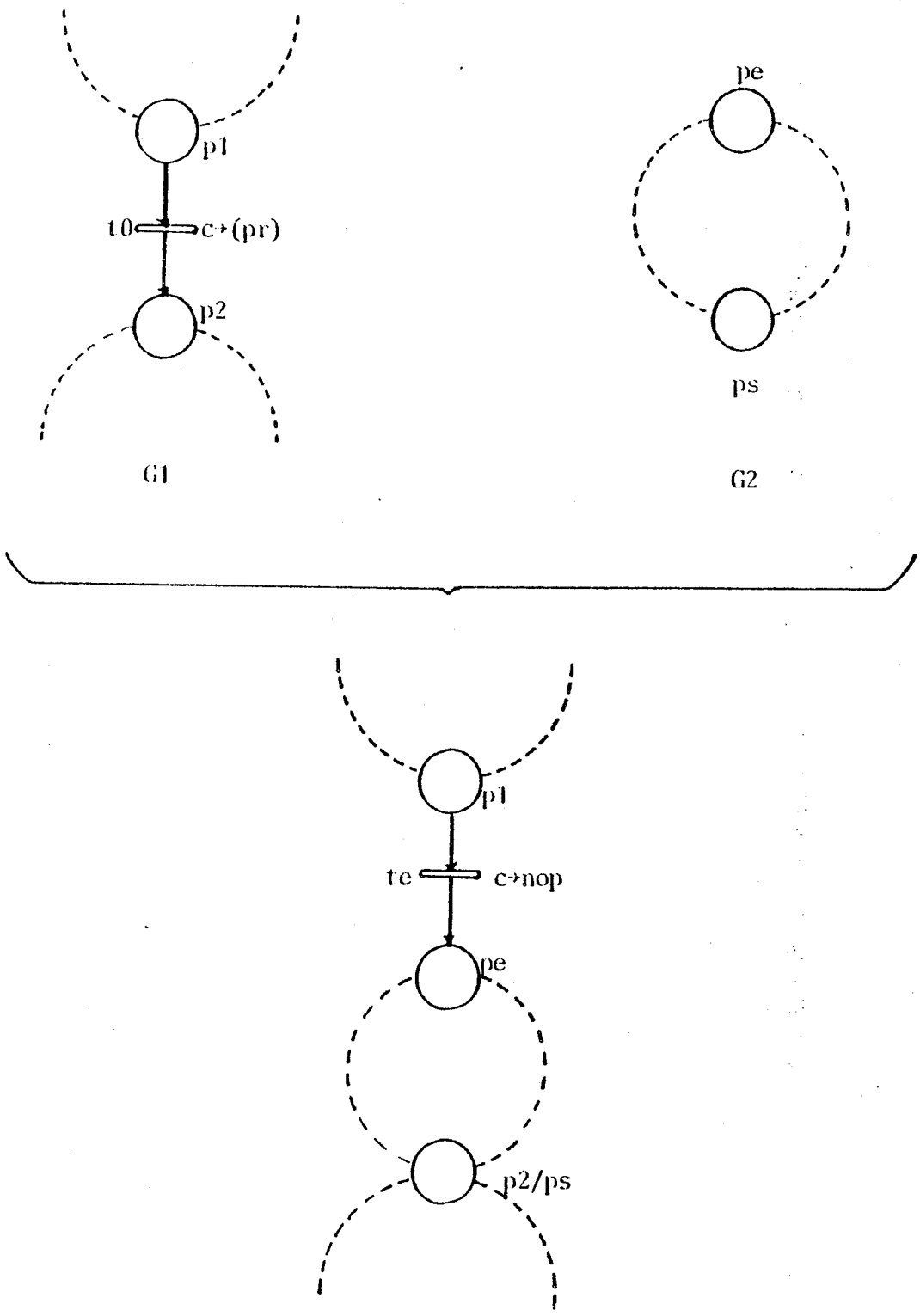


Figure 5 - Substitution de pr représenté par G2 dans G1

X1in et X1out de G1, car les vecteurs X2in et X2out après substitution sont inclus dans les vecteurs X1in et X1out (les règles de la syntaxe CESAR sur la visibilité des paramètres échangés l'imposent).

La fonction d'étiquetage des transitions est modifiée comme suit :

- la transition te est étiquetée exactement comme la transition t0 dans G1 avant substitution ;
- les transitions aboutissant à la place ps sont étiquetées (outre leurs étiquettes dans G2) par les étiquettes "x_" pour toute étiquette "_x" de t0 et par les étiquettes "_y_" pour toute étiquette "_y_" de t0 ;
- les transitions aboutissant à la place pt dans G2 sont étiquetées (en plus des étiquettes qui leurs sont propres dans G2) par les étiquettes "x_" pour toute étiquette "_x" ou "_x_" de t0 ;
- les autres transitions provenant de G2 sont étiquetées (outre leurs étiquettes dans G2) par les étiquettes "_x_" pour toute étiquette "_x_" ou "_x" de t0.

De façon pratique, les opérations de substitution sont en fait effectuées lors de la compilation à chaque appel de procédure. Une procédure devant être déclarée (avec son corps si elle est spécifiée) avant son appel, le GE interprété (ou communicant) la représentant existe nécessairement à ce moment là.

3.6 - Principe de l'opération de composition

Etant donné un ensemble de tâches formant une tâche composée, représentées par leurs réseaux (Gi)i, l'opération de composition permet, après substitution des paramètres échangés, d'obtenir le GP communicant G0 représentant cette tâche composée.

L'opération de composition utilise une opération de fusion des transitions représentant l'émission et la réception d'une même variable échangée, de façon à exprimer la sémantique du rendez-vous.

Supposons qu'une tâche composée T0 soit formée de deux tâches T1 et T2, telles que T1 émet la variable échangée v que T2 reçoit. Soit G1 (resp. G2) le réseau représentant T1 (resp. T2). Si T1 (resp. T2) ne possède qu'une instruction d'émission (resp. réception) de v dans son corps, le réseau G1 (resp. G2) ne possède qu'une transition t1 (resp. t2) représentant cette instruction. L'interprétation de t1 est de la forme (après substitution du paramètre échangé par la variable échangée v) :

c1 -> v1, !v := exp1, a1

et celle de t2 de la forme

c2 -> v2 := ?v, a2

où c1 (resp. c2) est une condition sur les variables de G1 (resp. G2), exp1 une expression sur les variables de G1, v1 (resp. v2) un ensemble de variables de G1 (resp. G2), et a1 (resp. a2) une affectation interne des variables de G1 (resp. G2), ne modifiant pas les variables de v1 (resp. v2).

La fusion de t1 et t2 est l'opération qui construit à partir de G1 et G2 le GP G0 (représentant T0 si aucune autre variable n'est échangée entre T1 et T2) obtenu en faisant l'union des réseaux G1 et G2 et en remplaçant les transitions t1 et t2 par une transition unique t0 telle que

>t0 = >t1 u >t2,

et t0> = t1> u t2>,

d'interprétation

c1.c2 -> v1, v2 := exp1, a1, a2

(le symbole "." représentant le produit booléen). Notons que les affectaions a1 et a2 peuvent être exécutées simultanément puisqu'elles portent sur des vecteurs de variables disjoints. Cette opération est illustrée par la figure 6 (où les transitions t1 et t2 n'ont qu'une place d'entrée et de sortie pour la simplicité du dessin).

Si E1 (resp. E2) est l'ensemble des étiquettes associées à t1 (resp. t2), l'ensemble des étiquettes associées à t0 est E1 u E2.

L'opération de composition se complique du fait que généralement les tâches à composer possèdent en leur corps plusieurs instructions (donc plusieurs transitions) représentant l'échange d'une même variable. Un rendez-vous pour l'échange de v peut être effectué lorsqu'une transition quelconque de G1 dont l'interprétation comporte l'émission de v est exécutée simultanément avec une transition quelconque de G2 dont l'interprétation comporte la réception de v. Si G1 comporte n1 transitions du type de t1 et G2 n2 du type de t2 (notons respectivement (t1i)i et (t2j)j les ensembles de ces transitions), le nombre de rendez-vous possibles est n1 X n2, chaque rendez-vous étant représenté par un couple (t1k, t2k') du produit cartésien (t1i)i X (t2j)j. On procèdera donc à une fusion pour chacun des couples de transitions de (t1i)i X (t2j)j.

Cette opération doit être effectuée pour toute variable émise par l'une des deux tâches et reçue par l'autre.

Une tâche composée comportant en général plusieurs tâches, l'opération de composition entre deux tâches devra être associative pour permettre la réalisation de la composition d'un ensemble de tâches de façon incrémentale. Elle doit de plus tenir compte du fait qu'il peut y avoir :

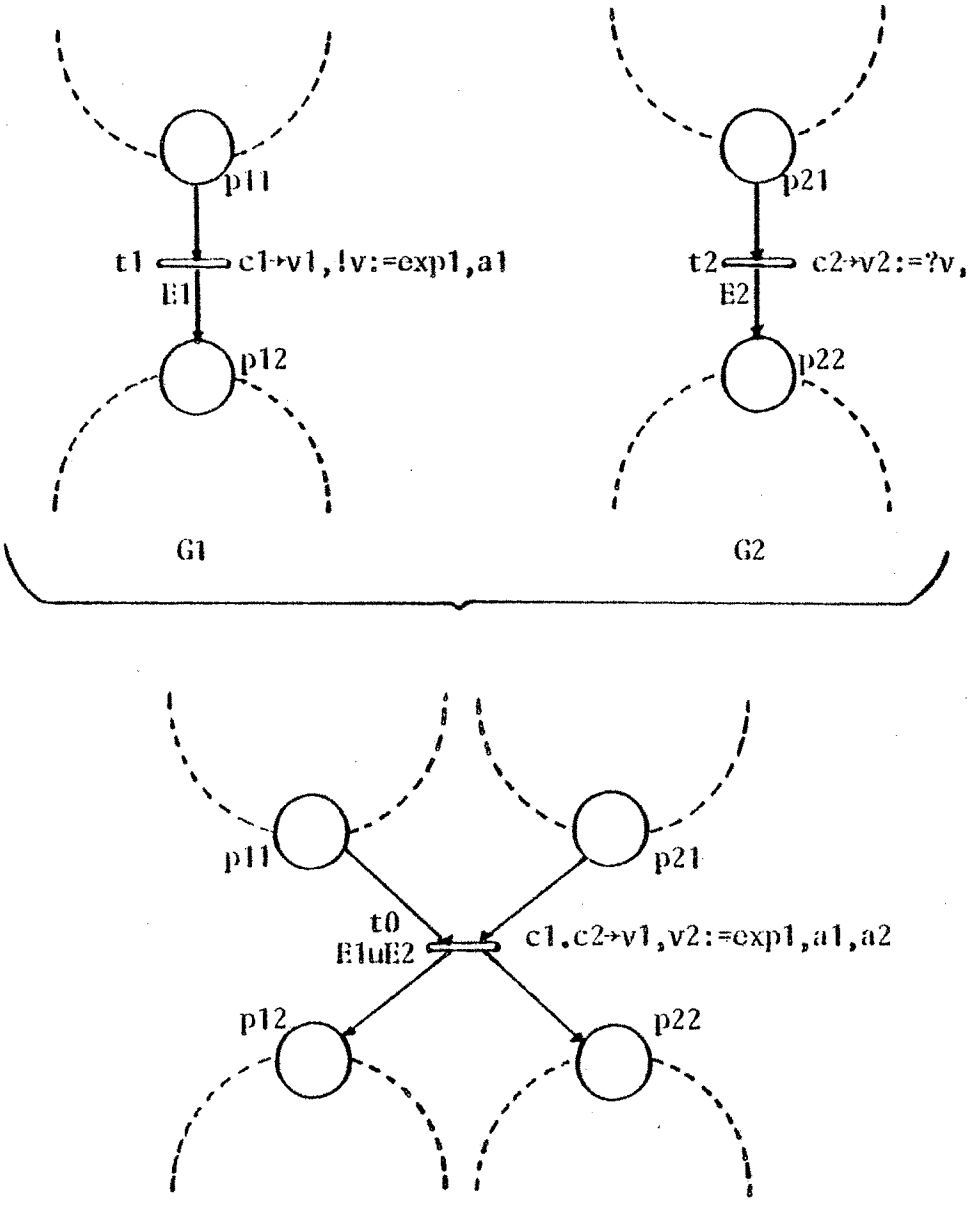


Figure 6 - Fusion de deux transitions

- plusieurs tâches émettrices de la même variable (il suffit que l'une quelconque l'émette),
- plusieurs tâches réceptrices de la même variable, en mode ANY (l'une quelconque doit la recevoir) ou en mode ALL (toutes doivent la recevoir).

Cela conduit à définir une opération de fusion nettement plus complexe que celle présentée plus haut, de façon à pouvoir dans tous les cas se réserver, après la composition de deux tâches, la possibilité de composer à nouveau le réseau obtenu, suivant les modes et sens d'échanges de chacune des variables échangées communes aux tâches à composer.

L'opération de composition est définie en 6.2 p. III-67.

3.7 - Principe de l'opération d'encapsulation

L'opération de composition laisse ouvertes toutes les possibilités de communication (ce qui garantit son associativité). Lorsque la totalité de la tâche composée a été obtenue de façon incrémentale, il est alors nécessaire de supprimer toutes ces possibilités qui ont été introduites pour permettre une opération de composition ultérieure, afin de ramener les possibilités de communication du réseau représentant la tâche composée à celles indiquées par sa partie formelle. C'est le rôle de l'opération d'encapsulation. Elle consiste à supprimer certaines transitions ou à en modifier l'interprétation, de façon à rendre invisibles certains échanges.

L'opération d'encapsulation est définie en 6.3 p. III-82.

4 - TRADUCTION DES INSTRUCTIONS

On donne ici sous forme graphique les méta-règles utilisées pour la traduction des instructions. Les symboles terminaux et non-terminaux figurant dans le contenu des briques sont ceux de la grammaire du langage CESAR figurant en annexe (à l'exception de certains symboles introduits ici pour simplifier les notations).

Au cours de la génération du réseau, les noms (de variables, procédures, actions, tâches, variables échangées) sont transformées en noms absolus (cf 5 p. III-55). Les variables non-significatives sont éliminées (cf 3.3 p. III-21).

Notations :

Dans toutes les méta-règles on notera :

- c le non-terminal <condition>,
- H un ensemble d'étiquettes associées à la brique ouverte en partie gauche de méta-règle (étiquettes "héritées" : elles sont toutes de la forme "_x_"),
- H_ l'ensemble d'étiquettes obtenu en transformant chaque étiquette "_x_" de H en "x_",
- e un symbole équivalent à [<étiquette>]*,
- E (resp. _E, _E_, E_) l'ensemble des étiquettes de la forme "x" (resp. "_x", "_x_", "x_") où x décrit l'ensemble des étiquettes absolues générées par e.

On notera dans l'interprétation des réseaux :
- ^ : la négation booléenne (NOT en CESAR),
- + : l'union booléenne (OR en CESAR),
- . : l'intersection booléenne (AND en CESAR).

4.1 - Séquence d'instructions

Règle de grammaire :

<séquence> ::= [<instruction>]+

Méta-règle associée : figure 1.

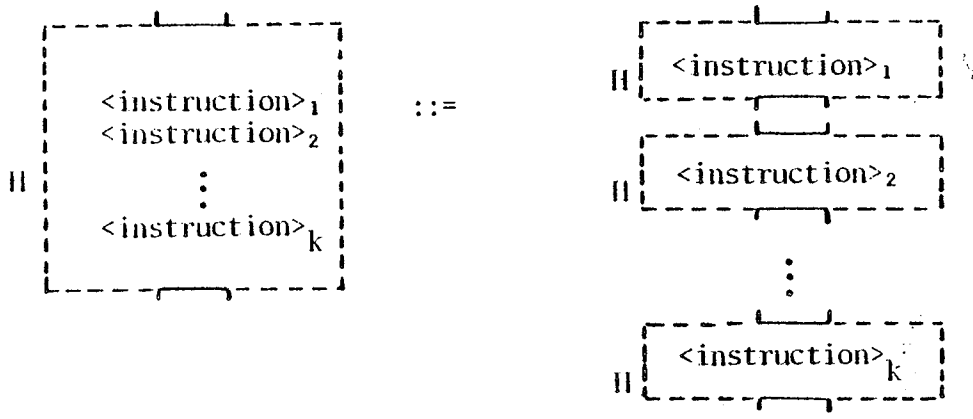


Figure 1

4.2 - Instruction_vide

Règle_de_grammaire_:

<instruction vide> ::= [<étiquette>]* NULL ;

Méta-règle_associée_: figure 2.

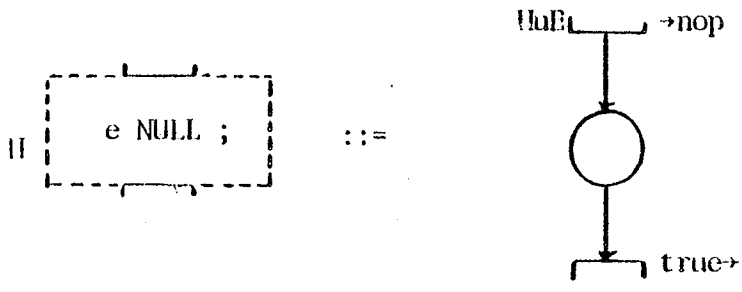


Figure 2

4.3 - Affectation

Règle_de_grammaire_:

<affectation> ::= [<étiquette>]* [<échange> ,]
<affectation simple> [, <affectation simple>]* ; |
[<étiquette>]* <échange> ;

Méta-règle associée : figure 3.

(On note a un symbole équivalent à :
[<échange>,] <affectation simple>
[,<affectation simple>]* |
<échange> .)

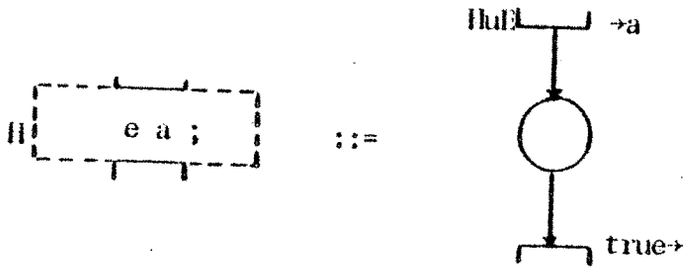


Figure 3

Remarque :

La syntaxe CESAR garantit que le réseau généré est à échanges simples.

4.4 - Instruction conditionnelle

Règle de grammaire :

```
<instruction if> ::= [<étiquette>]*
          IF <condition> THEN <séquence>
          [ ELSIF <condition> THEN <séquence> ]*
          [ ELSE <séquence> ] END [IF] ;
```

Quatre méta-règles sont employées, suivant la présence ou non des clauses ELSIF et ELSE.

Méta-règles associées :

* sans clauses ELSIF ni ELSE : figure 4.

* sans clauses ELSIF mais avec clause ELSE : figure 5.

Remarque :

Dans les deux règles précédentes, si la condition c a un résultat non spécifié, les conditions c et ~c du réseau sont toutes deux remplacées par true.

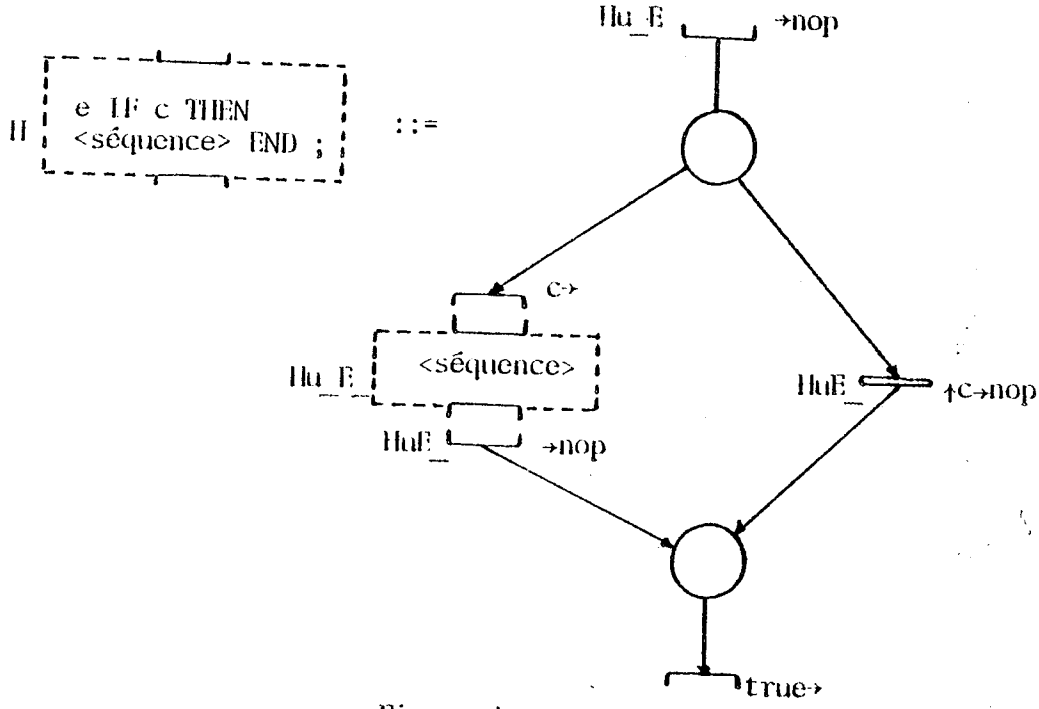


Figure 4

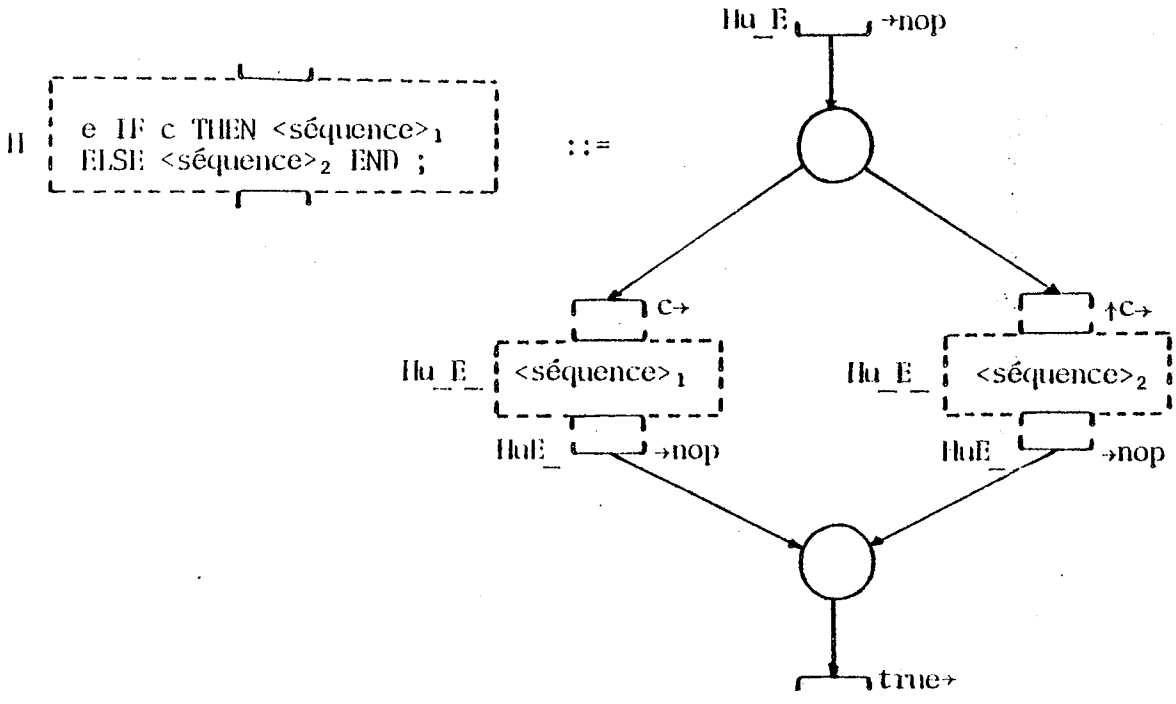


Figure 5

* avec clauses ELSIF mais sans clause ELSE : figure 6.

* avec clauses ELSIF et ELSE : figure 7.

Remarque :

Dans les deux règles précédentes, si une condition ci a un résultat non spécifié, les conditions ci et ~ci du réseau sont toutes deux remplacées par true.

4.5 - Instruction_cas

Règle de grammaire :

```
<instruction case> ::= [<étiquette>]*
          CASE <expression> IS
          [ WHEN <choix> [ | <choix> ]* => <séquence> ]*
          [ WHEN OTHERS => <séquence> ]
          END [CASE] ;
```

Deux méta-règles sont employées, suivant la présence ou non d'une clause WHEN OTHERS.

Notons :
- C un symbole équivalent à <choix> [| <choix>]* ,
- exp le non-terminal <expression>,
- K la condition associée à C qui est de la forme

$$k1 + \ldots + kp$$

où chaque ki correspond à un choix de C.

La grammaire CESAR donne pour le non-terminal <choix> la règle suivante :

```
<choix> ::= <expression> | <intervalle discret>
```

Suivant le cas la condition Ki sera donc de la forme :

$$exp0 = expi$$

ou $(expi1 \leq exp0) \,.\, (exp0 \leq expi2)$

si exp0 est l'expression testée dans l'instruction CASE.

Méta-règles associées :

* sans clause WHEN OTHERS : figure 8.

* avec clause WHEN OTHERS : figure 9.

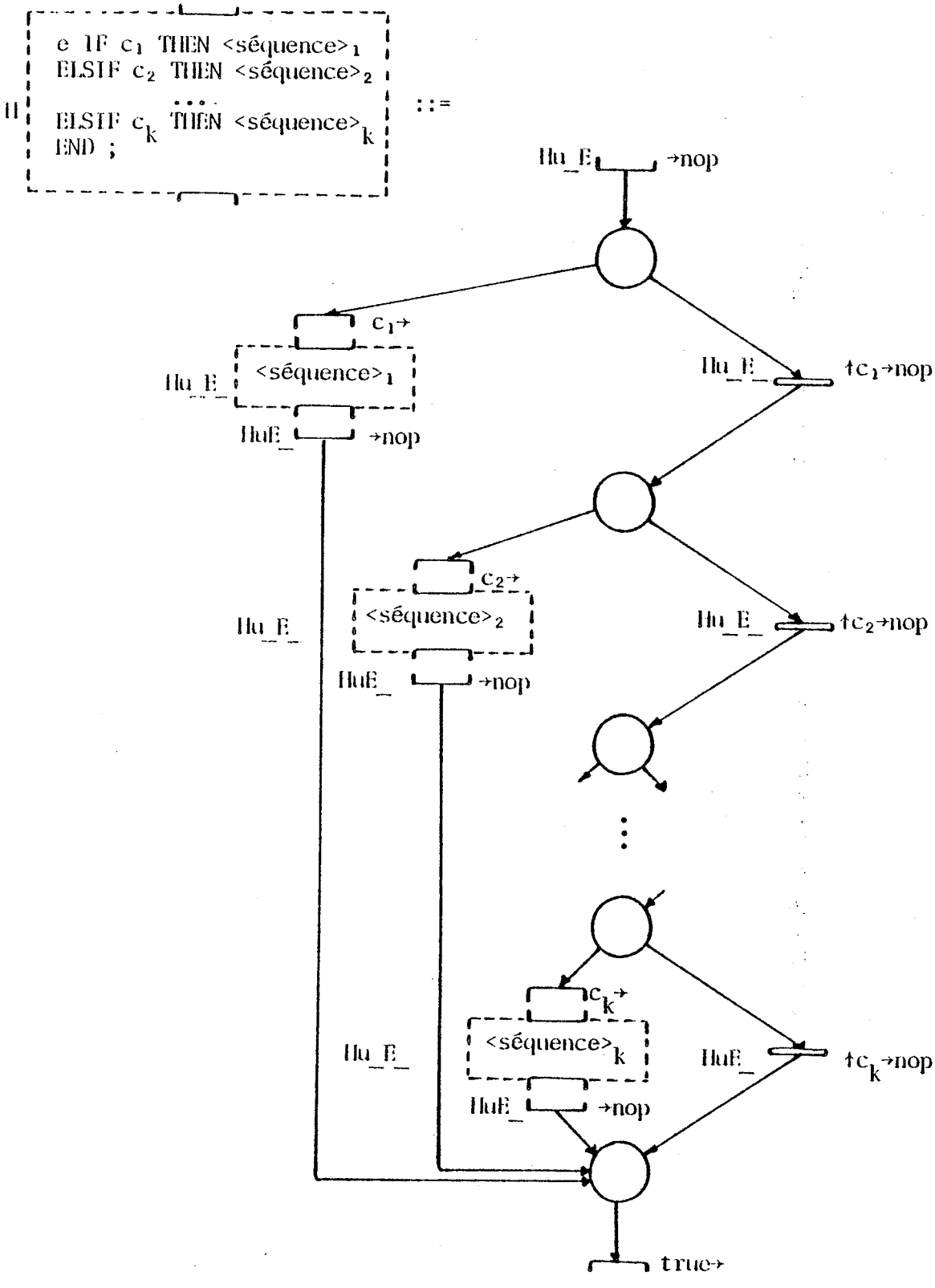

e IF c_1 THEN <séquence>$_1$
ELSIF c_2 THEN <séquence>$_2$
...
ELSIF c_k THEN <séquence>$_k$
END ;
::=
Hu_E
→nop
c_1→
<séquence>$_1$
HuE_
↑c_1→nop
c_2→
<séquence>$_2$
↑c_2→nop
c_k→
<séquence>$_k$
↑c_k→nop
true→

Figure 6

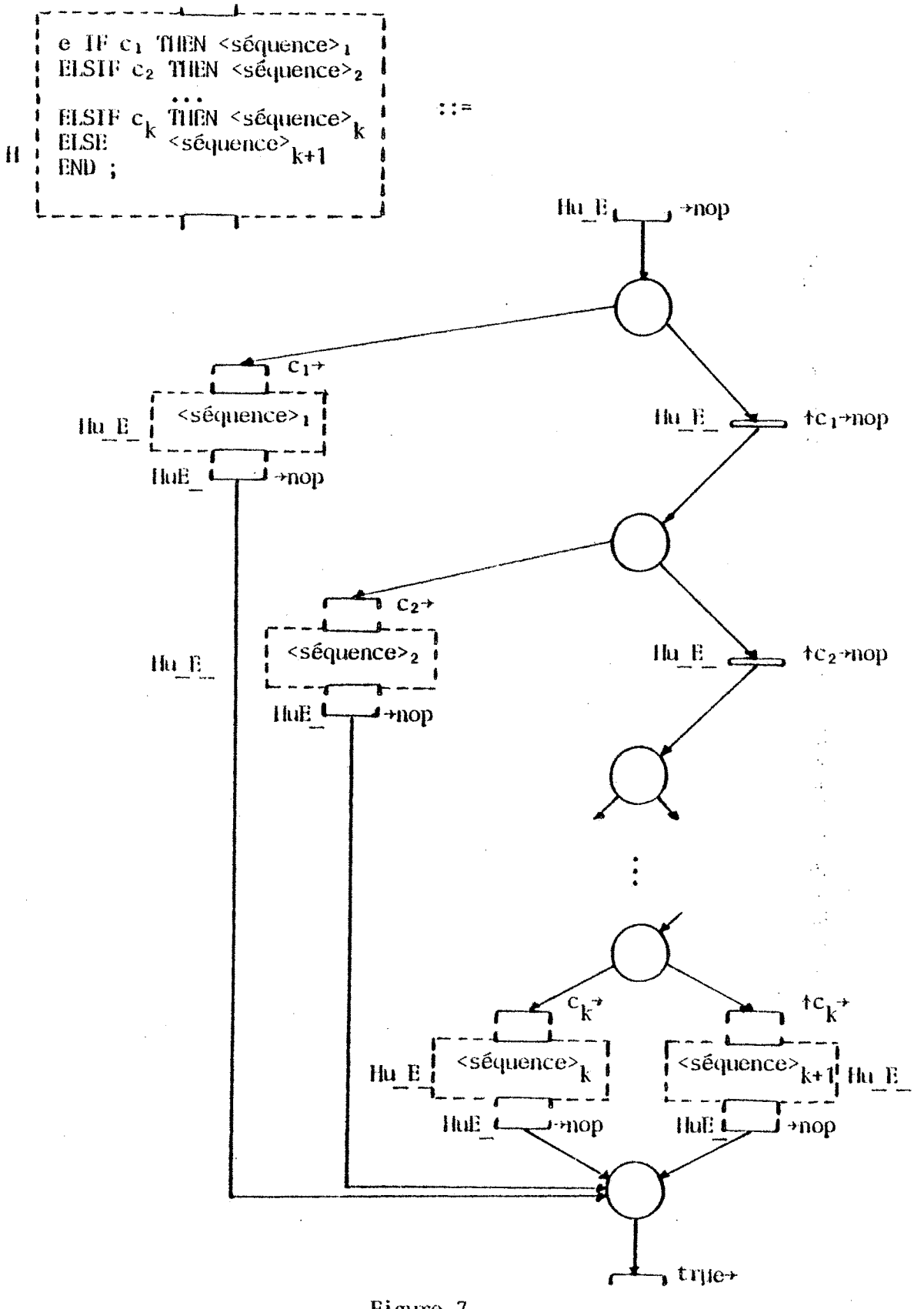

e IF c_1 THEN <séquence>$_1$
ELSIF c_2 THEN <séquence>$_2$
...
ELSIF c_k THEN <séquence>$_k$
ELSE <séquence>$_{k+1}$
END ;
::=
Hu_E
→nop
c_1→
<séquence>$_1$
↑c_1→nop
c_2→
<séquence>$_2$
↑c_2→nop
c_k→
<séquence>$_k$
↑c_k→
<séquence>$_{k+1}$
true→

Figure 7

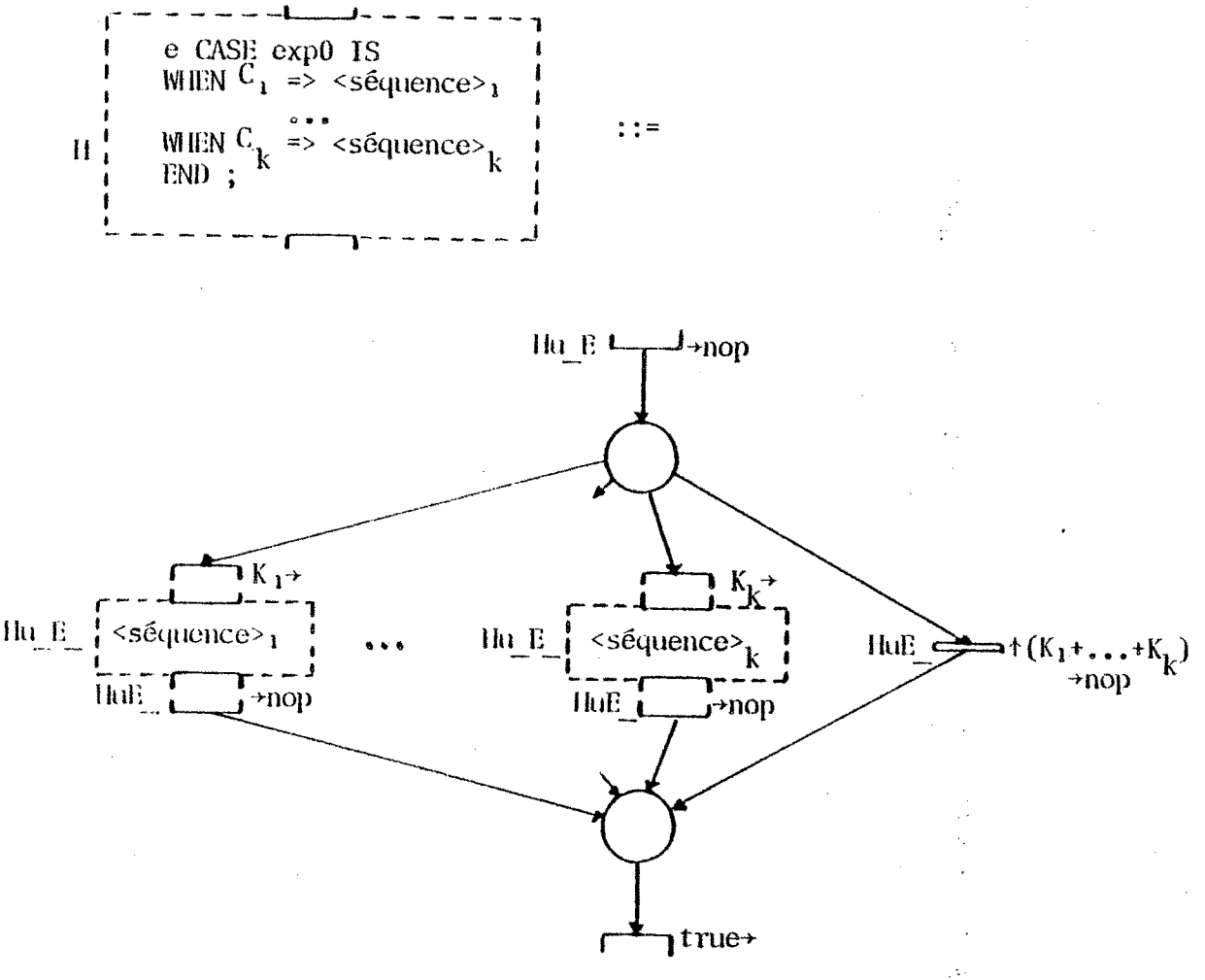

e CASE exp0 IS
WHEN C1 => <séquence>1
WHEN Ck => <séquence>k
END ;
H
::=
Hu_E
→nop
K1→
<séquence>1
HuE_
→nop
Hu_E
Kk→
<séquence>k
HuE_
→nop
HuE_
↑(K1+...+Kk)
→nop
true→

Figure 8

```
e CASE exp0 IS
WHEN C 1 => <séquence>1
...
WHEN C k => <séquence>k
WHEN OTHERS => <séquence>k+1
END ;
```

H ::=

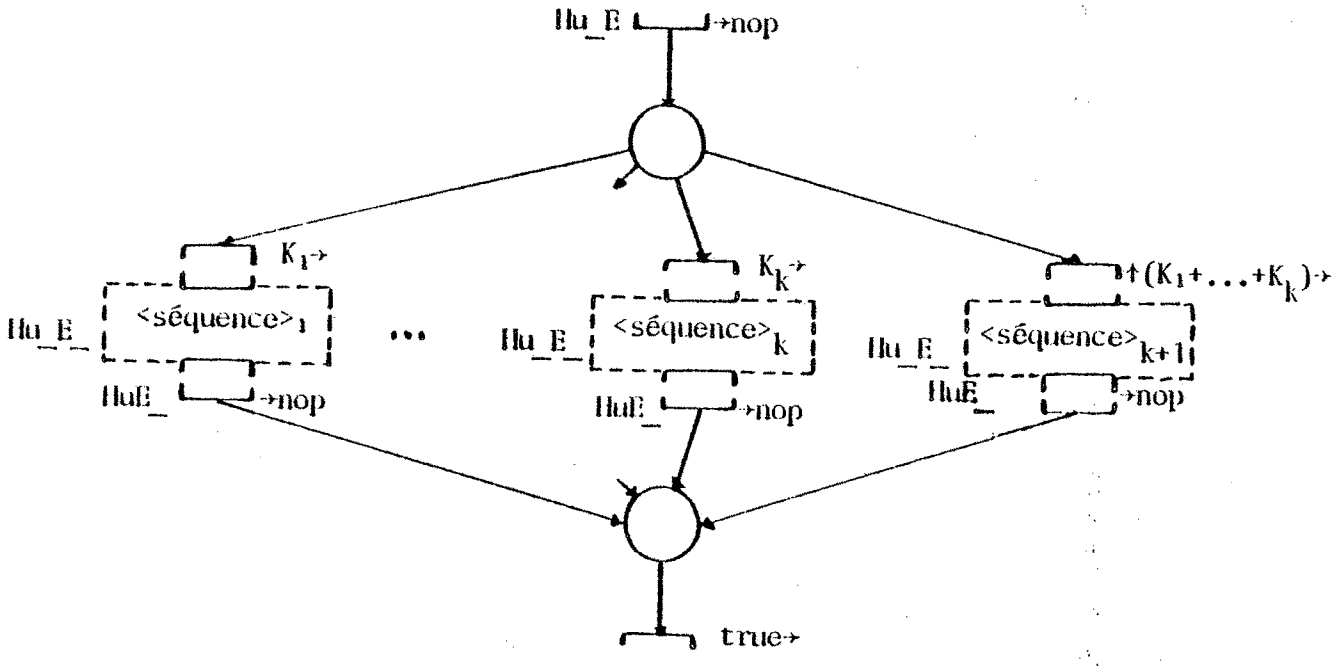


Figure 9

Remarque :

Si l'expression testée a un résultat non spécifié, toutes les conditions Ki, ainsi que la condition ~(K1 + K2 + ... + Kp) dans le deuxième cas, sont remplacées par true dans le réseau.

4.6 - Instructions_itératives

Règles_de_grammaire_:

```
<instruction loop> ::= [<étiquette>]* [<clause itérative>]
          <corps de loop>

<clause itérative> ::= WHILE <condition> |
          FOR <identificateur> IN [REVERSE]
          <contrainte d'appartenance>

<corps de loop> ::= LOOP <séquence> END [LOOP] ;
```

Quatre méta-règles sont employées suivant la présence ou l'absence d'une clause itérative et sa forme. Dans tous les cas, une place particulière px est générée, qui pourra être utilisée par la suite si la séquence à l'intérieur du LOOP comporte une instruction EXIT.

Méta-règles_associées_:

* sans clause itérative : figure 10.

* avec clause WHILE : figure 11.

Remarque_:

Si la condition c de la clause WHILE a un résultat non spécifié, les conditions c et ~c sont remplacées par true dans le réseau.

* avec clause FOR croissante : figure 12.

(On note :
- id le non-terminal <identificateur>,
- I le nom absolu de id,
- delta la contrainte d'appartenance,
- IT la variable auxiliaire utilisée pour mémoriser la borne à atteindre de delta,
- e1 et e2 les expressions représentant les bornes de delta.)

Remarques_:

Les variables I et IT sont rajoutées au vecteur de variables X de l'interprétation.

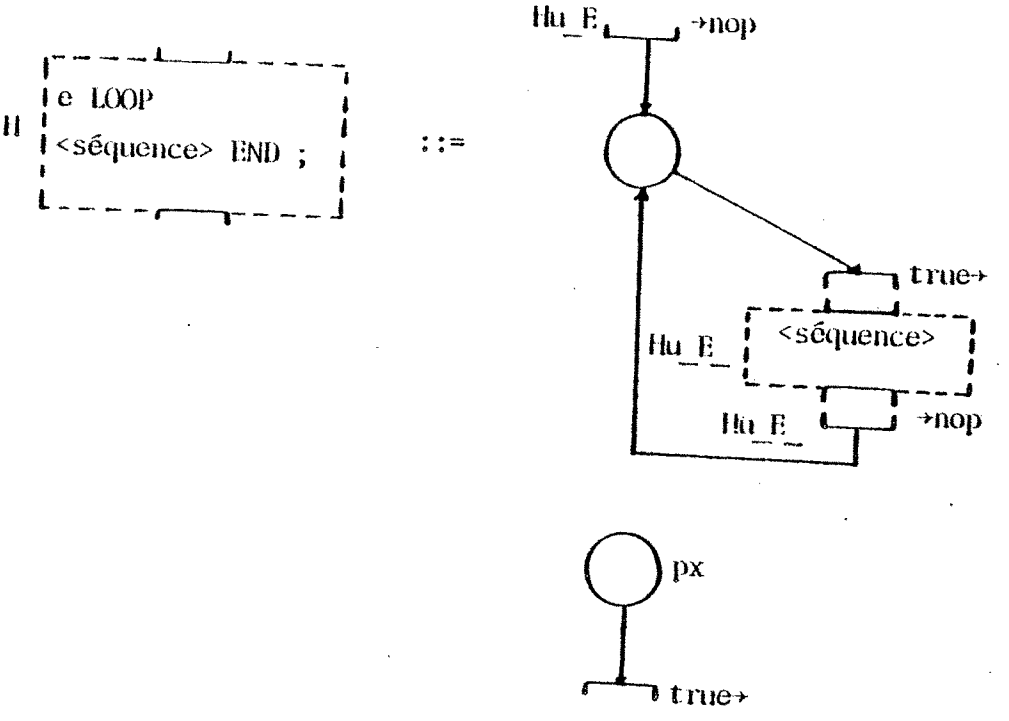


Figure 10

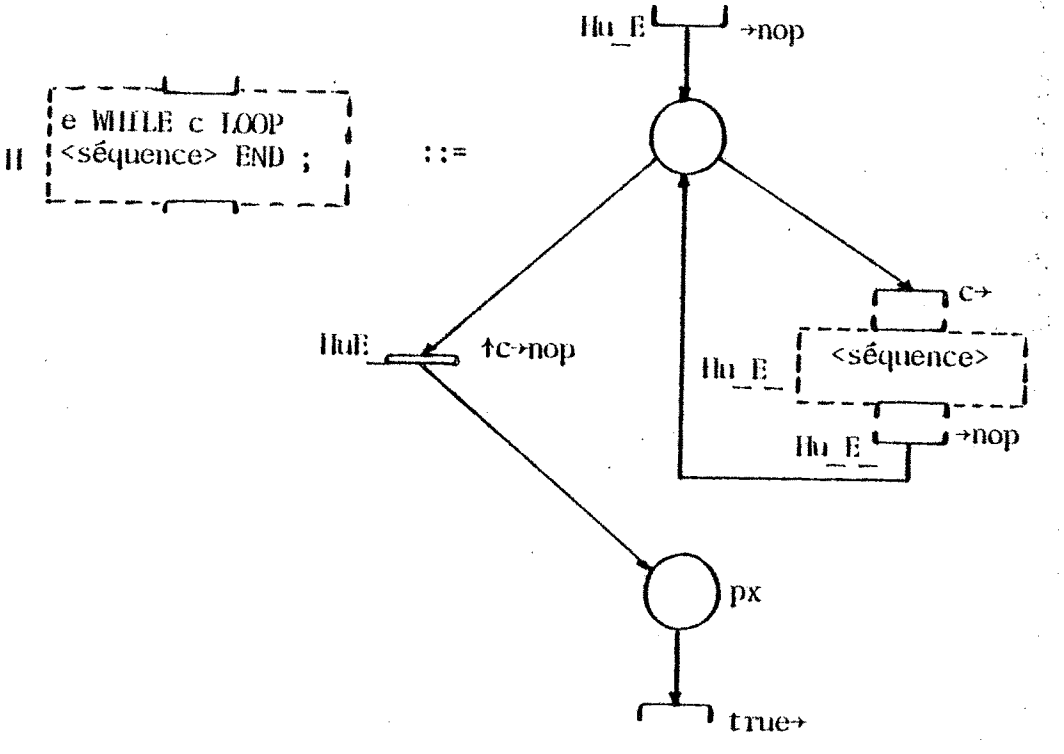


Figure 11

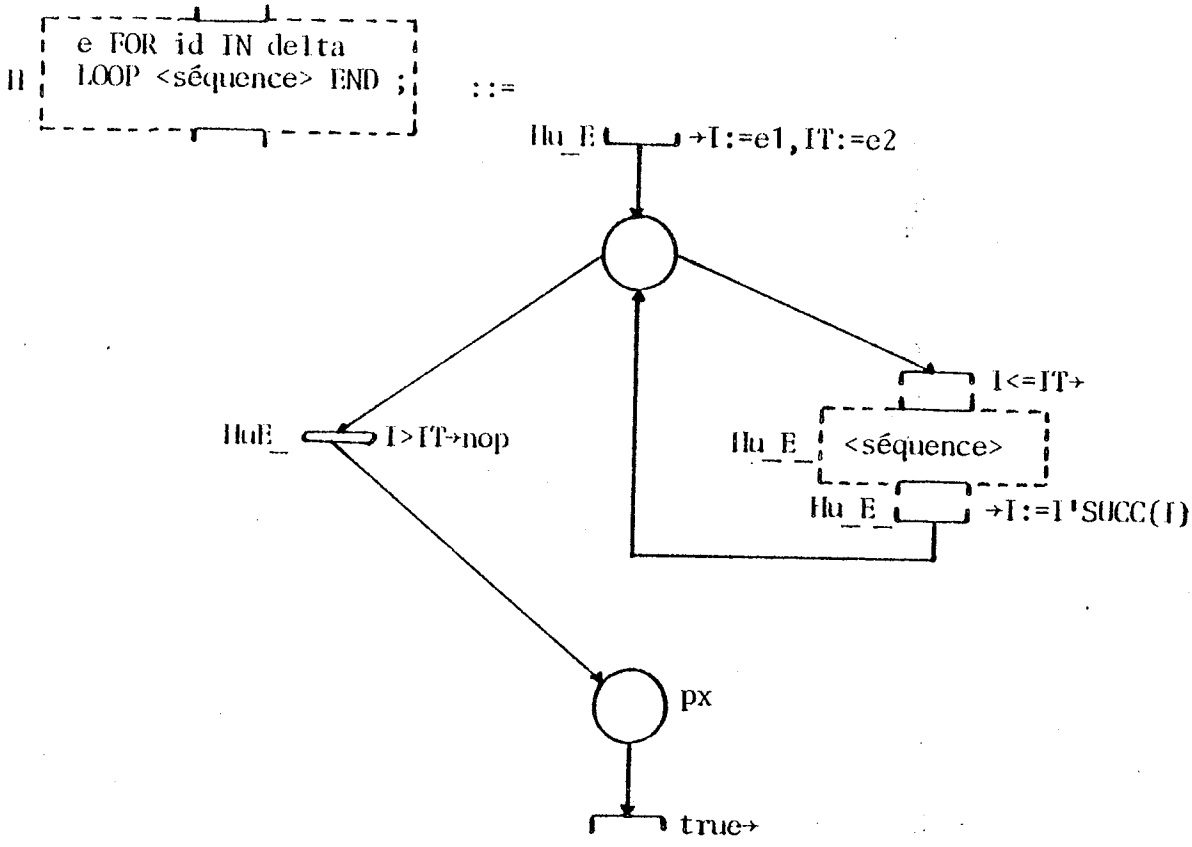


Figure 12

La borne e2 peut être
- soit infinie, auquel cas la condition I <= IT est remplacée par true, et la condition complémentaire par false,
- soit à résultat non spécifié, auquel cas ces deux conditions sont remplacées par true.

Dans les deux cas, les variables I et IT seront éliminées.

* avec clause FOR décroissante : figure 13.

Remarque :

Comme précédemment, la borne à atteindre (e1 dans ce cas) peut être infinie ou à résultat non spécifié.

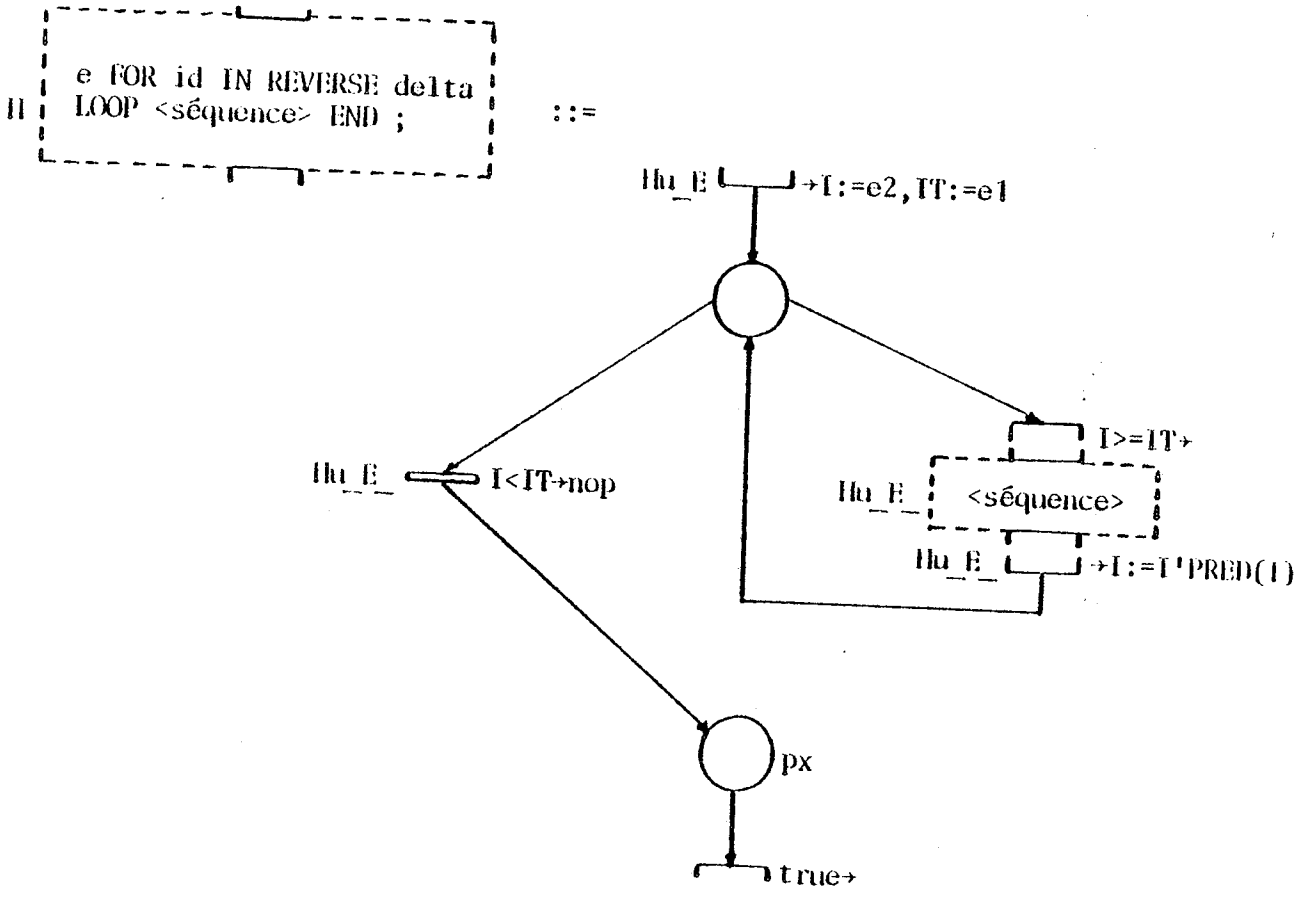


Figure 13

4.7 - Instruction_EXIT

Règle_de_grammaire_:

<exit> ::= [<étiquette>]* EXIT [WHEN <condition>] ;

Deux règles sont employées pour la traduction de l'instruction EXIT, suivant la présence ou l'absence d'une clause WHEN.

Pour la traduction de l'instruction EXIT, l'ensemble des étiquettes héritées H sera partitionné en deux ensembles H1 et H2 tels que :

* H1 est l'ensemble des étiquettes héritées correspondant à des actions commencées avant le début de l'instruction itérative immédiatement englobante de l'instruction EXIT en cours de traduction,
* H2 est l'ensemble des étiquettes héritées correspondant aux actions commencées depuis le début de cette même instruction itérative (y compris les étiquettes associées à

l'instruction itérative elle-même).

D'une façon pratique, cette partition sera obtenue à la compilation en gérant en pile les étiquettes et en insérant dans la pile un marqueur à chaque entrée de boucle.

La place px est notée en pointillés, pour signifier qu'elle n'est pas générée lors de la traduction de l'instruction EXIT. Elle a été générée lors de la traduction de l'instruction LOOP immédiatement englobante.

Méta-règles associées :

* sans clause WHEN : figure 14.

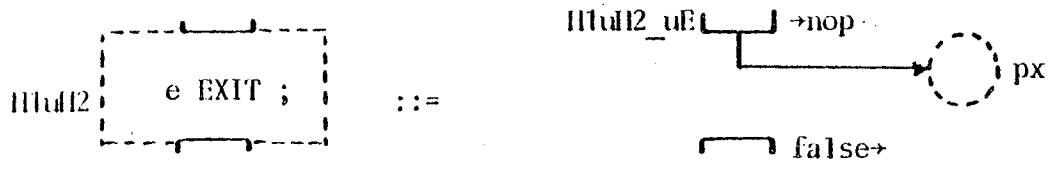


Figure 14

* avec clause WHEN : figure 15.

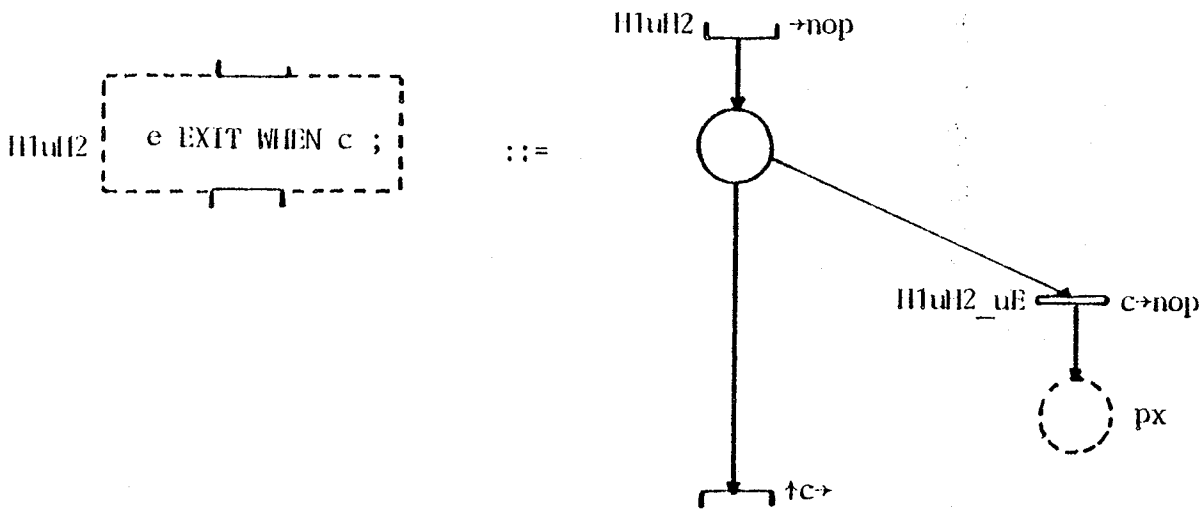


Figure 15

Remarque :

Si la condition c de la clause WHEN a un résultat non spécifié, les conditions c et ¯c du réseau sont toutes deux remplacées par true.

4.8 - Instruction choix non déterministe

Règle de grammaire :

```
<instruction either> ::= [<étiquette>]* EITHER
          [ WHEN <condition> => ] <séquence>
          [ OR [ WHEN <condition> => ] <séquence> ]*
          [ ELSE <séquence> ] END [EITHER] ;
```

Deux règles sont employées suivant la présence ou l'absence d'une clause ELSE.

L'absence de clause WHEN <condition> est exactement traitée comme WHEN TRUE.

Méta-règles associées :

* sans clause ELSE : figure 16.

Remarque :

Si une condition ci a un résultat non spécifié elle est remplacée par true dans le réseau.

* avec clause ELSE : figure 17.

Remarque :

Si une condition ci a un résultat non spécifié elle est remplacée par true dans le réseau, et son complément figurant dans la condition correspondant à la clause ELSE également.

4.9 - Appel de procédure

Règle de grammaire :

```
<appel de procédure> ::= [<étiquette>]* <nom de procédure>
          [( <liste de paramètres effectifs> )] ;
```

Méta-règle associée : figure 18.

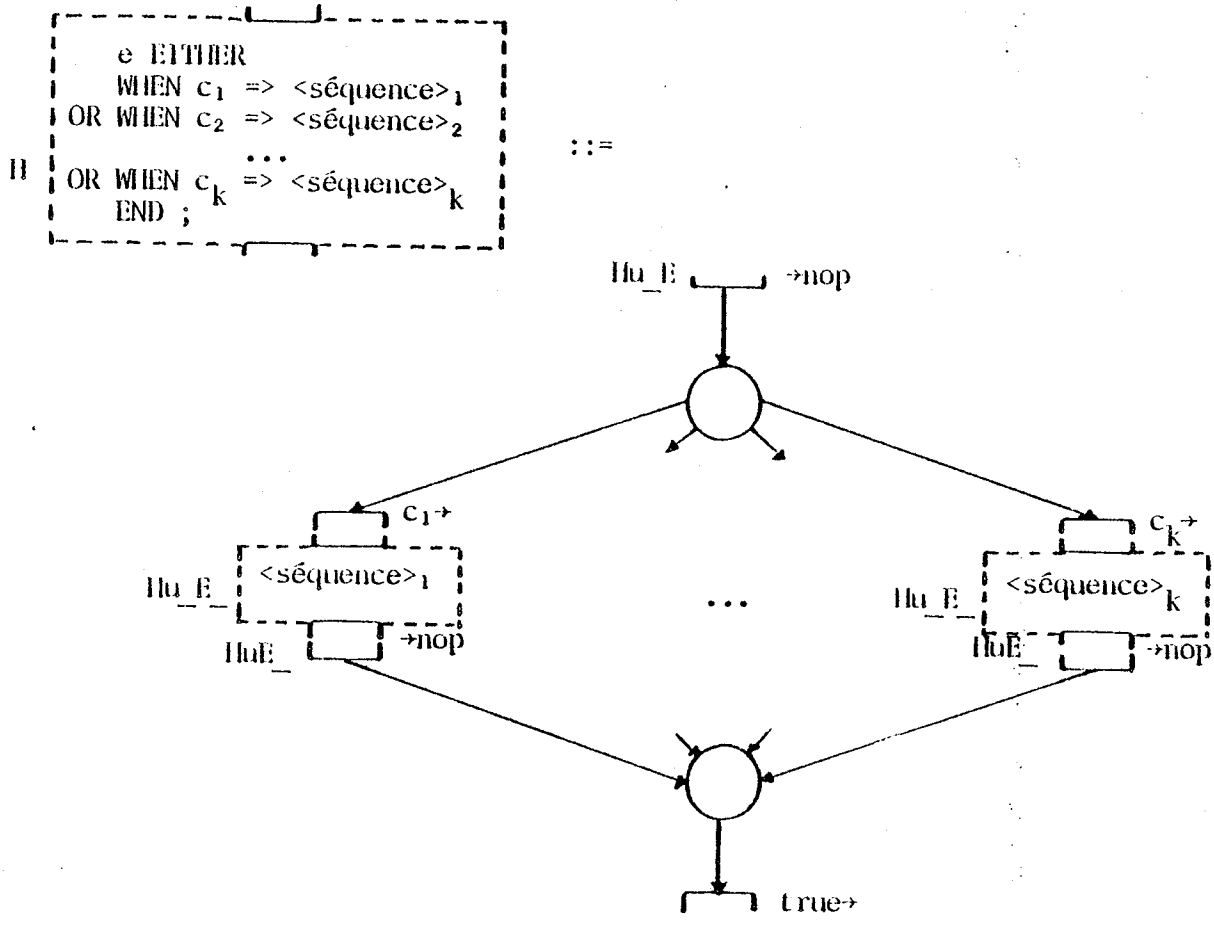

e EITHER
WHEN c1 => <séquence>1
OR WHEN c2 => <séquence>2
...
OR WHEN ck => <séquence>k
END ;
::=
Hu_E
→nop
c1→
<séquence>1
HuE_
→nop
...
ck→
<séquence>k
true→

Figure 16

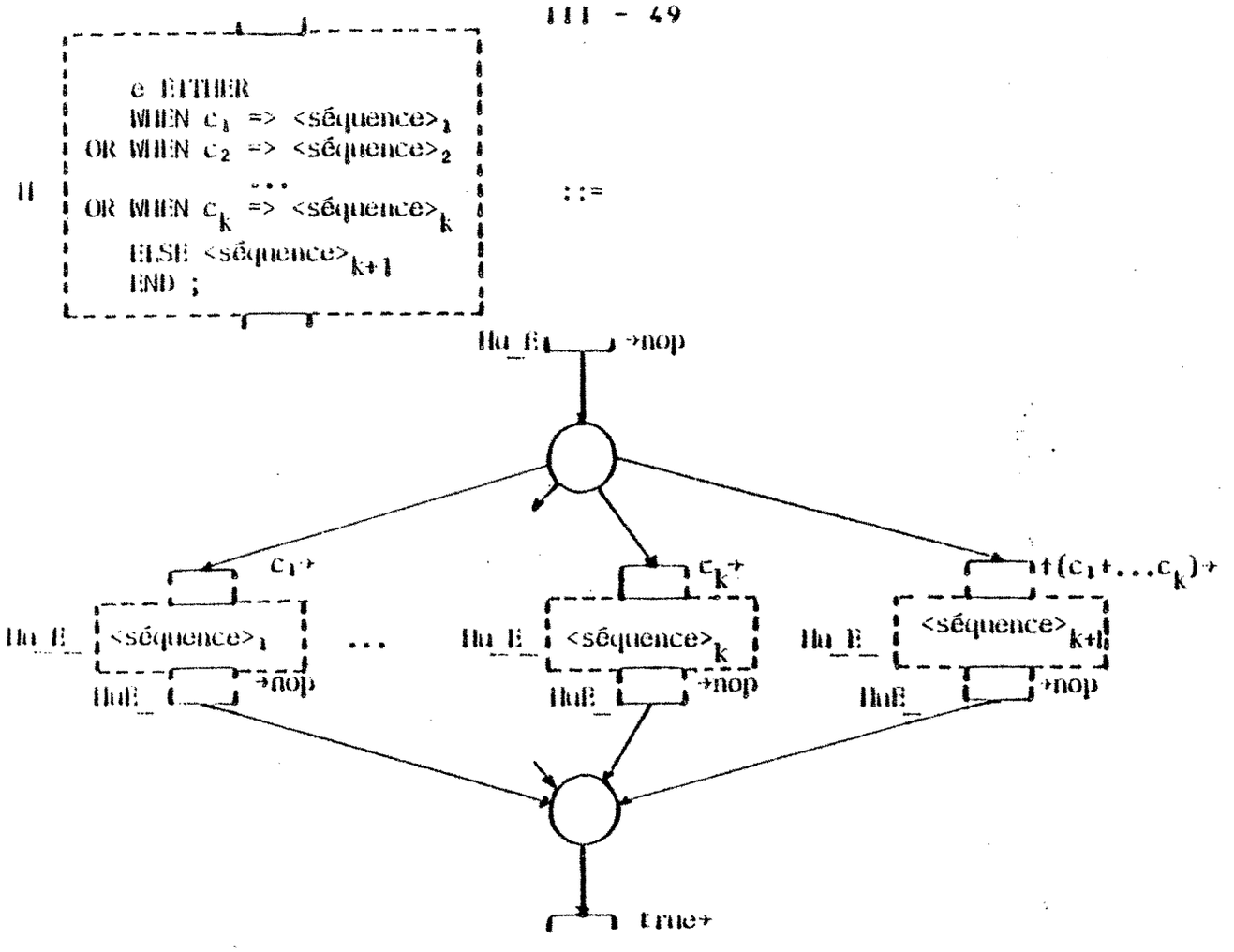


Figure 17

(On note parm un symbole équivalent à
[(<liste de paramètres effectifs>)].)

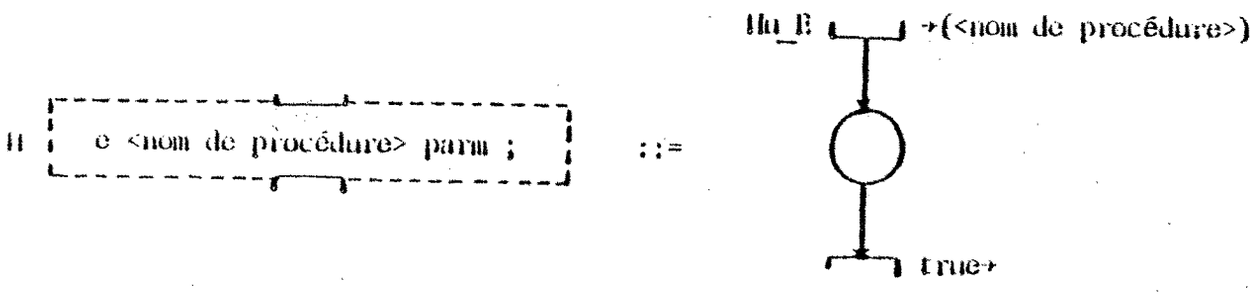


Figure 18

La liste des paramètres effectifs est construite, elle sera utilisée pour l'opération de substitution de la procédure appelée (cf 3.5 p. III-28).

4.10 - Instruction RETURN

Règle de grammaire :

<étiquette> ::= [<étiquette>]* RETURN ;

Méta-règle associée : figure 19.

(La place ps est en pointillés pour signifier qu'elle n'est pas générée lors de la traduction de l'instruction RETURN, mais qu'elle l'a été au moment de l'application de la règle associée à la déclaration de la procédure - cf 5 p. III-55).

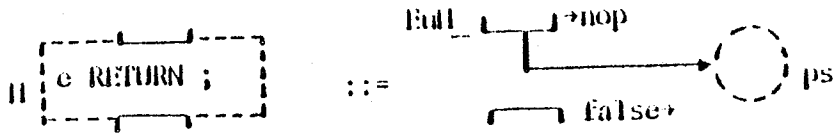


Figure 19

4.11 - Instruction TERMINATE

Règle de grammaire :

<terminate> ::= [<étiquette>]* TERMINATE ;

Deux règles sont employées suivant que l'instruction TERMINATE figure dans une procédure ou dans la séquence principale d'une tâche élémentaire.

Les places pt et pf sont en pointillés pour signifier qu'elles ne sont pas générées lors de la traduction de l'instruction TERMINATE mais lors de l'application de la règle associée à la déclaration de la procédure ou de la tâche où figure l'instruction TERMINATE (cf 5 p. III-55).

Méta-règles associées :

* si l'instruction TERMINATE figure dans une procédure : figure 20.

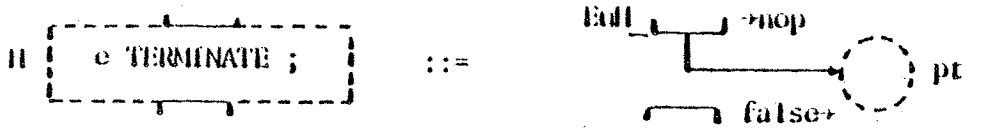


Figure 20

* si l'instruction TERMINATE figure dans une tâche élémentaire : figure 21.

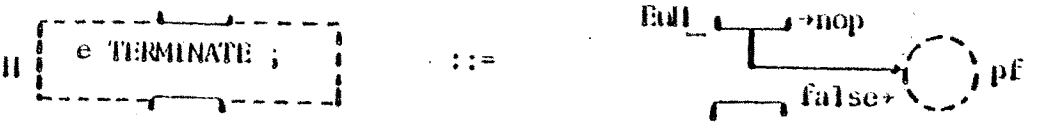


Figure 21

4.12 - Elimination_des_branches_mortes

L'algorithme de génération par application des règles précédentes produit, dans le cas de l'instruction EXIT sans clause WHEN et des instructions RETURN et TERMINATE, des transitions dont l'interprétation est de la forme :

false -> a ou false -> (pr)

ne possédant pas de place d'entrée.

Si l'affectation a est l'identité nop, ces transitions peuvent être supprimées. Sinon il y a une erreur dans le programme de description, puisqu'elles ne sont jamais exécutables.

Si aucune erreur n'est rencontrée lors de l'élimination des branches mortes, le réseau généré pour représenter une procédure ou une tâche (ou type de tâche) élémentaire ou non-spécifiée est un graphe d'états. Cette propriété découle immédiatement de la forme des méta-règles données.

4.13 - Réduction_des_réseaux_générés

L'utilisation des règles précédentes produit un assez grand nombre de transitions dont l'interprétation est :

true -> nop.

Sous certaines conditions, il est possible de réduire ces transitions, selon le schéma exprimé par la figure 22, où E1 (resp. E2, E) sont les ensembles d'étiquettes associés à t1 (resp. t2, t).

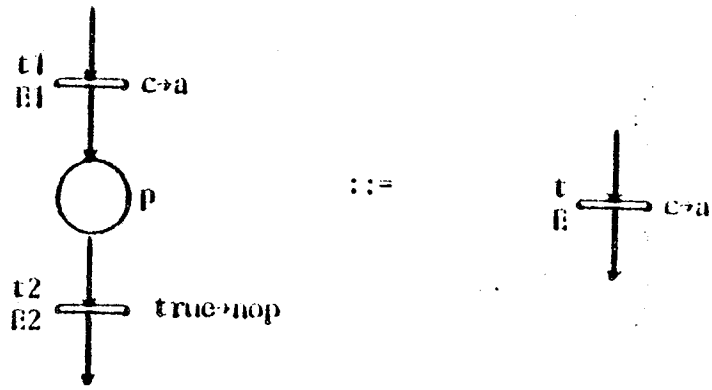


Figure 22

Les conditions sont les suivantes :

1) |>p| = |p>| = 1,

2) p n'est pas initialement marquée,

3) l'ensemble E2 ne contient pas d'étiquettes de la forme "_x" ou "x" (t2 ne marque pas le début d'une action, élémentaire ou non).

L'ensemble E est construit de la façon suivante :

1) si E1 contient une étiquette de la forme "_x", alors
- ou bien E2 contient l'étiquette "_x_", et dans ce cas, E contiendra l'étiquette "_x",
- ou bien E2 contient l'étiquette "x_", et dans ce cas, on mettra l'étiquette "x" dans E (l'action composée x est devenue élémentaire par fusion des deux transitions la représentant) ;

2) si E1 contient une étiquette de la forme "_x_", alors
- ou bien E2 contient la même étiquette, et E la contiendra également,
- ou bien E2 contient l'étiquette "x_", et dans ce cas, E contiendra cette dernière ;

3) les étiquettes de E1 de la forme "x" ou "x_" figurent

telles quelles dans E.

D'une façon pratique, cette règle de réduction est appliquée chaque fois que cela est possible lors de la génération des réseaux.

5 - Traduction des blocs, procédures et tâches élémentaires ou non-spécifiées

Chaque procédure ou tâche élémentaire ou non-spécifiée est représentée par un GE interprété (éventuellement communicant ou procédural) étiqueté.

Les notations employées sont les mêmes que celles du paragraphe "Traduction des instructions" (cf 4 p. III-34).

5.1 - Traduction des blocs

Règle de grammaire :

<bloc> ::= [DECLARE <partie déclarative de bloc>]
[<étiquette>]* BEGIN <séquence> END ;

Méta-règle associée : figure 1.

(On note init l'affectation simultanée de toutes les variables locales significatives par leur valeur initiale.)

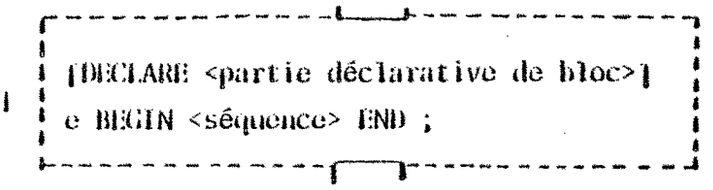
Le bloc étant un environnement anonyme (contrairement à une procédure ou une tâche qui possède un nom), il est nécessaire de lui définir un nom absolu, qui soit différent pour chaque bloc d'une même procédure ou tâche. On lui donne le nom

nea.BLOCi

où nea est le nom absolu de l'environnement et i un compteur incrémenté chaque fois qu'un bloc est rencontré au cours de la compilation.

La partie déclarative est analysée afin de rajouter au vecteur de variables X de l'interprétation les variables déclarées dans le bloc (en transformant leurs noms en noms absolus par préfixation par le nom absolu du bloc).

Les variables de type non-spécifié sont marquées comme devant être éliminées (y compris les tableaux d'éléments non spécifiés, et les champs non spécifiés des structures). Les variables du bloc non initialisées le sont également.



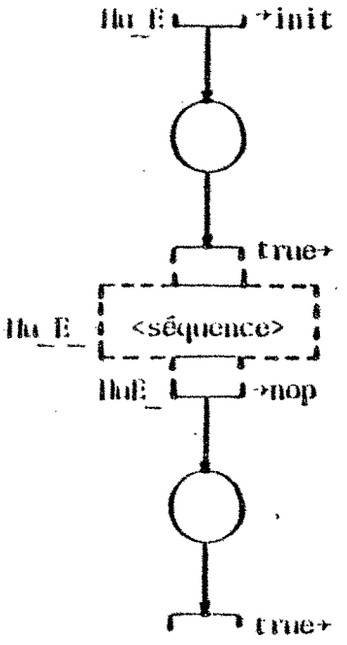


Figure 1

5.2 - Traduction des procédures

Règles de grammaire :

```
<déclaration de procédure> ::= PROCEDURE
          <en-tête de procédure> [ IS
          <partie déclarative de bloc>
          <corps de procédure> ] ; |
          <référence procédures externes>

<corps de procédure> ::= [<étiquette>]*
          BEGIN <séquence> END
          [<fin de procédure>]
```

Pour la traduction des procédures, on emploiera la méta-grammaire d'axiome une brique fermée de contenu le

symbole non-terminal <déclaration de procédure>.

5.2.1 - Procédure spécifiée

Méta-règle associée : figure 2.

(On note init l'affectation simultanée de toutes les variables locales significatives par leur valeur initiale.)

PROCEDURE <en-tête de procédure> IS
<partie déclarative de bloc>
e BEGIN <séquence> END ;

::=

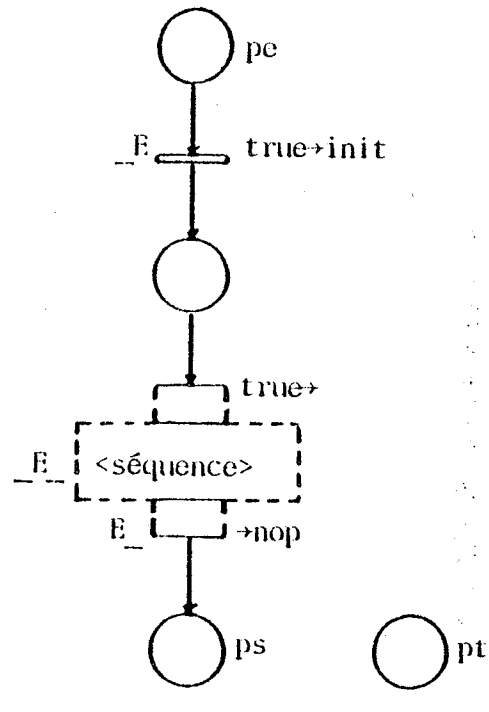


Figure 2

L'en-tête et la partie déclarative sont analysées afin de :

- définir le nom absolu de la procédure

nap = nae.n

si nae est le nom absolu de l'environnement où est déclaré la procédure et n le nom déclaré (identificateur) de la

procédure. Ce nom sera donné au réseau généré.

- construire la liste des paramètres avec leurs types, chaque paramètre d'identificateur p ayant un nom absolu nap.p . Cette liste de paramètres sera exploitée par l'algorithme de substitution.

- construire le vecteur de variables X de l'interprétation qui est formé :
  * du vecteur de variables de l'environnement où est déclaré la procédure,
  * des paramètres (non échangés) de la procédure (considérés comme des variables),
  * des variables déclarées dans la partie déclarative de le procédure (en transformant leurs noms en noms absolus).

- construire si nécessaire les vecteurs de variables échangées Xin et Xout de l'interprétation à partir de la liste des paramètres échangés de la procédure.

Les variables et paramètres de la procédure de types non-spécifiés sont marqués comme devant être éliminés (y compris les tableaux d'éléments non spécifiés, et les champs non spécifiés des structures). Les variables de la procédure non initialisées le sont également.

5.2.2 - Procédure non spécifiée

Méta-règle associée : figure 3.

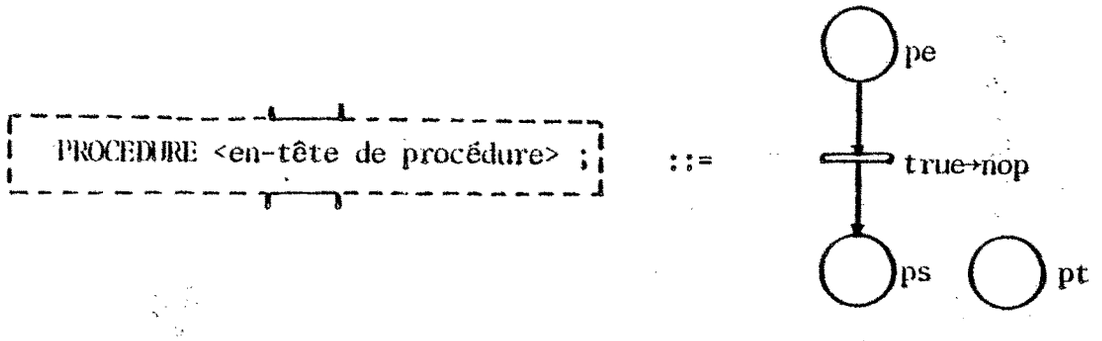


Figure 3

L'en-tête est analysé afin de :

- définir le nom absolu de la procédure, qui sera donné au réseau généré,

- construire la liste des paramètres avec leurs noms absolus et leurs types, ce qui définit le vecteur X de ce réseau ; (rappelons qu'une procédure non-spécifiée ne peut avoir que des paramètres non échangés).

Les vecteurs de variables échangées Xin, Xout de l'interprétation de ce réseau sont vides. L'ensemble de ses étiquettes également.

5.2.3 - Référence aux procédures externes

Aucune action de génération n'est à effectuer, les procédures ayant précédemment été compilées (au niveau train de compilation, donc leur nom absolu est égal à leur nom déclaré).

5.3 - Traduction des tâches élémentaires ou non-spécifiées

Une tâche élémentaire ou non-spécifiée peut être déclarée
- soit comme une tâche simple,
- soit comme une tâche d'un type.

Dans le deuxième cas, il est nécessaire d'avoir au préalable compilé la déclaration du type de tâche, ce qui aura permis de générer un réseau le représentant. On effectuera ensuite une "instanciation" (cf 5.3.3 p. III-62) de la tâche en utilisant le réseau représentant le type de tâche.

5.3.1 - Tâche élémentaire simple ou type de tâche élémentaire

Pour la traduction des tâches élémentaires simples ou des types de tâches élémentaires, on emploiera la méta-grammaire d'axiome une brique fermée de contenu le symbole non-terminal <déclaration de tâche élémentaire>.

Règles de grammaire :

```
<déclaration de tâche élémentaire> ::= TASK [TYPE]
          <en-tête de tâche> IS
          <partie déclarative de bloc>
          <corps de tâche élémentaire> ;

<corps de tâche élémentaire> ::=
          BEGIN <séquence> END
          [<fin de tâche élémentaire>]
```

Méta-règle associée : figure 4.

TASK [TYPE]<en-tête de tâche> IS
<partie déclarative de bloc>
BEGIN <séquence> END ; ::=

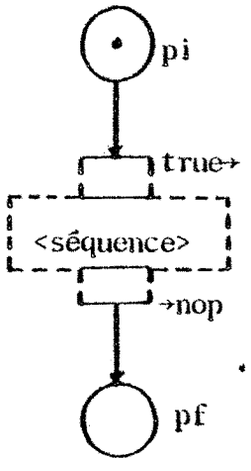


Figure 4

L'en-tête et la partie déclarative sont analysées afin de :

- définir le nom absolu de la tâche (ou du type de tâche)
nat = nae.n
si nae est le nom absolu de l'environnement où est déclaré la tâche (ou le type de tâche) avec le nom n. Ce nom sera donné au réseau généré.

- construire la liste des paramètres avec leurs types, chaque paramètre d'identificateur p ayant un nom absolu nat.p . Cette liste de paramètres sera utilisée pour la construction du réseau représentant une tâche composée ayant cette tâche (ou une tâche de ce type) pour composante.

- construire le vecteur de X de l'interprétation qui est formé
  * des paramètres (non échangés) de la tâche (considérés comme des variables, bien qu'en fait ils ne puissent être substitués que par des constantes),
  * des variables déclarées dans la partie déclarative de la tâche (en transformant leurs noms en noms absolus).

- construire les vecteurs de variables échangées Xin et Xout de l'interprétation à partir de la liste des paramètres

échangés de la tâche.

Les variables et paramètres de la tâche de types non-spécifiés sont marqués comme devant être éliminés (y compris les tableaux d'éléments non spécifiés, et les champs non spécifiés des structures). Les variables internes de la tâche non initialisées, mais d'un type spécifié, ne sont pas éliminées, car leur type permet de définir l'ensemble de leurs valeurs initiales possibles.

5.3.2 - Tâche non-spécifiée simple ou type de tâche non-spécifiée

Pour la traduction des tâches non-spécifiées simples ou des types de tâches non-spécifiées, on emploiera la méta-grammaire d'axiome une brique fermée de contenu le symbole non-terminal <déclaration de tâche non spécifiée>.

Règle de grammaire :

<déclaration de tâche non spécifiée> ::=
TASK [TYPE] <en-tête de tâche> ;

Méta-règle associée : figure 5.

L'en-tête est analysée afin de :

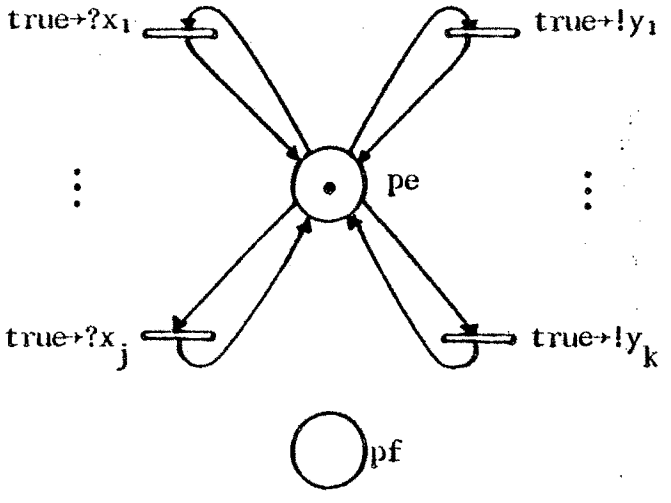
- définir le nom absolu de la tâche (ou du type de tâche), qui sera donné au réseau généré,

- construire la liste des paramètres avec leurs noms absolus et leurs types,

- construire les vecteurs de variables échangées Xin et Xout à partir de la liste des paramètres échangés.

Le vecteur de variables X de l'interprétation de ce réseau est formé des paramètres (non échangés) de la tâche. L'ensemble de ses étiquettes est vide.

TASK [TYPE] <en-tête de tâche> ; ::=



(pour tout x_1, ... x_j de Xin et tout y_1, ... y_k de Xout)

Figure 5

5.3.3 - Instanciation d'une tâche typée élémentaire ou non-spécifiée

Règle de grammaire :

```
<déclaration de tâches typées> ::=
          <liste d'identificateurs> :
          [ ARRAY ( <index spécifié>
          [ , <index spécifié> ]* ) OF ]
          <nom de type de tâche> ;
```

Aucune action de génération proprement dite n'est à effectuer, un réseau représentant le type de tâche ayant déjà été généré lors de la compilation de la déclaration du type de tâche (avec un nom absolu dépendant du contexte de sa déclaration).

La déclaration est analysée de façon à définir pour chaque tâche déclarée son nom absolu

nat = nae.n

où nae est le nom absolu de l'environnement de la déclaration

et n le nom déclaré,
ou nat = nae.n(i1, i2, ... , ik)
pour chaque k-uplet de valeurs d'indices possible si c'est un tableau de tâches à k dimensions qui est déclaré.

L'utilisation des règles de visibilité permet de retrouver la déclaration du type de tâche et son nom absolu NAT . Le réseau représentant la tâche de nom nat sera obtenue par recopie de celui représentant le type NAT en remplaçant dans l'interprétation les noms de variables (internes ou échangées) de la forme NAT.x par nat.x . La même opération est effectuée sur les vecteurs de variables X, Xin et Xout. Cette opération est appelée "instanciation" de la tâche nat à partir du type NAT. Elle est effectuée individuellement pour chaque tâche déclarée (c.a.d. pour chaque élément dans le cas d'un tableau de tâches).

5.3.4 - Référence aux tâches externes

La syntaxe CESAR ne permet pas de faire référence à une tâche compilée séparément, mais seulement à un type de tâche compilé séparément.

Aucune action de génération n'est à effectuer, les types de tâches ayant précédemment été compilés (au niveau train de compilation, donc leur nom absolu est égal à leur nom déclaré).

Une instanciation sera effectuée pour chaque tâche ultérieurement déclarée d'un tel type de la même manière que précédemment.

6 - INTRODUCTION DES TACHES COMPOSEES

Chaque tâche composée sera représentée par un GP interprété (communicant si la tâche est munie d'une partie formelle qui comporte des paramètres échangés) obtenu à partir des GP (ou GE) communicants représentant les tâches qui la composent.

Une tâche composée peut être déclarée :
- soit comme une tâche simple,
- soit comme une tâche d'un type.

Dans le deuxième cas, il est nécessaire d'avoir au préalable compilé la déclaration du type de tâche, ce qui aura permis de générer un réseau le représentant. On effectuera ensuite une instanciation (cf 5.3.3 p. III-62) de la tâche en utilisant le réseau représentant le type de tâche, comme pour une tâche élémentaire ou non spécifiée.

Après compilation de la partie déclarative de la tâche composée tous les réseaux représentant les tâches composantes ont été générés. Le GP représentant la tâche composée est alors obtenu grâce aux opérations suivantes :

1) Substitution des paramètres formels des tâches composantes par les paramètres effectifs fournis dans le corps de la tâche composée.

Ces paramètres effectifs peuvent être :
- pour les paramètres non échangés : des expressions dépendant uniquement de constantes ou de paramètres (non échangés) de la tâche composée,
- pour les paramètres échangés : des variables échangées déclarées dans la tâche composée, ou des paramètres échangés par celle-ci. (Les éléments d'un tableau de variables échangées sont traitées comme autant de variables échangées distinctes, cf 6.1 p. III-65).

Cette substitution est effectuée dans l'interprétation des GP représentant les tâches composantes et dans le vecteur de variables X (pour les paramètres non échangés, avec suppression de ceux-ci du vecteur X lorsqu'ils sont substitués par des constantes) et dans les vecteurs Xin et Xout (pour les paramètres échangés).

2) Construction du GP représentant la tâche composée d'une façon incrémentale, par utilisations successives de l'opération binaire de composition entre GP définie en 6.2 p. III-67.

Si l'opération de composition est notée ● et si les tâches composantes sont représentées par les GP P1, P2, ... Pn, la tâche composée est donc représentée par le GP

(... ((P1 ● P2) ● P3) ● ...) ● Pn .

Cette opération étant commutative et associative, l'ordre n'importe pas.

3) Encapsulation du GP obtenu au moyen de l'opération définie en 6.3 p. III-82. Cette opération a pour but
- de supprimer les possibilités de communication éventuelle introduites pour assurer l'associativité de l'opération de composition,
- de rendre invisibles tous les échanges des variables échangées déclarées dans la tâche composée, de façon à ne conserver que ses paramètres échangés dans les vecteurs Xin et Xout du GP communicant la représentant.

L'en-tête et la partie déclarative de la tâche composée sont analysés afin de :

- définir le nom absolu de la tâche (ou du type de tâche)
nat = nae.n
si nae est le nom absolu de l'environnement où est déclaré la tâche (ou le type de tâche) avec le nom n. Ce nom sera donné au réseau construit.

- construire la liste des paramètres avec leurs types, chaque paramètre d'identificateur p ayant un nom absolu nat.p . Cette liste de paramètres sera utilisée pour la construction du réseau représentant une tâche composée ayant pour composante cette tâche (ou une tâche de ce type).

- construire les vecteurs de variables échangées Xin et Xout de l'interprétation à partir de la liste des paramètres échangés de la tâche.

Le corps de la tâche composée est analysé afin de construire pour chaque tâche invoquée la liste de ses paramètres effectifs (avec leurs noms absolus) et leur mode de réception (ANY ou ALL) dans le cas des variables ou paramètres échangés. (Si cette variable échangée n'a qu'une tâche la recevant, le mode d'échange est indifférent).

6.1 - Remarque importante : les tableaux de variables échangées

Chaque élément d'un tableau de variables ou paramètres échangés est traité comme une variable ou paramètre échangé distinct.

Si dans les instructions d'échanges, les indices utilisés pour désigner un tel élément de tableau sont tous statiquement évaluables, l'élément de tableau effectivement échangé est connu lors de la traduction. Dans le cas contraire, un problème se pose pour la composition qui nécessite de connaître exactement les variables échangées entre les tâches à composer.

Cette situation peut se présenter :

- soit parce que dans le corps d'une tâche élémentaire figure l'échange d'un élément d'un tableau de paramètres échangés, sans que tous ses indices soient statiquement évaluables ;
- soit parce que lors de la substitution des paramètres formels par les paramètres effectifs (soit lors d'un appel de procédure, soit lors de l'utilisation d'une tâche à l'intérieur d'une tâche composée) un paramètre échangé simple est substitué par un élément d'un tableau de variables ou paramètres échangés dont tous les indices ne sont pas statiquement évaluables.

Préalablement à l'opération de composition, mais après la substitution des paramètres, on effectue la transformation du réseau suivante, qui permet de résoudre ce problème :

* chaque transition comportant l'échange d'un élément de tableau dont tous les indices ne sont pas statiquement évaluables, est dupliquée autant de fois qu'il y a de combinaisons possibles de valeurs pour les indices non statiquement évaluables (en fonction du type des index), en substituant chaque fois les indices par leurs valeurs. Par exemple, si v est un tableau de variables échangées à une dimension, dont le type de l'index est 1..n, la transition t ayant pour interprétation

c -> !v(exp), a

(où exp est une expression quelconque à résultat entier, non statiquement évaluable), est remplacée par n transitions t1, ..., tn ayant respectivement pour interprétation :

c.(exp = 1) -> !v(1), a
c.(exp = 2) -> !v(2), a
...
c.(exp = n) -> !v(n), a.

Les transitions t1 à tn sont telles que :

>t1 = >t2 = ... = >tn = >t

et t1> = t2> = ... = tn> = t>.

Leurs ensembles d'étiquettes sont tous identiques à celui de t. Le reste de l'interprétation n'est pas modifié.

Après composition, l'opération d'encapsulation fera disparaître celles de ces transitions qui n'ont pas été exploitées, car correspondant à des échanges jamais réalisables.

Notons enfin qu'un indice non statiquement évaluable lors de la composition peut le devenir plus tard, dans le cas où il dépend d'un paramètre par nom de la tâche composée, lorsque celui-ci est substitué ultérieurement par une expression statiquement évaluable. Lors de cette substitution, certaines des transitions correspondant à des rendez-vous peuvent voir leurs conditions devenir identiquement fausses. Cela peut d'ailleurs se produire

chaque fois qu'un paramètre non échangé d'une tâche ou d'une procédure est substitué par une expression statiquement évaluable. Afin de simplifier le réseau, on complètera donc toute substitution de paramètres par un algorithme d'évaluation des conditions où une substitution est faite, et d'élimination des transitions dont les conditions sont identiquement fausses.

6.2 - Opération de composition

Soit deux GP communicants P1 et P2 représentant deux tâches à composer, et soit :
- Xi le vecteur des variables (internes) de Pi,
- Xout(i) l'ensemble des variables émises par Pi (ou le vecteur de variables Xout de Pi),
- Xin1(i) l'ensemble des variables échangées en mode ANY reçues par Pi,
- Xin*(i) l'ensemble des variables échangées en mode ALL reçues par Pi.

Xin1(i) et Xin*(i) forment une partition du vecteur Xin de Pi.

L'opération de composition appliquée à P1 et P2 fournit un GP communicant P ayant pour vecteurs de variables X, Xin et Xout, les ensembles X1 u X2, Xin(1) u Xin(2), Xout(1) u Xout(2). L'ensemble des places est l'union des ensembles des places de P1 et P2. L'opération de composition agit uniquement sur les transitions comportant l'échange d'une variable échangée entre les deux GP.

On a nécessairement :

pour tout i, j : Xin1(i) n Xin*(j) = ∅

car le mode de réception d'une variable échangée ne dépend pas des tâches qui la reçoivent (à l'intérieur d'une même tâche composée, le mode de réception est toujours le même pour une variable donnée).

Par contre, chacune des intersections suivantes peut ne pas être vide :
- Xout(1) n Xout(2) : les deux tâches émettent la même variable ;
- Xin1(1) n Xin1(2) : les deux tâches reçoivent la même variable, échangée en mode ANY ;
- Xin*(1) n Xin*(2) : les deux tâches reçoivent la même variable, échangée en mode ALL ;
- Xout(i) n Xin1(j) (i ≠ j) : une des tâches émet une variable que reçoit l'autre, en mode ANY ;
- Xout(i) n Xin*(j) (i ≠ j) : une des tâches émet une variable que reçoit l'autre, en mode ALL.

Pour chaque variable de chacune de ces intersections, un traitement (différent suivant les cas) est réalisé afin d'effectuer la composition des deux tâches, d'une façon qui garantisse la possibilité ultérieure de pouvoir composer le GP obtenu avec un autre de façon incrémentale.

Le tableau suivant récapitule les différents cas de rendez-vous :

P1	P2	Règle
Xout	Xout	1
Xin1	Xin1	2
Xin*	Xin*	3
Xout	Xin1	4
Xout	Xin*	5

Seuls ces cinq cas sont envisagés car P1 et P2 jouent un rôle symétrique.

L'ordre dans lequel sont considérées les variables échangées entre les tâches à composer est indifférent, car il n'y a pas d'interactions entre leurs échanges (tous les échanges étant simples).

L'opération de composition est basée sur une opération de fusion entre deux transitions, qui a pour effet de remplacer deux transitions t1 et t2 par une transition t telle que

$$>t = >t1 \cup >t2 \quad \text{et} \quad t> = t1> \cup t2>$$

dont l'interprétation est obtenue à partir de celle de t1 et t2 (d'une façon qui dépend du cas de rendez-vous envisagé). Dans la suite, les opérations de fusion entre transitions sont représentées graphiquement, en supposant

$$|>t1| = |t1>| = |>t2| = |t2>| = 1$$

uniquement pour la simplicité du dessin.
L'ensemble des étiquettes associées à t est l'union des étiquettes associées à t1 et t2.

Le réseau P = P1 ⊕ P2 est obtenu comme suit :
- son ensemble de places est l'union des ensembles des places de P1 et P2,
- son ensemble de transitions est formé
* des transitions de P1 ou P2 dont l'interprétation ne comporte pas d'échange, ou comportant l'échange d'une variable échangée par l'un des réseaux mais pas par l'autre,
* des transitions obtenues au moyen des règles de composition (qui sont exposées ci-dessous).

Pour l'opération d'encapsulation, il est nécessaire de posséder un moyen d'identifier certaines transitions créées par la composition ; nous supposerons donc que nous disposons d'une technique de marquage des transitions. Sur

les figures, les transitions marquées seront indiquées par une astérisque.

Notations pour la suite :

On note :

- t1 (resp. t2) une transition de P1 (resp. P2),
- x1 (resp. x2) une variable (ou un ensemble de variables) interne(s) de P1 (resp. P2),
- c1 (resp. c2) une condition sur les variables internes de P1 (resp. P2),
- e1 (resp. e2) une expression sur les variables internes de P1 (resp. P2),
- a1 (resp. a2) une affectation simultanée sur les variables internes de P1 (resp. P2), ne modifiant pas les variables de x1 (resp. x2),
- t : xxx le fait que la transition t est munie de l'interprétation xxx,
- t *: xxx le fait que la transition t est munie de l'interprétation xxx et est marquée.

Sur les figures illustrant les règles de composition on n'indiquera pas l'interprétation complète des transitions, mais on signalera les échanges qu'elles comportent.

6.2.1 - Règle 1 : Variable émise par les deux tâches

Dans ce cas, les tâches consommatrices (composées ultérieurement) effectueront un rendez-vous soit avec P1, soit avec P2, sans qu'il y ait interaction entre P1 et P2.

Les transitions comportant l'émission de ces variables ne sont donc pas affectées par la composition de P1 et P2. Elles sont donc introduites telles quelles dans le réseau composé.

6.2.2 - Règle 2 : Variable reçue par les deux tâches en mode ANY

Lorsque cette variable sera émise par une tâche composée ultérieurement, un rendez-vous aura lieu entre celle-ci et P1 ou P2, sans qu'il y ait interaction entre P1 et P2.

Les transitions comportant la réception de ces variables ne sont donc pas affectées par l'opération de composition. Elles sont donc introduites telles quelles dans le réseau composé.

6.2.3 - Règle 3 : Variable reçue par les deux tâches en mode ALL

Lorsque cette variable (notons-la x) sera émise (par une tâche composée ultérieurement), un rendez-vous aura lieu entre elle et les tâches P1 et P2 simultanément (ainsi que toutes les tâches recevant x composées ultérieurement).

Soit T?x(1) (resp. T?x(2)) l'ensemble des transitions de P1 (resp. P2) dont l'interprétation comporte une réception de x. Un rendez-vous pour l'échange de x est effectué avec une transition quelconque de T?x(1) et simultanément une transition quelconque de T?x(2). Donc l'ensemble des rendez-vous possibles impliquant P1 et P2 est décrit par l'ensemble des couples de T?x(1) X T?x(2). L'algorithme de composition (relatif à la variable échangée x) est alors le suivant :

- pour chaque couple (t1, t2) de T?x(1) X T?x(2), construire par fusion de t1 et t2 la transition t de P telle que :

```
t1 : c1 -> x1 := ?x, a1
t2 : c2 -> x2 := ?x, a2
t  : c1.c2 -> x1, x2  := ?x, a1, a2.
```

Cette opération (si |T?x(1)| = |T?x(2)| = 1) est exprimée par la figure 1.

La transition construite comporte la réception de x. Ceci permettra d'effectuer ultérieurement une composition du GP obtenu avec un autre GP représentant une tâche émettant ou recevant x.

Cette opération est effectuée pour chaque variable échangée x de Xin*(1) n Xin*(2) (l'ordre est indifférent).

6.2.4 - Règle 4 : Variable émise par une tâche et reçue par l'autre en mode ANY

Supposons que la variable x soit émise par P1 et reçue en mode ANY par P2.

Dans ce cas, un rendez-vous peut avoir lieu entre P1 et P2 pour l'échange de x, mais P1 peut également effectuer un rendez-vous avec une autre tâche recevant x (composée ultérieurement), ou P2 avec une autre tâche émettant x.

Soit T!x(1) (resp. T?x(2)) l'ensemble des transitions de P1 (resp. P2) dont l'interprétation comporte une émission (resp. une réception) de x. L'ensemble des rendez-vous possibles de

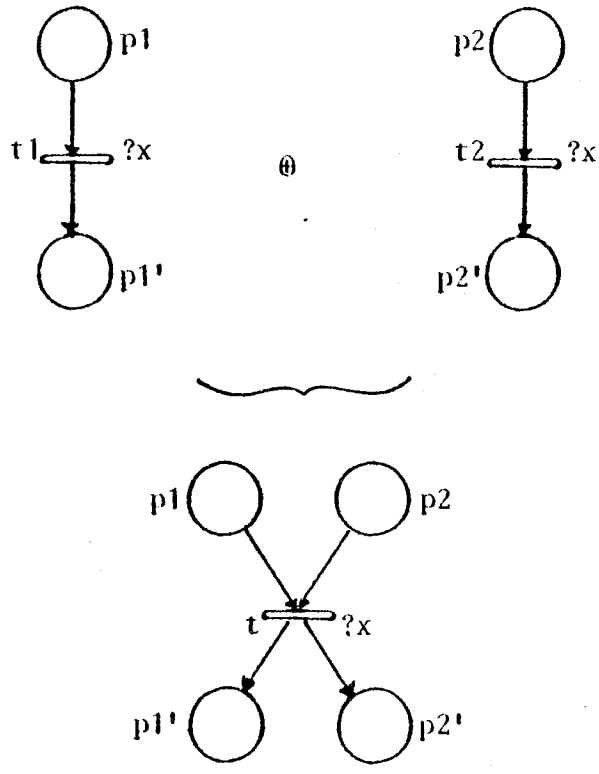


Figure 1 - Règle 3

P1 est décrit par l'ensemble T!x(1) X (T?x(2) u {e}), les couples (t1, t2) de T!x(1) X T?x(2) décrivant les rendez-vous entre P1 et P2, et les couples (t1, e) représentant les rendez-vous potentiels de P1 avec d'autres tâches que P2. De même, les rendez-vous possibles de P2 sont décrits par l'ensemble (T!x(1) u {e}) X T?x(2), les couples (e, t2) représentant les rendez-vous potentiels de P2 avec d'autres tâches que P1. L'algorithme de composition (relatif à la variable échangée x) est alors le suivant :

- pour chaque couple (t1, t2) de T!x(1) X T?x(2), construire par fusion de t1 et t2 la transition t de P telle que :

```
t1 : c1 -> x1, !x := e1, a1
t2 : c2 -> x2 := ?x, a2
t  : c1.c2 -> x1, x2  := e1, a1, a2.
```

- rajouter à P les transitions de T!x(1) et T?x(2) telles quelles, pour représenter les communications potentielles de P1 ou P2 avec d'autres partenaires.

Cette opération (si |T!x(1)| = |T?x(2)| = 1) est exprimée par la figure 2.

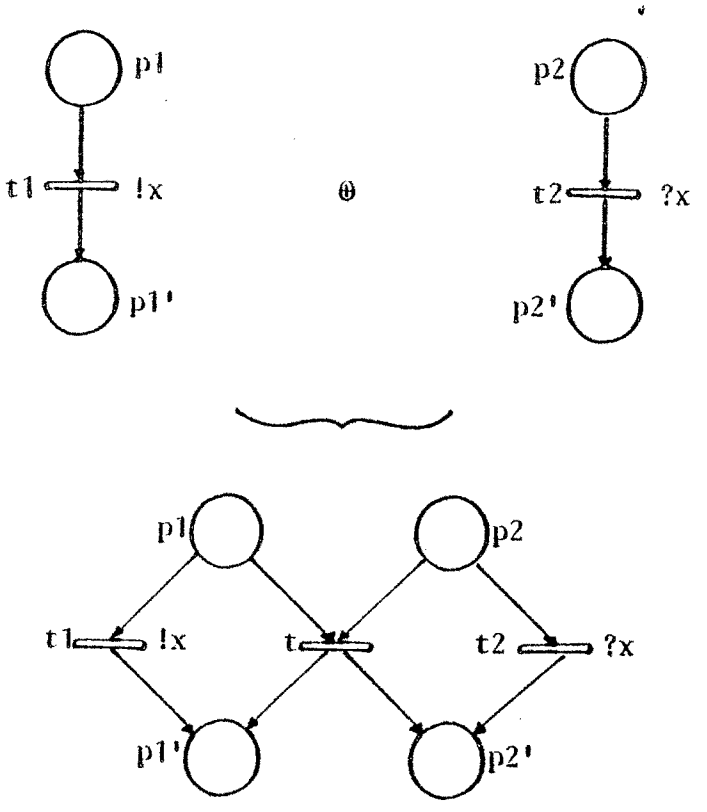


Figure 2 - Règle 4

Remarquons que cette opération laisse un exemplaire non fusionné de t1 et un de t2 (comportant respectivement une émission ou une réception de x) alors que les transitions obtenues par fusion ne comportent plus d'échanges de x. Ceci permettra d'effectuer ultérieurement une composition du GP obtenu, avec un GP représentant une tâche recevant ou emettant x.

Cette opération est répété pour chaque variable x de Xout(1) n Xin1(2). On procède de même (symétriquement) pour les variables de Xin1(1) n Xout(2).

6.2.5 - Règle 5 : Variable émise par une tâche et reçue par l'autre en mode ALL

Supposons que la variable x soit produite par P1 et consommée en mode ALL par P2.

Dans ce cas, un rendez-vous ne peut avoir lieu qu'entre P1 et l'ensemble des tâches recevant x (dont P2), ou entre une tâche émettant x autre que P1 et cet ensemble.

Soit T!x(1) (resp. T?x(2)) l'ensemble des transitions de P1 (resp. P2) dont l'interprétation comporte une émission (resp. une réception) de x. L'ensemble des rendez-vous possibles de P1 est décrit par l'ensemble T!x(1) X T?x(2) (puisque P2 participe toujours au rendez-vous). Par contre, les rendez-vous possibles de P2 (et l'ensemble des autres tâches recevant x) sont décrits par l'ensemble (T!x(1) u {e}) X T?x(2), les couples (e, t2) représentant les rendez-vous potentiels de P2 (et l'ensemble des autres tâches recevant x) avec d'autres tâches que P1 (composées ultérieurement). L'algorithme de composition (relatif à la variable échangée x) est le suivant :

- pour chaque couple (t1, t2) de T!x(1) X T?x(2), construire par fusion de t1 et t2 la transition t telle que :

  t1 : c1 -> x1, !x := e1, a1
  t2 : c2 -> x2 := ?x, a2
  t *: c1.c2 -> x1, x2, !x := e1, a1, a2.

Cette transition sera marquée (pour signifier qu'elle représente un rendez-vous, en mode ALL, bien que comportant l'émission de x).

- rajouter à P les transitions de T?x(2) telles quelles, pour représenter les communications potentielles de P2 avec d'autres partenaires.

Cette opération (si |T!x(1)| = |T?x(2)| = 1) est exprimée par la figure 3.

Remarquons que cette opération laisse un exemplaire non fusionné de t2 comportant la réception de x alors que les transitions obtenues par fusion comportent son émission. Ceci permettra d'effectuer ultérieurement une composition du GP obtenu, avec un GP représentant une tâche recevant ou émettant x.

Cette opération est répété pour chaque variable x de Xout(1) n Xin*(2). On procède de même (symétriquement) pour les variables de Xin*(1) n Xout(2).

6.2.6 - Variable émise et reçue par un même GP

L'application des règles 4 et 5 définies plus haut produit un GP dont les vecteurs Xin et Xout ne sont pas disjoints (alors qu'ils le sont pour les GE représentant les tâches élémentaires ou non spécifiées).

Lorsque ce GP est lui-même composé avec un autre, plusieurs règles deviennent applicables pour une même variable, comme l'exprime le tableau suivant :

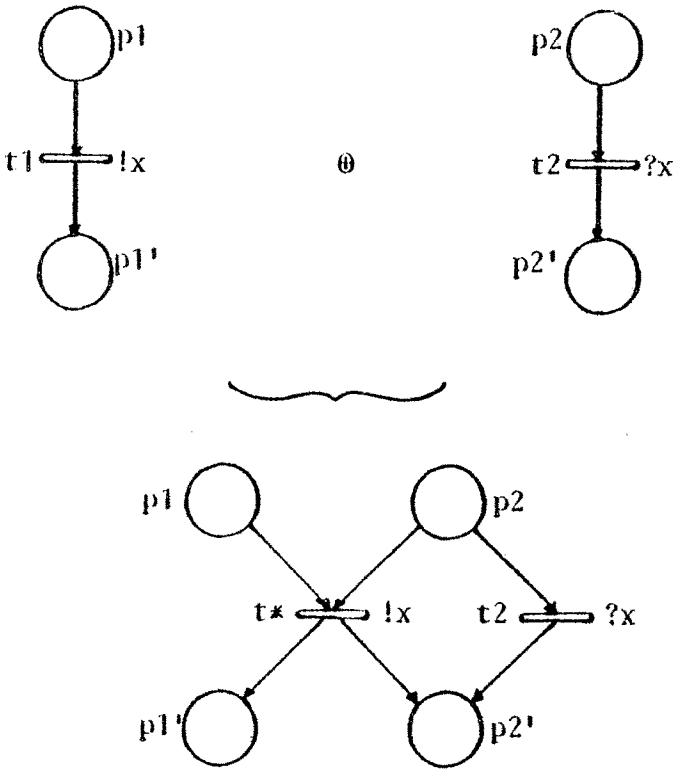


Figure 3 - Règle 5

cas	P1	P2	Règles applicables
1	Xout n Xin1	Xout	1, 4
2	Xout n Xin1	Xin1	2, 4
3	Xout n Xin1	Xout n Xin1	1, 2, 4
4	Xout n Xin*	Xout	1, 5
5	Xout n Xin*	Xin*	3, 5
6	Xout n Xin*	Xout n Xin*	1, 3, 5

(Nous ne considérons que ces six cas, car les rôles de P1 et P2 sont symétriques).

En fait, ces règles ne sont pas applicables telles qu'elles ont été définies précédemment, et nous sommes conduits à définir six règles supplémentaires, notées 1' à 6', correspondant à ces six cas, illustrées par les figures 4 à 9. (Les notations sont similaires à celles employées plus haut ; pour toute lettre mij, l'indice i valant 1 ou 2 identifie le GP P1 ou P2).

Règle_1' : figure 4 ;

avec :
  t11 : c11 -> x11, !x := e11, a11
  t12 : c12-> x12 := ?x, a12
  t2 : c2 -> x2, !x := e2, a2
  t   : c12.c2 -> x12, x2 := e2, a12, a2

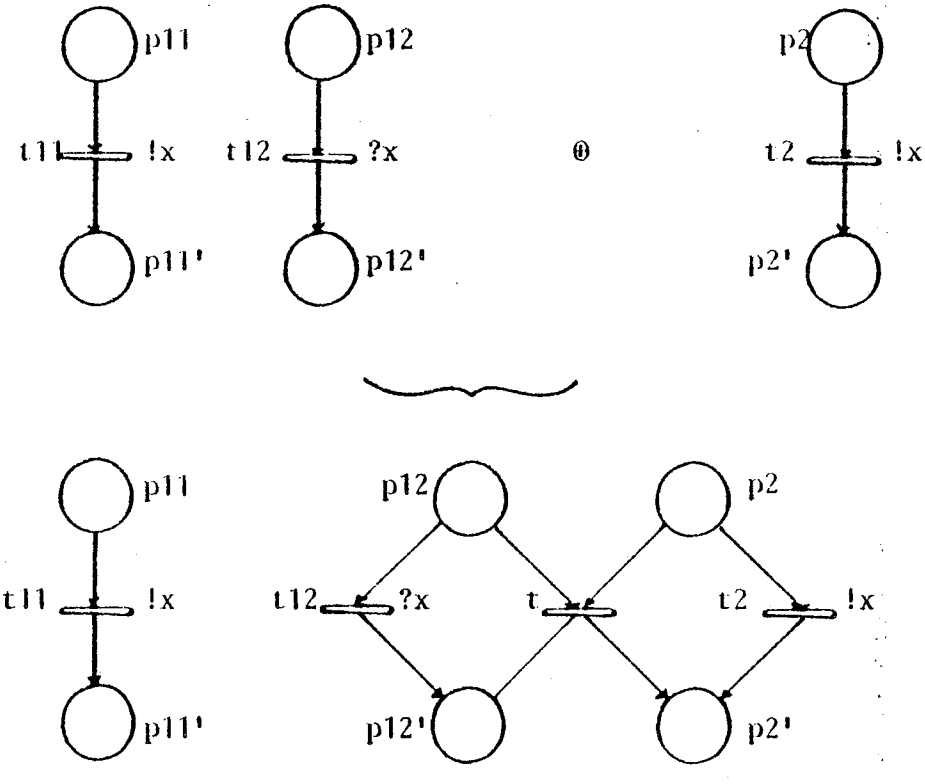


Figure 4 - Règle 1'

Règle_2' : figure 5 ;

avec :
  t11 : c11 -> x11 := ?x, a11
  t12 : c12 -> x12, !x := e12, a12
  t2 : c2 -> x2 := ?x, a2
  t   : c12.c2 -> x12, x2 := e12, a12, a2

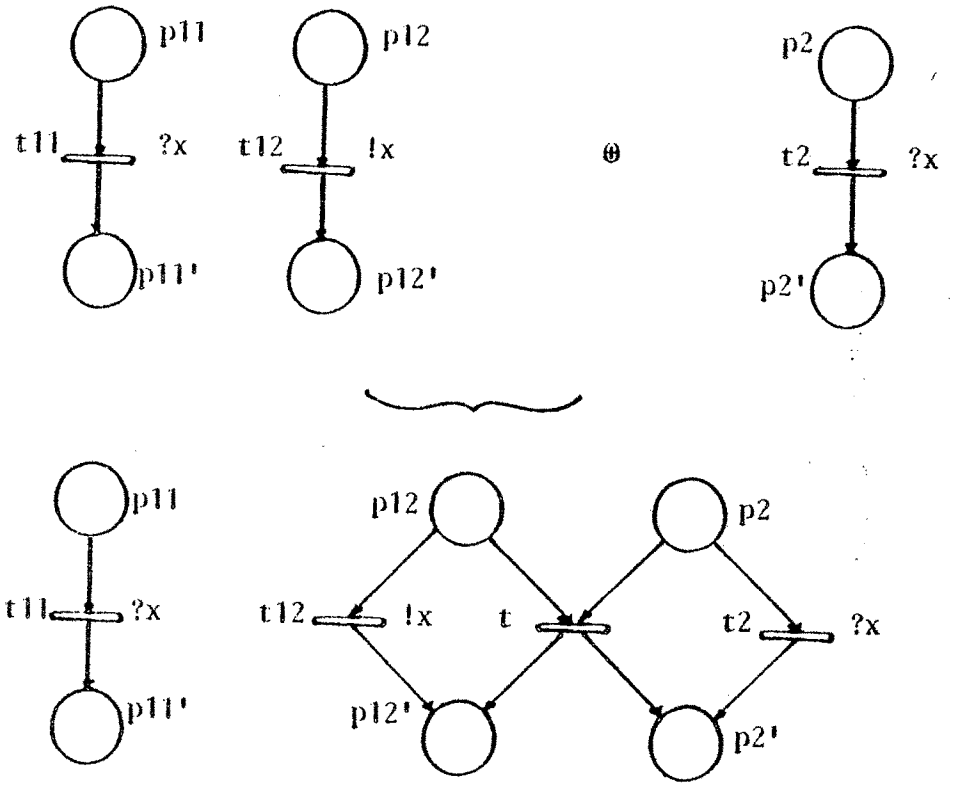


Figure 5 - Règle 2'

Règle_3' : figure 6 ;

avec :
  t11 : c11 -> x11, !x := e11, a11
  t12 : c12 -> x12 := ?x, a12
  t21 : c21 -> x21 := ?x, a21
  t22 : c22 -> x22, !x := e22, a22
  t   : c11.c21 -> x11, x21 := e11, a11, a21
  t' : c12.c22 -> x12, x22 := e22, a12, a22

Règle_4' : figure 7 ;

avec :
  t11 : c11 -> x11, !x := e11, a11
  t12 : c12-> x12 := ?x, a12
  t2 : c2 -> x2, !x := e2, a2
  t   : c12.c2 -> x12, x2, !x := e2, a12, a2

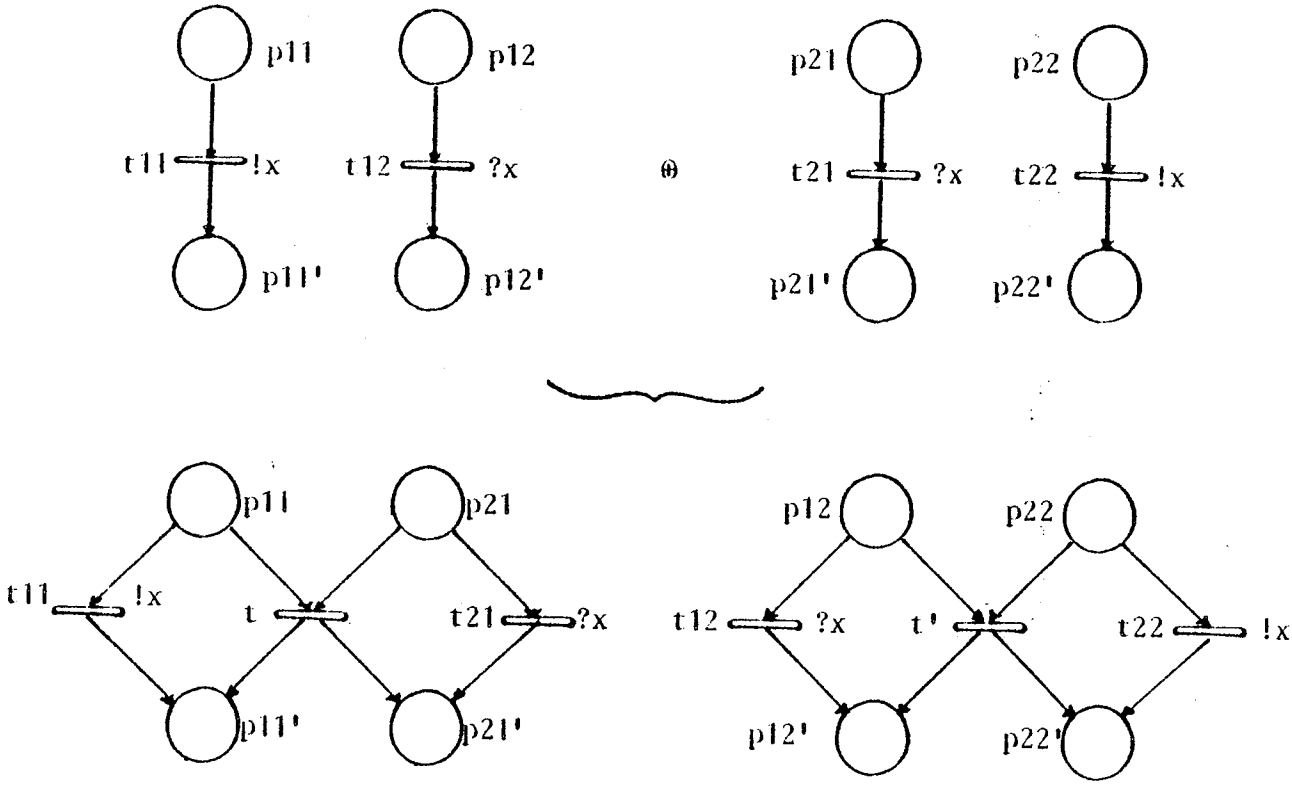


Figure 6 - Règle 3'

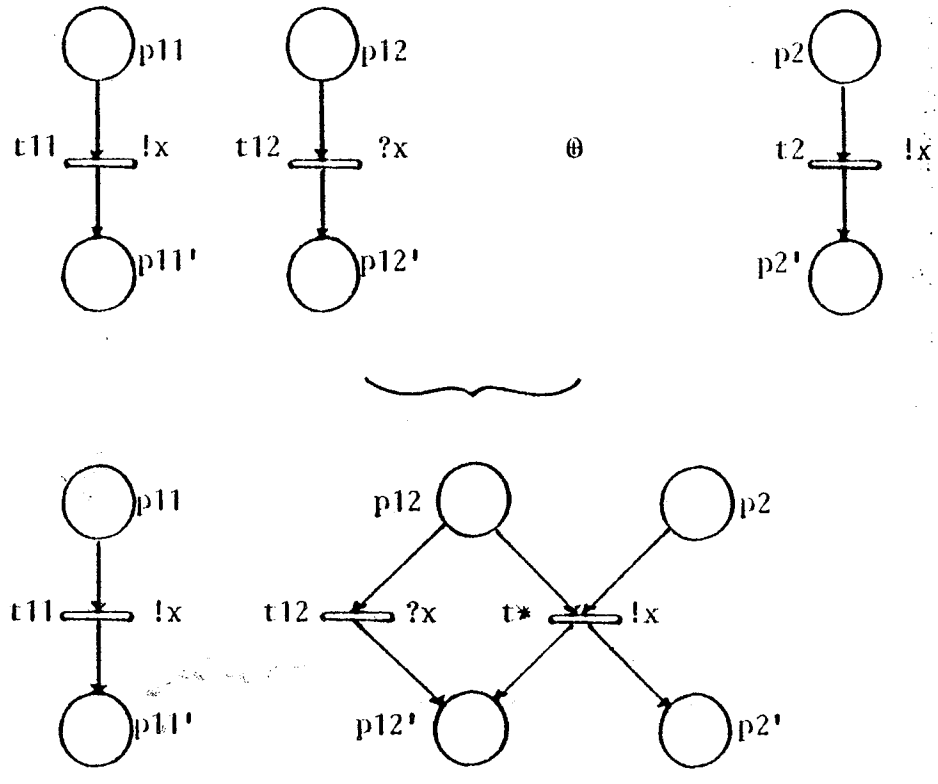


Figure 7 - Règle 4'

Règle_5' : figure 8 ;

avec :
  t11 : c11 -> x11, !x := e11, a11
  t12 : c12 -> x12 := ?x, a12
  t2 : c2 -> x2 := ?x, a2
  t *: c11.c2 -> x11, x2, !x := e11, a11, a2
  t' : c12.c2 -> x12, x2 := ?x, a12, a2

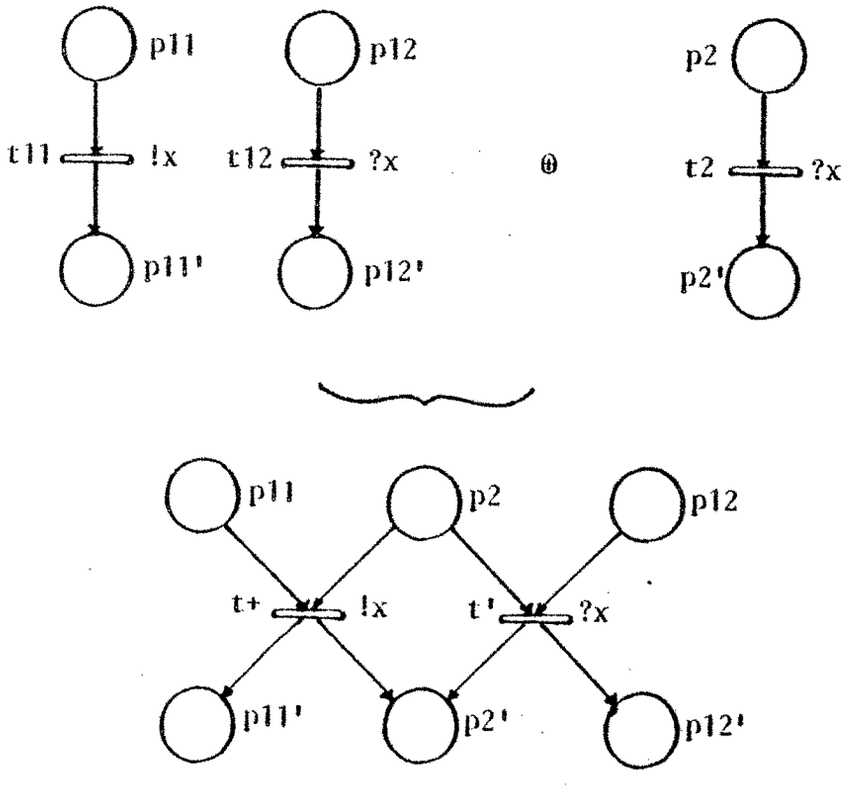


Figure 8 - Règle 5'

Règle_6' : figure 9 ;

avec :
  t11 : c11 -> x11, !x := e11, a11
  t12 : c12 -> x12 := ?x, a12
  t21 : c21 -> x21 := ?x, a21
  t22 : c22 -> x22, !x := e22, a22
  t *: c11.c21 -> x11, x21, !x := e11, a11, a21
  t' *: c12.c22 -> x12, x22, !x := e22, a12, a22
  t'' : c21.c12 -> x21, x12 := ?x, a21, a12

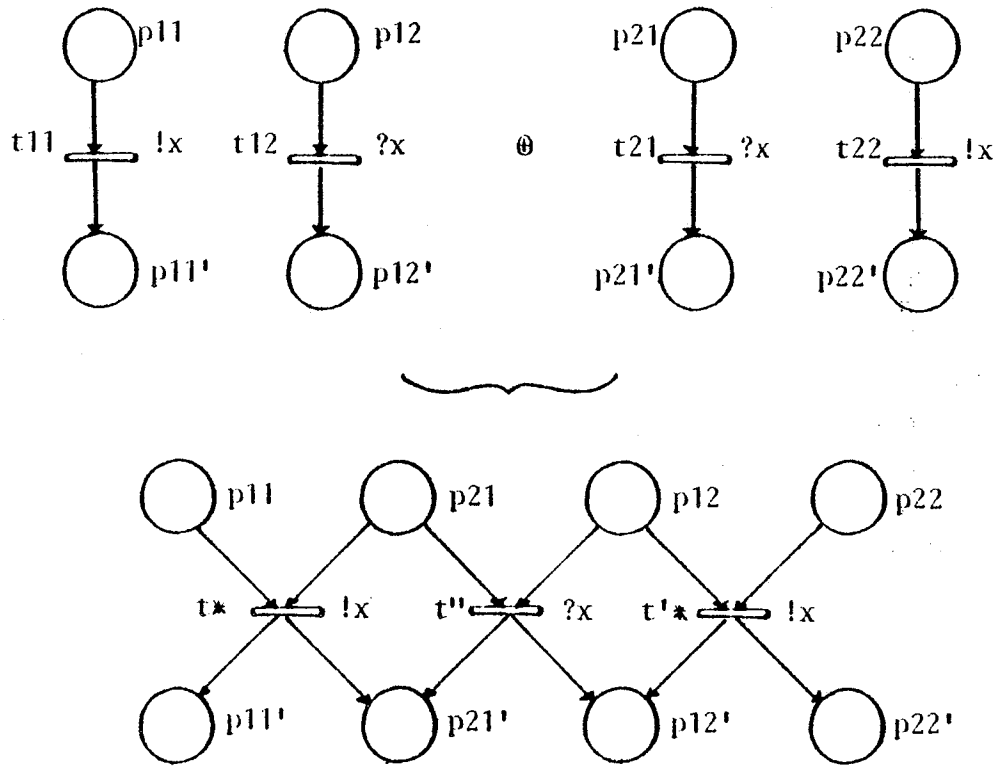


Figure 9 - Règle 6'

La conclusion de cette étude cas par cas est que les onze règles envisagées sont des cas particuliers des deux règles suivantes :

REGLE_A_:_Variable_émise_et_reçue_en_mode_ANY_par_les_deux_réseaux

-A1- pour chaque couple (t1, t2) de T!x(1) X T?x(2), construire une transition t telle que :

t1 : c1 -> x1, !x := e1, a1
t2 : c2 -> x2 := ?x, a2
t : c1.c2 -> x1, x2 := e1, a1, a2.

-A2- procéder de même (symétriquement) pour chaque couple de T?x(1) X T!x(2).

-A3- rajouter telles quelles les transitions de T!x(1), T?x(1), T!x(2), T?x(2) pour représenter les échanges potentiels avec d'autres partenaires.

REGLE_B_:_Variable_émise_et_reçue_par_les_deux_réseaux en_mode_ALL

-B1- pour chaque couple (t1, t2) de T!x(1) X T?x(2), construire une transition t telle que :
t1 : c1 -> x1, !x := e1, a1
t2 : c2 -> x2 := ?x, a2
t *: c1.c2 -> x1, x2, !x := e1, a1, a2.

-B2- procéder de même (symétriquement) pour chaque couple de T?x(1) X T!x(2).

SI T?x(1) et T?x(2) sont tous les deux non vides :
-B3- pour chaque couple (t1, t2) de T?x(1) X T?x(2), construire une transition t telle que :
t1 : c1 -> x1 := ?x, a1
t2 : c2 -> x2 := ?x, a2
t : c1.c2 -> x1, x2 := ?x, a1, a2.

SINON SI T?x(1) est non vide et T?x(2) vide (ou l'inverse) :
-B4- rajouter les transitions de T?x(1) et celles de T!x(1) telles quelles (ou celles de T?x(2) et T!x(2) dans le cas inverse).

SINON (dans ce cas T?x(1) et T?x(2) sont tous deux vides) :
-B5- prendre telles quelles les transitions de T!x(1) et T!x(2).

Les règles 1, 2, 4, 1', 2', 3' sont des cas particuliers de la règle A. Les règles 1, 3, 5, 4', 5', 6' sont des cas particuliers de la règle B.

6.2.7 - Propriétés de l'opération de composition

1) L'opération de composition entre deux GP fournit un GP (puisqu'elle est réalisée à partir de la fusion de transitions qui préserve cette propriété).

2) L'opération de composition est commutative :
$$P1 \bullet P2 = P2 \bullet P1 .$$

Cette propriété découle immédiatement des rôles symétriques joués par P1 et P2 dans l'algorithme de composition.

3) L'opération de composition est associative :
$$P1 \bullet (P2 \bullet P3) = (P1 \bullet P2) \bullet P3$$
(sous la restriction suivante : cette opération ne peut être employée de façon répétée, sans opération d'encapsulation intermédiaire, qu'à l'intérieur d'une même tâche composée).

La restriction ci-dessus garantit qu'une même variable échangée l'est toujours selon le même mode. Les échanges de variables différentes n'interagissant pas, il suffit de vérifier l'associativité de chacune des règles de composition A et B pour une variable x donnée.

Règle_A : cette règle
- reproduit sans modification toutes les transitions comportant des échanges,
- introduit des transitions représentant des rendez-vous deux-à-deux qui ne comportent plus d'échanges.

L'associativité de la règle A est donc évidente.

Règle_B : cette règle
- introduit des transitions représentant des rendez-vous entre une tâche produisant x et une tâche recevant x (si cela est possible), qui comportent l'émission de x, ou reproduit telles quelles les transitions comportant une émission de x (s'il n'y a pas de tâche recevant x),
- introduit des transitions représentant des rendez-vous entre deux tâches recevant x (si cela est possible), qui comportent une réception de x, ou reproduit telles quelles les transitions comportant une réception de x (lorsqu'il n'y a qu'une tâche recevant x).

Lors d'une nouvelle application de l'opération de composition, ces transitions sont respectivement considérées comme des émissions et des réceptions, et traitées comme telles, d'où l'associativité de la règle B.

6.2.8 - Remarque : Valeurs échangées non spécifiées

Lorsque l'instruction d'émission ne comporte pas de membre droit, les valeurs émises sont supposées non spécifiées. Cette situation peut être provoquée, soit par l'existence dans le programme de description d'une instruction d'émission sans membre droit, soit par une élimination de variables non significatives qui a conduit à éliminer le membre droit qui existait dans le programme initial.

Quelle que soit la raison pour laquelle une valeur non-spécifiée est émise, cette valeur peut être reçue et affectée à une ou plusieurs variables d'autres tâches. D'après le principe de l'élimination des variables affectées par des valeurs non-spécifiées (cf 3.3 p. III-21), ces variables doivent également être éliminées. L'algorithme de composition devra donc être complété d'un nouvel appel de l'algorithme d'élimination des variables non significatives.

6.3 - Opération d'encapsulation

L'opération d'encapsulation
- a pour premier argument un GP communicant P, ayant pour vecteurs de variables échangées Xin et Xout (ce GP ayant été produit par l'opération de composition, certaines de ses transitions sont marquées, pour différencier les transitions représentant les rendez-vous en mode ALL des transitions d'émission non utilisées),
- a pour second argument un couple de vecteurs de variables échangées Xin' et Xout' respectivement inclus dans Xin et Xout (éventuellement vides),
- produit un GP communicant P' ayant pour vecteurs de variables échangées les vecteurs Xin' et Xout'.

Elle est effectuée de la façon suivante :

- toutes les transitions comportant une réception de x pour toutes les variables x de Xin - Xin' sont supprimées ;

- toutes les transitions comportant une émission de x pour toutes les variables x de Xout - Xout' et marquées ont leur interprétation modifiée comme suit :

t : c1 -> x1, !x := e1, a1
devient t : c1 -> x1 := e1, a1 ;

- toutes les transitions comportant une émission de x pour toutes les variables x de Xout - Xout' non marquées sont supprimées ;

- le marquage des transitions est supprimé (dans tous les cas).

L'encapsulation du GP P de façon à conserver les vecteurs Xin' et Xout' est notée :

[P](Xin', Xout').

Pour la construction du GP représentant une tâche composée, le GP obtenu par composition de tous les GP représentant les tâches composantes est encapsulé de façon à conserver uniquement les vecteurs Xin' et Xout' définis par la partie formelle de la tâche composée.

Propriété :

Si P est un GP ayant pour vecteurs de variables échangées Xin et Xout, si Xin' et Xout' sont deux vecteurs de variables échangées respectivement inclus dans Xin et Xout, et si Xin" et Xout" sont deux vecteurs de variables échangées respectivement inclus dans Xin' et Xout' :

[[P](Xin', Xout')](Xin", Xout") = [P](Xin", Xout").

Cette propriété découle immédiatement de l'inclusion de Xin"et Xout" à la fois dans Xin et Xout et dans Xin' et Xout'.

7 - DEUX_EXEMPLES

Nous donnons ici les réseaux générés pour les deux exemples traités au paragraphe II-10.

7.1 - Echanges_périodiques_tramés

Le réseau généré est donné par la figure 1. Remarquons que toutes les valeurs échangées étant non-spécifiées, les variables S_1 et S_2 de la tâche TRAME ont été éliminées. Le réseau ne comporte donc plus aucune interprétation (c.a.d. toutes les transitions ont l'interprétation true -> nop). Nous avons noté à proximité des transitions représentant des échanges les noms des variables échangées correspondantes.

Malgré son aspect un peu embrouillé ce réseau est extrêmement simple. Les trois processus POSITION, DERIVE(1), DERIVE(2) comportent chacun une seule place (marquée), qui n'influe aucunement sur le fonctionnement du réseau (ce sont des places implicites qui peuvent être éliminées [BER 78]). Le processus TRAME est formé d'une boucle qui séquentialise tous les échanges, et d'une transition (la seule non porteuse d'inscription) qui permet d'interrompre définitivement le fonctionnement du réseau en retirant de cette boucle la marque qui y circule.

7.2 - Le_protocole_du_bit_altercé

Nous avons donné deux versions de ce protocole. On trouvera donc ici deux réseaux, chacun correspondant à une version du programme de description.

La taille du réseau généré reflétant la longueur du programme de description, le réseau représentant la deuxième version est deux fois plus volumineux que pour la première version. Par contre, le premier réseau est interprété (car les valeurs de certaines variables, celles du type BOOLEAN, sont significatives pour le comportement du protocole) alors que le second réseau ne l'est plus après élimination des variables des types non spécifiés. Cela signifie que tout le comportement est décrit par la structure du réseau elle-même.

Réseau_généré_-_Première_version_: figure 2.

Les interprétations et les étiquettes associées aux transitions sont les suivantes (nous n'avons pas procédé à la

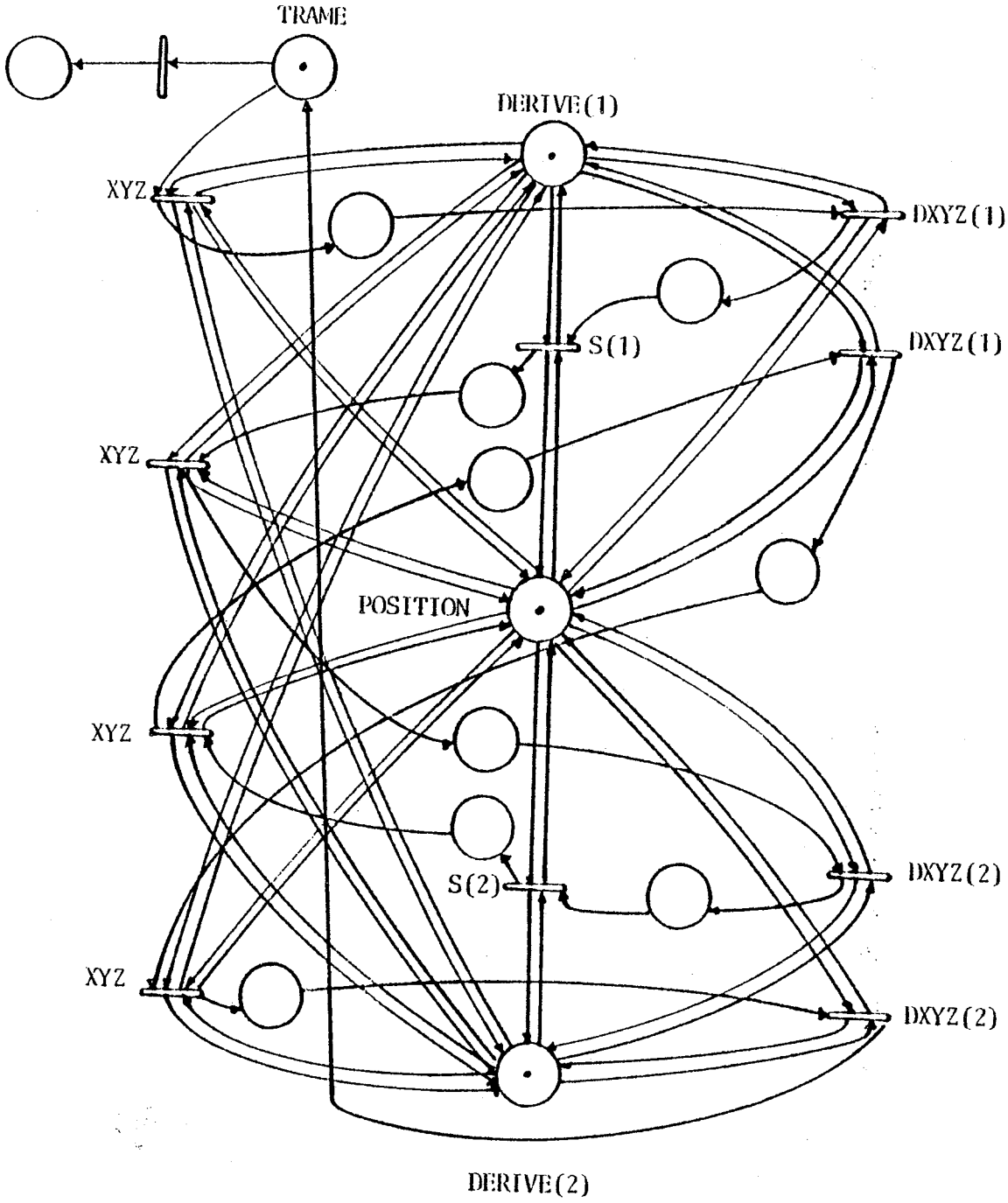

TRAME
DERIVE(1)
XYZ
DXYZ(1)
S(1)
DXYZ(1)
XYZ
POSITION
XYZ
S(2)
DXYZ(2)
XYZ
DXYZ(2)
DERIVE(2)

Figure 1

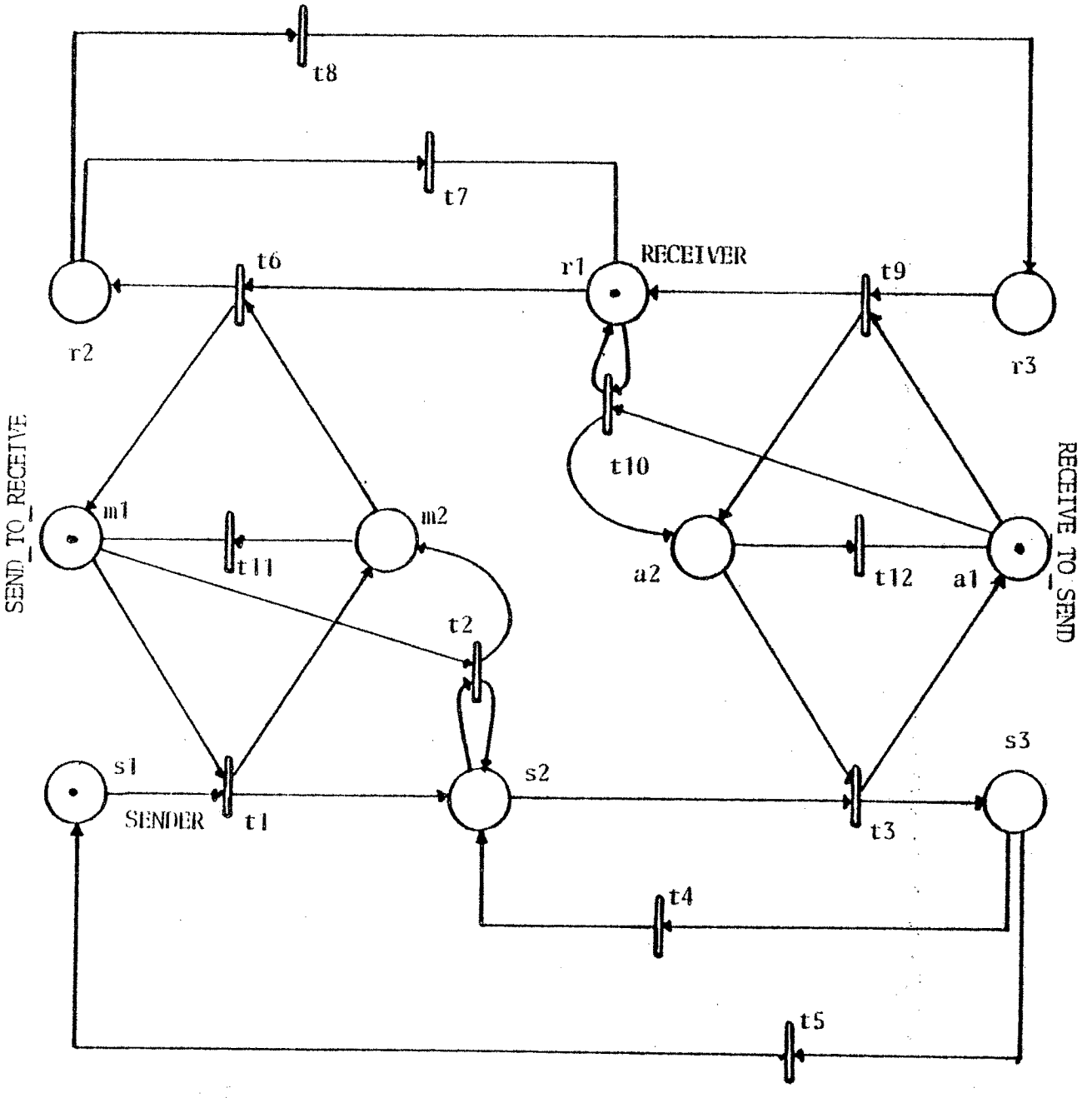


Figure 2

transformation des noms en noms absolus ; seules les variables BIT des tâches SENDER et RECEIVER sont différenciées par préfixation) :

```
t1   : "SEND" "GET"
       true -> MMM.B   := SENDER.BIT
```

```
t2  : "REPEAT" "GET"
      true -> MMM.B  := SENDER.BIT
t3  : "TRANSMIT_ACK" "RECEIVE_ACK"
      true -> RECEIVED_ACK.B  := AAA.B
t4  : "SKIP_ACK"
      RECEIVED_ACK.B /= SENDER.BIT -> nop
t5  : "ACCEPT_ACK"
      RECEIVED_ACK.B = SENDER.BIT
           -> SENDER.BIT  := NOT SENDER>BIT
t6  : "TRANSMIT" "RECEIVE"
      true -> RECEIVED_MSG.B  := MMM.B
t7  : "SKIP"
      RECEIVED_MSG.B /= RECEIVER.BIT -> nop
t8  : "ACCEPT"
      RECEIVED_MSG.B = RECEIVER.BIT -> nop
t9  : "GET_ACK" "SEND_ACK"
      true -> AAA.B  := RECEIVER.BIT,
              RECEIVER.BIT  := NOT RECEIVER.BIT
t10 : "REPEAT_ACK" "GET_ACK"
      true -> AAA.B := NOT RECEIVER.BIT
t11 : "LOOSE"
      true -> nop
t12 : "LOOSE_ACK"
      true -> nop
```

Réseau généré : Deuxième version : figure 3.

On notera que les transitions marquées A, B, C, D sont les mêmes en haut et en bas du dessin (qui forme ainsi une bande de Moebius).

Les étiquettes associées aux transitions sont les suivantes :

```
t1  : "SEND0" "GET0"
t2  : "RECEIVE_ACK0" "TRANSMIT_ACK0"
t3  : "SEND1" "GET1"
t4  : "RECEIVE_ACK1" "TRANSMIT_ACK1"
t5  : "REPEAT0" "GET0"
t6  : "REPEAT1" "GET1"
t7  : "SKIP_ACK0" "TRANSMIT_ACK0"
t8  : "SKIP_ACK1" "TRANSMIT_ACK1"
t9  : "LOOSE0"
t10 : "LOOSE1"
t11 : "RECEIVE0" "TRANSMIT0"
t12 : "SEND_ACK0" "GET_ACK0"
t13 : "RECEIVE1" "TRANSMIT1"
t14 : "SEND_ACK1" "GET_ACK1"
t15 : "REPEAT_ACK0" "GET_ACK0"
t16 : "REPEAT_ACK1" "GET_ACK1"
t17 : "SKIP1" "TRANSMIT1"
```

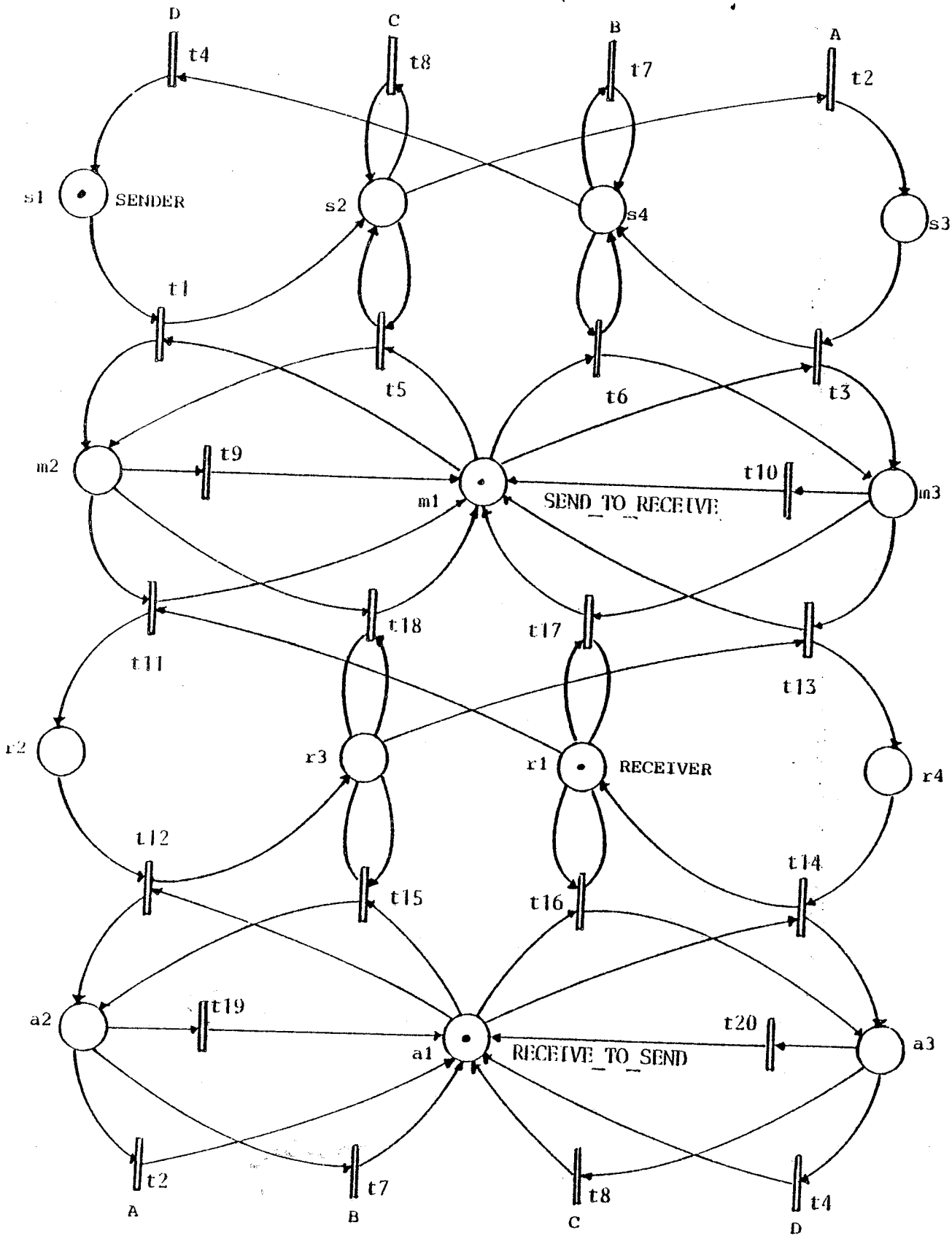

D
C
B
A
t4
t8
t7
t2
s1
SENDER
s2
s4
s3
t1
t5
t6
t3
m2
t9
t10
m1
SEND_TO_RECEIVE
m3
t11
t18
t17
t13
r2
r3
r1
RECEIVER
r4
t12
t15
t16
t14
a2
t19
t20
a1
RECEIVE_TO_SEND
a3
t2
t7
t8
t4
A
B
C
D

Figure 3

t18 : "SKIP0" "TRANSMIT0"
t19 : "LOOSE_ACK0"
t20 : "LOOSE_ACK1"

Le réseau obtenu est identique à celui donné dans [SIG 81], à l'exception de la ligne, qui y est représentée par un buffer de longueur 3.

8 - CONCLUSION

Dans ce chapitre nous avons présenté les techniques variées mises en oeuvre dans le système CESAR pour l'obtention d'un réseau de Petri interprété représentant l'application répartie étudiée, décrite en langage CESAR. Ceci est mené en deux phases :

- d'abord la traduction de chaque tâche élémentaire en un RdPI (en général communicant). Pour ce travail nous avons défini une "méta-grammaire de briques" dont chaque règle est associée à une (ou plusieurs) règles de la grammaire du langage de description CESAR. Le but visé était purement pratique, aussi n'avons nous pas jugé intéressant d'effectuer une analyse théorique poussée de cet objet. Due à la présentation graphique systématique des règles de la méta-grammaire de briques, ce chapitre est assez volumineux, mais sa lecture en est, nous le croyons, grandement facilitée. La traduction des instructions proprement dite est complétée de l'algorithme de substitution des procédures, seule méthode pour exprimer de façon statique la sémantique de l'appel de procédure dans un système multi-tâches (le "code" de la procédure étant dupliqué, aucun problème de réentrance ne se pose).

- ensuite la composition des réseaux obtenus dans la première phase, pour représenter les rendez-vous. Cette opération de composition est dans le principe identique à celle de [GUE 81] ou [JOR 82], mais elle est toutefois nettement plus complexe par la généralité du cas dans lequel nous nous plaçons (nombre quelconque d'émetteurs et de récepteurs ; mode "anonyme" et mode "diffusion" pouvant coexister). Cette opération est complétée de l'opération d'encapsulation, qui permet de restreindre la portée des échanges, et donc de réaliser une véritable structure hiérarchisée de tâches composées.

L'introduction des objets non spécifiés dans le langage de description a des répercussions sur le modèle généré. Des réseaux particuliers sont utilisés pour représenter les procédures et les tâches non spécifiées. L'algorithme d'élimination des variables non significatives permet de tirer toutes les conséquences de l'utilisation de constantes et d'expressions non spécifiées, en modifiant l'interprétation du réseau de façon à n'exclure aucune des situations possibles.

Le but de ce travail de génération est d'obtenir un modèle sur lequel l'analyse des propriétés spécifiées pourra être menée. Ce point sera traité dans le chapitre suivant.

L'introduction de quatre types d'étiquettes peut sembler une méthode complexe et artificielle. Nous la justifierons également dans le chapitre suivant.

Notons enfin que la méthode de traduction que nous proposons ici définit totalement et de façon non ambigue la sémantique du langage CESAR, dans la mesure où celle du RdPI est définie par ses règles de fonctionnement. Les réseaux de Petri ont déjà été utilisés dans ce but, en particulier pour les expressions de chemins [LC 74] [LC 75] et pour un langage de description de systèmes [JKM 79] [JK 82].

Chapitre IV

LE LANGAGE DE SPECIFICATION

1. Introduction
2. Le langage de spécification
3. Les variables propositionnelles
4. Les opérateurs temporels
5. Le protocole du bit alterné
6. Evaluation des opérateurs temporels
7. Conclusion

1 - INTRODUCTION

L'objet de ce chapitre est de présenter le langage de spécification du système CESAR. Ce langage est composé de formules exprimant des propriétés du fonctionnement (propriétés comportementales) devant être vérifiées par l'application décrite (dans le langage de description du système CESAR). Ces formules se présentent comme les formules bien formées (c.a.d. respectant l'arité des opérateurs) d'une logique temporelle. Une telle logique utilise, outre les opérateurs booléens classiques, des opérateurs (dits "temporels") permettant de caractériser le comportement d'un prédicat au cours de l'évolution du système. La liaison avec les tâches, les variables et les actions étiquetées du programme de description s'effectue grâce à un ensemble de variables propositionnelles introduites automatiquement par le système CESAR à partir du programme de description.

Cette logique permet d'exprimer des propriétés telles que :
- une relation entre les variables doit rester vérifiée pour tout comportement possible du système,
- une action doit être exécutée avant une autre,
- après une action, une relation entre les variables doit être vérifiée, etc.

Les relations entre la logique temporelle utilisée pour la spécification des propriétés d'une application décrite et le modèle obtenu à partir de la description de cette application seront explicitées au moyen d'une interprétation des variables propositionnelles et des opérateurs temporels de la logique. L'interprétation des variables propositionnelles sera donnée en termes de prédicats sur le modèle réseau de Petri interprété et étiqueté. Pour donner l'interprétation des opérateurs temporels, nous introduirons un modèle plus primitif que le RdPI, appelé "Système de Transitions", montrerons que la sémantique d'un RdPI peut être aisément définie en termes de Systèmes de Transitions, et donnerons l'interprétation des opérateurs temporels dans un Système de Transitions.

Dans la dernière section du chapitre on montrera comment il est possible d'évaluer les opérateurs temporels en les considérant comme des points fixes de transformateurs de prédicats. Il est donc possible de vérifier effectivement que le modèle obtenu à partir de la description satisfait les propriétés spécifiées.

2 - LE LANGAGE DE SPECIFICATION

2.1 - Définition syntaxique

Le langage de spécification du système CESAR est un langage de formules L construit à partir :
- d'un ensemble F d'objets (appelés variables propositionnelles),
- de la constante TRUE,
- de l'opérateur unaire de négation booléenne, noté NOT,
- de l'opérateur binaire de conjonction booléenne, noté AND,
- d'un ensemble d'opérateurs temporels unaires (OP1i)i,
- d'un ensemble d'opérateurs temporels binaires (OP2j)j,

par application des règles suivantes :
- TRUE appartient à L,
- tout élément de F est élément de L,
- si f est élément de L, NOT f est élément de L,
- si f1 et f2 sont éléments de L, f1 AND f2 est élément de L,
- si f est élément de L, OP1 (f) est élément de L pour tout opérateur temporel unaire OP1,
- si f1 et f2 sont éléments de L, OP2 [f1] (f2) est élément de L pour tout opérateur temporel binaire OP2. (La notation OP2 [ ] ( ) a été choisie pour mettre bien en évidence les rôles différents des deux arguments des opérateurs temporels binaires.)

Les notations suivantes seront également employées :
- FALSE pour NOT TRUE,
- f1 OR f2 pour NOT (NOT f1 AND NOT f2),
- f1 => f2 pour (NOT f1) OR f2,
- pour tout opérateur temporel unaire OP1, on définit l'opérateur dual de OP1, noté dual (OP1) par :

  dual (OP1) (f) ≡ NOT OP1 (NOT f),
- pour tout opérateur temporel binaire OP2, on définit également l'opérateur dual (par rapport au deuxième opérande) de OP2, noté dual (OP2) par :

  dual (OP2) [f1] (f2) ≡ NOT OP2 [f1] (NOT f2).

(Nous n'utiliserons jamais la dualisation par rapport au premier opérande).

L'ensemble F des variables propositionnelles est définie au paragraphe 3 p. IV-6. Les ensembles d'opérateurs temporels (unaires et binaires) sont définis au paragraphe 4 p. IV-15. On trouvera en annexe la syntaxe exacte reconnue par le système CESAR pour les formules temporelles.

2.2 - Le_modèle_Système_de_Transitions

Définition

Un Système de Transitions est un doublet S = (Q, ->), où Q est un ensemble d'objets, appelés **états**, et -> est une relation binaire sur Q.

La relation -> représente les **transitions** entre états du système.

Une **séquence d'exécution** à partir d'un état donné q0 est un séquence d'états s telle que :
- ou bien s est finie : s = q0q1q2 ... qt avec qi -> q(i+1) pour tout i de 0 à t-1, et qt est un état **puit**, c.a.d. il n'existe pas d'état q' tel que qt -> q',
- ou bien s est infinie : s = q0q1q2 ... qkq(k+1)... avec qi -> q(i+1) pour tout i.

On notera s(k) :
- le (k+1)-ième élément de s (c.a.d. qk) s'il existe,
- sinon, on prendra s(k) égal à un élément fictif omega rajouté à Q, non accessible (c.a.d. tel qu'il n'existe pas d'état q tel que q -> omega).

On notera -k-> la puissance k-ième de la relation ->. La relation s(0)-k->s(k) sera fausse ssi s(k) est égal à omega.

L'ensemble des séquences d'exécution à partir d'un état q sera noté EXq.

Un réseau de Petri interprété RI = (R, X, I) peut être considéré comme un système de transitions, dont l'ensemble des états est Q = MM(R) X D, où MM(R) est l'ensemble des marquages du RdP R, et D le domaine de définition de la variable X. La relation -> est définie par q -> q' ssi il existe dans RI une transition validée par q dont la mise à feu transforme l'état q en q'.

2.3 - Interprétation_de_la_logique_temporelle

Pour donner une sémantique au langage L que nous venons de définir, il nous faut définir pour chaque constante, variable propositionnelle et opérateur une interprétation sur un modèle. Plutôt que de travailler directement sur le modèle RdPI, nous définirons l'interprétation de la logique temporelle sur le modèle Système de Transitions. Outre le fait que l'expression en est plus facile, nous avons ainsi la volonté de montrer que cette logique est un système général d'expressions des propriétés, et n'est pas intrinsèquement

liée au modèle RdPI (à l'exception des variables propositionnelles introduites par le système CESAR, cf 3 p. IV-6).

Les constantes TRUE et FALSE étant respectivement interprétées comme les prédicats universellement vrai et faux, les variables propositionnelles le sont comme des prédicats sur les états du modèle. L'interprétation des opérateurs NOT, AND, OR et "=>" est celle des opérateurs booléens usuels. Ainsi étant donné un langage de formules L, et un système de transitions S = (Q, ->), on peut définir l'interprétation de L comme la fonction associant à chaque formule f de L un prédicat sur S (c.a.d. une application de Q dans {vrai, faux}) noté |f|, tel que :

- |TRUE| (q) = vrai, quel que soit q de Q,
- toute variable propositionnelle f de F est telle que |f| soit un prédicat sur S,
- pour toute formule f de L : |NOT f| (q) = vrai ssi |f| (q) = faux,
- pour tout f1, f2 de L : |f1 AND f2| (q) = vrai ssi |f1| (q) = vrai et |f2| (q) = vrai.

Il reste à donner une interprétation des opérateurs temporels. Cela sera fait au paragraphe 4 p. IV-15. Nous utiliserons pour cela la notion de séquence d'exécution.

3 - LES_VARIABLES_PROPOSITIONNELLES

Les variables propositionnelles figurant dans les formules de spécification sont de trois types :
- les variables propositionnelles associées aux tâches,
- les variables propositionnelles associées aux actions étiquetées dans le programme de description,
- des prédicats sur les variables du programme de description.

A chaque variable propositionnelle est associée une interprétation sur le modèle RdPI étiqueté en termes :
- de prédicats sur le marquage courant,
- de prédicats sur les variables.

Nous dirons qu'une tâche T1 est à_l'intérieur d'une tâche composée T2 ssi (définition récursive) :
- soit T1 est une composante de T2,
- soit T1 est une composante d'une tâche composée T', elle-même à l'intérieur de T2.

3.1 - Note_préalable_:_Le_problème_des_noms

Le système CESAR, de façon à différencier tous les objets qu'il manipule, utilise des noms absolus (qu'il s'agisse des noms de tâches, des étiquettes d'actions ou des noms des variables). Il est toutefois trop contraignant d'obliger l'utilisateur à manipuler de tels noms dans les formules de spécification. C'est pourquoi le système CESAR doit être muni d'une bibliothèque de tous les noms absolus (pour les noms de variables, d'actions et de tâches) qu'il a créés et doit rechercher, pour tout nom fourni par l'utilisateur dans une formule de spécification, toute coincidence possible avec un nom absolu. (Ceci n'a pas été réalisé dans la version expérimentale actuelle.)

Comme l'utilisateur peut également fournir des noms préfixés (pour accéder à un champ d'une structure ou à une variable d'une procédure possédant le même identificateur qu'une autre variable, par exemple) le système CESAR doit considérer qu'il y a coincidence entre le nom fourni

x1.x2. ... .xn

(où chaque xi est un identificateur, éventuellement indexé) et le nom absolu

y0.x1.y1.x2. ... .y(n-1).xn

où chaque yi est une suite quelconque (éventuellement vide) d'identificateurs (éventuellement indexés).

Si plusieurs coincidences sont possibles, correspondant à des objets de natures différentes (variables, actions,

tâches), ou à des variables distinctes, le système CESAR rejettera la formule de spécification en précisant quel symbole ne peut être identifié et quelles coincidences sont possibles. L'utilisateur devra alors préciser le symbole rejeté, par l'emploi d'une préfixation adéquate de façon à ne pouvoir obtenir qu'une seule coincidence avec un nom absolu du système.

Si un nom e peut coincider avec plusieurs noms absolus d'actions ea1, ea2, ... ean, (cas d'une action étiquetée figurant dans une procédure appelée dans des environnements différents, par exemple), le système CESAR interprètera le symbole e comme la suite des symboles ea1, ea2, ... ean. Il procèdera de même pour les noms de tâches.

De plus, en ce qui concerne les noms de tâches d'un tableau, il devra être possible d'employer le symbole "*" à la place d'une valeur d'indice, pour signifier l'ensemble de toutes les tâches du tableau pour toutes les valeurs possibles de l'index correspondant.

3.2 - Variables propositionnelles associées aux tâches

Il y a deux sortes de variables propositionnelles associées aux tâches, caractérisant respectivement :
- leur état initial,
- leur état final.

3.2.1 - Etat initial

Pour chaque tâche de nom absolu Ta le système CESAR reconnaît la variable propositionnelle

INIT (Ta)

signifiant que la tâche est dans son état initial.

L'interprétation de cette variable propositionnelle dépend de la nature de la tâche Ta :

1) si Ta est une tâche élémentaire : l'interprétation de la variable propositionnelle INIT(Ta) est la conjonction des prédicats suivants :
- la place initiale pi de la tâche Ta est marquée,
- chaque variable de l'interprétation déclarée avec valeur initiale spécifiée dans la partie déclarative de la tâche est telle que sa valeur soit égale à sa valeur initiale déclarée. Les variables non-significatives sont naturellement éliminées. Il est à remarquer que les variables significatives déclarées sans valeur initiale spécifiée dans la partie déclarative de la tâche ne figurent pas dans

l'interprétation de la variable propositionnelle INIT(Ta), ce qui est naturel puisque leurs valeurs initiales peuvent être n'importe quelle valeur de leur type. Il en est de même des variables déclarées dans les blocs ou procédures puisqu'elles ne sont initialisées qu'à l'entrée dans le bloc ou la procédure où elles sont déclarées.

2) si Ta est une tâche non spécifiée, l'interprétation de la variable propositionnelle INIT(Ta) est le prédicat toujours vrai. Cela est justifié par le fait qu'une tâche non-spécifiée est sans mémoire, et se comporte toujours comme si elle venait d'être initialisée. En termes de réseaux, cela signifie que la place pi d'une tâche non-spécifiée est toujours marquée, ce qui peut facilement être montré au vu du réseau représentant une telle tâche.

3) si Ta est une tâche composée, l'interprétation de la variable propositionnelle INIT(Ta) est la conjonction des interprétations des variables propositionnelles INIT(Ti) pour toutes les tâches Ti composantes de Ta. Par transitivité, cela revient à dire que INIT(Ta) a pour interprétation la conjonction des interprétations des variables propositionnelles INIT(Tei) pour toutes les tâches élémentaires Tei à l'intérieur de Ta.

On définit de même la variable propositionnelle

INIT (Ta1, ... , Tap)

comme la conjonction des variables propositionnelles INIT(Tai) pour i allant de 1 à p. Elle caractérise le fait que toutes les tâches Tai sont en leur état initial.

Ainsi, par application des règles de reconnaissance des noms dans le système CESAR, la variable propositionnelle INIT(T) sera interprétée comme INIT(Ta) si Ta est le seul nom absolu en coincidence avec T, ou au contraire comme INIT(Ta1, ... Tak) si Ta1, ... Tak sont tous les noms absolus de tâches en coincidence avec T.

Lorsque le système à analyser est décrit au moyen d'une tâche composée unique S, on notera

INIT

la variable propositionnelle INIT(S). Elle signifie que le système au complet est dans son état initial, c.a.d. par transitivité que toutes les tâches élémentaires le sont.

3.2.2 - Etat_final

Pour chaque tâche de nom absolu Ta le système CESAR reconnaît la variable propositionnelle

END (Ta)

signifiant que la tâche est dans son état final.

L'interprétation de cette variable propositionnelle dépend de la nature de la tâche Ta :

1) si Ta est une tâche élémentaire, l'interprétation de END(Ta) est le prédicat sur le marquage : la place finale pf de Ta est marquée.

2) si Ta est une tâche non-spécifiée, l'interprétation de END(Ta) est le prédicat toujours faux. On ne peut en effet jamais affirmer qu'une tâche non-spécifiée est terminée. Du point de vue du réseau cela signifie que la place pf d'une tâche non-spécifiée n'est jamais marquée (ce qui est facile à vérifier).

3) si Ta est une tâche composée, l'interprétation de END(Ta) est la conjonction des interprétations des variables propositionnelles END(Ti) pour toutes les tâches spécifiées Ti composantes de Ta, c.a.d. par transitivité la conjonction des interprétations des variables propositionnelles END(Tei) pour toutes les tâches élémentaires Tei (spécifiées) à l'intérieur de Ta.

On définit de même la variable propositionnelle

END (Ta1, ... , Tap)

comme la conjonction des variables propositionnelles END(Tai) pour i allant de 1 à p. Elle signifie que toutes les tâches Tai sont dans leur état final. Remarquer qu'elle est fausse si l'une des tâches Tai est non spécifiée.

Le système CESAR pourra ainsi interpréter la variable propositionnelle END(T) quel que soit le nombre de noms absolus de tâches en coincidence avec T.

Lorsque le système à analyser est décrit au moyen d'une tâche composée unique S, on notera

END

la variable propositionnelle END(S). Elle signifie que toutes les tâches spécifiées du système sont dans leur état final, c.a.d. par transitivité que toutes les tâches élémentaires le sont.

3.3 - Variables propositionnelles associées aux actions étiquetées

Remarque préliminaire :

L'algorithme de génération du réseau de Petri interprété représentant une application est tel que chaque processus du graphe de processus obtenu corresponde à une tâche élémentaire ou non-spécifiée du système décrit. Par la suite

nous confondrons le nom absolu de la tâche élémentaire ou non spécifiée et le nom du processus (ensemble de places du réseau) la représentant. Fin de remarque.

Pour chaque action étiquetée définie dans le programme de description, le système CESAR reconnaît des variables propositionnelles :
- caractérisant la possibilité d'exécuter cette action,
- caractérisant la position relative (immédiatement avant, à l'intérieur, immédiatement après) du contrôle d'une tâche par rapport à cette action.

3.3.1 - Exécutabilité d'une action

Pour chaque action étiquetée de nom absolu ea, le système CESAR reconnaît la variable propositionnelle

ENABLE (ea)

signifiant que l'action ea est exécutable.

L'action ea est considérée comme exécutable lorsque la transition (unique) étiquetée "_ea" ou "ea" est validée, c.a.d. lorsque toutes ses places d'entrée sont marquées et que la condition qui lui est associée est vraie. L'interprétation de ENABLE(ea) est donc la conjonction des deux prédicats (t étant la transition étiquetée par "ea" ou "_ea") :
- la condition c associée à t est vérifiée,
- et chaque place de >t est marquée.

Le système CESAR reconnaît de plus la variable propositionnelle

ENABLE (ea1, ... , ean)

définie comme l'union des variables propositionnelles ENABLE(eai). Elle caractérise le fait que l'une au moins des actions eai est exécutable.

Le système CESAR pourra ainsi interpréter la variable propositionnelle ENABLE(e) quel que soit le nombre de noms absolus d'actions en coincidence avec e.

Si Ta est un nom absolu de tâche, le système CESAR reconnaît la variable propositionnelle

ENABLE (Ta)

signifiant que la tâche Ta peut encore évoluer.

Si Ta est une tâche élémentaire ou non-spécifiée, ENABLE(Ta) est vraie lorsque l'une quelconque des transitions de sortie de l'une quelconque des places du processus Ta est validée. ENABLE(Ta) prend donc en compte l'exécutabilité de toutes les transitions, qu'elles correspondent à des actions étiquetées ou non.

Si Ta est une tâche composée, ENABLE(Ta) est définie comme l'union des variables propositionnelles ENABLE(Tei) pour toutes les tâches élémentaires ou non-spécifiées Tei à l'intérieur de Ta.

On définit de même la variable propositionnelle

ENABLE (Ta1, ... , Tap)

comme l'union des variables propositionnelles ENABLE(Tai) pour i allant de 1 à p. Elle caractérise le fait que l'une des tâches Tai au moins peut évoluer.

Le système CESAR pourra ainsi interpréter la variable propositionnelle ENABLE(T) quel que soit le nombre de noms absolus de tâches en coincidence avec T.

Lorsque le système à analyser est décrit au moyen d'une tâche composée unique S, on notera

ENABLE

la variable propositionnelle ENABLE(S). Remarquons que la variable ENABLE est vraie tant que le système peut évoluer, et donc que NOT ENABLE caractérise son blocage définitif.

Enfin on pourra utiliser les abréviations suivantes :

SINK (T) pour NOT ENABLE (T)

SINK (T1, ... , Tp) pour NOT ENABLE (T1, ... , Tp)

SINK pour NOT ENABLE.

3.3.2 - Position par rapport à une action

Pour toute action étiquetée de nom absolu ea, et pour tout nom absolu de tâche spécifiée Ta, le système CESAR reconnaît les trois variables propositionnelles

BEFORE (ea, Ta)

IN (ea, Ta)

et AFTER (ea, Ta)

caractérisant le fait que le contrôle de la tâche Ta se trouve respectivement :

- immédiatement avant le début de l'action ea (ce qui ne préjuge pas de l'exécutabilité de celle-ci),
- au cours de l'exécution de l'action ea,
- immédiatement après la fin de l'action ea.

Si Ta est une tâche élémentaire, l'interprétation de ces variables propositionnelles est définie comme suit :

- BEFORE (ea, Ta) est vraie si Ta contient une place d'entrée de la transition (unique) étiquetée "ea" ou "_ea", et si celle-ci est marquée ;
- IN (ea, Ta) est vraie si ea est une action composée dont un chemin d'exécution est dans Ta, et la place d'entrée dans Ta de l'une des transitions étiquetées par "_ea_" ou "ea_"

est marquée ;
- AFTER (ea, Ta) est vraie si Ta contient une place de sortie d'une transition étiquetée "ea_", ou de la transition étiquetée "ea", et si celle-ci est marquée.

Si Ta est une tâche composée, les variables propositionnelles BEFORE (ea, Ta) (resp. IN (ea, Ta), AFTER (ea, Ta)) sont définies comme les unions des variables propositionnelles BEFORE (ea, Tei) (resp. IN (ea, Tei), AFTER (ea, Tei)) pour toutes les tâches élémentaires Tei à l'intérieur de Ta.

On définit de même en termes d'union sur les couples (eai, Taj) :

BEFORE (ea1, ... , ean, Ta) BEFORE (ea, Ta1, ... , Tap)
IN (ea1, ... , ean, Ta) IN (ea, Ta1, ... , Tap)
AFTER (ea1, ... , ean, Ta) AFTER (ea, Ta1, ... , Tap)
BEFORE (ea1, ... , ean, Ta1, ... , Tap)
IN (ea1, ... , ean, Ta1, ... , Tap)
AFTER (ea1, ... , ean, Ta1, ... , Tap).

Le système CESAR pourra ainsi interpréter les variables propositionnelles BEFORE (e, T), IN (e, T), AFTER (e, T) quel que soit le nombre de noms absolus en coincidence avec e ou T.

Si le système à analyser est décrit au moyen d'une tâche composée unique S, on notera :

BEFORE (e) BEFORE (e1, ... , en)
IN (e) IN (e1, ... , en)
AFTER (e) AFTER (e1, ... , en)

pour BEFORE (e, S), IN (e, S) etc.

3.3.3 - Remarque sur la variable propositionnelle AFTER

L'interprétation de la variable propositionnelle AFTER (ea, Ta) pour une tâche Ta élémentaire donnée précédemment en termes du marquage du réseau de Petri interprété ne signifie effectivement que l'action ea vient d'être exécutée par la tâche Ta que si toutes les transitions étiquetées "ea_", ou la transition unique étiquetée "ea", sont telles que leur place p de sortie dans Ta ne peut être marquée par une transition ne correspondant pas à la fin de l'action e (c.a.d. non étiquetée par "ea" ou "ea_"). De plus il est nécessaire que p ne soit pas initialement marquée.

Cette propriété n'est absolument pas assurée par les règles de génération employées. Toutefois il est possible de transformer le réseau de façon à la garantir, en utilisant la transformation suivante (on suppose que l'ensemble Ea est l'ensemble des actions ea pour lesquelles on désire utiliser

une variable propositionnelle AFTER (ea, Ta) avec le sens que ea vient d'être exécuté) :
pour chaque action ea de Ea,
pour chaque place p de sortie dans Ta des transitions étiquetées "ea" ou "ea_" :
partitionner l'ensemble >p en {t1} et {t2} tels que
{t1} soit l'ensemble des transitions étiquetées "ea" ou "ea_" et {t2} = >p - {t1} ;
si {t2} ≠ ∅ ou si p est initialement marquée alors transformer les transitions de {t1} selon le schéma de la figure 1 (on note t1 ... t1' les transitions de {t1}, t2 ... t2' les transitions de {t2}).

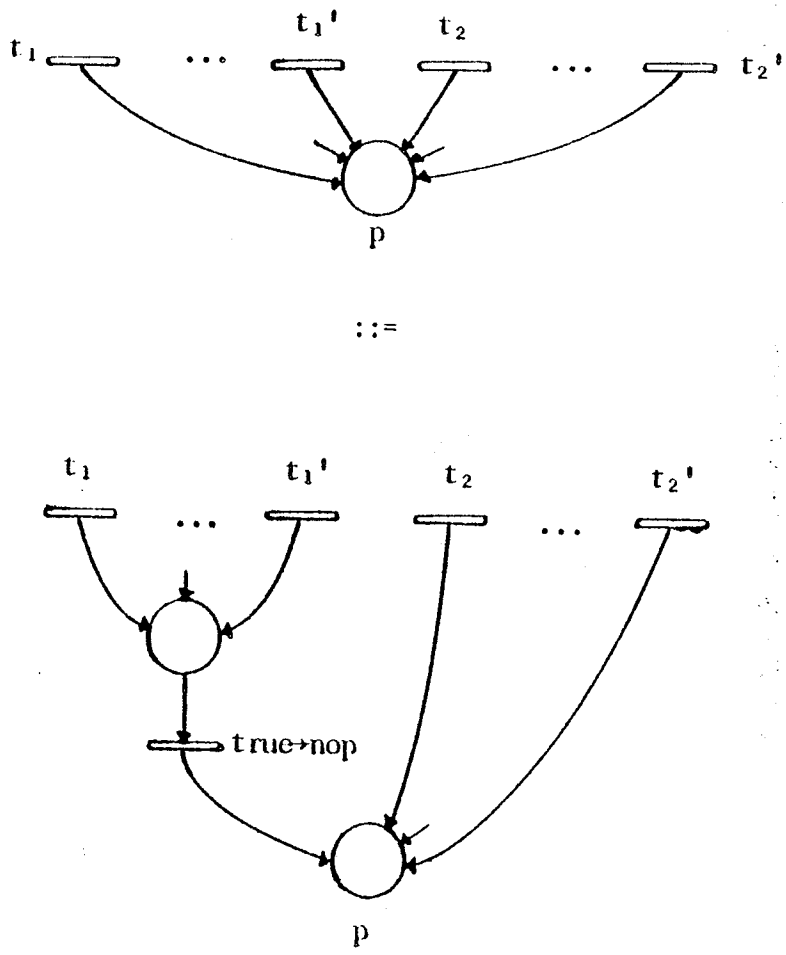


Figure 1

Cette transformation n'est pas incluse dans les règles de génération, car d'un point de vue pratique on a constaté que bien souvent l'ensemble Ea est un ensemble bien plus petit que l'ensemble de tous les noms d'actions figurant dans le programme de description. On a donc jugé préférable d'appliquer ponctuellement cette transformation pour chaque formule de spécification où figurent des variables propositionnelles AFTER.

3.3 - Prédicats sur les variables de l'interprétation

L'utilisateur peut utiliser comme variable propositionnelle tout prédicat exprimé par une condition (selon la syntaxe du langage de description) sur les variables du programme de description. Le système CESAR doit retrouver les noms absolus des variables et transformer la condition fournie en une condition sur les variables du réseau par élimination des variables non significatives. Les variables éliminées doivent être signalées à l'utilisateur.

4 - LES OPERATEURS TEMPORELS

Dans cette partie sont définis les opérateurs temporels du langage de spécification du système CESAR. On introduit d'abord les opérateurs unaires, qui sont formés, d'une part, des opérateurs POT, INEV, OBL, et WPOT, exprimant certaines modalités d'atteignabilité (possible, inévitable, etc.) et, d'autre part, de leurs duaux ALL, SOME, SONT et ALW. En fait, les opérateurs OBL et WPOT peuvent eux-mêmes s'exprimer à l'aide des opérateurs POT et INEV, qui sont donc les deux seuls opérateurs primitifs de cette logique.

On introduit ensuite les opérateurs binaires, qui sont les extensions conditionnelles des opérateurs unaires.

Enfin le problème de l'équitabilité (connu en anglais sous le nom de "fairness") est soulevé, ce qui nous conduit à introduire un nouvel opérateur, noté FAIR, pour exprimer l'atteignabilité inévitable sous l'hypothèse de l'équitabilité. Nous montrons ensuite que cet opérateur peut être approximé au moyen des opérateurs temporels unaires, et peut exactement être exprimé en utilisant les opérateurs conditionnels.

4.1 - Notion d'invariant et atteignabilité potentielle

4.1.1 - Définitions

L'opérateur temporel unaire POT est utilisé pour exprimer qu'un prédicat peut devenir vrai au cours de l'évolution du système décrit. Son interprétation est définie par :

|POT (f)| (q) ≡
il existe une séquence d'exécution s de EXq,
il existe k entier tels que :
q-k->s(k) et |f| (s(k)) = vrai.

En termes d'ensembles caractéristiques, |POT (f)| est l'ensemble des états q de S tels qu'il existe une séquence d'exécution à partir de q contenant un état satisfaisant |f|. |POT (f)| sera appelé l'ensemble des états à partir desquels un état de |f| est potentiellement atteignable.

L'opérateur temporel ALL est défini comme le dual de POT :
ALL ≡ dual (POT).

Son interprétation obtenue par dualisation est la suivante :
|ALL (f)| (q) ≡
quelle que soit la séquence d'exécution s de EXq,
quel que soit k entier :
q-k->s(k) implique |f| (s(k)) = vrai.

Si un état q vérifie |ALL (f)|, alors tous les états de toutes les séquences d'exécution à partir de q vérifient |f|. Tout ensemble d'états vérifiant cette propriété sera appelé invariant inclus dans |f|.

Remarque :

|ALL (f)| contient tous les états puits vérifiant |f| (c.a.d. les états q tels que |f|(q) et il n'existe pas q' : q->q').

4.1.2 - Propriétés

L'opérateur POT vérifie les propriétés suivantes :

$$f \Rightarrow POT(f)$$
$$POT(f \text{ OR } f') = POT(f) \text{ OR } POT(f')$$
$$POT(POT(f)) = POT(f).$$

Par dualisation, l'opérateur ALL vérifie les propriétés suivantes :

$$ALL(f) \Rightarrow f$$
$$ALL(f \text{ AND } f') = ALL(f) \text{ AND } ALL(f')$$
$$ALL(ALL(f)) = ALL(f).$$

Démonstration : Ces propriétés se déduisent aisément de l'expression de l'interprétation de POT et ALL et des propriétés des quantificateurs employés.

4.1.3 - Utilisations

Nous donnons ici quelques exemples typiques d'utilisation des opérateurs POT et ALL.

a) Expression de propriétés invariantes

Ce sont des propriétés qui doivent être vérifiées en permanence par le système décrit. Elles peuvent être exprimées par une formule de la forme :

$$INIT \Rightarrow ALL(f).$$

En fait un grand nombre de propriétés ont cette forme, avec une formule f plus ou moins compliquée. Afin de simplifier l'écriture, nous introduisons la notation

[] f

pour INIT => ALL (f).

Nous réserverons l'appellation propriété invariante au cas où f est une formule sans opérateur temporel. A l'intérieur de cette classe, plusieurs sous-classes de propriétés sont intéressantes.

* propriété_globalement_invariante

[] f

où f est un prédicat sur les variables du système décrit.

Exemple :

[] (0 <= NOMBRE_D_ELEMENTS AND NOMBRE_D_ELEMENTS <= N)

exprimant le fait qu'un buffer est borné.

* propriété_localement_invariante

[] (p => f)

où p est une variable propositionnelle exprimant une exécutabilité ou une position (ENABLE, BEFORE, IN, AFTER) et f un prédicat sur les variables du système décrit.

Exemples

[] (IN (ECRITURE) => (SEMAPHORE = 0))

exprimant le fait que lorsqu'un accès en écriture a lieu, le sémaphore doit rester à zéro durant toute l'opération.

[] (AFTER (ECRITURE) => (NOMBRE_D_ELEMENTS >= 1))

exprimant le fait qu'après une écriture le buffer ne peut être vide.

* exclusion_dirigée

[] (p1 => NOT p2)

où p1 et p2 sont deux variables propositionnelles exprimant des positions.

Exemple

[] (IN (ACCES, REDACTEUR) => NOT IN (ACCES, LECTEUR(*))

exprimant une exclusion des lecteurs par le rédacteur.

* exclusion_mutuelle

[] NOT (p1 AND p2)

où p1 et p2 sont deux variables propositionnelles exprimant

des positions.

Exemple

[] NOT (IN (SECTION, T1) AND IN (SECTION, T2))
exprimant l'exclusion mutuelle de T1 et T2 dans la section critique.

b) Atteignabilité_potentielle

Ce sont des propriétés exprimant qu'un certain prédicat peut (éventuellement) être atteint. Elles s'expriment par une formule de la forme :

INIT => POT (f).

Il est rare que l'on s'intéresse à de telles propriétés, car la possibilité d'atteindre seule (sans garantie que l'on atteigne effectivement) n'est en général pas suffisante.

On s'intéresse plutôt à la non atteignabilité d'une situation dangereuse, qui s'exprime par

INIT => NOT POT (f)

formule équivalente à

INIT => ALL (NOT f)

c.a.d. [] (NOT f)

ce qui classe cette propriété dans la catégorie des propriétés invariantes. Les propriétés de cette forme sont parfois appelées propriétés de sureté.

c) Vivacité

On peut exprimer qu'une action e est vivante (au sens usuellement employé pour les transitions des RdP [SIF 79]) par la formule :

INIT => ALL (POT (ENABLE (e)))

signifiant que, quelque soit l'évolution ultérieure du système décrit, on pourra toujours rendre e à nouveau exécutable. Par dualisation cette formule peut s'écrire :

INIT => ALL (NOT ALL (NOT ENABLE (e)))

ou INIT => NOT POT (ALL (NOT ENABLE (e)))

exprimant qu'il n'est pas possible d'atteindre un invariant contenu dans NOT ENABLE (e). En effet, si l'on pénètre dans un tel invariant, l'action e ne sera jamais plus exécutable. Un tel invariant est usuellement appelé verrou (en anglais "deadlock") pour l'action e.

Remarquons que la vivacité d'un ensemble d'actions e1, ... , en s'exprime par :

[] (POT (ENABLE (e1)) AND ... AND POT (ENABLE (en)))

alors que la formule

[] POT (ENABLE (e1, ... , en))

exprime que l'une au moins des actions ei est vivante.

Ainsi, la formule

[] POT (ENABLE (T))

où T est un nom de tâche signifie que la tâche T pourra toujours évoluer quelle qu'ait été son évolution antérieure.

Enfin, le fait que le système décrit n'est jamais totalement bloqué s'exprime par

INIT => NOT POT (NOT ENABLE)

ce qui est équivalent à

INIT => ALL (ENABLE)

c.a.d. [] ENABLE.

4.2 - Notion de trajectoire et atteignabilité inévitable

4.2.1 - Définitions

L'opérateur temporel unaire INEV est utilisé pour exprimer qu'un prédicat doit devenir vrai au cours de l'évolution du système décrit. Son interprétation est définie par :

|INEV (f)| (q) ≡
pour toute séquence d'exécution s de EXq,
il existe k entier tel que
q-k->s(k) et |f| (s(k)) = vrai.

En termes d'ensembles caractéristiques, |INEV (f)| est l'ensemble des états q de S tels que toute séquence d'exécution à partir de q contienne un état satisfaisant |f|. |INEV (f)| sera appelé l'ensemble des états à partir desquels un état de |f| est inévitablement atteignable.

L'opérateur temporel SOME est défini comme le dual de INEV :

SOME ≡ dual (INEV).

Son interprétation obtenue par dualisation est la suivante :

|SOME (f)| (q) ≡
il existe une séquence d'exécution s de EXq telle que,
pour tout entier k :
q-k->s(k) implique |f| (s(k)) = vrai.

Si un état q vérifie |SOME (f)|, alors il existe une séquence d'exécution à partir de q dont tous les états vérifient |f|. Un ensemble d'états vérifiant cette propriété sera appelé trajectoire incluse dans |f|.

Remarque :

|SOME (f)| contient les états puits vérifiant |f|.

4.2.2 - Propriétés

L'opérateur INEV vérifie les propriétés suivantes :

$$f \Rightarrow INEV(f)$$

$$INEV(INEV(f)) = INEV(f).$$

L'opérateur INEV n'est pas comme POT distributif par rapport à OR, mais il vérifie toutefois :

$$\text{si } (f \Rightarrow f') \text{ alors } (INEV(f) \Rightarrow INEV(f')).$$

Par dualisation, l'opérateur SOME vérifie les propriétés suivantes :

$$SOME(f) \Rightarrow f$$

$$SOME(SOME(f)) = SOME(f)$$

$$\text{si } (f \Rightarrow f') \text{ alors } (SOME(f) \Rightarrow SOME(f')).$$

Démonstration : Ces propriétés se déduisent des propriétés des quantificateurs figurant dans l'interprétation de SOME et INEV.

4.2.3 - Utilisations

Nous donnons ici quelques exemples typiques d'utilisation de l'opérateur INEV. On peut également obtenir des variantes intéressantes des propriétés données précédemment en remplaçant l'opérateur POT par l'opérateur INEV (en particulier pour la vivacité), ou l'opérateur ALL par l'opérateur SOME.

a) Atteignabilité inévitable

Ce sont des propriétés exprimant qu'un certain prédicat doit être atteint. Elles s'expriment par une formule de la forme :

$$INIT \Rightarrow INEV(f).$$

Ainsi, la correction totale d'un programme s'exprime par le fait

- que ce programme se termine,
- qu'un certain résultat (caractérisé par un prédicat f) est alors atteint.

Cela s'exprime par la formule :

$$INIT \Rightarrow INEV(END \text{ AND } f).$$

Exemple

Avec

$f = (T(1) \leq T(2) \text{ AND } T(2) \leq T(3) \text{ AND } \ldots \text{ AND } T(n-1) \leq T(n))$

cette formule permet d'exprimer la correction d'un programme de tri sur un tableau T de longueur n (donné).

b) Réponse_à_une_action

Il s'agit de spécifier la situation où une action e1 en provoque une autre e2. Elle s'exprime par une formule de la forme :

[] (p1 => INEV (p2))

où p1 et p2 sont des variables propositionnelles indiquant une position par rapport à e1 et e2. Le cas le plus fréquent est celui où p1 est de la forme AFTER (e1, ...) et p2 de la forme ENABLE (e2, ...), BEFORE (e2, ...) ou IN (e2, ...).

Exemple

[] (AFTER (EMISSION) => INEV (ENABLE (RECEPTION)))

exprimant que toute émission doit permettre d'effectuer une réception.

Une forme plus faible de cette propriété (réponse possible) peut être exprimée en employant l'opérateur POT à la place de l'opérateur INEV.

c) Exécution_certaine

Nous dirons qu'une action e a une exécution certaine si elle vérifie la propriété suivante :

[] (ENABLE (e) => INEV (AFTER (e))).

Cette formule exprime que l'action e, si elle est validée au moins une fois, sera inévitablement exécutée. Cette propriété n'est en général pas vraie pour toutes les actions du système décrit, mais il est intéressant pour certains systèmes que certaines actions particulières la vérifient.

Exemple

[] (ENABLE (INTERRUPTION) => INEV (AFTER (INTERRUPTION))

exprimant que toute interruption finira par être prise en compte. Noter qu'il n'y a pas effet de comptage, c.a.d. que cela ne signifie qu'il y aura autant d'exécution de l'action INTERRUPTION que de fois où elle est validée.

Si on remplace l'opérateur INEV par POT, on obtient une propriété triviale, vraie pour tout système de transitions.

4.3 - Trajectoire_sans_fin_et_atteignabilité_obligatoire

4.3.1 - Définitions

L'opérateur temporel unaire OBL est utilisé pour exprimer qu'un prédicat doit devenir vrai au cours de l'évolution du système décrit, à moins que ce dernier ne se bloque avant.

Son interprétation est définie par :

|OBL (f)| (q) ≡
pour toute séquence d'exécution s de EXq,
il existe k entier tel que :
q-k->s(k) implique |f| (s(k)) = vrai.

En termes d'ensembles caractéristiques, |OBL (f)| est l'ensemble des états q de S tels que toute séquence d'exécution à partir de q contienne un état satisfaisant |f|, ou aboutisse à un état puits. |OBL (f)| sera appelé l'ensemble des états à partir desquels un état de |f| est obligatoirement atteignable.

L'opérateur temporel SONT est défini comme le dual de OBL :
SONT ≡ dual (OBL).
Son interprétation obtenue par dualisation est la suivante :

|SONT (f)| (q) ≡
il existe une séquence d'exécution s de EXq telle que,
pour tout entier k :
q-k->s(k) et |f| (s(k)) = vrai.

Si un état vérifie |SONT (f)|, alors il existe une séquence d'exécution infinie à partir de q dont tous les états vérifient |f|. Un ensemble d'états vérifiant cette propriété sera appelé trajectoire sans fin incluse dans |f|.

Remarque :

Il est facile de vérifier, pour tout f :
OBL (f) = INEV (f OR NOT ENABLE)
SONT (f) = SOME (f AND ENABLE).

4.3.2 - Utilisations

Nous donnons ici un exemple d'utilisation des opérateurs OBL et SONT. On peut également obtenir certaines variantes intéressantes des propriétés données précédemment, en remplaçant les opérateurs POT ou INEV par l'opérateur OBL, ou les opérateurs ALL ou SOME par l'opérateur SONT.

a) Absence de famine

Nous dirons qu'il y a famine pour l'action e, si le système décrit peut toujours évoluer sans jamais valider e. Ceci s'exprime par le fait qu'il existe une trajectoire sans fin dans NOT ENABLE (e), atteignable depuis l'état initial. La propriété exprimant qu'il n'y a pas possibilité de famine pour e est donc :
INIT => NOT POT (SONT (NOT ENABLE (e)))

c.a.d. par dualisation :

INIT => ALL (OBL (ENABLE (e)))

ou [] OBL (ENABLE (e)).

Exemple

[] OBL (ENABLE (MANGER, PHILOSOPHE (i)))

exprimant que le philosophe numéro i (donné) ne souffre pas de famine (au sens propre dans cet exemple bien connu), ce qui ne garantit évidemment pas qu'il mange effectivement.

4.4 - Invariant_sans_blocage_et_atteignabilité_faible

4.4.1 - Définitions

L'opérateur temporel unaire WPOT est utilisé pour exprimer qu'un prédicat peut devenir vrai au cours de l'évolution du système décrit, à moins que ce dernier ne se bloque avant. Son interprétation est définie par :

|WPOT (f)| ≡
il existe une séquence d'exécution s de EXq,
il existe k entier tel que :
q-k->s(k) implique |f| (s(k)) = vrai.

En termes d'ensembles caractéristiques, |WPOT (f)| est l'ensemble des états q de S tels qu'il existe une séquence d'exécution à partir de q contenant un état satisfaisant |f| ou aboutissant à un état puits. |WPOT (f)| sera appelé l'ensemble des états à partir desquels un état de |f| est faiblement atteignable.

L'opérateur ALW est défini comme le dual de WPOT :

ALW ≡ dual (WPOT).

Son interprétation,, obtenue par dualisation, est la suivante :

|ALW (f)| ≡
pour toute séquence d'exécution s de EXq,
pour tout k entier :
q-k->s(k) et |f| (s(k)) = vrai.

Si un état vérifie |ALW (f)|, alors toutes les séquences d'exécution à partir de q sont infinies, et tous leurs états vérifient |f|. Un ensemble d'états vérifiant cette propriété sera appelé invariant sans blocage.

Remarque :

On peut aisément vérifier, pour tout f :

WPOT (f) = POT (f OR NOT ENABLE)

ALW (f) = ALL (f AND ENABLE)

ou par distributivité :

WPOT (f) = POT (f) OR POT (NOT ENABLE)

ALW (f) = ALL (f) AND ALL (ENABLE).

4.4.2 - Utilisations

Parmi toutes les substitutions possibles (de ALL, SOME, SONT par ALW et de POT, INEV, OBL par WPOT) la plus intéressante est celle de ALL par ALW pour exprimer les propriétés invariantes des systèmes à évolution cyclique. On peut comme pour ALL distinguer les propriétés globalement invariantes exprimées par des formules de la forme :

INIT => ALW (f)

des propriétés localement invariantes exprimées par des formules de la forme :

INIT => ALW (p => f).

4.5 - Opérateurs temporels conditionnels

Nous définissons ici quatre opérateurs temporels binaires également notés ALL, POT, SOME et INEV, car ils sont en fait des généralisations des opérateurs unaires précédemment définis. Rappelons que comme les deux opérandes ne jouent pas un rôle symétrique, on emploie la notation ALL[f1](f2) (et une notation analogue pour les autres opérateurs). Ceci permet également de considérer l'opérateur unaire ALL[f1]( ) pour f1 donné. On appellera ces opérateurs opérateurs conditionnels, le premier opérande étant la condition sous laquelle l'opérateur temporel est considéré.

4.5.1 - Invariant conditionnel

L'interprétation de ALL[f1](f2) est définie comme suit :

|ALL[f1](f2)| (q) =
pour toute séquence d'exécution s de EXq,
pour tout k entier :
(q=k=>s(k) et pour tout i de 0 à k-1 : |f1|(s(i)))
implique |f2|(s(k)).

Si un état q vérifie |ALL[f1](f2)|, alors tous les états de toutes les séquences d'exécution à partir de q vérifient |f2|

tant que ces séquences sont incluses dans |f1|.

4.5.2 - Trajectoire conditionnelle

L'interprétation de SOME[f1](f2) est définie comme suit :

|SOME[f1](f2)| (q) =
il existe une séquence d'exécution s de EXq telle que
pour tout k entier :
(q-k->s(k) et pour tout i de 0 à k-1 : |f1|(s(i)))
implique |f2|(s(k)).

Si un état q vérifie |SOME[f1](f2)|, alors il existe une séquence d'exécution à partir de q dont tous les états vérifient |f2| tant que la trajectoire reste dans |f1|.

4.5.3 - Atteignabilités conditionnelles

On définit les opérateurs duaux (par rapport au 2 ème opérande) :

POT ≡ dual (ALL)
INEV ≡ dual (SOME)

c.a.d. :

POT[f1](f2) ≡ NOT ALL[f1](NOT f2)
INEV[f1](f2) ≡ NOT SOME[f1](NOT f2).

Leurs interprétations obtenues par dualisation sont les suivantes :

|POT[f1](f2)| (q) ≡
il existe une séquence d'exécution s de EXq,
il existe k entier tels que :
q-k->s(k) et |f2|(s(k)) et
pour tout i de 0 à k-1 : |f1|(s(i)).

|INEV[f1](f2)| (q) ≡
pour toute séquence d'exécution s de EXq,
il existe k entier tel que :
q-k->s(k) et |f2|(s(k)) et
pour tout i de 0 à k-1 : |f1|(s(i)).

|POT[f1](f2)| (resp. |INEV[f1](f2)|) est l'ensemble des états q de S tels qu'il est possible (resp. inévitable) d'atteindre à partir de q un état de |f2| tout en restant dans |f1|.

4.5.4 - Propriétés

Les propriétés suivantes montrent que les opérateurs unaires précédemment introduits sont des cas particuliers des opérateurs temporels conditionnels :

pour tout f :

$$ALL[TRUE](f) = ALL(f)$$
$$SOME[TRUE](f) = SOME(f)$$
$$POT[TRUE](f) = POT(f)$$
$$INEV[TRUE](f) = INEV(f).$$

De plus, pour tout f1, f2, f3 : si f2 => f1 alors

$$ALL[f1](f3) \Rightarrow ALL[f2](f3)$$
$$SOME[f1](f3) \Rightarrow SOME[f2](f3)$$
$$POT[f2](f3) \Rightarrow POT[f1](f3)$$
$$INEV[f2](f3) \Rightarrow INEV[f1](f3).$$

Démonstration : immédiate à partir de la définition des opérateurs conditionnels.

4.5.5 - Utilisations

Nous donnons ici quelques exemples d'utilisation des opérateurs conditionnels.

a) Correction conditionnelle

Une forme de correction partielle peut être exprimée par le fait qu'un prédicat reste invariant tant qu'une certaine condition (exprimant le fonctionnement "normal") est vérifiée. Elle s'exprime par une formule de la forme :

$$(INIT \text{ AND } c) \Rightarrow ALL\ [c]\ (f)$$

où c est la condition exprimant le fonctionnement normal et f est un prédicat quelconque.

Exemple

Avec c = (NBRE_REQUETES_EN_ATTENTE < 1000)
et f = ENABLE (ALLOCATEUR)
cette formule exprime le fait qu'un allocateur de ressources ne se bloque pas tant qu'il n'est pas saturé (le nombre de requêtes en attente devenant trop élevé).

Ce type de formule peut être généralisée pour exprimer une propriété conditionnelle quelconque, c.a.d. une propriété exprimée par une formule temporelle qui est vraie tant qu'une certaine condition est vérifiée.

Exemple

Avec la même condition c et
f = (AFTER (REQUETE) => INEV (ENABLE (ACCES)))
la formule exprime le fait que l'allocateur accorde toujours la ressource dans un temps fini, tant qu'il n'est pas saturé.

b) Réponse_conditionnelle_à_une_action

Des propriétés complexes de réponses conditionnelles peuvent être exprimées en utilisant les opérateurs conditionnels. Nous en donnons ici deux exemples.

Supposons qu'une action e1 admet pour réponse l'action e2. Nous pouvons exprimer la situation suivante : l'action e1, après qu'elle ait été exécutée une fois, ne peut être répétée tant que la réponse e2 n'a pas été effectuée. Cela s'exprime par une formule de la forme :
AFTER (e1) => ALL [NOT AFTER (e2)] (NOT ENABLE (e1)).

Cette propriété ne garantit pas que la réponse soit effectivement exécutée, mais dans ce cas e1 ne sera jamais plus exécutée. Si l'on désire avoir la propriété précédente, tout en imposant que la réponse e2 soit effectuée, on pourra employer une formule de la forme :
AFTER (e1) => INEV [NOT ENABLE (e1)] (AFTER (e2)).

Exemple

Dans un système représentant un protocole de transmission de données, où l'action send représente l'émission d'un nouveau message et où l'action receive représente sa réception, la propriété précédente (en prenant e1 = send et e2 = receive) expriment qu'un nouveau message ne peut être émis tant que le précédent n'est pas parvenu à destination. La deuxième propriété garantit de plus que tous les messages émis finissent par être reçus.

4.5.6 _ Autres_opérateurs_conditionnels

Comme pour les opérateurs unaires, le système CESAR reconnaît les opérateurs binaires (opérateurs conditionnels) OBL, SONT, WPOT et ALW qui s'expriment en fonction des opérateurs ALL, POT, SOME et INEV de la façon suivante :
OBL[f1](f2) = INEV[f1](f2 OR NOT ENABLE)
SONT[f1](f2) = SOME[f1](f2 AND ENABLE)
WPOT[f1](f2) = POT[f1](f2 OR NOT ENABLE)
ALW[f1](f2) = ALL[f1](f2 AND ENABLE).

4.6 - Le problème de l'équitabilité

Nous préférons traduire le terme anglais "fairness" par le néologisme "équitabilité" plutôt que par "équité" comme cela est parfois fait, car ce dernier terme nous paraît impliquer une idée plus forte (trop fortement "égalitaire") que celle que nous voulons exprimer.

4.6.1 - Illustration du problème

a) Considérons une ligne (entre un émetteur et un récepteur) qui peut perdre des messages. Ce système est sommairement représenté par le RdPI de la figure 1.

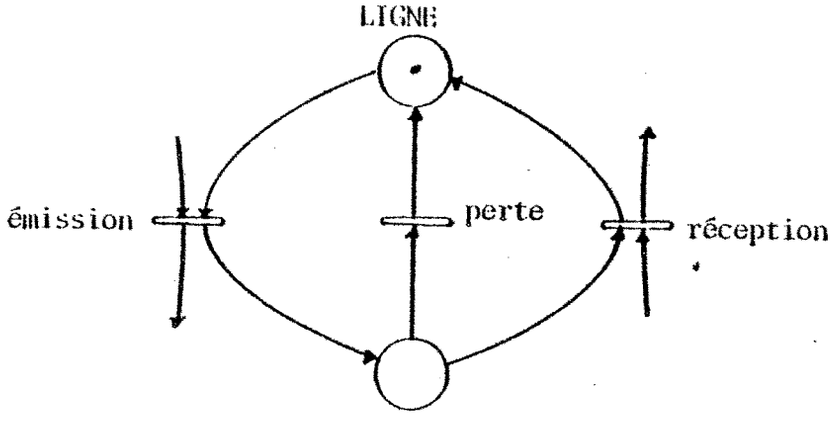


Figure 1

Chaque fois qu'un message est émis, la ligne peut transmettre le message ou le perdre. Le choix étant non déterministe à chaque fois, il est possible que la ligne perde tous les messages émis et donc qu'aucun ne parvienne au récepteur. Nous dirons qu'un tel comportement est non équitable, car il privilégie une action (la perte du message) par rapport à une autre (sa transmission). En fait, tout comportement dans lequel un nombre infini de messages est émis et un nombre fini seulement est reçu est non équitable, car il existe un instant à partir duquel la ligne ne transmet plus aucun message.

Remarquons que si la ligne ne perd pas également un nombre infini de messages son comportement n'est pas non plus

équitable (pour l'action perte de message).

b) Considérons un système formé de trois processus :
- deux processus utilisateurs U1 et U2 d'une ressource en exclusion mutuelle,
- un processus gestionnaire G de cette ressource (ayant le fonctionnement d'un sémaphore).

Un RdPI étiqueté simplifié représentant ce système est donné figure 2, où les actions ai correspondent à l'acquisition de la ressource, et les actions li à sa libération.

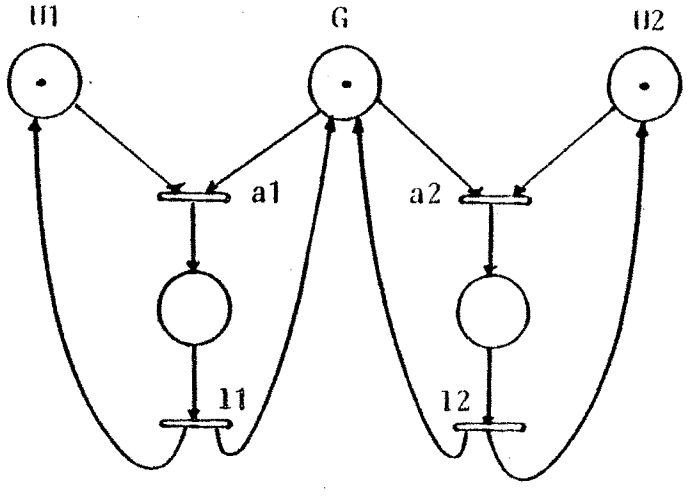


Figure 2

Il est facile de voir (l'exemple est bien connu) que ce système est tel que chaque fois que ENABLE(a1) est vrai, ENABLE(a2) l'est également. Il y a donc chaque fois un choix non déterministe entre les actions a1 et a2, choix qui peut toujours être résolu en faveur de U1. U2 peut donc ne jamais entrer en section critique bien qu'il soit toujours présent à son entrée chaque fois que la ressource se libère. Ce comportement est non équitable, ainsi que tout comportement infini où l'un des deux utilisateurs n'a accès à la ressource qu'un nombre fini de fois.

4.6.2 - Discussion et principe

Les deux exemples précédents mettent en évidence le problème de la non équitabilité des systèmes de transitions utilisés pour modéliser des systèmes réels dont on peut presque toujours supposer qu'ils sont équitables. En effet, l'équitabilité des systèmes informatiques réels est en général assurée par la statégie de résolution du non-déterminisme, implémentée au moyen de files d'attente ou d'une technique de datation privilégiant les requêtes les plus anciennes. Les systèmes non-informatiques dans lesquels la résolution du non determinisme est aléatoire (réellement) le sont également (du fait de la loi des grands nombres).

Principe

L'équitabilité c'est : tout évènement possible infiniment souvent doit être exécuté infiniment souvent.

4.6.3 - Notion de séquence d'exécution équitable

Dans les systèmes de transitions chaque comportement est caractérisé par une séquence d'exécution. On pourra donc parler d'une séquence d'exécution équitable ou non.

Définition

Une séquence d'exécution infinie s = q0q1 ... qk ... est OOO équitable ssi il existe un état qj figurant un nombre infini de fois dans s tel que :
il existe un état q', successeur direct de qj (qj->q') tel que la sous-séquence qjq' ne figure pas un nombre infini de fois dans s.

On notera EXFq l'ensemble des séquences équitables à partir de q.

Remarque

Les systèmes de transitions manipulés par le système CESAR ont un nombre fini d'états, donc chaque état a un nombre fini de successeurs.

Propriété

Pour tout système de transitions S ayant un nombre fini d'états, pour tout état q de S, EXFq n'est jamais vide.

Démonstration

Si q est élément de |POT(SINK)| alors il existe une séquence finie (donc équitable) à partir de q.

Dans le cas contraire, on peut construire une séquence s infinie à partir de q de la façon suivante :
* pour chaque état q' de S atteignable à partir de q on numérote les sucesseurs de q' (en nombre fini),
* chaque fois que l'on choisit un successeur q" de q' on mémorise le numéro du successeur choisi,
* pour choisir un successeur q" de q' on prend celui de numéro n+1 (modulo le nombre de successeur de q') si n est le numéro de celui choisi au précédent passage à q' (l'initialisation des numéros mémorisés peut être quelconque).

La séquence construite ainsi est manifestement telle que si un état q' figure un nombre infini de fois dans s, toutes les sous-séquences q'q" pour tous les successeurs q" de q' également.

Corolaire

Pour toute séquence non équitable s, pour tout k entier, le préfixe s(0) ... s(k) de longueur k+1 de s peut être prolongé de façon à obtenir une séquence s' équitable de même préfixe de longueur k+1 que s.

Démonstration : Application du résultat précédent à l'état s(k).

Remarque : On a montré dans [QS 82c] que cette propriété est également vraie pour les systèmes ayant un nombre infini (dénombrable) d'états.

4.6.4 - L'opérateur d'atteignabilité équitable FAIR

Tous les opérateurs temporels que nous avons introduits jusqu'ici sont définis en considérant l'ensemble EXq des séquences d'exécution à partir d'un état q, qu'elles soient équitables ou non. Ils ne prennent donc pas en compte l'éventuelle équitabilité du système décrit. Nous sommes donc amenés à introduire des opérateurs définis en considérant uniquement l'ensemble EXFq des séquences équitables à partir de q.

L'opérateur temporel unaire FAIR est utilisé pour exprimer qu'un prédicat doit devenir vrai au cours de l'évolution du système décrit, si celui-ci est équitable. Son interprétation est définie par :

|FAIR (f)| (q) ≡
pour toute séquence d'exécution s de EXFq,
il existe k entier tel que :
q-k->s(k) et |f| (s(k)) = vrai.

L'opérateur FAIR est donc analogue à l'opérateur INEV, mais on ne considère pour l'évaluer que les comportements équitables du système décrit.

Propriété : pour tout f :
INEV (f) => FAIR (f).

Démonstration : Pour tout état q, EXFq est inclus dans EXq et n'est pas vide.

Remarque :

Il est inutile de définir un opérateur analogue à POT mais ne considérant que les comportements équitables, car il serait identique à POT. Notons POT' cet éventuel opérateur. On a, pour tout f :
POT' (f) => POT (f)
puisque pour tout q, EXFq est inclus dans EXq.

Supposons que la réciproque ne soit pas vraie. Alors il existe un état q tel que :
POT(f)(q) = vrai et POT'(f)(q) = faux.
D'après la première hypothèse il existe une séquence non équitable s à partir de q telle qu'il existe k entier tel que :
q-k->s(k) et |f|(s(k)).
On peut donc prendre le préfixe q ... s(k) de s et le prolonger de façon équitable, ce qui nous donne une séquence équitable à partir de q permettant d'atteindre f. Contradiction.

On définit également l'opérateur conditionel FAIR de la façon suivante :
|FAIR [f1](f2)|(q) ≡
pour toute séquence d'exécution s de EXFq,
il existe k entier tel que :
q-k->s(k) et |f2|(s(k)) et
pour tout i de 0 à k-1 : |f1|(s(i)).

4.6.5 - Approximations de l'opérateur FAIR.

Proposition 1

Il n'est pas possible d'exprimer l'opérateur unaire FAIR à l'aide des autres opérateurs temporels unaires (et des opérateurs booléens).

Démonstration :

Considérons le système de transitions représenté par le graphe d'états de la figure 3.

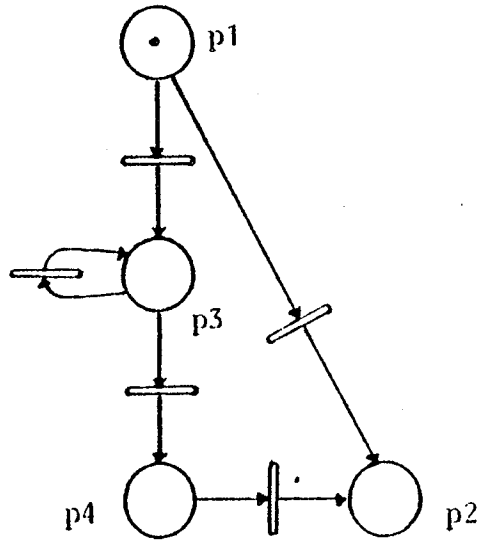


Figure 3

L'ensemble des états est $Q = \{x1, x2, x3, x4\}$ où xi est l'état où la place pi est marquée et les autres non.

S'il est possible d'exprimer FAIR à partir des opérateurs temporels unaires et des opérateurs booléens, on doit pouvoir construire le prédicat FAIR(f) à l'aide des opérateurs POT et INEV, des opérateurs booléens NOT et AND, de la constante FALSE et du prédicat f lui-même.

Prenons f tel que |f| = {x4}. Alors |FAIR(f)| = {x3, x4}.

- Par application des opérateurs booléens à {x4}, on obtient les ensembles :
  ∅, {x4}, {x1, x2, x3}, Q.
- L'application de POT à ces ensembles donne respectivement :
  ∅, {x1, x3, x4}, Q, Q.
- L'application de INEV aux mêmes ensembles donne respectivement :
  ∅, {x4}, Q, Q.
- On a donc obtenu les ensembles :
  ∅, {x4}, {x1, x2, x3}, {x1, x3, x4}, Q.
- Par application des opérateurs booléens, on obtient les ensembles :
  ∅, {x2}, {x4}, {x1, x3}, {x2, x4}, {x1, x2, x3}, {x1, x3, x4}, Q.
- L'application de POT à ces ensembles donne respectivement :
  ∅, {x1, x3, x4}, Q, {x1, x3}, Q, Q, {x1, x3, x4}, Q.
- L'application de INEV aux mêmes ensembles donne respectivement :
  ∅, {x4}, {x2, x4}, {x1, x3}, {x2, x4}, Q, {x1, x3, x4}, Q.

Aucun nouvel ensemble n'est apparu, il n'est donc pas possible d'exprimer {x3, x4} de cette façon.

Proposition_2

Pour tout f :

(INEV (f) OR ALL POT (f)) => FAIR (f)

et FAIR (f) => (INEV (f) OR SOME POT (f)).

Démonstration_:

a) (INEV (f) OR ALL POT (f)) => FAIR (f).

On sait que INEV (f) => FAIR (f).

Supposons que |ALL POT (f)| ne soit pas inclus dans |FAIR (f)|. Alors il existe un état q tel que :

ALL POT (f) (q) = vrai et FAIR (f) (q) = faux.

Donc il existe une séquence équitable s à partir de q dont tous les états sont dans |NOT f|. Comme q vérifie ALL POT (f) tous les états de s vérifient POT (f), donc s est infinie (il n'est pas possible de se bloquer dans |NOT f|). L'ensemble des états étant fini, cette séquence passe infiniment souvent par des états permettant d'atteindre |f|, et puisque les choix permettant d'atteindre |f| ne sont jamais faits, cette séquence est non équitable. Contradiction.

b) FAIR (f) => (INEV (f) OR SOME POT (f)).
Supposons que pour un état q :
FAIR (f) (q) = vrai et INEV (f) (q) = faux.
Cela signifie qu'il existe une séquence non équitable s à partir de q n'atteignant pas |f|. Tout préfixe de s pouvant être prolongé de façon à obtenir une séquence équitable s', et toute séquence équitable à partir de q devant atteindre |f|, cela signifie que tous les états de s vérifient POT (f). Donc q vérifie SOME POT (f).

Exemple :

Dans l'exemple utilisé pour démontrer la proposition 1, on a, avec f tel que |f| = {x4} :

$$|INEV\ (f)| = \{x4\}$$
$$|FAIR\ (f)| = \{x3, x4\}$$
$$|POT\ (f)| = \{x1, x3, x4\}$$
$$|ALL\ POT\ (f)| = \emptyset$$
$$|SOME\ POT\ (f)| = \{x1, x3\}.$$

On a donc les inclusions strictes entre :

$$|INEV\ (f)\ OR\ ALL\ POT\ (f)| = \{x4\}$$
$$|FAIR\ (f)| = \{x3, x4\}$$
$$|INEV\ OR\ SOME\ POT\ (f)| = \{x1, x3, x4\}.$$

Proposition 3

Pour tout f1, f2 :
(INEV[f1](f2) OR ALL POT[f1](f2)) => FAIR[f1](f2)
et FAIR[f1](f2) => (INEV[f1](f2) OR SOME POT[f1](f2)).

Démonstration : Similaire à la démonstration précédente, en ne considérant que les séquences dont tous les éléments (avant d'atteindre |f2|) vérifient f1, puisque FAIR[f1](f2), POT[f1](f2) et INEV[f1](f2) expriment des atteignabilités en restant dans |f1|.

4.6.6 - Caractérisation exacte de l'opérateur FAIR

Proposition 4

Pour tout f :
FAIR (f) = ALL [NOT f] (POT (f)).

Démonstration :

a) ALL[NOT f](POT(f)) => FAIR(f).
Supposons que ALL[NOT f](POT(f)) n'implique pas FAIR(f). Alors il existe un état q tel que :
ALL[NOT f](POT(f))(q)=vrai et FAIR(f)(q)=faux.
Donc il existe une séquence équitable s à partir de q dont tous les états sont dans |NOT f|. Comme q vérifie

ALL[NOT f](POT(f)), tous les états de s vérifient POT(f), et s est infinie. Donc s est non équitable. Contradiction.

b) FAIR(f) => ALL[NOT f](POT(f)).
Remarquons que :

INEV(f) => ALL[NOT f](POT(f))

puisque tant que |f| n'est pas atteint il doit être possible de l'atteindre.

Supposons donc que pour un état q :

FAIR(f)(q)=vrai et INEV(f)(q)=faux.

Cela signifie que toutes les séquences à partir de q
- ou bien sont équitables et finissent par passer par |f|. Les états de telles séquences sont donc dans |POT(f)| tant que |f| n'est pas atteint ;
- ou bien sont non équitables, et dans ce cas sont des séquences infinies dans |POT(f)| (puisque tout préfixe de longueur quelconque d'une séquence non équitable peut être prolongé de façon équitable, et dans ce cas on doit obtenir une séquence passant par |f| d'après la 1ère hypothèse).

Donc toutes les séquences à partir de q restent dans |POT(f)| tant que |f| n'est pas atteint. Donc q vérifie ALL[NOT f](POT(f)).

Proposition_5

Pour tout f1, f2 :

FAIR [f1] (f2) = ALL [NOT f2] (POT [f1] (f2)).

Démonstration_: Similaire à la démonstration précédente, en ne considérant que les séquences dont tous les états (avant d'atteindre |f2|) sont dans |f1|.

5 - LE PROTOCOLE DU BIT ALTERNE

Dans cette section nous avons regroupé un certain nombre de propriétés devant être vérifiées par le protocole du bit alterné. La plupart ne s'expriment pas de la même façon dans les deux versions du protocole, et leurs expressions ne sont d'ailleurs pas toujours équivalentes car dans la deuxième version on ne peut pas dissocier la valeur des bits de contrôle émis des actions d'émission proprement dite, ce qui oblige parfois à exprimer plus de choses que désiré.

De plus nous n'avons pas cherché à être exhaustif, ni à éviter des redondances. La recherche d'un ensemble minimal et complet de propriétés caractérisant le fonctionnement d'un protocole (ou d'une application quelconque) est un problème que nous n'avons pas abordé, et qui reste donc posé. Il est lié au problème de l'équivalence dans la logique temporelle induite par le programme de description, c.a.d. en tenant compte des propriétés implicites du modèle (invariants dus à la structure du contrôle et propriétés des données), problème pour lequel la connaissance d'une procédure de décision dans la logique seule est insuffisante.

Sans donc avoir cherché à comparer les propriétés énoncées ici, nous les avons classées en trois grandes sections :

- propriétés invariantes,
- propriétés de vivacité,
- propriétés de réponse aux actions.

Cette façon de procéder a un intérêt méthodologique, et nous pensons que c'est un bon guide à suivre pour une recherche plus systématique des propriétés.

5.1 - Propriétés invariantes

Les propriétés invariantes étant essentiellement des propriétés sur les données, elles ne sont intéressantes que pour la première version du protocole.

5.1.1 - Correction des interfaces avec la ligne de transmission des messages

```
* [] (AFTER(SEND, REPEAT) =>
                    (SEND_TO_RECEIVE.MMM.B = SENDER.BIT))
```

Après l'émission (ou la répétition) d'un message, la valeur du bit de contrôle du message reçu par SEND_TO_RECEIVE est égale au bit BIT de l'émetteur (SENDER).

* [] (AFTER(RECEIVE) =>
(RECEIVER.RECEIVED_MSG.B = SEND_TO_RECEIVE.MMM.B))

Après la réception d'un message, la valeur du bit de contrôle du message reçu par le récepteur (RECEIVER) est égale au bit de contrôle du message transmis par SEND_TO_RECEIVE.

5.1.2 - Correction_des_interfaces_avec_la_ligne_de transmission_des_acquittements

* [] (AFTER(SEND_ACK, REPEAT_ACK) =>
(RECEIVE_TO_SEND.AAA.B = NOT RECEIVER.BIT))

Après émission (ou répétition) d'un acquittement, la valeur du bit de contrôle de l'acquittement reçu par RECEIVE_TO_SEND est la valeur complémentaire du bit BIT du récepteur (qui a changé puisque ce dernier attend le message suivant).

* [] (AFTER(RECEIVE_ACK) =>
(SENDER.RECEIVED_ACK.B = RECEIVE_TO_SEND.AAA.B))

Après la réception d'un acquittement, la valeur du bit de contrôle de l'acquittement reçu par l'émetteur est égale au bit de contrôle de l'acquittement transmis par RECEIVE_TO_SEND.

5.2 - Propriétés_de_vivacité

5.2.1 - Absence_de_blocage (Versions 1 et 2)

* [] ENABLE

Absence de blocage total définitif.

* [] POT ENABLE(SENDER)

Absence de blocage définitif de l'émetteur.

* [] POT ENABLE(RECEIVER)

Absence de blocage définitif du récepteur.

5.2.2 - Vivacité des actions principales du protocole

Par actions principales, nous entendons les actions d'émettre et de recevoir de nouveaux messages (qui sont la finalité même du protocole).

a) Version_1

* [] POT ENABLE(SEND)

L'action d'émettre un nouveau message est vivante.

* [] POT ENABLE(ACCEPT)

L'action d'accepter un message reçu comme un nouveau message est vivante.

b) Version_2

* [] (POT ENABLE(SEND0) AND POT ENABLE(SEND1))

Les actions d'émettre de nouveaux messages avec un bit de contrôle à 0 ou à 1 sont toutes deux vivantes.

* [] (POT ENABLE(RECEIVE0) AND POT ENABLE(RECEIVE1))

Les actions de recevoir de nouveaux messages avec un bit de contrôle à 0 ou à 1 sont toutes deux vivantes.

5.2.3 - Fonctionnement périodique (Versions 1 et 2)

* [] FAIR INIT

Le système repasse périodiquement par son état initial (uniquement si ses comportements sont équitables ; cette propriété n'est pas vérifiée avec l'opérateur INEV).

5.3 - Propriétés de réponse aux actions

5.3.1 - Séquencement des actions de l'émetteur

a) Version_1

```
* [] (AFTER(SEND, REPEAT) => INEV BEFORE(RECEIVE_ACK))
```

Après avoir émis un message, on attend un acquittement.

```
* [] ((AFTER(RECEIVE_ACK) AND
                    (SENDER.RECEIVED_ACK.B /= SENDER.BIT)
          => INEV (AFTER(SKIP_ACK) AND BEFORE(REPEAT) AND
                                        NOT ENABLE(SEND)))
```

Si le bit de contrôle de l'acquittement reçu n'est pas égal à celui attendu, on saute l'acquittement, on se prépare à réémettre le message, et on ne permet pas l'émission d'un nouveau message.

```
* [] ((AFTER(RECEIVE_ACK) AND
                    (SENDER.RECEIVED_ACK.B = SENDER.BIT)
          => INEV (AFTER(ACCEPT_ACK) AND BEFORE(SEND))
```

Si le bit de contrôle de l'acquittement reçu est égal au bit attendu, on accepte l'acquittement et on se prépare à émettre un nouveau message.

b) Version_2

```
* [] (AFTER(SEND0, REPEAT0) =>
                    INEV BEFORE(RECEIVE_ACK1, SKIP_ACK0))
  [] (AFTER(SEND1, REPEAT1) =>
                    INEV BEFORE (RECEIVE_ACK1, SKIP_ACK0))
```

Après avoir émis un message, on attend un acquittement, que l'on sautera s'il n'a pas le même bit de contrôle que le message émis.

```
* [] (AFTER(RECEIVE_ACK0) => INEV BEFORE(SEND1))
  [] (AFTER(RECEIVE_ACK1) => INEV BEFORE(SEND0))
```

Après avoir reçu l'acquittement attendu, on se prépare à émettre un nouveau message (avec le bit complémentaire).

```
* [] (AFTER(SKIP_ACK0) => INEV (BEFORE(REPEAT1) AND
                                NOT ENABLE(SEND0, SEND1)))
  [] (AFTER(SKIP_ACK1) => INEV (BEFORE(REPEAT0) AND
                                NOT ENABLE(SEND0, SEND1)))
```

Après avoir sauté un acquittement non attendu, on se prépare à répéter le message précédemment émis, mais on ne peut pas émettre un nouveau message (quel que soit son bit de contrôle).

5.3.2 - Séquencement des actions du récepteur

a) Version_1

```
* [] (AFTER(SEND_ACK, REPEAT_ACK) => INEV BEFORE(RECEIVE))
```

Après avoir émis un acquittement, on attend un message.

```
* [] ((AFTER(RECEIVE) AND
                (RECEIVER.RECEIVED_MSG.B /= RECEIVER.BIT)
          => INEV (AFTER(SKIP) AND BEFORE(REPEAT_ACK)))
```

Si le bit de contrôle du message reçu n'est pas égal à celui attendu, on saute le message et on se prépare à réémettre l'acquittement précédent (avec le bit complémentaire).

```
* [] ((AFTER(RECEIVE) AND
                (RECEIVER.RECEIVED_MSG.B = RECEIVER.BIT)
          => INEV (AFTER(ACCEPT) AND BEFORE(SEND_ACK)))
```

Si le bit de contrôle du message reçu est égal au bit attendu, on accepte le message et on se prépare à émettre l'acquittement correspondant.

b) Version_2

```
* [] (AFTER(SEND_ACK0, REPEAT_ACK0) =>
                                INEV BEFORE(RECEIVE1, SKIP0))
  [] (AFTER(SEND_ACK1, REPEAT_ACK1) =>
                                INEV BEFORE(RECEIVE0, SKIP1))
```

Après avoir émis un acquittement, on attend un message que l'on sautera s'il n'a pas le bit de contrôle attendu (le complémentaire de celui de l'acquittement émis).

```
* [] (AFTER(RECEIVE0) => INEV BEFORE(SEND_ACK0))
  [] (AFTER(RECEIVE1) => INEV BEFORE(SEND_ACK1))
```

Après avoir reçu un nouveau message, on se prépare à émettre l'acquittement correspondant.

```
* [] (AFTER(SKIP0) => INEV BEFORE(REPEAT_ACK0))
  [] (AFTER(SKIP1) => INEV BEFORE(REPEAT_ACK1))
```

Après avoir reçu un message dupliqué, on se prépare à réémettre l'acquittement correspondant.

5.3.3 - Fonctionnement correct du protocole

Pour éviter de considérer les comportements non équitables où la ligne perd tous les messages, on emploiera l'opérateur FAIR, chaque fois où l'on est tenté d'utiliser l'opérateur INEV.

a) Version_1

```
* [] (AFTER(SEND) => FAIR AFTER(ACCEPT))
```

Tout nouveau message émis finit par provoquer la réception d'un message (reconnu comme nouveau).

```
* [] (AFTER(SEND) =>
                  ALL [NOT AFTER(ACCEPT)] (NOT ENABLE(SEND)))
```

Tant qu'un message émis n'a pas été reçu il n'est pas possible d'en émettre un autre (nouveau).

```
* [] (AFTER(SEND) => FAIR [NOT ENABLE(SEND)] (AFTER(ACCEPT)))
```

Tout message émis finit par être reçu (sans qu'il y ait émission de nouveaux messages entre temps).

b) Version_2

```
* [] (AFTER(SEND0) => FAIR AFTER(RECEIVE0))
  [] (AFTER(SEND1) => FAIR AFTER(RECEIVE1))
```

Tout message émis finit par provoquer une réception (avec le même bit de contrôle).

```
* [] (AFTER(SEND0) =>
       ALL [NOT AFTER(RECEIVE0)] (NOT ENABLE(SEND0, SEND1)))
  [] (AFTER(SEND1) =>
       ALL [NOT AFTER(RECEIVE1)] (NOT ENABLE(SEND0, SEND1)))
```

Tant qu'un message émis n'a pas été reçu, il n'est pas possible d'en émettre un autre (quel que soit son bit de contrôle).

```
* [] (AFTER(SEND0) =>
          FAIR [NOT ENABLE(SEND0, SEND1)] (AFTER(RECEIVE0)))
  [] (AFTER(SEND1) =>
          FAIR [NOT ENABLE(SEND0, SEND1)] (AFTER(RECEIVE1)))
```

Tout message émis finit par être reçu (sans qu'il y ait d'autre émission entre temps).

6 - EVALUATION DES OPERATEURS TEMPORELS

Etant donné d'une part un programme de description d'une application, représenté par un RdPI étiqueté, et d'autre part un ensemble de formules temporelles exprimant les propriétés souhaitées pour cette application, le système CESAR doit répondre (ou au moins s'il ne le peut pas, fournir des éléments de réponse) à la question : quelles sont parmi ces formules celles que l'application décrite vérifie, et celles qu'elle ne vérifie pas ?

L'approche adoptée dans le système CESAR, due à J. SIFAKIS ([SIF 79] [SIF 82]), présente l'originalité de caractériser les opérateurs temporels comme des points fixes de transformateurs de prédicats monotones construits à partir d'un même transformateur élémentaire de prédicats, la fonction PRE, et fournit ainsi une méthode d'évaluation par calcul itératif de ces opérateurs.

Pour plus de généralité, nous raisonnerons sur le modèle Système de Transitions.

Etant donné un système de transitions S = (Q, ->), l'ensemble des prédicats sur Q peut être identifié avec le treillis PQ des parties de Q, muni des opérateurs d'union (u), d'intersection (n) et de complémentation (¯).

Un transformateur de prédicats est une application de PQ dans lui-même. Etant donné deux transformateurs de prédicats f et g, on peut définir les transformateurs de prédicats f u g, f n g, ¯f, ¯f¯, [f]n et Id tels que, pour tout X de PQ :

$(f \cup g)(X) \equiv f(X) \cup g(X)$
$(f \cap g)(X) \equiv f(X) \cap g(X)$
$\bar{f}(X) \equiv \overline{(f(X))}$
$\bar{f}^{-}(X) \equiv \overline{f(\bar{X})}$
$Id(X) \equiv X$

et pour tout $k \geq 0$:

$[f]^{k+1}(X) \equiv f([f]^{k}(X))$
avec $[f]^{0} \equiv Id$.

On introduit également les notations suivantes :
$[f]^{*} \equiv Id \cup f \cup [f]^{2} \cup \ldots \cup [f]^{k} \cup \ldots$
$[f]^{\times} \equiv Id \cap f \cap [f]^{2} \cap \ldots \cap [f]^{k} \cap \ldots$.

6.1 - Les transformateurs de prédicats PRE et ~PRE~

Définition :

Etant donné un système de transition S = (Q, ->), on définit le transformateur de prédicats PRE tel que, pour tout X de PQ, pour tout q de Q :

PRE(X)(q) ≡ il existe q' tel que q -> q' et X(q')

c.a.d. PRE(X) est l'ensemble des états à partir desquels il est possible d'atteindre un état de X en exécutant une transition de S.

On utilisera également le transformateur de prédicats ~PRE~ dual de PRE, c.a.d. tel que :

~PRE~(X)(q) ≡ pour tout q' : q -> q' => X(q')

c.a.d. ~PRE~(X) est l'ensemble des états à partir desquels il n'est pas possible d'atteindre des états de ~X en exécutant une transition de S.

Notons ENABLE et SINK les prédicats sur Q tels que :
ENABLE (q) ≡ il existe q' : q->q',
SINK (q) ≡ ~ ENABLE (q).

Propriétés :

Les propriétés suivantes ont été démontrées dans [SIF 79]. Pour tout système de transitions S = (Q, ->) :

a) PRE(∅) = ∅
~PRE~(Q) = Q ;

b) PRE(Q) = ENABLE
~PRE~(∅) = SINK ;

c) pour tout X :
PRE(X) est inclus dans ENABLE
~PRE~(X) contient SINK ;

d) pour tout X :
(PRE ∩ ~PRE~)(X) = ~PRE~(X) ∩ ENABLE
(PRE ∪ ~PRE~)(X) = PRE(X) ∪ SINK ;
Cette propriété est illustrée par la figure 1.

e) PRE est distributif par rapport à l'union
~PRE~ est distributif par rapport à l'intersection ;

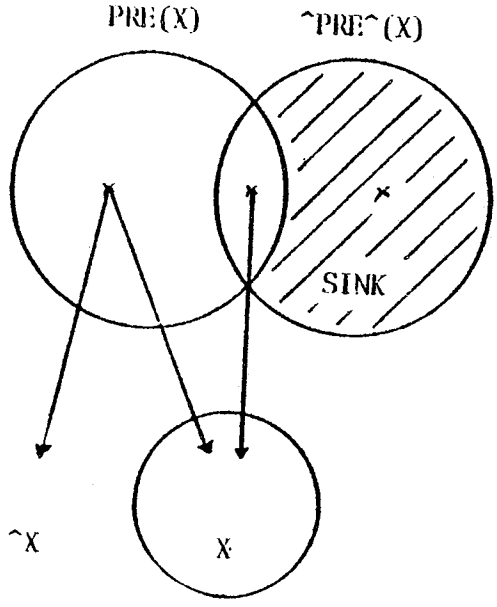


Figure 1

f) PRE et ^PRE^ sont monotones, c.a.d. si X1 est inclus dans X2, alors PRE(X1) est inclus dans PRE(X2) (idem pour ^PRE^).

6.2 - Invariants et trajectoires

Les notions d'invariants et trajectoires introduites informellement lors de la discussion sur les opérateurs temporels ALL, ALW, SOME et SONT peuvent être définies comme suit.

Définitions :

Etant donné un système de transitions S = (Q, ->), et C un prédicat sur Q,

- un invariant conditionnel (sous la condition C) est un prédicat X sur Q tel que, pour tout q, q' de Q :
  X(q) et C(q) et q->q' implique X(q') ;

- une trajectoire conditionnelle (sous la condition C) est un prédicat W sur Q tel que, pour chaque q de Q :
  W(q) et C(q) et ENABLE(q) implique
  il existe q' de Q tel que (q->q' et W(q')).

Un invariant (resp. trajectoire) est un invariant conditionnel (resp. trajectoire conditionnelle) dont la condition est identiquement vraie. Un invariant (resp. invariant conditionnel) sans blocage est un invariant (resp. invariant conditionnel) contenu dans le prédicat ENABLE. Une trajectoire (resp. trajectoire conditionnelle) sans fin est une trajectoire (resp. trajectoire conditionnelle) contenue dans le prédicat ENABLE.

Propositions :

Etant donné un système de transitions $S = (Q, \rightarrow)$, et C un prédicat sur Q :

a) X est un invariant ssi
$$X = X \cap \tilde{}PRE\tilde{}(X),$$

b) Y est un invariant sans blocage ssi
$$Y = Y \cap (PRE \cap \tilde{}PRE\tilde{})(Y),$$

c) W est une trajectoire ssi
$$W = W \cap (PRE \cup \tilde{}PRE\tilde{})(W),$$

d) V est une trajectoire sans fin ssi
$$V = V \cap PRE(V),$$

e) X est un invariant conditionnel (sous la condition C) ssi
$$X = X \cap (\tilde{}C \cup \tilde{}PRE\tilde{}(X)),$$

f) Y est un invarant conditionnel sans blocage (sous la condition C) ssi
$$Y = Y \cap (\tilde{}C \cup (PRE \cap \tilde{}PRE\tilde{})(X)),$$

g) W est une trajectoire conditionnelle (sous la condition C) ssi
$$W = W \cap (\tilde{}C \cup (PRE \cup \tilde{}PRE\tilde{})(W)),$$

h) V est une trajectoire conditionnelle sans fin (sous la condition C) ssi
$$V = V \cap (\tilde{}C \cup PRE(V)).$$

Démonstration : Les propositions a) à d) ont été démontrées dans [SIF 79]. Les quatres autres en sont les extensions conditionnelles, et se démontrent aisément.

6.3 - Rappel de résultats sur les points fixes des fonctions monotones

Nous rappelons ici des résultats connus sur les points fixes des fonctions monotones ([TAR 55] [PAR 69] [SIF 79]).

Définitions :

Une fonction monotone dans un treillis est dite u-continue si elle préserve les limites des suites croissantes, c.a.d. si P1, P2, ... est une suite croissante (Pi est inclus dans P(i+1)) de limite P, alors f(P) est la limite de la suite croissante f(P1), f(P2), ...

Dualement, f est dite n-continue si elle préserve la limite des suites décroissantes.

Propositions :

Soit f une fonction monotone dans un treillis PQ, et X0 un élément de PQ :

a) si f est u-continue et telle que X0 est inclus dans f(X0), alors [f]*(X0) est le plus petit point fixe de f contenant X0 ;

b) si f est n-continue et telle que f(X0) est inclus dans X0, alors [f]x(X0) est le plus grand point fixe de f contenu dans X0.

6.4 - Calcul itératif des opérateurs temporels

Nous avons vu que PRE (resp. ^PRE^) est distributif par rapport à l'union (resp. l'intersection) et donc PRE (resp. ^PRE^) est u-continu (resp. n-continu). SIFAKIS [SIF 79] a montré de plus que PRE (resp. ^PRE^) est n-continu (resp. u-continu) ssi la relation -> de S est à image finie c.a.d. pour chaque q de Q, il existe k entier tel que

$$|(q' \mid q \to q')| \leq k.$$

Les RdPI manipulés par le système CESAR sont dans ce cas (a fortiori puisqu'ils n'ont qu'un nombre fini d'états).

D'où les résultats suivants :

Propositions

Etant donné S = (Q, ->) un système de transitions, et C un prédicat sur Q :

a) [Id n ^PRE^]x (X0) = [^PRE^]x (X0) est le plus grand invariant contenu dans X0,

b) [Id n PRE n ^PRE^]x (Y0) est le plus grand invariant sans blocage contenu dans Y0,

c) [Id n (PRE u ¯PRE¯)]x (W0) est la plus grande trajectoire contenue dans W0,

d) [Id n PRE]x (V0) est la plus grande trajectoire sans fin contenue dans V0,

e) [Id n (¯C u ¯PRE¯)]x (X0) = [¯C u ¯PRE¯]x (X0) est le plus grand invariant conditionnel (sous la condition C) contenu dans X0,

f) [Id n (¯C u PRE n ¯PRE¯)]x (Y0) est le plus grand invariant conditionnel sans blocage (sous la condition C) contenu dans Y0,

g) [Id n (¯C u PRE u ¯PRE¯)]x (W0) est la plus grande trajectoire conditionnelle (sous la condition C) contenue dans W0,

h) [Id n (¯C u PRE)]x (V0) est la plus grande trajectoire conditionnelle sans fin (sous la condition C) contenue dans V0,

(la simplification effectuée en a) et e) est due à la distributivité de ¯PRE¯ par rapport à l'intersection).

Théorème 1 :

Etant donné un système de transitions S = (Q, ->) dont la relation -> est à image finie, C un prédicat sur Q, et f une formule temporelle de L :

a) |ALL(f)| = [¯PRE¯]x(|f|),
b) |ALW(f)| = [Id n PRE n ¯PRE¯]x(|f|),
c) |SOME(f)| = [Id n (PRE u ¯PRE¯)]x(|f|),
d) |SONT(f)| = [Id n PRE]x(|f|),
e) |ALL[C](f)| = [¯C u ¯PRE¯]x(|f|),
d) |ALW[C](f)| = [Id n (¯C u PRE n ¯PRE¯)]x(|f|),
f) |SOME[C](f)| = [Id n (¯C u PRE u ¯PRE¯)]x(|f|),
h) |SONT[C](f)| = [Id n (¯C u PRE)]x(|f|).

Démonstration (pour ALL) :

a) |ALL(f)| est un invariant : se déduit de la définition de l'interprétation de ALL(f) et de celle d'un invariant,

b) |ALL(f)| est inclus dans |f| : immédiat d'après sa définition,

c) |ALL(f)| est le plus grand invariant contenu dans |f| : par l'absurde en tenant compte des propriétés des quantificateurs universel et existenciel.

Théorème 2 (dual du théorème 1) :

Etant donné un système de transitions S = (Q, ->) dont la relation -> est à image finie, C un prédicat sur Q, et f une formule temporelle de L :

a) |POT(f)| = [PRE]*(|f|),
b) |WPOT(f)| = [Id u PRE u ¯PRE¯]*(|f|),
c) |INEV(f)| = [Id u (PRE n ¯PRE¯)]*(|f|),
d) |OBL(f)| = [Id u ¯PRE¯]*(|f|),
e) |POT[C](f)| = [C n PRE]*(|f|),
f) |WPOT[C](f)| = [Id u C n (PRE u ¯PRE¯)]*(|f|),
g) |INEV[C](f)| = [Id u C n PRE n ¯PRE¯)]*(|f|),
h) |OBL[C](f)| = [Id u C n ¯PRE¯]*(|f|).

Conséquence pratique :

On peut calculer itérativement l'interprétation des opérateurs temporels comme la limite des suites suivantes :

a1) ALL(f) : Xk+1 = Xk n ¯PRE¯(Xk)
b1) ALW(f) : Xk+1 = Xk n (PRE n ¯PRE¯)(Xk)
c1) SOME(f) : Xk+1 = Xk n (PRE u ¯PRE¯)(Xk)
d1) SONT(f) : Xk+1 = Xk n PRE(Xk)
e1) ALL[C](f) : Xk+1 = Xk n (¯C u ¯PRE¯(Xk))
f1) ALW[C](f) : Xk+1 = Xk n (¯C u (PRE n ¯PRE¯)(Xk))
g1) SOME[C](f) : Xk+1 = Xk n (¯C u (PRE u ¯PRE¯)(Xk))
h1) SONT[C](f) : Xk+1 = Xk n (¯C u PRE(Xk))

a2) POT(f) : Xk+1 = Xk u PRE(Xk)
b2) WPOT(f) : Xk+1 = Xk u (PRE u ¯PRE¯)(Xk)
c2) INEV(f) : Xk+1 = Xk u (PRE n ¯PRE¯)(Xk)
d2) OBL(f) : Xk+1 = Xk u ¯PRE¯(Xk)
e2) POT[C](f) : Xk+1 = Xk u C n PRE(Xk)
f2) WPOT[C](f) : Xk+1 = Xk u C n (PRE u ¯PRE¯)(Xk)
g2) INEV[C](f) : Xk+1 = Xk u C n (PRE n ¯PRE¯)(Xk)
h2) OBL[C](f) : Xk+1 = Xk u C n ¯PRE¯(Xk)

en initialisant avec : X0 = |f|.

6.5 - Réalisation pratique

Les résultats précédents fournissent une technique de calcul des opérateurs temporels dans tous les modèles où il existe une méthode d'évaluation de la fonction PRE. Dans le cas des réseaux de Petri interprétés, deux méthodes ont été étudiées (et implémentées), l'une opérant sur l'expression des prédicats, l'autre sur leurs ensembles caractéristiques. Ces méthodes sont en fait applicables à un grand nombre de modèles.

6.5.1 - Première méthode : calcul sur les prédicats

Cette méthode ne présuppose pas connu l'ensemble effectif des états atteints par le RdPI à partir du ou des états initiaux. De ce fait l'ensemble des marquages est considéré être (0, 1)n si le réseau a n places (rappelons qu'il est sauf), et l'ensemble des valeurs du vecteur X de l'interprétation est considéré être

$$D = D1 \times D2 \times \ldots \times Dk$$

s'il y a k composantes de X, chaque composante xi ayant Di pour domaine de définition déclaré.

On notera Y le vecteur (M, X). Tout état du RdPI est caractérisé par une valeur unique de Y. Les prédicats manipulés par le système CESAR se composant de prédicats sur X et de variables propositionnelles qui sont des prédicats sur M, ils peuvent être vu comme des prédicats sur Y.

Chaque transition ti est exécutable si la condition

$$ci(Y) = c1i(M) \cap c2i(X)$$

est vraie, où c1i est la condition sur M représentant le fait que ti est validée dans le RdP, et c2i est la condition sur X associée à la transition par l'interprétation.

De même l'action effectuée lors de la mise à feu de ti se compose d'une modification de M et d'une affectation sur X, et peut donc être représentée par une affectation (vectorielle) unique sur Y, de la forme

$$Y := ai(Y)$$

équivalente à

$$M := a1i(M),\ X := a2i(X).$$

Un prédicat P(Y) peut devenir vrai du fait de la mise à feu de ti ssi ci(Y) est vraie et P(ai(Y)) également. D'où l'expression de la fonction PRE (p étant le nombre de transitions du RdPI) :

$$PRE(P(Y)) = \text{union pour } i=1 \text{ à } p \text{ des } ci(Y) \cap P(ai(Y)).$$

L'expression de "PRE" se déduit de la précédente :

$$\text{"PRE"}(P(Y)) = \text{intersection pour } i=1 \text{ à } p \text{ des } \overline{ci}(Y) \cup P(ai(Y)).$$

De plus les prédicats ENABLE et SINK s'expriment de la façon suivante :

$$ENABLE = \text{union pour } i=1 \text{ à } p \text{ des } ci$$

$$SINK = \text{intersection pour } i=1 \text{ à } p \text{ des } \overline{ci}$$

ce qui permet de calculer :

$$(PRE \cap \text{"PRE"})(P) = \text{"PRE"}(P) \cap ENABLE$$

$$(PRE \cup \text{"PRE"})(P) = PRE(P) \cup SINK.$$

D'un point de vue pratique, on réduit le calcul sur les prédicats au calcul booléen, en codant chacune des variables (composantes de X) à l'aide de variables booléennes. Deux codes ont été envisagés :

- le codage dit "extensif" : une variable x pouvant prendre n valeurs est codée à l'aide de n variables booléennes x1, ... , xn, la variable xi représentant le fait que x a sa i-ème valeur ;
- le codage dit "compact" : les variables sont codées au moyen de leur représentation en binaire (les types énumérés ayant été transformés en intervalle d'entiers).

En ce qui concerne le vecteur M, chacune de ses composantes est déjà une variable booléenne (puisque le RdP est sauf).

Le premier codage bien qu'introduisant plus de variables permet des opérations plus simples. De plus on peut utiliser au cours des calculs des relations de la forme :

$$x1 + x2 + \ldots + xn = 1$$

exprimant le fait qu'une seule des variables xi est vraie à tout moment (représentant la valeur courante de x).

Une relation similaire peut être exprimée sur les composantes du vecteur M représentant les marquages des places appartenant à un même processus (chaque processus étant un graphe d'états où circule une seule marque).

D'un point de vue pratique, le codage extensif s'est révélé plus performant que le codage compact, essentiellement au niveau de la manipulation des prédicats, grâce à l'utilisation des relations indiquées ci-dessus. Quelle que soit le codage employé, cette méthode est limitée par le temps de traitement, qui croit exponentiellement avec le nombre de monômes de la fonction booléenne représentant le prédicat en cours de calcul.

Cette méthode est applicable à tout système de transitions dont chaque transition peut être représentée par un couple (condition, action). Ces systèmes sont parfois appelés <u>systèmes conditions-actions</u>.

Une première version de l'analyseur a été réalisée selon cette méthode (utilisant le codage extensif). Elle admet des systèmes conditions-actions booléens quelconques et utilise toutes les relations de la forme

$$x1 + x2 + \ldots + xn = 1$$

qu'on lui fournit. Elle ne calcule que les opérateurs ALL, SOME et SONT non conditionnels (les opérateurs POT, INEV et OBL sont calculés par dualisation).

Exemple

Dans l'exemple du protocole du bit alterné (Version 1) la vérification de la propriété

INIT => ALL POT ENABLE(SENDER)

conduit aux calculs suivants :

* ENABLE(SENDER) = s1m1 u s2m1 u s2a2 u s3.
(les variables si, mi, ai, ri représentent le fait que la place de même nom sur la figure 2 p. III-86 est marquée)

* POT(ENABLE(SENDER)) est calculé par dualisation comme NOT ALL(NOT ENABLE(SENDER)), ce qui conduit aux calculs suivants :

* NOT ENABLE(SENDER) = s1m2 u s2m2a1

* ALL (NOT ENABLE(SENDER)) est calculé comme la limite de Pk+1 = Pk n ⁻PRE⁻(Pk) avec

P0 = s1m2 u s2m2a1

On obtient successivement :

⁻PRE⁻(P0) = false,

P1 = P0 n ⁻PRE⁻(P0) = false,

donc ALL(NOT ENABLE(SENDER)) = false.

* POT ENABLE(SENDER) = true.

* ALL POT ENABLE(SENDER) = true.

* INIT => ALL POT ENABLE(SENDER) = true.

6.5.2 - Deuxième méthode : calcul sur les ensembles d'états

Cette méthode nécessite de connaître a priori l'ensemble des états effectivement atteints par le RdPI à partir du ou des états initiaux possibles, ainsi que la relation -> (en fait, d'un point de vue pratique, la relation inverse de ->, que nous noterons [->]-1). On doit donc faire précéder toute analyse des spécifications par un algorithme de simulation exhaustive afin de construire le graphe des états du système et la relation [->]-1.

L'expression de PRE est alors immédiate. Etant donné un prédicat P représenté par son ensemble caractéristique :

PRE(P) = {q / q->q' et q' élément de P),

c.a.d. ⁻PRE(P) = {q / q' [->]-1 q et q' élément de P).

L'ensemble SINK est déterminé au moment de la construction du graphe des états. Les autres fonctions utilisées sont calculées par complémentation :

$$ENABLE = \neg SINK$$
$$\neg PRE \neg (P) = \neg\ PRE(\neg P)$$
$$(PRE \cap \neg PRE \neg)(P) = \neg\ PRE(\neg P) - SINK$$
$$(PRE \cup \neg PRE \neg)(P) = PRE(P) \cup SINK.$$

Une deuxième version de l'analyseur a été réalisée selon cette méthode. Elle admet des systèmes conditions-actions à variables entières bornées (les types énumérés doivent être transformés en intervalles d'entiers). Cet analyseur est formé de deux parties :

- un "simulateur exhaustif" qui construit le graphe d'états du système à analyser. Il procède d'abord à une traduction du système conditions-actions fourni en une procédure en langage Pascal, qu'il utilise ensuite pour déterminer l'effet de chaque transition ;

- l'analyseur proprement dit, qui évalue sous forme d'ensembles d'états des formules parenthésées où peuvent figurer tous les opérateurs de la logique. Il reconnait les variables propositionnelles INIT, ENABLE et SINK (toutes trois sans arguments), ainsi que toute relation (exprimée par une condition) sur les variables du système conditions-actions, qu'il évalue par traduction vers des fonctions en langage Pascal avant de les utiliser.

Cette méthode présente l'inconvénient d'obliger à construire le graphe des états. Toutefois cette opération plutôt coûteuse n'est effectuée qu'une seule fois, et l'évaluation des différents opérateurs temporels par calcul itératif sur les ensembles caractéristiques des prédicats est ensuite beaucoup plus performante que par la première méthode. Cette méthode peut s'appliquer à tous les systèmes de transitions pour lesquels on peut construire aisément le graphe des états et la relation [->]-1. Une limite de cette méthode est la taille du graphe des états.

Exemple

Le protocole du bit alterné (Version 1) admet un graphe de 182 états, alors que l'ensemble des états possibles, compte tenu du domaine de définition des différentes variables, a un cardinal de 2304. La première méthode travaille en fait sur cet espace d'états potentiels plutôt que sur l'espace d'états réel. On voit donc le gain qu'il y a à construire le graphe d'états préalablement à l'analyse.

Pour la même propriété que précédemment, le prédicat ENABLE(SENDER) est représenté par un ensemble de 149 états et POT(ENABLE(SENDER)) est trouvé recouvrir la totalité de l'espace d'états dès la première itération.

7 - CONCLUSION

Dans ce chapitre on a présenté un langage pour la spécification des propriétés des systèmes, chaque propriété étant exprimée par une formule d'une logique temporelle. Cette logique est bâtie à partir :

- d'une part d'un ensemble de variables propositionnelles (INIT, END, ENABLE, BEFORE, IN et AFTER) dont l'interprétation est faite sur le RdPI généré à partir du programme de description. Cette partie du langage de spécification dépend donc étroitement du langage de description (chapitre II), du modèle RdPI retenu, et de la façon de le générer (chapitre III).

- d'autre part, d'un ensemble d'opérateurs temporels unaires et binaires, pouvant tous être exprimés à partir des opérateurs POT et INEV binaires. Leur interprétation est définie de façon générale en termes de systèmes de transitions, un modèle plus primitif que les RdPI. De ce fait, ces opérateurs définissent une logique d'intérêt général, qui peut être appliquée avec tout modèle dont on peut exprimer simplement la sémantique en termes de systèmes de transitions (à condition de définir un ensemble de variables propositionnelles adéquates).

Cette logique s'inscrit dans la lignée des travaux sur le temps arborescent [ABR 79] [CE 81] [BMP 81] par opposition à l'approche temps linéaire [PNU 77] [GPSS 80] qui a déjà été utilisée pour la spécification des propriétés des protocoles [SMS 81].

Nous pensons que la logique en temps arborescent présente deux avantages :

- d'une part elle permet de différencier les modalités "il est possible que" (POT) et "il est inévitable que" (INEV) [LAM 80], et par suite les modalités WPOT et OBL (dont les duaux surtout sont intéressants), alors que la logique en temps linéaire ne le permet pas.

- d'autre part, donnant la possibilité de raisonner en termes d'ensembles d'états plutôt qu'en termes de séquences, elle permet de résoudre le problème de l'équitabilité posé par le non déterminisme des modèles utilisés pour représenter les systèmes parallèles. Les résultats présentés ici sont exposés dans le cadre du projet CESAR où les systèmes étudiés ont un nombre fini d'états, mais ils sont développés dans le cas des systèmes ayant un nombre infini (dénombrable) d'états dans [QS 82c].

L'originalité de notre approche réside à notre avis dans trois points :

- utilisation de la logique temporelle pour la spécification des propriétés des systèmes, en relation avec un langage de description de ces systèmes,
- résolution du problème de l'équitabilité (dans le cadre des systèmes de transitions où parallélisme et non déterminisme sont confondus),
- utilisation pratique de la caractérisation en points fixes des opérateurs temporels pour la vérification des propriétés spécifiées (vérification effectuée automatiquement par la machine sur un modèle lui-même généré automatiquement, par opposition avec certaines "méthodes de preuves"" totalement manuelles et reposant sur l'intuition du "prouveur").

Chapitre V

CONCLUSION
BILAN ET PERSPECTIVES

1. Bilan de l'expérience
2. Propositions pour un véritable prototype
3. Extensions envisageables à plus long terme
4. Conclusion

1 - BILAN DE L'EXPERIENCE

Le système CESAR est un système ambitieux, mettant en oeuvre des techniques diverses, certaines bien connues (e.g. compilation), d'autres moins classiques (e.g. calcul itératif de prédicats), et dont certains des fondements théoriques sont récents (e.g. logique temporelle conditionnelle). Il était donc nécessaire de tester la validité de l'approche et la faisabilité des algorithmes qu'elle implique avant de procéder à une réalisation complète du système CESAR. Cette tâche (et la réflexion théorique correspondante) a mobilisé l'activité d'un "groupe CESAR" au sein de l'équipe "Langages" de l'IMAG, groupe formé principalement de Patrick Martinet, Jean-Pierre Queille, Jean-Philippe Schwartz et Joseph Sifakis, et auquel ont collaboré à des titres divers d'autres personnes (Jean-Pierre Charbonnier, Susanne Graf, Jean-Charles Marty, Marc Rozier, Jacques Voiron).

Tout le travail de programmation a été réalisé en Pascal sur le système Multics du CICG.

Nous avons considéré que la compilation du langage CESAR dans sa totalité n'était pas l'objet essentiel des travaux à mener pour cette tâche de validation préalable, car ne soulevant que peu de problèmes théoriques ou pratiques (gestion des étiquettes, substitution des procédures), par rapport à l'approche déjà réalisée dans [QUE 78a]. C'est pourquoi nous avons défini un sous-langage du langage CESAR, ayant la même expressibilité que celui-ci, mais allégé d'un certain nombre de fioritures syntaxiques pratiques mais non indispensables (e.g. instructions FOR et CASE), et de certains concepts non essentiels (e.g. packages) [QUE 82]. Une version expérimentale de son compilateur a été réalisée (analyse syntaxique, génération des réseaux correspondant aux tâches élémentaires, composition, élimination des variables non-significatives [RS 81] [CHA 82]). Quoique simplifié par certains cotés (traitement d'erreurs sommaire, par exemple), ce compilateur est assez évolué pour être démonstratif. Il permet de traiter les types de tâches et les tableaux de tâches et de variables échangées, ainsi que la compilation séparée des types de tâches.

En ce qui concerne l'analyse des formules de spécification, deux maquettes d'analyseur ont été réalisées. La première (cf IV-6.5.1) admet des systèmes conditions-actions booléens, et ne calcule que les opérateurs ALL, SOME et SONT non conditionnels (par calcul littéral sur des expressions booléennes). Elle utilise toutes les relations de la forme

$$x1 + x2 + \ldots + xn = 1$$

qu'on lui fournit pour simplifier les calculs qu'elle effectue. L'interprétation des formules parenthésées est laissée à la charge de l'utilisateur qui doit guider le

calcul pas à pas. Le codage des RdPI générés par le compilateur en systèmes conditions-actions booléens a été étudié ([MA 81]) mais n'a pas été réalisé.

La deuxième version (cf IV-6.5.2) de l'analyseur admet des systèmes conditions-actions à variables entières bornées. Elle est formée d'un programme de construction du graphe d'états du système à analyser et de l'analyseur proprement dit, qui évalue sous forme d'ensembles d'états des formules parenthésées où peuvent figurer tous les opérateurs de la logique (conditionnels ou non). Il reconnaît les variables propositionnelles INIT, ENABLE et SINK (toutes trois sans arguments), ainsi que toute relation sur les variables du système conditions-actions.

Dans les deux versions, l'interprétation des variables propositionnelles INIT avec arguments, END avec ou sans arguments, ENABLE ou SINK avec arguments, et BEFORE, IN ou AFTER en termes de prédicats sur le système conditions-actions analysé n'a pas été effectuée. Toutefois ce qui a été réalisé est suffisant pour comparer les deux techniques d'analyse :

- la première pose essentiellement un problème de temps de calcul, qui croit exponentiellement avec le nombre de monômes de la fonction booléenne représentant le prédicat que l'on est en train de calculer. Cette solution occupe par contre assez peu de place mémoire, et on peut donc envisager son emploi sur des matériels où la place disponible est faible, mais où le temps de calcul n'est pas critique ;

- la deuxième méthode paraît pénalisée du fait de l'obligation de construire le graphe d'états préalablement à l'analyse. En fait cette opération pose plus un problème de place que de temps, et n'est de toute façon exécutée qu'une fois avant l'évaluation de toutes les formules de spécification. De plus, la connaissance préalable du graphe d'états rend le calcul des formules en termes d'ensembles d'états très nettement plus performant que dans la première version (bien que des mesures systématiques n'aient pu être effectuées, on a constaté que le rapport des temps d'exécution variait entre 1/2 et 1/60 suivant les formules du protocole du bit alterné à prouver, et que le temps de génération du graphe est lui-même faible devant le temps de démonstration d'une seule formule dans la première version). Sur un matériel tel que celui que nous avons utilisé, où la contrainte de place n'est pas très importante mais où le temps de calcul coûte cher, c'est cette deuxième méthode qui est la plus intéressante (et c'est pourquoi la première version de l'analyseur n'a pas été plus développée).

Il faut enfin signaler qu'un troisième outil a été développé parallèlement. Il s'agit d'un outil d'aide à la

spécification, utilisant une procédure de décision dans la logique temporelle. Cet outil permet de savoir si une formule est valide (c.a.d. est un théorème découlant des axiomes de la logique). Il reconnaît les opérateurs ALL et POT, SOME et INEV, conditionnels ou non. Lorsqu'une formule n'est pas valide, il exhibe un contre-exemple, c.a.d. un modèle dans lequel la formule n'est pas toujours vraie, ce qui permet de mieux comprendre pourquoi elle n'est pas valide. C'est donc essentiellement un système d'aide au raisonnement dans la logique temporelle. Il permet par exemple de comparer formellement des propriétés intuitivement proches, ou de relever des contradictions dans un ensemble de spécifications fournies.

L'ensemble des programmes réalisés nous paraît suffisant pour justifier la démarche choisie. La faisabilité des algorithmes a été démontrée. Le compilateur expérimental est peu performant, mais il pourrait être optimisé assez facilement, ce que nous n'avons pas cherché à faire. L'analyseur dans sa deuxième version se révèle à notre avis très satisfaisant, et un travail d'optimisation supplémentaire permettrait certainement de l'améliorer (c.a.d. essentiellement de diminuer la place occupée par le graphe d'états sans nuire aux performances).

Le bilan de l'expérience CESAR est donc incontestablement positif. On a en effet prouvé :

- qu'il est possible de générer automatiquement sans perte d'information un modèle (en l'occurrence un RdPI) à partir d'une description d'une application en un langage algorithmique non-déterministe de haut-niveau, manipulant les concepts de tâches et d'échanges par rendez-vous ; bien d'autres formes de synchronisation peuvent d'ailleurs s'exprimer en termes de RdP interprétés ou non [QUE 78b] ;

- que l'utilisation d'une logique temporelle pour l'expression des propriétés requises pour cette application fournit une possibilité de spécification abstraite d'un large éventail de propriétés comportementales (même si son emploi n'est pas toujours facile) ;

- que la caractérisation en termes de points fixes de transformateurs de prédicats des opérateurs de cette logique fournit une méthode de preuve automatique de ces propriétés, que l'on peut mettre en oeuvre de façon efficace sous forme de calcul sur des ensembles d'états ;

ce qui nous paraît justifier l'étude et la réalisation d'une version plus développée.

2 - PROPOSITIONS POUR UN VERITABLE PROTOTYPE DU SYSTEME CESAR

Les deux entrées du système CESAR sont, d'une part, la description de l'application à analyser, et d'autre part, la spécification des propriétés qu'elle doit satisfaire. Le langage de description proposé apparaît dans l'ensemble satisfaisant (des propositions d'extensions figurent dans [QUE 82], mais ne remettent pas en cause les choix faits lors de sa définition). Par contre l'utilisation du langage de spécification dans sa forme actuelle apparaît parfois complexe, à tel point que l'utilisation d'un système de décision dans cette logique a été jugée nécessaire afin d'en examiner de plus près les propriétés. Pour tenter d'améliorer les conditions d'utilisation du langage de spécification, plusieurs choses peuvent être faites :

- essayer de donner au langage de spécification un formalisme moins mathématique et plus abordable à l'utilisateur terminal (les concepts manipulés restant toutefois ceux de la logique temporelle, puisque c'est sur eux que notre méthode de preuve s'applique) ;

- classifier les propriétés les plus couramment employées et définir des "macro-opérateurs" leur correspondant (comme cela a déjà été esquissé avec l'emploi de la notation []f à la place de INIT => ALL f) ;

- réaliser un véritable système d'aide à la spécification, permettant la manipulation, la transformation, la comparaison et le stockage des formules de spécification, utilisant la procédure de décision dans la logique temporelle [SCH 83].

En ce qui concerne le compilateur, il faudrait étudier une technique de compilation ascendante et la comparer avec la solution descendante adoptée dans la maquette. Il devra être complété par un système permettant l'archivage et le catalogage de modules, avant et après compilation, et d'un éditeur de liens vérifiant la cohérence des types des données définies dans des modules distincts. Enfin, l'édition de tables de "références croisées" sur les données des programmes de description serait utile, surtout par la possibilité qu'elle offre de mesurer l'impact d'une modification faite à la conception initiale de l'application.

La méthode de calcul employée par l'analyseur (dans sa deuxième version) a donné de bons résultats et ne sera pas remise en cause. Elle peut être améliorée par l'emploi de plusieurs techniques de réduction : réduction du graphe d'états au cours de sa génération, ou en vue de la démonstration d'une propriété particulière ; réduction du réseau représentant l'application par l'utilisation de techniques de réduction de RdPI [MAR 81] avant la

construction du graphe d'états ; réduction par l'emploi des mêmes techniques des réseaux représentant les tâches élémentaires avant composition. Un effort tout particulier doit être mené pour améliorer l'interaction entre l'analyseur et l'utilisateur, de façon à lui faciliter la manipulation et le catalogage des prédicats, lui permettre d'orienter la politique d'analyse (ordre d'évaluation des opérateurs, nombre maximum d'itérations, mémorisation de résultats partiels, etc.) et d'interpréter les résultats en termes des variables du programme de description et des variables propositionnelles [MAR 83].

Nous pensons qu'une version opérationnelle de CESAR ainsi étoffée serait un outil extrêmement utile pour la conception et l'intégration d'applications réparties aussi bien en milieu universitaire qu'en milieu industriel. Sa réalisation nous semble toutefois plus dans la vocation d'un industriel que dans celle de l'université.

3 - EXTENSIONS ENVISAGEABLES A PLUS LONG TERME

Pour un système d'intérêt aussi général que le système CESAR, de nombreuses extensions sont envisageables. Elles concernent l'expressivité du langage de description et de celui de spécification, l'utilisation d'autres méthodes d'analyse, et l'extension du système CESAR à d'autres domaines (fiabilité et performances).

On peut ainsi introduire la notion de types abstraits [GUT 77] (et leur extension les types abstraits génériques [BJ 78]) dans le langage de description (avec la possibilité de définir des types abstraits incomplètement spécifiés, pour respecter la philosophie de ce langage). Il serait également intéressant de compléter le programme de description de la définition d'une architecture matérielle (en termes d'un réseau de processeurs et de canaux) sur laquelle l'architecture logique (en termes de tâches) doit être implémentée, des vérifications de cohérence pouvant être effectuées.

Malgré ces extensions, l'emploi du système CESAR, par la forme algorithmique de son langage de description, se situe à un stade de la conception déjà assez avancé. Un outil d'aide à la conception est également nécessaire aux stades antérieurs, mais cela sort du cadre que nous nous sommes assignés pour le projet CESAR. L'étude d'un outil à ce niveau, permettant une description non algorithmique des processus vus comme des boites noires uniquement capables d'envoyer et de recevoir des messages, est en cours (projet SPARC [JQS 82]).

Pour améliorer l'expressibilité du langage de spécification, on peut envisager l'utilisation d'expressions de chemins (expressions régulières sur les noms d'actions [LC 75]) pour une expression naturelle des contraintes de séquencement (qui sont souvent assez difficiles à exprimer en logique temporelle), ainsi que l'emploi de compteurs d'occurences d'actions qui permettent l'expression de certaines propriétés [RV 77] [QS 82a] qu'il n'est pas facile d'exprimer autrement.

En ce qui concerne l'analyse, plusieurs approches peuvent être envisagées (outre les méthodes de réduction signalées au paragraphe précédent) :

- décomposer les propriétés à vérifier en un ensemble de propriétés locales (vraies à certains points du contrôle) qui peuvent être évaluées comme points fixes d'un système d'équations, individuellement plus simples que l'équation globale [SIF 81] ;

- utiliser des méthodes d'approximation dans le calcul de l'interprétation des opérateurs en tant que points fixes ([COU 78], [CH 78], [CLA 80]). Ces méthodes fournissent des conditions nécessaires ou suffisantes suivant le sens de l'approximation effectuée (inférieure ou supérieure) ;

- étudier une méthode permettant de synthétiser les axiomes de la logique définie par un programme de description, de façon à leur appliquer une méthode de preuve formelle.

Enfin, en ce qui concerne les extensions du système CESAR aux domaines de la fiabilité et de l'évaluation de performances, il est facile de compléter la description de l'architecture matérielle de données numériques sur la fiabilité de chaque organe, de façon à pouvoir évaluer la fiabilité par fonction (c.a.d. par tâche) ou par variable échangée (par l'utilisation de graphes série-parallèle). En associant aux traitements des durées, il est également possible d'effectuer une analyse de performances en termes de Réseaux de Petri temporisés [SIF 77] (en supposant une évolution à vitesse maximale), ou en termes de Réseaux de Petri stochastiques [NAT 80] [FN 82].

Ces dernières propositions soulèvent le problème du "mélange des genres", en ce sens que l'on peut se demander s'il vaut mieux un système général utilisant des techniques différentes pour résoudre des problèmes de natures diverses, ou si au contraire une pluralité d'outils variés n'est pas préférable. Le problème de la spécification et de l'analyse comportementale est peut-être suffisamment complexe pour ne pas être mêlé à d'autres problèmes. D'un autre coté, il est tentant d'essayer de regrouper au sein d'un même système d'analyse tous les outils nécessaires au concepteur pour l'évaluation de l'architecture qu'il est en train de définir. L'approche suivie est déjà en elle-même une approche unificatrice, car regroupant au sein d'un même système des techniques venues d'horizons différents. Le réseau de Petri, en tant que modèle primitif du parallélisme, et grâce à ses nombreuses possibilités d'extensions, peut être (et nous le souhaitons) le ferment d'unifications ultérieures.

4 - CONCLUSION

L'un des points les plus positifs, à notre avis, de l'approche que nous avons adoptée est de proposer une véritable méthode de preuve des propriétés comportementales des systèmes où intervient le parallélisme. Qui plus est, cette méthode est automatisable lorsque l'on se limite aux systèmes ayant un espace d'états fini. Cette restriction ne nous semble pas une limitation pour les problèmes que nous avons pour objectif d'étudier, c.a.d. les problèmes que l'on peut regrouper sous le vocable très général de "problèmes de synchronisation". Notre but n'était en effet pas de fournir un outil pour la preuve des programmes parallèles, mais pour celle des interactions du comportement de différents processus composant un système.

Les réseaux de Petri interprétés dans le cas où l'interprétation ne manipule que des variables bornées, n'ont pas, en définitive, une puissance de modélisation supérieure à celle des réseaux de Petri nus. Nous pensons toutefois avoir fait un bon usage des réseaux de Petri en leur réservant le rôle de représenter la partie contrôle de l'application décrite, rôle auquel ils sont parfaitement bien adaptés. De plus nous n'avons pas essayé d'employer ce modèle comme outil de description premier, préférant le générer automatiquement à partir d'une description dans un langage de haut-niveau conçu dans ce but.

Un point faible de notre approche est que la façon d'employer le langage de description influe sur les propriétés vérifiées par l'application décrite (et sur la facilité de leur démonstration). L'emploi du système CESAR en tant qu'outil de preuve n'induit en effet aucune méthodologie particulière de conception (et donc de description). C'est une lacune à combler par l'étude d'une méthodologie de conception incrémentale, qui devra reposer sur une méthode de preuve elle-même incrémentale (preuve de systèmes "ouverts" et synthèse de propriétés). Une telle approche devrait permettre de dépasser les limites (quantitatives) du système CESAR, dont l'approche "de front" suppose des moyens de calcul considérables, si la taille du système à analyser devient trop importante. Atténuons toutefois l'impression laissée par cette dernière phrase en soulignant qu'une "bonne modélisation" est celle qui ne met en évidence que le contrôle (évènements significatifs pour le comportement) en laissant de coté les données et les traitements qui n'influent pas sur celui-ci, et qu'une "bonne modélisation" peut être de taille modeste, même pour une application réelle de dimensions apparemment trop importantes pour être traitée par le système CESAR. Ce travail de modélisation est un travail humain, qui reste à notre avis un préalable à l'utilisation de tout système de preuve.

ANNEXE 1

Grammaire du langage de description du système CESAR

Les règles de syntaxe hors-contexte sont données sous forme de Backus-Naur avec quelques extensions.

Les symboles non-terminaux sont notés en minuscules entre les caractères "<" et ">". Les mots-clés sont notés en majuscules. Chaque règle de la grammaire est formée d'une partie gauche qui est un symbole non-terminal, et d'une partie droite qui est une suite d'alternatives séparées par le symbole "|", de la forme suivante :

```
<non-terminal> ::= alternative 1 |
                   alternative 2 |
                   ...
```

Chaque alternative est une suite de symboles non-terminaux et terminaux (ces derniers étant des mots-clés, ou des symboles composés d'un ou plusieurs caractères spéciaux).

Les notations suivantes ont été introduites (où xxx est une suite quelconque de symboles terminaux et non terminaux) :

- [xxx] signifie l'occurence éventuelle de xxx,
- [xxx]* signifie l'occurence éventuelle de xxx un nombre quelconque de fois (0, 1 ou plusieurs fois),
- [xxx]+ signifie l'occurence de xxx un nombre quelconque de fois (au moins une fois).

Lorsque les caractères "|", "[" et "]" doivent figurer en partie droite de règle pour eux-mêmes, ils sont écrits "'|", "'[" et "']". Les caractères "*" et "+" ne suivant pas immédiatement le caractère "]" figurent toujours pour eux-mêmes. Le caractère "<" suivi d'un blanc et le caractère ">" précédé d'un blanc figurent également pour eux-mêmes.

L'axiome de la grammaire est <train de compilation>.

Les identificateurs prédéclarés ne figurent pas dans les règles de grammaire, mais sont mentionnés en notes. Ils ne sont pas des mots réservés.

```
<caractère> ::= <lettre majuscule> | <chiffre> |
          <caractère spécial> | <lettre minuscule> |
          <autre caractère>

<lettre majuscule> ::= A | B | C | D | E | F | G | H | I |
          J | K | L | M | N | O | P | Q | R | S | T | U |
          V | W | X | Y | Z

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

<caractère spécial> ::= <blanc> | # | ' | ( | ) | * | + | , |
          / | : | ; | < | = | > | _ | . | - | ! | ? | " | |

<blanc> ::= \SP | \HT

<lettre minuscule> ::= a | b | c | d | e | f | g | h | i |
          j | k | l | m | n | o | p | q | r | s | t | u |
          v | w | x | y | z

<fin de ligne> ::= [ <caractère de fin de ligne> ]+

<caractère de fin de ligne> ::= \CR | \LF | \FF | \VT

<autre caractère> ::= % | & | $ | @ | [ | \ | ] | ^ |
          ^ | { | } | ~ | \BS | \BEL

<identificateur> ::= <lettre> [ [_] <lettre ou chiffre> ]*

<lettre ou chiffre> ::= <lettre> | <chiffre>

<lettre> ::= <lettre majuscule> | <lettre minuscule>

<liste d'identificateurs> ::= <identificateur>
          [ , <identificateur> ]*

<nombre littéral> ::= <nombre base 10> | <nombre base 2> |
          <nombre base 16>

<nombre base 10> ::= <entier> [<exposant>]

<entier> ::= <chiffre> [ [_] <chiffre> ]*

<exposant> ::= E [+] <entier>

<nombre base 2> ::= 2 # <entier base 2> #

<nombre base 16> ::= 16 # <entier base 16> #

<entier base 2> ::= <chiffre base 2>
          [ [_] <chiffre base 2> ]*

<entier base 16> ::= <chiffre base 16>
          [ [_] <chiffre base 16> ]*
```

```
<cchiffre base 2> ::= 0 | 1

<chiffre base 16> ::= <chiffre> | A | B | C | D | E | F

<commentaire> ::= -- [<caractère>]* <fin de ligne>

<déclaration en bloc> ::= <déclaration de constantes> |
          <déclaration de variables> |
          <déclaration de type> |
          <déclaration de procédure>

<déclaration en package> ::= <déclaration de constantes> |
          <déclaration de type> |
          <déclaration de procédure> |
          <déclaration de tâche>

<déclaration en tâche composée> ::=
          <déclaration en package> |
          <déclaration de variables échangées>

<déclaration de constantes> ::= <liste d'identificateurs> :
          CONSTANT [<type>] [ := <expression> ] ;

<déclaration de variables> ::= <liste d'identificateurs> :
          <type> [ := <expression> ] ;

<déclaration de type> ::= <déclaration de type énuméré> |
          <déclaration de type entier> |
          <déclaration de tableau> |
          <déclaration de structure> |
          <déclaration de type dérivé> |
          <déclaration de type non spécifié>

<nom de type> ::= [ <nom de package> . ] <identificateur>

Note : les types [STANDARD.]BOOLEAN, [STANDARD.]INTEGER et
[STANDARD.]NATURAL sont prédéclarés.

<déclaration de type énuméré> ::= TYPE <identificateur> IS
          ( <liste d'identificateurs> ) ;

<contrainte d'appartenance énumérée> ::=
           RANGE <intervalle énuméré>

<intervalle énuméré> ::=
          <expression> .. <expression>

<déclaration de type entier> ::= TYPE <identificateur> IS
          <contrainte d'appartenance entière> ;

<contrainte d'appartenance entière> ::=
          RANGE <intervalle entier>
```

```
<intervalle entier> ::=
          <expression simple> .. <expression simple> |
          < .. <expression simple> |
          <expression simple> .. > |
          < .. >

<déclaration de tableau> ::= TYPE <identificateur> IS ARRAY
          ( <indication d'index> [,<indication d'index>]* )
          OF <type contraint> ;

<indication d'index> ::= <index spécifié> |
          <index non spécifié>

<index spécifié> ::= <intervalle discret>

<intervalle discret> ::= <type discret contraint> |
          <intervalle>

<intervalle> ::= <expression> .. <expression>

<index non spécifié> ::= <nom de type> RANGE <>

<contrainte d'index> ::= ( <élément de contrainte d'index>
          [ , <élément de contrainte d'index> ]* )

<élément de contrainte d'index> ::= <index spécifié> | *

<déclaration de structure> ::= TYPE <identificateur>
          [<partie discriminante>] IS RECORD
          <liste de composants> END [<fin de structure>] ;

<liste de composants> ::= [<déclaration de composants>]* |
          NULL ;

<déclaration de composants> ::= <liste d'identificateurs> :
          <type> ;

<fin de structure> ::= RECORD | <identificateur>

<partie discriminante> ::= ( <déclaration de discriminants>
          [ ; <déclaration de discriminants> ]* )

<déclaration de discriminants> ::= <liste d'identificateurs>
          : <type discret contraint>

<contrainte de discriminant> ::= ( <valeur de discriminant>
          [ , <valeur de discriminant> ]* )

<valeur de discriminant> ::= <expression> | *

<déclaration de type dérivé> ::= TYPE <identificateur> IS
          <type contraint> ;
```

```
<type contraint> ::= <type discret contraint> |
          <type tableau contraint> |
          <type à discriminants contraint>

<type discret contraint> ::= <nom de type>
          [ <contrainte d'appartenance> ]

<contrainte d'appartenance> ::=
          <contrainte d'appartenance énumérée> |
          <contrainte d'appartenence entière>

<type tableau contraint> ::= <nom de type>
          [ <contrainte d'index> ]

<type à discriminants contraint> ::= <nom de type>
          [ <contrainte de discriminant> ]

<type> ::= [ ARRAY ( <index spécifié>
          [ , <index spécifié> ]* ) OF ]
          <type contraint>

<déclaration de type non spécifié> ::= TYPE <identificateur>
          [<partie discriminante>] ;

<expression> ::= <relation> [ AND <relation> ]* |
          <relation> [ OR <relation> ]* |
          <relation> [ XOR <relation> ]*

<relation> ::= <expression simple>
          [<opérateur de comparaison> <expression simple>]

<expression simple> ::= [<opérateur unaire>] <terme>
          [ <opérateur additif> <terme> ]*

<terme> ::= <facteur> [ <opérateur multiplicatif <facteur> ]*

<facteur> ::= <primaire>
          [ <opérateur exponentiel> <primaire> ]

<primaire> ::= <littéral> |
          <nom de variable> |
          <nom de constante> |
          <attribut> |
          <expression convertie> |
          <expression qualifiée> |
          <fonction standard> |
          ( <liste d'expressions> )

<liste d'expressions> ::= <expression> [ , <expression> ]*

<opérateur de comparaison> ::= = | /= | < | <= | > | >=
```

```
<opérateur unaire> ::= + | - | NOT

<opérateur additif> ::= + | -

<opérateur multiplicatif> ::= * | / | MOD

<opérateur exponentiel> ::= **

<fonction standard> ::= ABS ( <expression simple> )

<littéral> ::= <nombre littéral> |
          <littéral énuméré> |
          <littéral non spécifié>

<littéral énuméré> ::= <identificateur>
```

Note : les littéraux énumérés FALSE et TRUE (du type BOOLEAN) sont prédéclarés.

```
<littéral non spécifié> ::= " [<caractère>]* "

<nom de variable> ::= <variable>

<variable> ::= <identificateur> |
          <variable> ( <liste d'expressions> ) |
          <variable> . <identificateur>

<nom de constante> ::= [ <nom de package> . ] <variable>

<attribut> ::= <nom d'objet à attributs> '
          <identificateur d'attribut>
          [ ( <expression simple> ) ]

<nom d'objet à attributs> ::= <nom de type> |
          <nom de variable> |
          <nom de constante>

<identificateur d'attribut> ::= FIRST | LAST | LENGTH |
          SUCC | PRED

<expression convertie> ::= <nom de type>
        ( <liste d'expressions> )

<expression qualifiée> ::=
          <nom de type> ' <nom de variable> |
          <nom de type> ' <nom de constante> |
          <nom de type> ' <nombre littéral> |
          <nom de type> ' <littéral énuméré> |
          <nom de type> ' ( <liste d'expressions> )
```

```
<instruction> ::= <instruction vide> |
          <affectation> |
          <instruction if> |
          <instruction case> |
          <instruction loop> |
          <instruction either> |
          <appel de procédure> |
          <exit> |
          <return> |
          <terminate> |
          <bloc>

<séquence> ::= [ <instruction> ]+

<étiquette> ::= << <identificateur> >>

<instruction vide> ::= [ <étiquette> ]* NULL ;

<affectation> ::= [ <étiquette> ]* [ <échange> , ]
          <affectation simple> [, <affectation simple>]* ; |
          [ <étiquette> ]* <échange> ;

<affectation simple> ::= <nom de variable>
          [ , <nom de variable> ]* := <expression>

<instruction if> ::= [ <étiquette> ]*
          IF <condition> THEN <séquence>
          [ ELSIF <condition> THEN  <séquence> ]*
          [ ELSE <séquence> ] END [IF] ;

<condition> ::= <expression>

<instruction case> ::= [ <étiquette> ]*
          CASE <expression> IS
          [ WHEN <choix> [ | <choix> ]* => <séquence> ]*
          [ WHEN OTHERS => <séquence> ]
          END [CASE] ;

<choix> ::= <expression> |
          <intervalle discret>

<instruction loop> ::= [<étiquette>]* [<clause itérative>]
          <corps de loop>

<corps de loop> ::= LOOP <séquence> END [LOOP] ;

<clause itérative> ::= WHILE <condition> |
          FOR <identificateur> IN [ REVERSE ]
          <contrainte d'appartenance>

<exit> ::= [ <étiquette> ]* EXIT [ WHEN <condition> ] ;
```

```
<instruction either> ::= [ <étiquette> ]* EITHER
          [ WHEN <condition> => ] <séquence>
          [ OR [ WHEN <condition> => ] <séquence> ]*
          [ ELSE <séquence> ] END [EITHER] ;

<bloc> ::= [ DECLARE <partie déclarative de bloc> ]
          [ <étiquette> ]*
          BEGIN <séquence> END ;

<partie déclarative de bloc> ::=
          [ <déclaration en bloc> ]*

<déclaration de procédure> ::= PROCEDURE
          <en-tête de procédure> [ IS
          <partie déclarative de bloc>
          <corps de procédure> ] ; |
          <référence procédures externes>

<en-tête de procédure> ::= <identificateur>
          [ <partie formelle> ]

<partie formelle> ::= ( <déclaration de paramètre>
          [ ; <déclaration de paramètre> ]* )

<déclaration de paramètre> ::=
          <déclaration de paramètre non échangé> |
          <déclaration de paramètre échangé>

<déclaration de paramètre non échangé> ::=
          <liste d'identificateurs> : <type>

<déclaration de paramètre échangé> ::=
          <liste d'identificateurs> :
          [ ARRAY ( <indication d'index>
          [ , <indication d'index> ]* ) OF ]
          <mode échange> <type>

<mode échange> ::= INPUT | OUTPUT

<corps de procédure> ::= [<étiquette>]* BEGIN <séquence> END
          [ <fin de procédure> ]

<fin de procédure> ::= PROCEDURE | <identificateur>

<return> ::= [ <étiquette> ]* RETURN ;

<appel de procédure> ::= [ <étiquette> ]* <nom de procédure>
          [ ( <liste de paramètres effectifs> ) ] ;

<nom de procédure> ::= [ <nom de package> . ]
          <identificateur>
```

```
<liste de paramètres effectifs> ::= <paramètre effectif>
          [ , <paramètre effectif> ]*

<paramètre effectif> ::= <expression> |
          <nom de paramètre échangé> |
          <nom de type> ( <nom de paramètre échangé> )

<déclaration de tâche> ::=
          <déclaration de tâche simple> |
          <déclaration de tâches typées> |
          <référence tâches externes>

<déclaration de tâche simple> ::=
          <déclaration de tâche élémentaire> |
          <déclaration de tâche composée> |
          <déclaration de tâche non spécifiée>

<déclaration de tâche élémentaire> ::= TASK [TYPE]
          <en-tête de tâche>
          IS <partie déclarative de bloc>
          <corps de tâche élémentaire> ;

<déclaration de tâche composée> ::= COTASK [TYPE]
          <en-tête de tâche>
          IS <partie déclarative de tâche composée>
          <corps de tâche composée> ;

<partie déclarative de tâche composée> ::=
          [ <déclaration en tâche composée> ]*

<déclaration de tâche non spécifiée> ::=
          TASK [TYPE] <en-tête de tâche> ;

<en-tête de tâche> ::= <identificateur>
          [ <partie formelle de tâche> ]

<partie formelle de tâche> ::=
          ( <déclaration de paramètre de tâche>
          [ ; <déclaration de paramètre de tâche> ]* )

<déclaration de paramètre de tâche> ::=
          <déclaration de paramètre non échangé> |
          <déclaration de paramètre échangé>

<déclaration de tâches typées> ::=
          <liste d'identificateurs> :
          [ ARRAY ( <index spécifé>
          [ , <index spécifié> ]* ) OF ]
          <nom de type de tâche> ;

<nom de type de tâche> ::= [ <nom de package> . ]
          <identificateur>
```

```
<déclaration de variables échangées> ::=
          <liste d'identificateurs> :
          [ ARRAY ( <index spécifié>
          [ , <index spécifié> ]* ) OF ]
          EXCHANGED <type> ;

<échange> ::= [ <nom de variable> , ]*
          ! <nom de paramètre échangé> := <expression> !
          ! <nom de paramètre échangé> !
          [ <nom de variable> [ , <nom de variable> ]* := ]
          ? <nom de paramètre échangé>

<nom de paramètre échangé> ::=  <identificateur>
          [ ( <liste d'expressions> ) ]

<terminate> ::= [ <étiquette> ]* TERMINATE ;

<corps de tâche élémentaire> ::=
          BEGIN <séquence> END
          [ <fin de tâche élémentaire> ]

<fin de tâche élémentaire> ::= TASK ! <identificateur>

<corps de tâche composée> ::=
          BEGIN [ <modes de rendez-vous> ]
          <invocation de tâche>
          [ // <invocation de tâche> ]*
          END [ <fin de tâche composée> ]

<fin de tâche composée> ::= COTASK ! <identificateur>

<invocation de tâche> ::= <nom de tâche>
          [ <liste de paramètres effectifs de tâche> ]

<nom de tâche> ::= [ <nom de package> . ]
          <identificateur> [ ( <liste d'expressions> ) ]

<liste de paramètres effectifs de tâche> ::=
          ( <paramètre effectif de tâche>
          [ , <paramètre effectif de tâche> ]* )

<paramètre effectif de tâche> ::= <expression> !
          <nom de variable échangée> !
          <nom de type> ( <nom de variable échangée> )

<nom de variable échangée> ::=
          <identificateur> [ ( <liste d'expressions> ) ]

<modes de rendez-vous> ::= MODE [<spécification de mode>]*
          END [MODE] ;
```

```
<spécification de mode> ::= <nom de variable échangée>
          [ , <nom de variable échangée> ]* :
          <mode de réception> ;

<mode de réception> ::= ANY | ALL

<déclaration de package> ::= PACKAGE <identificateur> IS
          <partie déclarative de package>
          END [<fin de package>] ;

<corps de package> ::= PACKAGE BODY <identificateur> IS
          <partie déclarative de package>
          END [<fin de package>] ;

<partie déclarative de package> ::=
          [ <déclaration en package> ]*

<fin de package> ::= PACKAGE | <identificateur>

<nom de package> ::= <identificateur>

Note : le package STANDARD est prédéclaré.

<train de compilation> ::=
          [<déclaration de type ou constante>]*
          [ [<clause with>]*
          <unité de compilation> ]*

<déclaration de type ou constante> ::=
          <déclaration de type> |
          <déclaration de constantes>

<unité de compilation> ::=
          <déclaration de tâche> |
          <déclaration de procédure> |
          <déclaration de package> |
          <corps de package>

<clause with> ::= WITH <nom de package>
          [ , <nom de package> ]* ;

<référence tâches externes> ::= EXTERNAL TASK TYPE
          <liste d'identificateurs> ;

<référence procédures externes> ::= EXTERNAL PROCEDURE
          <liste d'identificateurs> ;
```

ANNEXE 2

Grammaire du langage de spécification du système CESAR

Les conventions sont les mêmes que celles employées dans la grammaire du langage de description. Les non-terminaux soulignés sont les mêmes que ceux de la grammaire du langage de description.

L'axiome de la grammaire est <formule temporelle>.

```
<formule temporelle> ::=
          <prédicat> [ => <prédicat> ] |
          <prédicat> [ AND <prédicat> ]* |
          <prédicat> [ OR <prédicat> ]*

<prédicat> ::= [ <opérateur> ]* <variable propositionnelle>

<opérateur> ::= NOT | [] |
          <opérateur temporel> [ [ <formule temporelle> ] ]

<opérateur temporel> ::= ALL | POT | SOME | INEV |
          SONT | OBL | ALW | WPOT | FAIR

<variable propositionnelle> ::= <variable prédéfinie> |
          ( <expression> ) |
          ( <formule temporelle> )

<variable prédéfinie> ::=
          <état> [ ( <liste de noms de tâches> ) ] |
          ENABLE [ ( <argument de enable> ) ] |
          SINK [ ( <liste de noms de tâches> ) ] |
          <position> ( <argument de position> )

<état> ::= INIT | END

<position> ::= BEFORE | IN | AFTER

<argument de position> ::=
          <liste de noms d'actions>
          [ , <liste de noms de tâches> ]
```

<argument de enable> ::= <liste de noms d'actions> |
<liste de noms de tâches>

<liste de noms d'actions> ::= <nom d'action>
[, <nom d'action>]*

<nom d'action> ::= [<nom d'environnement> .]* <identificateur>

<liste de noms de tâches> ::= <nom de tâche>
[, <nom de tâches>]

<nom de tâche> ::= [<nom d'environnement> .]*
<identificateur> [(<liste d'indices>)]

<liste d'indices> ::= <indice> [, <indice>]*

<indice> ::= <expression> | *

<nom d'environnement> ::= <identificateur>
[(<liste_d'expressions>)]

BIBLIOGRAPHIE

[ABR 79] - ABRAHAMSON K. - "Modal Logic of Concurrent Non Deterministic Programs" - Semantics of Concurrent Computation, Lecture Notes in Computer Science vol. 70, Springer-Verlag pp. 21-33.

[ADA 80] - "Reference Manual for the Ada Programming Language" - Compagnie Cii-Honeywell Bull (July 80).

[BER 78] - BERTHELOT G. - "Vérification des Réseaux de Petri" - Thèse de Troisième Cycle (Janvier 78), Université Pierre et Marie Curie (Paris VI).

[BH 78] - BRINCH HANSEN P. - "Distributed Processes : A Concurrent Programming Concept" - Communications of ACM 21-12 (November 78), pp. 934-941.

[BJ 78] - BERT D. & JACQUET P. - "Some Validation Problems with Parametrized and Generic Functions" - 3ème Colloque International sur la Programmation, Paris.

[BMP 81] - BEN-ARI M., MANNA Z. & PNUELI A. - "The Temporal Logic of Branching Time" - 8th ACM Symposium on Principles of Programming Languages (January 81), pp. 164-176.

[BSW 69] - BARTLETT K.A., SCANTLEBURRY R.A. & WILKINSON P.T. - "A Note on Reliable Full-Duplex Transmission over Half-Duplex Links" - Communications of ACM 12-5 (May 69), pp. 260-261.

[BT 82] - BERTHELOT G. & TERRAT R. - "Modeling and Proofs of a Data Transfer Protocol by Predicate/Transition Nets" - Application and Theory of Petri Nets, Informatik-Fachberichte vol. 52, Springer-Verlag pp. 251-257.

[CE 81] - CLARKE E.M. & EMERSON E.A. - "Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic" - Technical Report 12-81, Aiken Computation Laboratory, Harvard University.

[CH 78] - COUSOT P. & HALBWACHS N. - "Automatic Discovery of Linear Restraints among Variables of a Program" - Proceedings of the 5th ACM Symposium on Principles of Programming Languages, pp.84-96.

[CHA 82] - CHARBONNIER J.P. - "Composition des Réseaux de Petri Interprétés dans le cadre du Projet Cesar" - Rapport de projet ENSIMAG (Juin 82).

[CLA 80] - CLARKE E.M. Jr. - "Synthesis of Ressource Invariants for Concurrent Programs" - ACM Transactions on Programming Languages and Systems 2-3 (July 80), pp. 338-358.

[CMPS 80] - DE CINDIO F., DE MICHELIS G., POMELLO L. & SIMONE C. - "A Petri Net Model of CSP" - Instituto de Cibernetica, Université de Milan.
[COO 76] - COOPRIDER L. - "Petri Nets and the Representation of Standard Synchronizations" - Technical Report, Computer Science Department, Carnegie Mellon University.
[COU 78] - COUSOT P. - "Méthodes Itératives de Construction et d'Approximation de Points Fixes d'Opérateurs Monotones sur un Treillis, Analyse Sémantique des Programmes" - Thèse d'Etat (Mars 78), Université de Grenoble.
[DIJ 75] - DIJKSTRA E.W. - "Guarded Commands, Nondeterminacy and Formal Derivation of Programs" - Communications of ACM 18-8 (August 75), pp. 453-457.
[EC 80] - EMERSON E.A. & CLARKE E.M. - "Characterizing Correctness Properties of Parallel Programs using Fixpoints" - Proceedings of ICALP 80, Lecture Notes in Computer Science vol. 85, Springer-Verlag pp. 165-181.
[FN 82] - FLORIN G. & NATKIN S. - "Evaluation based upon Stochastic Petri Nets of the Maximum Throughput of a Full Duplex Protocol" - Application and Theory of Petri Nets, Informatik-Fachberichte vol. 52, Springer-Verlag pp. 280-288.
[GDS 82] - GUIDACCI DA SILVEIRA G.F. - "Sur la Preuve des Systèmes Parallèles et Distribués par la Logique Temporelle" - Thèse d'Etat (Janvier 82), Université Paul Sabatier, Toulouse.
[GIR 81] - GIRAULT C. - "Proof of Protocols in the Case of Failures" - Rapport de Recherche (Janvier 81), Institut de Programmation.
[GLT 79] - GENRICH H.J., LAUTENBACH K. & THIAGARAJAN P.S. - "Predicate/Transition Nets" - Proceedings of the Advanced Course on General Net Theory of Processes and Systems, Lecture Notes in Computer Science vol. 84, Springer-Verlag.
[GPSS 80] - GABBAY D., PNUELI A., SHELAH S. & STAVI J. - "On the Temporal Analysis of Fairness" - 7th ACM Symposium on Principles of Programming Languages, pp. 163-173.
[GUE 81] - GUERREIRO P. - "Relational Semantics of Strongly Communicating Sequential Processes" - Formalization of Programming Concepts, Lecture Notes in Computer Science vol. 107, Springer-Verlag.
[GUT 77] - GUTTAG J. - "Abstract Data Types and the Development of Data Structures" - Communications of ACM 20-6 (June 77).
[HC 68] - HUGHES G.E. & CRESWELL M.J. - "An Introduction to Modal Logic" - Methuen & Co. Ltd, London.
[HOA 78] - HOARE C.A.R. - "Communicating Sequential Processes" - Communications of ACM 21-8 (August 78), pp. 666-677.
[JEN 81] - JENSEN K. - "Coloured Petri Nets and the Invariant Method" - Theoretical Computer Science 14, pp. 317-336.
[JK 82] - JENSEN K. & KYNG M. - "Petri Nets and Semantics of System Descriptions" - Application and Theory of Petri Nets, Informatik-Fachberichte vol. 52, Springer-Verlag pp. 64-71.

[JKM 79] - JENSEN K., KYNG M. & MADSEN O.L. - "A Petri Net Definition of a System Description Language" - Semantics of Concurrent Computation, Lecture Notes in Computer Science vol. 70, Springer-Verlag pp.348-368.
[JOR 82] - JORRAND P. - "Description and Composition of Communicating Processes - Problems of Analysis and Correctness" - Rapport de Recherche IMAG RR 290 (Février 82).
[JQS 82] - JORRAND P. QUEILLE J.P. & SIFAKIS J. - "Conception et Vérification des Applications Réparties : Présentation du système CESAR et des développements en cours" - Projet Pilote SURF, Bilan et Perspectives (Janvier 82), ADI pp. 139-153.
[KEL 72] - KELLER R.M. - "Vector Replacement Systems : A Formalism for Modelling Asynchronous Systems" - Technical Report 117, Princeton University (December 72).
[KEL 76] - KELLER R.M. - "Formal Verification of Parallel Programs" - Communications of ACM 19-7 (July 76), pp. 371-384.
[KOT 76] - KOTOV V.E. - "Control Types : an Approach to the Problem of Parallel Languages" - International Conference on Information Processing, IFIP-INFOPOL, North-Holland pp. 339-352.
[LAM 80] - LAMPORT L. - ""Sometime" is Sometimes "Not Never" - On the Temporal Logic of Programs" - 7th ACM Symposium on Principles of Programming Languages (January 80), pp. 174-185.
[LC 74] - LAUER P.E. & CAMPBELL R.M. - "A Description of Path Expressions by Petri Nets" - Computer Laboratory, Technical Report 64, University of Newcastle Upon Tyne.
[LC 75] - LAUER P.E. & CAMPBELL R.H. - "Formal Semantics of a Class of High Level Primitives for Coordinating Concurrent Processes" - Acta Informatica 5, pp. 297-332.
[MA 81] - MARTY J.C. - "Analyse de Systèmes 'Condition-Action' par Calcul Itératif de Prédicats" - Rapport de DEA (Juin 81), INP Grenoble.
[MAR 81] - MARTINET P. - "Réduction de Réseaux de Petri dans le cadre du Projet CESAR" - Rapport de DEA (Octobre 81), INP Grenoble.
[MAR 83] - MARTINET P. - Thèse en préparation, INP Grenoble.
[MIL 78] - MILNER R. - "Synthesis of Communicating Behaviour" - Mathematic Foundations of Computer Science, Lecture Notes in Computer Science, Springer-Verlag pp. 71-83.
[MM 79] - MILNE G. & MILNER R. - "Concurrent Processes and their Syntax" - Journal of ACM 26-2 (April 79), pp. 302-321.
[MUL 82] - MULLER H. - "Correctness Proof for the Alternating Bit Protocol by Assertion Systems" - Application and Theory of Petri Nets, Informatik-Fachberichte vol. 52, Springer-Verlag pp. 322-326.
[NAT 80] - NATKIN S. - "Les Réseaux de Petri Stochastiques et leur Application à l'Evaluation des Systèmes Informatiques" - Thèse de Docteur-Ingénieur (Juin 80), CNAM.

[PAR 69] - PARK D. - "Fixpoint Induction and Proofs of Program Properties" - Machine Intelligence 5, pp. 59-78.
[PNU 77] - PNUELI A. - "The Temporal Logic of Programs" - 18th IEEE Annual Symposium on Foundations of Computer Science, pp. 46-57.
[QS 82a] - QUEILLE J.P. & SIFAKIS J. - "Iterative Methods for the Analysis of Petri Nets" - Application and Theory of Petri Nets, Informatik-Fachberichte vol. 52, Springer-Verlag pp. 161-167.
[QS 82b] - QUEILLE J.P. & SIFAKIS J. - "Specification and Verification of Concurrent Systems in CESAR" - Proceedings of 5th International Symposium on Programming, Lecture Notes in Computer Science vol. 137, Springer-Verlag pp. 337-351.
[QS 82c] - QUEILLE J.P. & SIFAKIS J. - "Fairness and Related Properties in Transition Systems - A Time Logic to Deal with Fairness" - Rapport de recherche IMAG RR 292 (Mars 82).
[QUE 78a] - QUEILLE J.P. - "Une Technique de Modélisation des Systèmes de Processus Parallèles adaptée à l'Analyse Statique" - Rapport de projet ENSIMAG (Juin 78), INP Grenoble.
[QUE 78b] - QUEILLE J.P. - "Techniques et Modèles pour la Spécification et l'Analyse de la Synchronisation" - Rapport de DEA (Juillet 78), INP Grenoble.
[QUE 81] - QUEILLE J.P. - "The CESAR System : an Aided Design and Certification System for Distributed Applications" - Proceedings of the 2nd International Conference on Distributed Computing Systems (April 81), Computer Society Press pp. 149-161.
[QUE 82] - QUEILLE J.P. - "Rapport de Définition du Langage de Description du Système CESAR" - Rapport de recherche IMAG, à paraître.
[ROZ 81] - ROZIER M. - "Description et Spécification de Protocoles dans le Cadre du Projet CESAR" - Rapport de DEA (Octobre 81), INP Grenoble.
[RS 81] - ROZIER M. & SCHWARTZ J.P. - "Réalisation d'un Programme de Traduction du Langage de Description des Applications Réparties, 'DADA', en Réseaux de Petri Interprétés" - Rapport de projet ENSIMAG (Juin 81).
[RV 77] - ROBERT P. & VERJUS J.P. - "Towards Autonomous Descriptions of Synchronization Modules" - Proceedings of IFIP 77, pp. 981-986.
[SCH 83] - SCHWARTZ J.P. - Thèse en préparation, INP Grenoble.
[SIF 77] - SIFAKIS J. - "Etude du Comportement Permanent des Réseaux de Petri Temporisés" - Rapport de recherche IMAG RR 68.
[SIF 79] - SIFAKIS J. - "Le Contrôle des Systèmes Asynchrones : Concepts, Propriétés, Analyse Statique" - Thèse d'Etat (Juin 79), Université de Grenoble.
[SIF 80] - SIFAKIS J. - "Deadlocks and Livelocks in Transition Systems" - Mathematical Foundations of Computer Science, Lecture Notes in Computer Science vol. 88, Springer-Verlag pp. 587-600.

[SIF 81] - SIFAKIS J. - "Global and Local Invariants in Transitions Systems" - Rapport de recherche IMAG RR 274 (Novembre 81), à paraître dans ICALP 82.
[SIF 82] - SIFAKIS J. - "A Unified Approach for Studying the Properties of Transition Systems" - Rapport de recherche IMAG RR 179 (Décembre 79, révisé Décembre 80), à paraitre dans Theoretical Computer Science.
[SIG 81] - Special Interest Group on Petri Nets and Related System Models - Newsletter 7 (February 81), pp. 1720.
[SMS 81] - SCHWARTZ R.L. & MELLIAR-SMITH P.M. - "Temporal Logic Specification of Distributed Systems" - Proceedings of the 2nd International Conference on Distributed Computing Systems (April 81), Computer Society Press pp. 446-454.
[SZL 77] - SZLANKO J. - "Petri Nets for Proving Some Corectness Properties of Parallel Programs" - IFAC-IFIP Workshop on Real-Time Programming, Eidhoven (June 77).
[TAR 55] - TARSKI A. - "A Lattice-Theoretical Fixpoint Theorem and its Applications" - Pacific Journal of Mathematics 5, pp. 285-305.

Dernière page d'une thèse

VU

Grenoble, le 1er Juin 1982

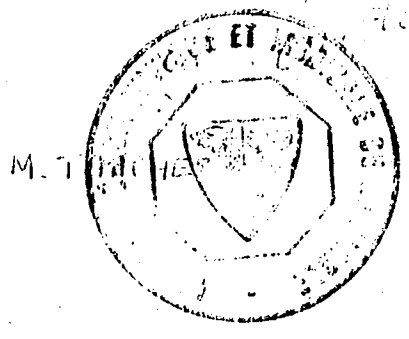
Le Président de la thèse

L. Bollinger

Vu, et permis d'imprimer,

Grenoble, le 3.6.82

Le Président de l'Université
Scientifique et Médicale



Résumé :

Le système CESAR proposé dans cette thèse est un système d'aide à la conception des applications réparties. Il permet :

- de décrire l'application étudiée dans un langage algorithmique en termes de processus communiquant par rendez-vous ;
- de spécifier les propriétés de comportement souhaitées au moyen d'une logique temporelle.

Le modèle sur lequel ces formules sont analysées est un réseau de Petri interprété généré automatiquement à partir de la description fournie. L'analyse repose sur une évaluation des opérateurs temporels comme points fixes de transformateurs de prédicats sur l'espace d'états du modèle.

Mots-clés :

Applications Réparties, Processus Communicants, Spécification, Logique Temporelle, Transformateurs de Prédicats, Réseaux de Petri.