QUASAR : une réalisation du système CESAR ; description, spécification et analyse des applications réparties

Jean-Philippe Schwartz

▶ To cite this version:

Jean-Philippe Schwartz. QUASAR : une réalisation du système CESAR ; description, spécification et analyse des applications réparties. Modélisation et simulation. Université Joseph-Fourier - Grenoble I, 1983. Français. NNT : . tel-00308504

HAL Id: tel-00308504

https://theses.hal.science/tel-00308504

Submitted on 30 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

SCHWARTZ Jean-Philippe

QUASAR, UNE REALISATION DU SYSTEME CESAR: DESCRIPTION, SPECIFICATION ET ANALYSE DES APPLICATIONS REPARTIES.

Thèse soutenue le 28 novembre 1983 devant la commission d'examen.

J. MOSSIERE	Président
K. APT	
P. AZEMA	
P. JORRAND	Examinateurs
J.P. QUEILLE	
J. SIFAKIS	

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

Année universitaire 1982-1983

Président de l'Université : M. TANCHE

MEMBRES DU CORPS ENSEIGNANT DE L'U.S.M.G.
(RANG A)
SAUF ENSEIGNANTS EN MEDECINE ET PHARMACIE

PROFESSEURS DE 1ère CLASSE

ARNAUD Paul	Chimie organique
ARVIEU Robert	Physique nucléaire I.S.N.
AUBERT Guy	Physique C.N.R.S.
AYANT Yves	Physique approfondie
BARBIER Marie-Jeanne	Electrochimie
BARBIER Jean-Claude	Physique expérimentale C.N.R.S. (labo de magnétisme)
BARJON Robert	Physique nucléaire I.S.N.
BARNOUD Fernand	Biosynthèse de la cellulose-Biologie
BARRA Jean-René	Statistiques - Mathématiques appliquées
BELORISKY Elie	Physique
BENZAKEN Claude (M.)	Mathématiques pures
BERNARD Alain	Mathématiques pures
BERTRANDIAS Françoise	Mathématiques pures
BERTRANDIAS Jean-Paul	Mathématiques pures
BILLET Jean	Géographie
BONNIER Jean-Marie	Chimie générale
BOUCHEZ Robert	Physique nucléaire I.S.N.
BRAVARD Yves	Géographie
CARLIER Georges	Biologie végétale
CAUQUIS Georges	Chimie organique
CHIBON Pierre	Biologie animale
COLIN DE VERDIERE Yves	Mathématiques pures
CRABBE Pierre (détaché)	C.E.R.M.O.
CYROT Michel	Physique du solide
DAUMAS Max	Géographie
DEBELMAS Jacques	Géologie générale
DEGRANGE Charles	Zoologie
DELOBEL Claude (M.)	M.I.A.G. Mathématiques appliquées
DEPORTES Charles	Chimie minérale
DESRE Pierre	Electrochimie
DOLIQUE Jean-Michel	Physique des plasmas
DUCROS Pierre	Cristallographie
FONTAINE Jean-Marc	Mathématiques pures
GAGNAIRE Didier	Chimie physique

.../...

GASTINEL Noël	Analyse numérique - Mathématiques appliquées
GERBER Robert	Mathématiques pures
GERMAIN Jean-Pierre	Mécanique
GIRAUD Pierre	Géologie
IDELMAN Simon	Physiologie animale
JANIN Bernard	Géographie
JOLY Jean-René	Mathématiques pures
JULLIEN Pierre	Mathématiques appliquées
KAHANE André (détaché DAFCO)	Physique
KAHANE Josette	Physique
KOSZUL Jean-Louis	Mathématiques pures
KRAKOWIAK Sacha	Mathématiques appliquées
KUPTA Yvon	Mathématiques pures
LACAZE Albert	Thermodynamique
LAJZEROWICZ Jeannine	Physique
LAJZEROWICZ Joseph	Physique
LAURENT Pierre	Mathématiques appliquées
DE LEIRIS Joël	Biologie
LLIBOUTRY Louis	Géophysique
LOISEAUX Jean-Marie	Sciences nucléaires I.S.N.
LOUP Jean	Géographie
MACHE Régis	Physiologie végétale
MAYNARD Roger	Physique du solide
MICHEL Robert	Minéralogie et pétrographie (géologie)
MOZIERES Philippe	Spectrométrie - Physique
OMONT Alain	Astrophysique
OZENDA Paul	Botanique (biologie végétale)
PAYAN Jean-Jacques (détaché)	Mathématiques pures
PEBAY PEYROULA Jean-Claude	Physique
PERRIAUX Jacques	Géologie
PERRIER Guy	Géophysique
PIERRARD Jean-Marie	Mécanique
RASSAT André	Chimie systématique
RENARD Michel	Thermodynamique
RICHARD Lucien	Biologie végétale
RINAUDO Marguerite	Chimie CERMAV
SENGEL Philippe	Biologie animale
SERGERAERT Francis	Mathématiques pures
SOUTIF Michel	Physique
VAILLANT François	Zoologie
VALENTIN Jacques	Physique nucléaire I.S.N.
VAN CUTSEN Bernard	Mathématiques appliquées
VAUQUOIS Bernard	Mathématiques appliquées
VIALON Pierre	Géologie

PROFESSEURS DE 2ème CLASSE

ADIBA Michel	Mathématiques pures
ARMAND Gilbert	Géographie

.../...

AURIAULT Jean-Louis	Mécanique
BEGUIN Claude (M.)	Chimie organique
BOEHLER Jean-Paul	Mécanique
BOITET Christian	Mathématiques appliquées
BORNAREL Jean	Physique
BRUN Gilbert	Biologie
CASTAING Bernard	Physique
CHARDON Michel	Géographie
COHENADDAD Jean-Pierre	Physique
DENEUVILLE Alain	Physique
DEPASSEL Roger	Mécanique des fluides
DOUCE Roland	Physiologie végétale
DUFRESNOY Alain	Mathématiques pures
GASPARD François	Physique
GAUTRON René	Chimie
GIDON Maurice	Géologie
GIGNOUX Claude (M.)	Sciences nucléaires I.S.N.
GUITTON Jacques	Chimie
HACQUES Gérard	Mathématiques appliquées
HERBIN Jacky	Géographie
HICTER Pierre	Chimie
JOSELEAU Jean-Paul	Biochimie
KERCKOVE Claude (M.)	Géologie
LE BRETON Alain	Mathématiques appliquées
LONGEQUEUE Nicole	Sciences nucléaires I.S.N.
LUCAS Robert	Physiques
LUNA Domingo	Mathématiques pures
MASCLE Georges	Géologie
NEMOZ Alain	Thermodynamique (CNRS - CRTBT)
OUDET Bruno	Mathématiques appliquées
PELMONT Jean	Biochimie
PERRIN Claude (M.)	Sciences nucléaires I.S.N.
PFISTER Jean-Claude (détaché)	Physique du solide
PIBOULE Michel	Géologie
PIERRE Jean-Louis	Chimie organique
RAYNAUD Hervé	Mathématiques appliquées
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
SOUTIF Jeanne	Physique
SCHOOL Pierre-Claude	Mathématiques appliquées
STUTZ Pierre	Mécanique
SUBRA Robert	Chimie
VIDAL Michel	Chimie organique
VIVIAN Robert	Géographie

REMERCIEMENTS

Je tiens à remercier :

- P. JORRAND, directeur de recherche au CNRS, pour m'avoir accueilli dans son équipe ;

- J. SIFAKIS, chargé de recherche au CNRS, pour les nombreuses discussions que nous avons eues ensemble depuis trois années, et pour tout le temps qu'il a consacré à l'encadrement et au développement du projet QUASAR ;

- J.P. QUEILLE, ingénieur à la SAGEM, pour l'aide efficace qu'il a apportée à la réalisation du projet QUASAR. Son expérience et ses conseils ont été très profitables pour l'avancement des travaux ; sa participation à la programmation a par ailleurs largement contribué à l'achèvement du projet ;

- J. MOSSIERE, professeur à l'université de Grenoble, pour m'avoir fait le grand honneur de présider le jury de cette thèse ;

- K. APT, chargé de recherche au CNRS, et P. AZEMA, maitre de recherche au CNRS, pour avoir accepté de participer au jury de cette thèse, et pour avoir bien voulu juger ce travail.

Je remercie également toutes les personnes qui ont participé de façon active à la réalisation du projet QUASAR : J.P. CHARBONNIER, S. GRAF, P. MARTINET, ainsi que les personnes qui ont contribué sous diverses formes au développement du projet CESAR : J.C. MARTY, M. ROZIER et J. VOIRON.

Je remercie aussi D. IGLESIAS et son équipe pour le soin qu'ils ont apporté à l'impression de cette thèse.

QUASAR , UNE REALISATION DU SYSTEME CESAR :

DESCRIPTION, SPECIFICATION ET ANALYSE
DES APPLICATIONS REPARTIES

I . Introduction - Présentation du système QUASAR
II . Le langage de description et sa traduction
III . Le langage de spécification
IV . Preuve par évaluation des formules de spécification
V . Manuel d'utilisation de QUASAR
VI . Conclusion - Bilan et Perspectives

SOMMAIRE DETAILLE

I. INTRODUCTION - PRESENTATION DE QUASAR

1. Objectifs
2. Présentation du système CESAR

   1. Le langage de description de CESAR
   2. Le langage de spécification de CESAR
   3. Vérification de propriétés dans CESAR

3. Origine du système QUASAR
4. Le système QUASAR

II. LE LANGAGE DE DESCRIPTION ET SA TRADUCTION

1. Introduction
2. Le langage de description de QUASAR

   1. Syntaxe du langage de description
   2. Déclaration de tâches élémentaires
   3. Déclaration des variables internes et initialisation
   4. Expressions
   5. Corps de tâche élémentaire
   6. Déclaration de tâche composée
   7. Déclaration de variables échangées
   8. Corps de tâche composée
   9. Remarque sur les identificateurs
   10. Exemples de description

       1. Exclusion mutuelle de Decker
       2. Protocole d'allocation de bus
       3. Transmission avec déconnection
       4. Réseau de trafic ferroviaire
       5. Transmission synchrone et asynchrone
       6. Réseau de distributeurs de billets

3. Traduction d'un programme de description

   1. Composition des tâches

      1. Principe de la composition
      2. Algorithme de composition
      3. Propriétés de la composition
      4. Opération d'encapsulation

2. Génération de programmes Pascal

1. Représentation des variables
2. Procédure d'initialisation des variables
3. Procédure d'exécution des commandes gardées

3. Réalisation de la traduction

1. Compilation de la description

1. Analyse syntaxique
2. Production d'un langage intermédiaire
3. Composition des tâches
4. Traitement des étiquettes

2. Génération d'un programme Pascal

1. Procédure d'initialisation des variables
2. Procédure d'exécution des commandes gardées

4. Un exemple de traduction

4. Conclusion

III. LE LANGAGE DE SPECIFICATION DE QUASAR

1. Introduction
2. La logique temporelle CTL

1. Introduction
2. Syntaxe et sémantique
3. Procédure de décision

1. Structure de Hintikka
2. Méthode des tableaux sémantiques

1. Construction d'un tableau
2. Algorithme d'évaluation
3. Algorithme de marquage
4. Algorithme de déroulement

3. Algorithme de décision

4. Application de la procédure de décision

3. Spécification de propriétés dans QUASAR

1. Interprétation des formules de la logique
2. Syntaxe du langage de spécification
3. Sémantique du langage de spécification

1. Les variables propositionnelles

1. Prédicats sur les variables de l'application
2. Variables propositionnelles associées aux actions

1. Exécutabilité d'une action
2. Confirmation de l'exécution d'une action

2. Les opérateurs temporels

4. Expression de propriétés
5. Le problème de l'équité
6. Exemples de spécification

1. Exclusion mutuelle de Decker
2. Protocole d'allocation de bus
3. Transmission avec déconnection
4. Réseau de trafic ferroviaire
5. Transmission synchrone et asynchrone
6. Réseau de distributeurs de billets

4. Conclusion

IV. PREUVE PAR EVALUATION DES FORMULES DE SPECIFICATION

1. Introduction
2. Principe de l'évaluation

1. Le transformateur de prédicats pre
2. Invariants et trajectoires
3. Rappels sur les points fixes des fonctions monotones
4. Calcul itératif des opérateurs temporels

3. Evaluation dans le graphe des états

1. Principe de la méthode
2. Construction du graphe des états
3. Evaluation des prédicats non temporels

1. Variables propositionnelles associées aux actions
2. Prédicats sur les variables de l'application

4. Evaluation des formules

1. Représentation des formules
2. Evaluation des formules
3. Calcul itératif des opérateurs

5. Un exemple d'évaluation

4. Le cas particulier de la logique S4

    1. Présentation du problème
    2. Construction du graphe réduit des états
    3. Evaluation des formules

        1. Evaluation des prédicats non temporels
        2. Calcul itératif des opérateurs

    4. Un exemple d'évaluation

5. Evaluation dans le treillis des prédicats

    1. Principe de la méthode
    2. Codage d'un CA-système

        1. Choix du codage
        2. Restrictions engendrées par le codage
        3. Codage des variables
        4. Codage des conditions
        5. Codage des actions

    3. Rappels sur le calcul booléen
    4. Evaluation des prédicats non temporels
    5. Evaluation des formules

        1. Calcul du transformateur de prédicats pre
        2. Calcul itératif des opérateurs

    6. Un exemple d'évaluation

6. Programmation et comparaison des méthodes

    1. Schéma de programmation de l'évaluation

        1. Structures de données
        2. Algorithme d'évaluation

    2. Comparaison des méthodes

7. Conclusion

V. UTILISATION DE QUASAR

1. Introduction
2. Rappels sur le système MULTICS
3. Présentation du système QUASAR
4. Liste des commandes utilisateur

    1. Commande cas
    2. Commande comp

3. Commande sim
4. Commande def
5. Commande eval
6. Commande decide
7. Commande del
8. Commande e
9. Commande help
10. Commande listcas
11. Commande print
12. Commande q
13. Commande @

5. Schéma d'une session
6. Exemples complets d'analyse

1. Exclusion mutuelle de Decker
2. Protocole d'allocation de bus
3. Transmission avec déconnection
4. Réseau de trafic ferroviaire
5. Transmission synchrone et asynchrone
6. Réseau de distributeurs de billets

7. Conclusion

VI. CONCLUSION - BILAN ET PERSPECTIVES

1. Bilan de l'expérience
2. Propositions d'améliorations
3. Conclusion

ANNEXE : ORGANISATION DES PROGRAMMES

REFERENCES

CHAPITRE I

```
*****************************************
*                                       *
*             INTRODUCTION              *
*   PRESENTATION DU SYSTEME QUASAR      *
*                                       *
*****************************************
```

1. Objectifs
2. Présentation du système CESAR
3. Origine du système QUASAR
4. Le système QUASAR

1. OBJECTIFS

L'objectif de cette thèse est de présenter une réalisation d'un système d'aide à la conception et à la validation d'architectures d'applications réparties. Le système, dénommé QUASAR, est une version expérimentale simplifiée d'un projet plus ambitieux, le système CESAR, qui tente de répondre en partie aux impératifs exprimés par les concepteurs d'applications réparties.

La notion de parallélisme est apparue ces dernières années dans les systèmes informatiques. La plupart des systèmes sont conçus aujourd'hui comme des ensembles de sous systèmes évoluant en parallèle. Les différents sous systèmes sont indépendants entre eux du point de vue de leur fonctionnement interne, mais doivent respecter un ensemble de contraintes, d'ordre physique (partage d'une ressource par plusieurs processus) ou logique (collaboration et communication entre constituants d'un même système). Nous dénommerons "application répartie" toute application mono- ou multi-tâches utilisant la notion de parallélisme. Le parallélisme peut être réel ou simulé (machines comportant plusieurs processeurs ou un processeur multi-programmé), les différentes tâches peuvent être logicielles ou réalisées par des organes matériels géographiquement distants, et des contraintes temporelles peuvent intervenir (applications temps-réel).

La mise en oeuvre de tels systèmes soulève de nombreux problèmes, tant sur le plan de la conception (modularisation et définition des constituants) que sur celui de la preuve du bon fonctionnement des systèmes réalisés. Aussi les concepteurs d'applications réparties expriment-ils le besoin pressant de disposer d'un système d'aide à la conception et à l'intégration qui leur permet :

- de décrire les différents modules de l'application à concevoir ;

- de définir l'architecture du système à concevoir à l'aide de ces modules ;

- d'évaluer la conformité de cette architecture à une spécification de son comportement souhaité.

Pour répondre à ces impératifs, nous proposons la réalisation d'un système d'aide à la conception qui fournit à l'utilisateur :

- 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 ;
* autorisant une description indépendante des contraintes d'implémentation matérielles, logicielles ou syntaxiques ;
* permettant la description algorithmique des processus physiques, et leurs réactions à des événements extérieurs (au moyen de commandes gardées comportant des échanges) ;

- un langage de spécification du fonctionnement des systèmes, sous la forme d'un ensemble de propriétés devant être vérifiées ;

- un système d'analyse permettant de comparer la description et la spécification, c.à.d. de vérifier que les propriétés requises sont garanties par l'architecture décrite.

Le système CESAR répond en grande partie à ces objectifs, en permettant au concepteur d'une application répartie d'évaluer l'architecture qu'il propose, en ce sens qu'il peut déterminer l'ensemble des propriétés qui sont vérifiées parmi celles qu'il a spécifiées. Il peut ainsi comparer différentes architectures possibles pour une même application, et éliminer celles qui ne vérifient pas une propriété fondamentale.

2. PRESENTATION DU SYSTEME CESAR

Le système CESAR (Certification, Evaluation et Spécification des Applications Réparties) [QUE82] constitue une proposition pour un système d'aide à la conception et à l'intégration des applications réparties. CESAR est un projet ambitieux, dont la complexité n'a pas permis une réalisation complète ; c'est pourquoi un prototype de ce système, QUASAR, a été construit dans le but de prouver la faisabilité des algorithmes de CESAR.
Une présentation détaillée de CESAR nous parait cependant indispensable pour mettre en relief la valeur et les limitations de ce système, et pour établir avec précision la distance qui sépare ce projet de la réalisation que nous présentons dans cette thèse.

Le principe du système CESAR repose sur la traduction de la description d'une application à étudier vers un modèle mathématique statiquement analysable. Les spécifications sont exprimées par des formules d'une logique temporelle qui sont interprétées dans le modèle. La conformité des propriétés avec le programme de description est vérifiée par évaluation dans le modèle de l'interprétation des formules temporelles, par des calculs de points fixes d'opérateurs monotones.

2.1. LE LANGAGE DE DESCRIPTION DE CESAR

Le langage de description de CESAR, syntaxiquement proche du langage ADA [ADA80], permet de représenter une application répartie par un ensemble de tâches communiquant par échanges de messages.

Toutes les données sont typées. En plus des types et constructeurs de types usuels, l'utilisateur peut définir des types (dits non spécifiés) sans leur associer une implémentation en termes de types standard, et manipuler des constantes et variables de ces types.

Les instructions structurées habituelles (IF, CASE, FOR, WHILE) sont transformées pour permettre un choix non déterministe si les conditions dont elles dépendent sont non spécifiées.

Les particularités du langage de description de CESAR sont les suivantes :

- affectation simultanée (vectorielle) de plusieurs variables ;

- instructions d'échanges (notées ! et ?) inspirées du langage CSP [HOA78] ;

- instruction EITHER permettant le choix non déterministe entre commandes gardées [DIJ75] comportant éventuellement des échanges.

Contrairement au langage CSP, les valeurs échangées sont identifiées par des noms de variables échangées, mais les processus partenaires ne le sont pas. Chaque variable échangée peut avoir plusieurs producteurs possibles et plusieurs consommateurs possibles. Les échanges s'effectuent par rendez-vous selon deux modes : le mode "anonyme" entre l'un quelconque de ses producteurs et l'un quelconque de ses consommateurs ; le mode "diffusion" entre l'un quelconque de ses producteurs et l'ensemble de ses consommateurs.

Les traitements déclarés (regroupés en tâches et en procédures) peuvent ne pas être décrits, et sont alors 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 les 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 au sein des tâches composées.

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

2.2. LE LANGAGE DE SPECIFICATION DE CESAR

Chaque propriété comportementale de l'application est exprimée par une formule d'une logique comportant des opérateurs temporels, c.à.d. des opérateurs prenant en compte la notion de temps.

Les variables propositionnelles de cette logique représentent des prédicats sur l'état de l'application, et sont de deux types :
- des relations sur les données ;
- des états du contrôle des différentes tâches.

Les opérateurs temporels permettent d'exprimer des notions telles que :

- l'atteignabilité d'un prédicat (potentielle, inévitable, ...) ;

- le fait qu'un prédicat reste toujours vrai au cours de l'évolution du système.

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

2.3. VERIFICATION DES PROPRIETES DANS CESAR

La vérification de la conformité des propriétés de spécification avec la description d'une application se fait en deux étapes :

- le programme de description est traduit en un modèle conservant la totalité de l'information nécessaire à l'analyse ;

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

Le modèle retenu pour la traduction est le réseau de Petri interprété [SIF79]. Un réseau de Petri interprété est un réseau de Petri muni d'un ensemble X de variables, et dont les transitions sont étiquetées par un couple (c,a), où c est une condition booléenne sur les variables de X, et a une affectation des éléments de X. La mise à feu d'une transition n'est possible que si ses

places d'entrées sont marquées et si la condition c est vraie ; la mise à feu provoque, en plus du déplacement de marques habituel des réseaux de Petri, la modification de la valeur des variables de X par exécution de l'action a.

La traduction d'un programme de description s'effectue en deux phases : la traduction de chaque tâche élémentaire, puis la construction des réseaux représentant les tâches composées (pour aboutir au système complet) à partir des réseaux ainsi générés. La première étape relève de la compilation classique : analyse syntaxique, puis traduction des tâches élémentaires à l'aide d'une grammaire de "briques" de réseaux de Petri interprétés, dont chaque règle associe à une instruction composée un sous réseau. La deuxième étape utilise une opération de composition entre réseaux, fondée sur la fusion de transitions dont l'interprétation comporte des échanges, de façon à représenter les rendez-vous.

L'analyse repose sur l'évaluation, pour chaque formule 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.

3. ORIGINE DU SYSTEME QUASAR

Les ambitions initiales de l'équipe au sein de laquelle a été développé le projet CESAR ont été de concevoir une maquette réalisant intégralement le système d'analyse. Les objectifs étaient doubles : prouver tout d'abord la faisabilité de la méthode, par la programmation des différentes fonctions de l'analyseur, mener ensuite l'expérience à son terme pour disposer d'un système complet et (si possible) efficace d'aide à la conception et à l'intégration des systèmes répartis, prêt à l'emploi.

La mise au point des programmes souleva immédiatement des problèmes de natures diverses :

- le matériel utilisé pour tester les programmes venait d'être installé, et présentait de graves imperfections qui ont entrainé une grande perte de temps ;
- le projet CESAR n'était pas défini dans tous ses détails : le langage de description a suivi une évolution importante pendant cette période, et les programmes d'évaluation se sont heurtés à des problèmes de saturation mémoire du calculateur et à des coûts d'exécution prohibitifs ;
- le cahier de charges qui avait été fixé s'est avéré trop important, en partie à cause des problèmes rencontrés.

En fin de compte, des programmes utilisables ont été mis au point pendant ce laps de temps, mais les objectifs étaient loins d'être atteints. Pour ne pas abandonner une expérience malgré tout avancée, car les problèmes majeurs rencontrés avaient été résolus, il fut alors décidé de réduire les objectifs, et de mettre au point un système d'analyse moins ambitieux, mais conservant les caractéristiques du fonctionnement de CESAR.

Ainsi naquit QUASAR, un système d'aide à la conception moins riche que CESAR, mais qui a pu être entièrement achevé. Cette thèse constitue une description complète des principes employés pour l'analyse des systèmes répartis, en même temps qu'un manuel d'utilisation du programme qui réalise QUASAR.

4. LE SYSTEME QUASAR

Les objectifs du système QUASAR sont de réaliser un système d'aide à la conception et à l'intégration des applications réparties, similaire à CESAR, mais diminué d'un certain nombre de possibilités afin de simplifier la mise au point des programmes qui constituent le système d'analyse.

QUASAR est une version très simplifiée de CESAR, mais qui conserve toutes les idées essentielles de ce projet. QUASAR n'est pas toujours d'un emploi très pratique, mais il possède en théorie toute la puisssance du système CESAR initialement proposé, pourvu que l'utilisateur accepte de se plier aux contraintes qu'il impose.

QUASAR est organisé autour d'un mini-langage de description dans lequel les tâches élémentaires sont chacunes décrites sous la forme d'un ensemble plat (c.à.d. non ordonné et ne comportant pas de structures de blocs) de commandes gardées. Les commandes gardées sont des couples (condition, action), chaque action étant constituée d'un ensemble d'affectations simultanées, précédé éventuellement d'un échange. Toutes les variables sont de type intervalle borné d'entiers. La structuration en tâches élémentaires et composées a été conservée, ainsi que les deux modes de rendez-vous. La puissance du langage de description est ainsi conservée, mais l'utilisateur peut être amené à gérer explicitement dans chacune des tâches élémentaires le séquencement des actions, au moyen d'une variable jouant le rôle d'un compteur ordinal, alors que le langage intialement proposé utilisait pour cela les instructions composées usuelles.

Le modèle sous-jacent est un système conditions-actions, obtenu au moyen d'une opération de composition entre systèmes conditions-actions communicants [QUE 81], identique dans le principe à celle définie pour les réseaux de Petri interprétés [QUE82], en

faisant abstraction du réseau de Petri représentant le contrôle des tâches.

Le langage de description et sa traduction vers un système conditions-actions sont présentés de façon plus détaillée dans le chapitre II de la thèse.

Le langage de spécification de CESAR a été conservé intégralement. Par ailleurs, un outil d'aide à la spécification des systèmes étudiés a été constitué par la mise en oeuvre d'une procédure de décision de la logique temporelle utilisée pour l'expression des propriétés comportementales des systèmes. L'utilisateur peut ainsi étudier et comparer des formules de la logique de façon formelle, pour détecter des redondances ou des inconsistances dans un ensemble de spécifications.

La logique temporelle, sa procédure de décision, et le langage de spécification sont décrits dans le chapitre III.

L'analyse des propriétés de spécification passe par la construction du graphe des états d'une application. Chaque formule de spécification est interprétée comme un sous ensemble de l'espace d'états du système, et son évaluation consiste à calculer explicitement le sous ensemble des états du système qui vérifient la formule. La comparaison de ce sous ensemble avec l'ensemble de tous les états possibles du système permet de décider si la formule est valide pour l'application on non.

Nous présentons au quatrième chapitre la méthode d'évaluation qui a été retenue dans QUASAR, ainsi que d'autres formes d'analyse qui ont été expérimentées.

Le cinquième et dernier chapitre de cette thèse constitue un mode d'emploi du programme d'analyse qui a été réalisé. Nous décrivons les possibilités offertes à l'utilisateur, par l'intermédiaire des commandes dont il dispose pour décrire, spécifier et vérifier des systèmes répartis.

Nous présentons finalement en annexe un schéma de l'organisation du programme qui réalise le système QUASAR. Nous discutons les problèmes soulevés par la mise au point du système d'analyse, en particulier les contraintes imposées par l'implémentation du programme.

CHAPITRE II

```
**********************************
*                                *
*   LE LANGAGE DE DESCRIPTION    *
*       ET SA TRADUCTION         *
*                                *
**********************************
```

1. Introduction
2. Le langage de description
3. Traduction d'un programme
4. Conclusion

1. INTRODUCTION

L'objet de ce chapitre est de présenter le langage de description du système QUASAR, et sa traduction vers un programme PASCAL servant à générer le système de transitions correspondant à une application.

Le langage de description de QUASAR, inspiré du langage de description de CESAR, a été mis au point dans le but de simplifier au maximum l'aspect compilation dans le système d'analyse. Il nous est apparu dans CESAR, qu'une partie importante de la traduction d'un programme de description vers un réseau de Petri interprété relevait en définitive de la compilation classique (traduction des tâches élémentaires, vérification de la compatibilité des types ...) et ne présentait donc pas une importance fondamentale.

Il nous a semblé préférable de mettre l'accent sur les parties de CESAR qui présentaient, elles, des aspects nouveaux. C'est pourquoi nous avons délibérément affaibli le langage de description de CESAR, pour n'en conserver que les traits caractéristiques :

- les tâches élémentaires sont chacunes décrites sous forme d'un ensemble plat (non structuré) de commandes gardées, pouvant comporter des échanges ;

- toutes les variables, internes ou échangées, sont du type intervalle d'entiers borné ;

- la structuration des systèmes en tâches élémentaires et tâches composées a été conservée, ainsi que les deux modes de rendez-vous.

La puissance de description de CESAR est ainsi conservée dans QUASAR, et la part de la compilation dans le système d'analyse est réduite au minimum.

La description algorithmique d'un système distribué est exploitée pour générer le système de transitions représentant l'application. Cette opération s'effectue en deux phases :

- compilation de la description fournie : construction de la table des identificateurs, traduction des tâches élémentaires en un langage intermédiaire, construction des tâches composées à partir des tâches élémentaires, puis génération de procédures PASCAL représentant l'application décrite ;

- utilisation des procédures PASCAL de façon à générer le système de transitions : construction de l'ensemble des états initiaux possibles, puis génération de tous les états par exécution systématique de toutes les transitions exécutables à partir de chacun des états initiaux.

Nous présentons dans un premier temps le langage de description du système QUASAR. Nous illustrons les possibilités du langage par la donnée du programme de description de quelques exemples de systèmes répartis. Nous étudierons ensuite la phase de traduction d'une application vers un programme PASCAL.

2. LE LANGAGE DE DESCRIPTION DE QUASAR

Le langage de description de QUASAR est un langage algorithmique qui permet de représenter les applications réparties par un ensemble de tâches communiquant par échanges de messages. Une application est décrite de façon hiérarchisée selon une structure arborescente dont les feuilles sont les tâches élémentaires - ou unités d'exécution d'instructions - . Les autres noeuds de l'arbre sont des tâches composées qui permettent de regrouper plusieurs tâches, élémentaires ou composées, s'exécutant en parallèle. Cette représentation permet, outre la représentation de la structure fonctionnelle du système, la limitation de la portée des échanges, en les effectuant au sein des tâches composées.

Les processus élémentaires de l'application sont représentés par des tâches élémentaires. Chaque tâche élémentaire consiste en la déclaration de ses paramètres échangés et de ses variables internes, et en un corps de tâche, qui décrit le fonctionnement du processus. Le corps d'une tâche élémentaire est formé d'un ensemble plat de commandes gardées, les gardes étant des conditions booléennes sur les variables internes de la tâche, et les actions étant des affectations simultanées, précédées éventuellement d'un échange.

Le langage de description offre les possibilités suivantes :

- affectations simultanées des variables d'une tâche élémentaire, pour éviter les contraintes de séquentialité entre affectations, présentes dans les langages de programmation classiques ;
- instructions d'échanges (notées ! pour l'émission et ? pour la réception) inspirées du langage CSP [HOA78] ;
- choix non déterministe entre commandes gardées comportant éventuellement des échanges (comme dans CSP, mais sans privilégier un sens d'échange par rapport à l'autre).

Contrairement au langage CSP, les valeurs échangées sont identifiées par des noms de variables échangées (analogues aux noms de portes [MM79]) , mais les processus partenaires ne le sont pas. Chaque variable échangée peut avoir plusieurs producteurs et plusieurs consommateurs possibles. Les échanges s'effectuent par rendez-vous selon deux modes :

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

Nous donnons tout d'abord la syntaxe du langage de description de QUASAR, et nous détaillons ensuite les différents points importants du langage.

2.1. SYNTAXE DU LANGAGE DE DESCRIPTION

Le langage de description est défini par la grammaire ci-dessous, dont l´ axiome est le non terminal <programme>. Les notations suivantes seront utilisées :

<xxx> : symbole non terminal ;
xxx : mot-clé ;
[xxx] : occurrence éventuelle de xxx ;
[xxx]* : occurrence éventuelle de xxx , un nombre quelconque de fois (0,1 ou pusieurs) ;
[xxx]+ : occurrence de xxx , un nombre positif de fois (au moins une fois).

Comme d´habitude, le caractère "|" est utilisé pour délimiter les alternatives des parties droites de règles. Les symboles terminaux "[","]" et "|" de la grammaire sont écrits respectivement _[, _] et _| .

```
<programme> ::= <tâche> .

<tâche> ::= <tâche élémentaire> |
            <tâche composée>

<tâche élémentaire> ::=
            TASK <identificateur> ;
            [ <entrées> ]
            [ <sorties> ]
            [ <déclarations> ]
            [ <initialisations> ]
            DO
            <commande gardée>
            [ _| <commande gardée> ]*
            OD

<entrées> ::= INPUT
              <identificateur>
              [ , <identificateur> ]* ;

<sorties> ::= OUTPUT
              <identificateur>
              [ , <identificateur> ]* ;

<déclarations> ::= DECLARE  [ <dcl> ]+

<dcl> ::= <identificateur> : <nombre> .. <nombre> ;

<initialisations> ::= INIT
```

```
                        <affectation>
                        [ , <affectation> ]* ;

<commande gardée> ::= [ <étiquette> ]*
                        <condition> :
                        [ <échange> [ , <affectation ]* ]

<étiquette> ::= { [ <caractère> ]+ }

<échange> ::=
        ! <identificateur> [ := <expression> ] |
        [ <identificateur> , ]+
            ! <identificateur> := <expression> |
        [ [ <identificateur> , ]* <identificateur> := ]
            ? <identificateur>                  |
        <affectation>

<affectation> ::= <identificateur> := <expression>

<tâche composée> ::=
        COTASK <identificateur> ;
        [ <entrées> ]
        [ <sorties> ]
        [ <tâches> ; ]+
        [ <variables diffusées> ]
        [ <variables anonymes> ]
        BODY
        <instance de tâche>
        [ // <instance de tâche> ]*

<variables diffusées> ::= BROAD
                          <identificateur>
                          [ , <identificateur> ]* ;

<variables anonymes> ::= PORT
                         <identificateur>
                         [ , <identificateur> ]* ;

<instance de tâche> ::= <identificateur>
                        [ ( <identificateur>
                        [ , <identificateur> ]* ) ]
```

Les non terminaux suivants sont définis selon la syntaxe du langage PASCAL :

<identificateur> : le symbole "_" est admis ;
<nombre> : nombre entier signé ;
<condition> : condition à résultat booléen ;
<expression> : expression à résultat entier.

Les commentaires PASCAL (entre les symboles "(*" et "*)" unique-

ment) sont admis.

2.2. DECLARATION DE TACHE ELEMENTAIRE

Une tâche élémentaire est déclarée par le mot-clé TASK suivi de son nom (identificateur), et de la définition de ses éventuels paramètres échangés. Les paramètres reçus sont déclarés au moyen d'une directive INPUT et ceux émis au moyen d'une directive OUTPUT. Tous les paramètres formels échangés doivent avoir des noms distincts, et sont du type entier universel (entier signé non borné). L'ordre dans lequel les paramètres formels échangés sont déclarés dans les directives INPUT et OUTPUT est significatif (la correspondance paramètre formel / paramètre effectif est faite par position).

Une tâche élémentaire comporte ensuite la déclaration de ses variables internes, l'initialisation éventuelle de celles-ci, puis le corps de la tâche.

Exemples :

```
1 ) TASK SENDER ;
      INPUT ACKNOWLEDGEMENT ;
      OUTPUT MESSAGE ;

2 ) TASK PRODUCER ;
      OUTPUT PRODUCT ;
      (* pas de paramètre reçu *)
```

2.3. DECLARATION DES VARIABLES INTERNES ET INITIALISATION

Les variables internes des tâches élémentaires sont toutes d'un type intervalle d'entiers borné. Leur déclaration précise les bornes de l' intervalle de définition, sous la forme de deux entiers signés séparés par le symbole "..", le premier devant être inférieur ou égal au second. Les identificateurs des variables et les nombres doivent respecter la syntaxe PASCAL. Les valeurs maximum et minimum des nombres sont celles du type INTEGER du langage PASCAL sur le système MULTICS.

Les déclarations des variables forment un paragraphe introduit par le mot-clé DECLARE. Ce paragraphe est en principe suivi d'un paragraphe introduit par le mot-clé INIT, formé d'un ensemble d'affectations définissant la valeur initiale des variables déclarées. Toute variable non initialisée dans ce paragraphe est supposée avoir une valeur quelconque à l'intérieur de son intervalle

de définition.

Exemples :

```
1 ) DECLARE
        SEMAPHORE : 0..1 ;
    INIT
        SEMAPHORE := 1 ;

2 ) DECLARE
        POS_X : 1..10 ;
        POS_Y : 1..20 ;
        POS_Z : 1..5  ;
    INIT
        POS_X := 1 , POS_Y := 1 ;
        (* POS_Z n'est pas initialisée *)
```

2.4. EXPRESSIONS

Toutes les expressions doivent respecter la syntaxe PASCAL. Elles sont de deux types :

- celles utilisées dans les affectations, dont le résultat est nécessairement du type entier, et doit appartenir à l'intervalle de définition de la (des) variable (s) affectée (s) . De plus, les expressions utilisées dans les affectations du paragraphe d'initialisation doivent être statiquement évaluables, c'est à dire n'utiliser que des nombres littéraux et des opérateurs ou fonctions standard du langage PASCAL (pas de variables) ;
- celles utilisées comme conditions dans les commandes gardées, dont le résultat est nécessairement du type booléen.

Exemples :

```
1 ) 4 * ( 3 + POS_X )
    (* expression à résultat entier *)

2 ) ( POS_X <= 6 ) OR ( POS_Z in [ 2..4 ] )
    (* expression à résultat booléen utilisant
       l'opérateur d'appartenance PASCAL "in" *)
```

2.5. CORPS DE TACHE ELEMENTAIRE

Le corps des tâches élémentaires est une instruction DO...OD unique regroupant un ensemble de commandes gardées séparées par le sym-

bole "|". Chaque commande gardée est formée d'une partie condition, suivie du symbole ":", et d'un ensemble (éventuellement vide) d'affectations simultanées, précédé éventuellement par un échange.

Une émission (resp. réception) est une affectation dont la partie gauche (resp. droite) est un nom de paramètre formel échangé, précédé d'un point d'exclamation (resp. d'interrogation). Ce paramètre doit figurer dans la directive OUTPUT (resp. INPUT) définissant les paramètres échangés de la tâche élémentaire où figure l'échange. Une valeur émise ou reçue peut être simultanément affectée à une ou plusieurs variables internes. Lorsque la valeur émise est non significative (synchronisation pure), celle-ci peut être omise, ainsi que le symbole d'affectation ":=" ; la valeur reçue par le partenaire est alors elle aussi non significative, et ne doit pas être affectée à une variable interne.

L'exécution simultanée des affectations d'une commande gardée implique l'interdiction d'affecter une même variable plusieurs fois dans une commande gardée.

Les commandes gardées peuvent être précédées d'un ensemble d'étiquettes (chaînes de caractères quelconques entre les symboles "{" et "}", le premier caractère non blanc devant être différent de "#") utilisées pour clarifier le programme de description et aider à repérer l'origine des transitions figurant dans le système conditions-actions représentant l'application à l'issue de l'opération de composition entre tâches élémentaires.
Les étiquettes d'un programme de description pourront être utilisées dans les formules de spécification pour exprimer le positionnement d'une tâche par rapport à une action étiquetée.

L'exécution de l'instruction DO...OD formant le corps d'une tâche élémentaire se fait de la façon suivante :

1) évaluation de toutes les gardes ; si une garde est vraie, la commande gardée correspondante est toujours réputée exécutable, si elle ne comporte pas d'échange ; dans le cas contraire, la commande gardée n'est réputée exécutable que si, d'une part, la garde est vraie et, d'autre part, l'échange est lui-même exécutable, c.à.d. que toutes les tâches impliquées par le rendez-vous sont dans un état qui leur permet de participer à ce rendez-vous ;

2) choix (non déterministe) d'une commande gardée exécutable, et exécution de celle-ci ; si elle ne comporte pas d'échange, il s'agit de la simple exécution vectorielle de l'ensemble des affectations figurant dans la commande gardée ; dans le cas contraire, toutes les commandes gardées participant au rendez-vous représentant l'échange effectué sont exécutées simultanément dans les différentes tâches élémentaires où elles figurent (avec substitution de la valeur émise dans les instructions de réception).

Les différentes instructions DO...OD représentant les corps des différentes tâches élémentaires s'exécutent sans aucune interaction autre que les rendez-vous.

Exemples :

1) d'affectations simultanées sans échange :
```
    INDEX := INDEX MOD 100 + 1 , COUNT := COUNT - 1 ;
```

2) d'échanges :
```
    !X := Y , Y:= Y-1 ;
      (* émission de la valeur de Y, et décrémentation de Y *)
    X, Y, Z := ? ZO ;
      (* réception d'une valeur, et affectation de cette valeur
         au trois variables X,Y et Z *)
    X, Y, ! Z := ZO , COMPTEUR := 1 ;
      (* émission de la valeur de ZO, affectation de cette valeur
         aux variables X et Y, et affectation de la variable
         COMPTEUR *)
    ?P ;
      (* réception sans affectation, pour représenter une
         synchronisation pure *)
```

3) d'instruction DO...OD :
```
    DO
      {ranger} COUNT<100 : ? C_IN , COUNT:=COUNT+1   |
      {sortir} COUNT>0   : ! C_OUT, COUNT:=COUNT-1
    OD
```

2.6. DECLARATION DE TACHE COMPOSEE

Une tâche composée est déclarée par le mot-clé COTASK suivi de son nom (identificateur) et des (éventuelles) directives INPUT et OUTPUT définissant ses paramètres formels échangés. Ceux-ci constituent la seule partie visible de la tâche composée. Ils sont utilisés comme variables échangées à l'intérieur de la tâche composée.

Comme pour les tâches élémentaires, tous les paramètres échangés par la tâche composée doivent avoir des noms distincts, et sont du type entier universel.

La tâche composée comporte ensuite :

- les déclarations des tâches (élémentaires ou composées) dont celle-ci est formée ;
- la déclaration des noms des variables échangées utilisées pour établir les liens entre les paramètres échangés des tâches composantes ;

- le corps de la tâche.

Exemple :

```
COTASK CC ;
  INPUT C_IN ;
  OUTPUT C1_OUT,C2_OUT ;
```

2.7. DECLARATION DES VARIABLES ECHANGEES

Une variable échangée sert de porte de communication entre deux ou plusieurs tâches composantes, pour effectuer la liaison entre une émission et une (ou plusieurs) réception (s). Les variables échangées sont déclarées au moyen des directives BROAD et PORT, correspondant respectivement aux modes diffusion et anonyme. La signification des deux modes est la suivante :

- mode diffusion (directive BROAD) : un rendez-vous pour l'échange de cette variable ne peut avoir lieu qu'entre l'un quelconque des émetteurs potentiels de çette variable et la totalité des récepteurs de celle-ci, qui doivent donc être tous dans un état où le rendez-vous est possible ;
- mode anonyme (directive PORT) : un rendez-vous pour l'échange de cette variable peut avoir lieu entre l'un quelconque de ses émetteurs et l'un quelconque de ses récepteurs ;

Les variables échangées déclarées dans une tâche composée sont utilisées pour les rendez-vous internes entre les tâches composantes. Les paramètres échangés de la tâche composée sont aussi utilisés comme variables échangées, mais ils permettent, en plus des rendez-vous internes de la tâche, d'effectuer des rendez-vous entre les tâches composantes et des tâches extérieures à la tâche composée. Les paramètres de la tâche composée déclarés par les directives INPUT et OUTPUT doivent être redéclarés dans les directives BROAD et PORT de façon à définir leur mode d'échange entre les tâches composantes.

2.8. CORPS DE TACHE COMPOSEE

Le corps d'une tâche composée est introduit par le mot-clé BODY, et est formé d'un ensemble d'instances de tâches séparées par le symbole "//" . Chaque instance de tâche est composée d'un nom de tâche (déclarée à l'intérieur de la tâche composée) suivi par la liste entre parenthèses des noms des variables échangées (déclarées à l'intérieur de la tâche composée) ou des paramètres (de la tâche

composée) qu'elle reçoit ou émet. La correspondance entre nom de variable échangée (ou paramètre de la tâche composée) et paramètre formel est faite par position.

Exemples :

```
1 ) COTASK PROD_CONS ;
    (* système producteur consommateur simple *)

      TASK PRODUCER ;
        OUTPUT OUTOBJ ;
      DO ... OD ;

      TASK CONSUMMER ;
        INPUT INOBJ ;
      DO ... OD ;

      TASK BUFFER ;
        INPUT OBJIN ;
        OUTPUT OBJOUT ;
      DO ... OD ;

    PORT X_PROD, X_CONS ;
    BODY
      PRODUCER ( X_PROD ) //
      BUFFER ( X_PROD , X_CONS ) // CONSUMMER ( X_CONS )

2 ) COTASK SEND_RECEIVE ;
    (* réception simultanée d'un message par deux tâches,
       et envois séparés de compte rendus *)

      TASK SENDER ;
        INPUT ACKIN1, ACKIN2 ;
        OUTPUT MESSAGEOUT ;
      DO ... OD ;

      TASK RECEIVER1 ;
        INPUT MESSAGEIN1 ;
        OUTPUT ACKOUT1 ;
      DO ... OD ;

      TASK RECEIVER2 ;
        INPUT MESSAGEIN2 ;
        OUTPUT ACKOUT2 ;
      DO ... OD ;

    PORT ACK1, ACK2 ;
    BROAD MESSAGE ;
    BODY
      SENDER (ACK1,ACK2,MESSAGE) //
      RECEIVER1 (MESSAGE,ACK1) //
```

RECEIVER2 (MESSAGE,ACK2) ;

2.9. REMARQUE SUR LES IDENTIFICATEURS

Dans la maquette réalisée, il est nécessaire que tous les identificateurs de tâches, de paramètres formels, de variables internes et échangées soient différents. Les lettres majuscules et minuscules sont équivalentes, mais le caractère de soulignement est significatif.

La longueur des identificateurs est positive et quelconque, mais seuls les 30 premiers caractères sont pris en compte. De la même façon, les étiquettes ne comportent que 30 caractères significatifs.

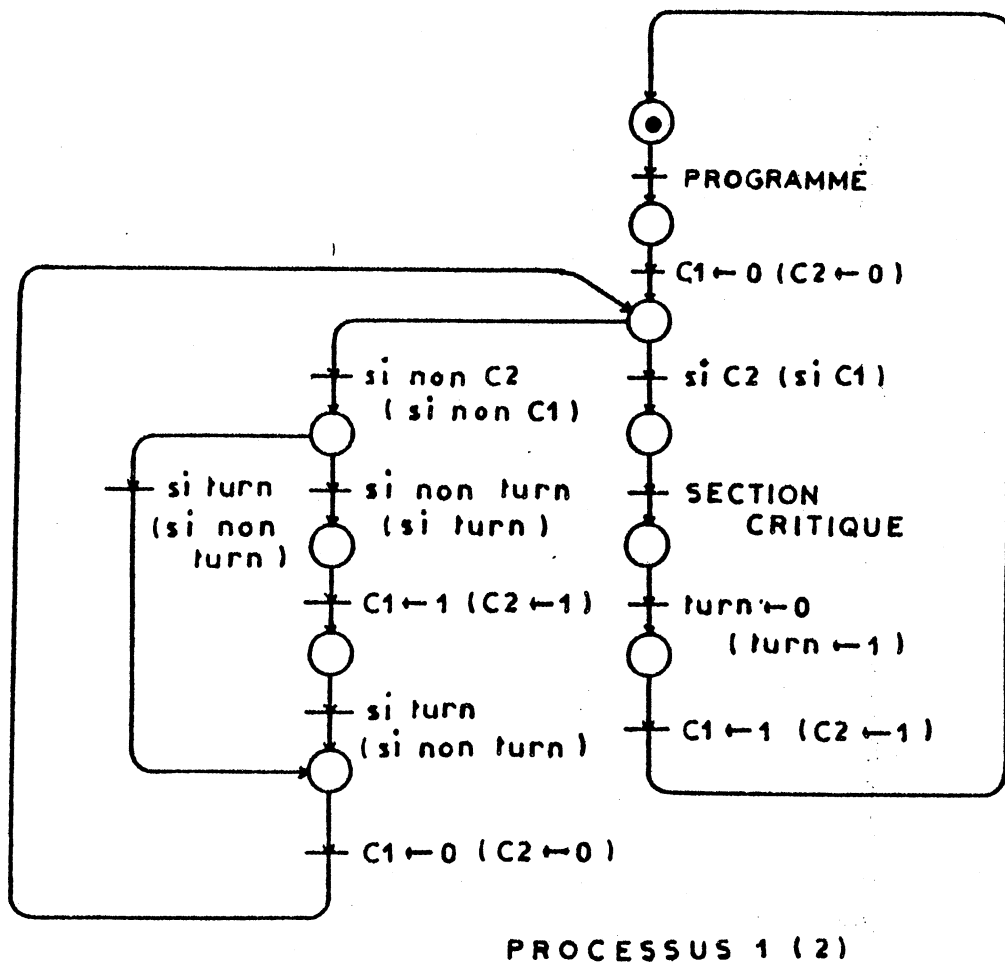
2.10. EXEMPLES

Nous donnons ici quelques exemples de description de systèmes répartis. Ces exemples illustrent les possibilités du langage du système QUASAR, et seront repris ultérieurement dans les chapitres présentant la traduction et l' analyse des applications réparties.

2.10.1. EXCLUSION MUTUELLE DE DECKER

L'exemple que nous allons voir a été proposé par DECKER et communiqué par DIJKSTRA [DIJ68]. Il s'agit d'un algorithme qui permet l'exclusion mutuelle entre deux processus P1 et P2 qui partagent une mémoire commune. Chaque processus Pi positionne une variable booléenne Ci à faux lorsqu'il souhaite utiliser la ressource, et une priorité alternée (représentée par un booléen TURN) permet de régler les conflits dus aux demandes simultanées. Nous donnons tout d'abord une représentation de l'application par un système de transitions [GUI82], et nous présentons ensuite le programme de description dans la langage de QUASAR.

figure 2.1



PROCESSUS 1 (2)

La description de l'application dans le langage de QUASAR soulève un problème important. S'il parait évident de représenter les deux processus P1 et P2 par deux tâches élémentaires, il semble plus délicat de matérialiser l' interface entre les deux processus : les tâches ne partagent pas de variables communes, alors que les processus de l'algorithme de DECKER peuvent tous deux tester la valeur des variables C1, C2 et TURN. Comment peut-on remédier à l' absence de variables partagées entre tâches ? On peut envisager la présence de copies locales des variables C1, C2 et TURN, et l'actualisation de celles-ci par échange entre les deux tâches. Cette façon de faire risque néanmoins de compliquer sérieusement le système et ne nous parait pas satisfaisante quant au fond du problème. Il semble plus opportun de regrouper les phases de "scheduling" locales aux processus P1 et P2 (tests des valeurs de C1,C2 et TURN) sous forme

d'un contrôleur explicite qui enregistre les demandes d'accès à la mémoire commune, et assure l'exclusion mutuelle par l'envoi d'autorisations conditionnant l'accès effectif à la ressource par les processus.

La description que nous proposons est constituée de trois tâches P1, P2 et CONTROLER. Les tâches P1 et P2 décrivent le fonctionnement cyclique des deux processus qui utilisent la mémoire. La tâche CONTROLER gère les accès des processus à la mémoire. Le traitement effectué par cette tâche est une transcription du réseau de Petri présenté dans le langage de QUASAR. Toutefois, plusieurs simplifications sont apportées pour condenser le programme :

- les transitions en séquence qui ne comportent que des tests des variables C1, C2 et TURN sont regroupées pour n'avoir qu'un seul test ;
- les transitions inconditionnelles en séquence, qui affectent des ensembles de variables disjoints sont regroupées ;
- la séquence de transitions "si non C2", "si TURN", "C1:=0" relative au processus P1 n'est pas reproduite, car elle ne provoque aucune modification de l'état du système. La même séquence, relative à P2, est elle aussi supprimée.

Le programme de description proposé est le suivant :

```
cotask DECKER ;

   task P1 ;
      input AU1 ; (* autorisation d'accès *)
      output DE1,FI1 ; (* demande, fin d'utilisation *)
      declare
         Y1 : 0..3 ; (* compteur ordinal interne *)
      init
         Y1 := 0 ;
      do
        {local_p1}
          Y1=0 : Y1:= 1 |
        {dem_res1}
          Y1=1 : !DE1, Y1:= 2  |
        {aut_acces1}
          Y1=2 : ?AU1, Y1:= 3  |
        {libere_1}
          Y1=3 : !FI1, Y1:= 0
      od ;

   task P2 ;
      input AU2 ;
      output DE2,FI2 ;
      declare
         Y2 : 0..3 ;
      init
         Y2 := 0 ;
```

```
      do
        {local_p2}
          Y2=0 : Y2:= 1 |
        {dem_res2}
          Y2=1 : !DE2, Y2:= 2  |
        {aut_acces1}
          Y2=2 : ?AU2, Y2:= 3  |
        {libere_2}
          Y2=3 : !FI2, Y2:= 0
      od ;

   task CONTROLER ;
      input DEM1, DEM2, FIN1, FIN2 ;
        (* demandes et fins d'accès *)
      output AUT1, AUT2 ;
        (* autorisations d'accès *)
      declare
         C1 : 0..1 ;
           (* C1=0 = demande d'accès processus 1 *)
         C2 : 0..1 ;
           (* C2=0 = demande d'accès processus 2 *)
         X1 : 0..3 ; (* compteur ordinal interne *)
         X2 : 0..3 ; (* compteur ordinal interne *)
         TURN : 0..1 ; (* priorité alternée *)
      init
         C1:=1, C2:=1 ,TURN:=1, X1:=0, X2:=0 ;
      do
        {dem1}
          X1=0 : ?DEM1, C1:=0, X1:=1  |
        {acces1}
          (X1=1)and(C2=1) : !AUT1, X1:=2  |
        {lib1}
          X1=2 : ?FIN1, TURN:=0, C1:=1, X1:=0  |
        {a1}
          (X1=1)and(C2=0)and(TURN=0) : C1:=1, X1:=3  |
        {a2}
          (X1=3)and(TURN=1) : C1:=0, X1:=1  |

        {dem2}
          X2=0 : ?DEM2, C2:=0, X2:=1  |
        {acces2}
          (X2=1)and(C1=1) : !AUT2, X2:=2  |
        {lib2}
          X2=2 : ?FIN2, TURN:=1, C2:=1, X2:=0  |
        {b1}
          (X2=1)and(C1=0)and(TURN=1) : C2:=1, X2:=3  |
        {b2}
          (X2=3)and(TURN=0) : C2=0, X2:=1  |
      od ;

   port A1, A2, D1, D2, F1,F2 ;
```

```
body
   P1 (A1, D1, F1) //
   P2 (A2, D2, F2) //
   CONTROLER (D1,D2,F1,F2,A1,A2) .
```

2.10.2. PROTOCOLE D'ALLOCATION DE BUS

Considérons un système formé de n processus utilisant un bus série en exclusion mutuelle. Le système est organisé en anneau virtuel, chaque processus connaissant l'identité de son prédécesseur et de son successeur et communiquant avec eux, le long duquel circule un jeton. Le processus qui possède le jeton est autorisé à utiliser le bus.

La description du système est simple si l'on considère que les différents processus fonctionnent toujours parfaitement. Il faut cependant considérer qu'un processus peut fonctionner temporairement de façon incorrecte, par exemple lorsqu'il est en possession du jeton, ce qui peut entrainer la perte du jeton. Nous allons décrire un protocole, présenté dans [LEL81], qui garantit que la perte du jeton ne peut être que temporaire. Nous n'envisageons pas les problèmes de reconfiguration de l'anneau (exclusion du processus fautif) et supposons que le mauvais fonctionnement d'un processus ne peut avoir pour conséquence que la perte du jeton.

Les hypothèses sur la configuration de l'anneau sont les suivantes :

- les identités des processus sont des entiers entre 1 et n, n étant le nombre de processus de l'anneau ;
- les processus peuvent distinguer les messages circulant sur l'anneau ;
- l'ordre des messages circulant sur l'anneau ne peut pas être modifié, (les messages reçus par un processus sont réémis dans le même ordre) ;
- chaque processus dispose d'une horloge locale lui permettant d'armer des "time-out".

Soit TC le jeton circulant sur l'anneau. Le principe de base du protocole est que chaque processus Pi qui n'a pas reçu TC avant l'échéance d'un certain délai (déclenchement de time-out), génère un jeton TCi portant l'identité du processus et qui est candidat pour devenir le nouveau TC, et qui va lui aussi circuler le long de l'anneau. Si le processus Pi reçoit TC avant que le candidat TCi n'ait fait le tour complet de l'anneau, Pi doit éliminer TCi. Sinon le processus Pj qui va générer le nouveau jeton est celui tel que j = min (S(i)), où S(i) est l'ensemble des identités des jetons candidats que Pi a vu passer pendant la rotation complète du jeton candidat TCi.

On peut représenter chaque processus Pi par trois états :

- Ai : état d'attente, le time-out étant déclenché ;
- Bi : le processus Pi est prêt à générer un nouveau TC. Lors de l'émission du jeton candidat, le time-out est armé ;
- Ci : Pi n'est pas responsable de la régénération de TC. Le time-out est armé ;

et pour chaque processus Pi l'ensemble des actions qui peuvent modifier ces états par cinq transitions :

- t0 : déclenchement du time-out ;
- t1 : réception de TC ;
- t2 : réception d'un jeton candidat dont l'identité j est telle que j<i ;
- t3 : réception d'un jeton candidat dont l'identité j est telle que j>i ;
- t4 : réception d'un jeton candidat dont l'identité est i.

Le tableau d'états pour un processus Pi est alors le suivant :

	t0	t1	t2	t3	t4
Ai	Bi	Ai	Ai	Ai	Ai
Bi	Bi	Ai	Ci	Bi	Ai**
Ci	Bi	Ai	Ci	Ci	Ai

(Ai** : passage à l'état Ai, après génération d'un nouveau jeton).

Nous présentons la description du système pour un anneau de trois processus. Chaque processus Pi est représenté par une tâche Pi dont les paramètres échangés sont : TCINi et TCOUTi pour le jeton reçu et émis, TCANDINi et TCANDOUTi pour les jetons candidats reçus et émis. Les valeurs transmises par les paramètres TCINi et TCOUTi sont non significatives, tandis que celles de TCANDINi et TCANDOUTi contiennent l'identité du jeton candidat et sont donc comprises entre 1 et 3. Les échanges entre deux processus Pi et Pj sont effectués directement entre les tâches Pi et Pj, pour ne pas alourdir la description par des lignes de transmission.

```
cotask ALLOC_RESSOURCE ;

  task P1 ;
    input TCIN1,TCANDIN1 ;
```

```
  output TCOUT1,TCANDOUT1 ;
declare
  C1 : 0..2 ; (* états A,B,C *)
  C11: 0..2 ; (* états intermédiaires utilisés pour
                mémoriser les messages, de façon à
                pouvoir les réémettre ensuite *)
  A1 : 1..3 ; (* valeur d'un jeton candidat reçu *)
init
  C1:=0 ,C11:=0, A1:=1 ;
do

  (* ETAT A1 : C1=0 *)

  {rec_jeton_a_1}
  C1+C11=0 : ?TCIN1, C11:=1 |
      (* réception du jeton,utilisation du bus *)
  (C1=0)and(C11=1) : !TCOUT1, C11:=0 |
      (* réémission du jeton *)

  C1+C11=0 : A1:=?TCANDIN1, C11:=2 |
      (* réception d'un jeton candidat *)
  (C1=0)and(C11=2) : !TCANDOUT1:=A1, C11:=0 |
      (* réémission du jeton candidat *)

  C1+C11=0 : !TCANDOUT1:=1, C1:=1 |
      (* time-out *)

  (* ETAT B1 : C1=1 *)

  {rec_jeton_b_1}
  (C1=1)and(C11=0) : ?TCIN1, C11:=1 |
      (* réception du jeton,utilisation du bus *)
  (C1=1)and(C11=1) : !TCOUT1, C1:=0, C11:=0 |
      (* réémission du jeton *)

  (C1=1)and(C11=0) : A1:=?TCANDIN1, C11:=2 |
      (* réception d'un jeton candidat *)
  (C1=1)and(C11=2)and(A1<1) : !TCANDOUT1:=A1, C1:=2, C11:=0 |
  (C1=1)and(C11=2)and(A1>1) : !TCANDOUT1:=A1, C11:=0 |
  (C1=1)and(C11=2)and(A1=1) : !TCOUT1, C1:=0, C11:=0 |
      (* distinctions suivant la valeur
         reçue par la variable A1 *)

  (C1=1)and(C11=0) : !TCANDOUT1:=1 |
      (* time-out *)

  (* ETAT C1 : C1=2 *)

  {rec_jeton_c_1}
  (C1=2)and(C11=0) : ?TCIN1, C11:=1 |
      (* réception du jeton,utilisation du bus *)
```

```
   (C1=2)and(C11=1) : !TCOUT1, C1:=0, C11:=0 |
       (* réémission du jeton *)

   (C1=2)and(C11=0) : A1:=?TCANDIN1, C11:=2 |
       (* réception d'un jeton candidat *)
   (C1=2)and(C11=2)and(A1<>1) : !TCANDOUT1:=A1, C11:=0 |
   (C1=2)and(C11=2)and(A1=1) : C1:=0, C11:=0 |
       (* distinctions suivant la valeur
          reçue par la variable A1 *)

   (C1=2)and(C11=0) : !TCANDOUT1:=1, C1:=1
       (* time out *)
 od ;

 task P2 ;
   input TCIN2,TCANDIN2 ;
   output TCOUT2,TCANDOUT2 ;
 declare
   C2 : 0..2 ;
   C21: 0..2 ;
   A2 : 1..3 ;
 init
   C2:=0 ,C21:=0, A2:=1 ;
 do
   {rec_jeton_a_2}
   C2+C21=0 : ?TCIN2, C21:=1 |
   (C2=0)and(C21=1) : !TCOUT2, C21:=0 |
   C2+C21=0 : A2:=?TCANDIN2, C21:=2 |
   (C2=0)and(C21=2) : !TCANDOUT2:=A2, C21:=0 |
   C2+C21=0 : !TCANDOUT2:=2, C2:=1 |

   {rec_jeton_b_2}
   (C2=1)and(C21=0) : ?TCIN2, C21:=1 |
   (C2=1)and(C21=1) : !TCOUT2, C2:=0, C21:=0 |
   (C2=1)and(C21=0) : A2:=?TCANDIN2, C21:=2 |
   (C2=1)and(C21=2)and(A2<2) : !TCANDOUT2:=A2, C2:=2, C21:=0 |
   (C2=1)and(C21=2)and(A2>2) : !TCANDOUT2:=A2, C21:=0 |
   (C2=1)and(C21=2)and(A2=2) : !TCOUT2, C2:=0, C21:=0 |
   (C2=1)and(C21=0) : !TCANDOUT2:=2 |

   {rec_jeton_c_2}
   (C2=2)and(C21=0) : ?TCIN2, C21:=1 |
   (C2=2)and(C21=1) : !TCOUT2, C2:=0, C21:=0 |
   (C2=2)and(C21=0) : A2:=?TCANDIN2, C21:=2 |
   (C2=2)and(C21=2)and(A2<>2) : !TCANDOUT2:=A2, C21:=0 |
   (C2=2)and(C21=2)and(A2=2) : C2:=0, C21:=0 |
   (C2=2)and(C21=0) : !TCANDOUT2:=2, C2:=1
 od ;

 task P3 ;
   input TCIN3,TCANDIN3 ;
```

```
      output TCOUT3,TCANDOUT3 ;
    declare
      C3 : 0..2 ;
      C31: 0..2 ;
      A3 : 1..3 ;
    init
      C3:=0 ,C31:=0, A3:=1 ;
    do
      {rec_jeton_a_3}
      C3+C31=0 : ?TCIN3, C31:=1 |
      (C3=0)and(C31=1) : !TCOUT3, C31:=0 |
      C3+C31=0 : A3:=?TCANDIN3, C31:=2 |
      (C3=0)and(C31=2) : !TCANDOUT3:=A3, C31:=0 |
      C3+C31=0 : !TCANDOUT3:=3, C3:=1 |

      {rec_jeton_b_3}
      (C3=1)and(C31=0) : ?TCIN3, C31:=1 |
      (C3=1)and(C31=1) : !TCOUT3, C3:=0, C31:=0 |
      (C3=1)and(C31=0) : A3:=?TCANDIN3, C31:=2 |
      (C3=1)and(C31=2)and(A3<3) : !TCANDOUT3:=A3, C3:=2, C31:=0 |
      (C3=1)and(C31=2)and(A3>3) : !TCANDOUT3:=A3, C31:=0 |
      (C3=1)and(C31=2)and(A3=3) : !TCOUT3, C3:=0, C31:=0 |
      (C3=1)and(C31=0) : !TCANDOUT3:=3 |

      {rec_jeton_c_3}
      (C3=2)and(C31=0) : ?TCIN3, C31:=1 |
      (C3=2)and(C31=1) : !TCOUT3, C3:=0, C31:=0 |
      (C3=2)and(C31=0) : A3:=?TCANDIN3, C31:=2 |
      (C3=2)and(C31=2)and(A3<>3) : !TCANDOUT3:=A3, C31:=0 |
      (C3=2)and(C31=2)and(A3=3) : C3:=0, C31:=0 |
      (C3=2)and(C31=0) : !TCANDOUT3:=3, C3:=1
    od ;

port JET12, JET23, JET31, CAND12, CAND23, CAND31 ;

body
   P1 (JET31,CAND31,JET12,CAND12) //
   P2 (JET12,CAND12,JET23,CAND23) //
   P3 (JET23,CAND23,JET31,CAND31) .
```

Les quatre exemples qui suivent sont extraits de [COSY83]. A partir d'une brève description d'un problème, nous constituons une modélisation des systèmes proposés. La description des systèmes est parfois incomplètement spécifiée ; nous sommes amenés alors à introduire certaines hypothèses supplémentaires. Par ailleurs, les solutions que nous donnons ne répondent pas toujours exactement aux exigences fixées par les énoncés : les problèmes posés permettent ainsi de juger les possibilités et les limites du langage de description de QUASAR [QSS83].

2.10.3. SYSTEME DE TRANSMISSION AVEC DECONNECTION

Dans ce troisième exemple, nous nous proposons de décrire un système constitué de deux sites ´a´ et ´b´, reliés par une ligne de transmission. La ligne peut échanger des messages simultanément dans les deux sens, jusqu´à ce qu´elle reçoive un message de déconnection venant d´un site, après quoi elle ne communique plus avec ce site. Elle poursuit la communication avec l´autre site, jusqu´à la réception d´un message de déconnection. L´ordre des messages dans une direction donnée doit être préservé par la ligne.

Nous représentons ce système par un ensemble de trois tâches, ENDA et ENDB pour les sites ´a´ et ´b´, et LINE pour la ligne de transmission. Les tâches ENDA et ENDB communiquent avec la tâche LINE par échanges de messages, effectués par rendez-vous.

Le contenu des messages échangés ne nous intéressant pas, les rendez-vous entre tâches relèvent de la synchronisation pure, sans communication de valeurs. Pour conserver l´ordre des messages échangés, nous incorporons dans la tâche LINE deux "buffers" (un par sens de transmission) bornés circulaires pour stocker les messages reçus par la ligne de transmission.

Les échanges entre la tâche ENDA et la ligne LINE s´effectuent par l´intermédiaire de trois variables échangées, la première (MAL) pour les messages envoyés par le site ´a´ à la ligne, la deuxième (MLA) pour les messages allant de la ligne vers le site ´a´, et la troisième (DISA) pour l´envoi d´un ordre de déconnection du site ´a´. De même, les communications entre la ligne et le site ´b´ sont effectuées à l´aide de trois variables MBL, MLB et DISB.

Pour décrire le site ´a´, nous exprimons ses possibilités de communication avec la ligne de transmission. Nous ne nous intéressons pas au fonctionnement interne du site, nous considérons uniquement les actions d´échanges de messages.
Le corps de la tâche ENDA est composé de quatre commandes gardées ; deux d´entre elles pour l´émission d´un message (message ou ordre de déconnection) par le site ´a´, une troisième pour la réception d´un message venant de la ligne, et une quatrième pour schématiser les opérations internes du site ´a´. Nous opérons de même pour le site ´b´.

La description de la tâche LINE est obtenue par énumération de ses communications avec les deux sites. En considérant le site ´a´, la ligne peut effectuer trois actions :

- si le buffer des messages de ´a´ vers ´b´ n´est pas saturé, elle peut recevoir un message venant du site ´a´ ; elle stocke alors le message dans le buffer ;

- elle peut recevoir un ordre de déconnection du site ´a´ ; dans ce cas, elle interrompt définitivement la communication avec ce site ;

- si le buffer des messages de ´b´ vers ´a´ est non vide, elle peut envoyer un message au site ´a´ ;
(ces trois opérations étant conditionnées par le fait que le site ´a´ ne soit pas déconnecté, c.à.d. qu´il n´ait pas déjà envoyé un message de déconnection).

Pour ne pas encombrer la description, nous ne représentons pas les deux buffers, mais simplement les maniements d´indices qui permettent de préserver l´ordre des messages transmis. La taille des buffers étant non significative, nous la prendrons égale à 2.

Nous obtenons alors la description suivante du système proposé :

```
cotask TRANSMISSION ;

  task ENDA ;
    input mb ;
    output ma,da ;
  do
    (* envoi d´un message *)
    {sendal} true : !ma |

    (* envoi d´un ordre de déconnection *)
    {senddisa} true : !da |

    (* réception d´un message *)
    {recla}   true : ?mb |

    (* traitement local *)
    {locala}  true :
  od;

  task ENDB ;
    input mma ;
    output mmb,db ;
  do
    (* envoi d´un message *)
    {sendbl} true : !mmb |

    (* envoi d´un ordre de déconnection *)
    {senddisb} true : !db |

    (* réception d´un message *)
    {reclb}   true : ?mma |

    (* traitement local *)
    {localb}  true :
```

```
od;

task LINE ;
  input ia,ib,dia,dib ;
  output oa,ob ;
declare
  inxab   : 1..2 ;
  outxab  : 1..2 ;
  countab : 0..2 ;
  (* pointeurs d'entrée et de sortie, nombre d'éléments
     du buffer des messages de 'a' vers 'b' *)
  inxba   : 1..2 ;
  outxba  : 1..2 ;
  countba : 0..2 ;
  (* pointeurs d'entrée et de sortie, nombre d'éléments
     du buffer des messages de 'b' vers 'a' *)
  disca : 0..1 ;
  discb : 0..1 ;
  (* valeurs booléennes indiquant si les
     sites sont déconnectés ou non *)
init
  countab := 0, countba := 0,
  outxab := 1, outxba := 1,
  inxab := 1, inxba := 1,
  disca := 0, discb := 0 ;
do
  (* communication avec 'a' *)

  (* réception d'un message de 'a' *)
  {recal}
  (disca=0) and (countab<2) : ?ia,
                              inxab := (inxab mod 2) +1,
                              countab := countab+1 |
  (* message de déconnection *)
  {discona}
  disca=0 : ?dia, disca := 1 |
  (* émission d'un message vers 'a' *)
  {sendla}
  (disca=0) and (countba>0) : !oa,
                              outxba := (outxba mod 2) +1,
                              countba := countba-1 |

  (* communication avec 'b' *)

  (* réception d'un message de 'b' *)
  {recbl}
  (discb=0) and (countba<2) : ?ib,
                              inxba := (inxba mod 2) +1,
                              countba := countba+1 |
  (* message de déconnection *)
  {disconb}
```

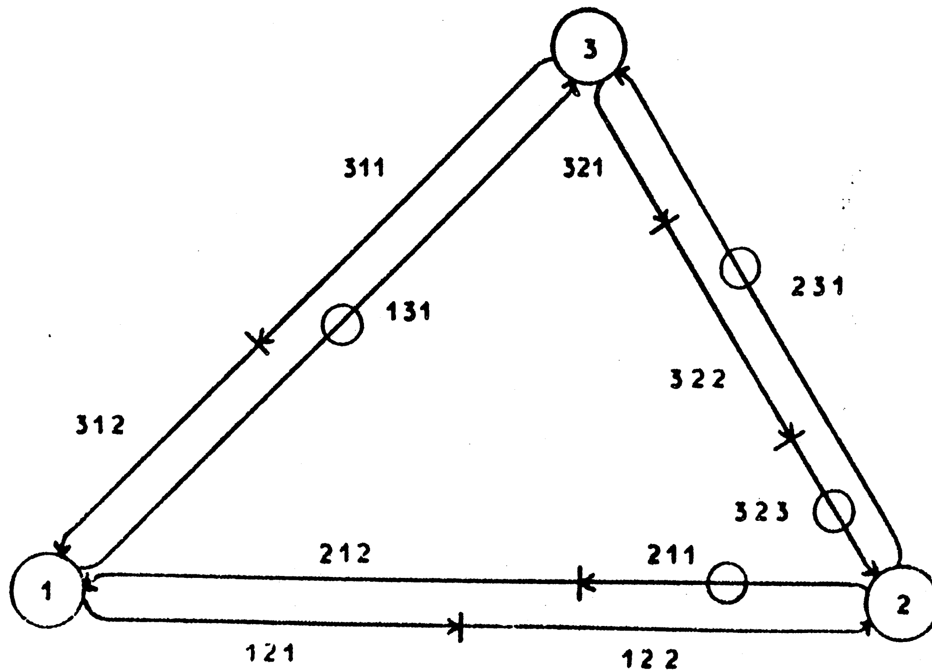
```
    discb=0 : ?dib, discb := 1 |
    (* émission d'un message vers 'b' *)
    {sendlb}
    (discb=0) and (countab>0) : !ob,
                                outxab := (outxab mod 2) +1,
                                countab := countab-1
  od;

port mal, mbl, mla, mlb, disa, disb ;

body
  ENDA (mla,mal,disa) //
  ENDB (mlb,mbl,disb) //
  LINE (mal,mbl,disa,disb,mla,mlb).
```

2.10.4. RESEAU DE TRAFIC FERROVIAIRE

Nous décrivons dans cet exemple un système de trafic ferroviaire, constitué d'un ensemble de noeuds (qui peuvent représenter des gares ou des aiguillages) et de voies ferrées en sens unique reliant chacune deux noeuds. Chaque voie est formée d'un ensemble non vide de secteurs. Un train peut changer de voie uniquement à un noeud, à condition que le secteur sur lequel il veut prendre place ne soit pas déjà occupé par un autre train. Le système est tel que chaque secteur peut contenir au plus un train à un instant donné.

Nous proposons une solution indépendante de la configuration du réseau ferroviaire considéré. Nous prenons pour exemple un réseau composé de trois noeuds reliés deux à deux par deux voies (une dans chaque sens), chaque voie comportant 1, 2 ou 3 secteurs (figure 2.2). La description d'un système plus complexe ne présente aucun problème, car la définition des objets qui le composent est indépendante aussi bien du nombre de trains que du nombre total de voies et de secteurs par voie.

figure 2.2



Nous décrivons le système par six tâches TRACKij, chaque tâche TRACKij représentant la voie qui mène du noeud i au noeud j. Le passage d'un train d'une voie à une autre se traduit par l'échange d'une variable (émission par la voie de sortie, réception par la voie d'entrée) entre les tâches qui correspondent aux deux voies, et n'est possible que si le premier secteur de la voie d'entrée est disponible.

Chaque voie TRACKij est elle-même composée d'un ensemble non vide de tâches SECTIONijk, chaque tâche SECTIONijk représentant le k-ième secteur de la voie menant du noeud i au noeud j.

Le passage d'un train d'un secteur à un autre s'effectue par l'échange d'une variable entre les tâches représentant les deux secteurs. Si les secteurs appartiennent à la même voie (deux secteurs consécutifs) ij, l'échange de la variable est interne à la tâche TRACKij. Si le secteur quitté par le train est le dernier (SECTIONijk) d'une voie ij, alors la variable émise par la tâche SECTIONijk est propagée par la tâche TRACKij, qui la transmet à la tâche TRACKjl représentant la voie sur laquelle le train veut se rendre. La tâche TRACKjl transmettra alors la variable au secteur SECTIONjl1.

Nous décrivons ainsi le système de trafic ferroviaire par l'intermédiaire d'une structure hiérarchisée, formée d'un ensemble

de voies communiquant entre elles, chaque voie étant elle-même composée d'un ensemble de secteurs consécutifs qui communiquent chacun avec leur successeur.

La description d'une tâche SECTIONijk est élémentaire. Elle comporte deux variables échangées, l'une d'entrée qui reçoit les demandes d'occupation du secteur, l'autre de sortie qui envoie une demande du même type à un autre secteur. L'occupation d'un secteur est représentée par une variable booléenne (0 pour libre, 1 pour occupé) qui peut être modifiée en deux occasions : une demande arrive lorsque le secteur est libre, ou bien un train qui se situe sur le secteur quitte ce secteur.

Une tâche SECTIONijk sera donc décrite par la donnée suivante :

```
task SECTIONijk ;

  input INijk ;
  output OUTijk ;

declare
  Bijk : 0..1 ;
  (* variable d'occupation du secteur *)

init
  Bijk := (* 0,1 suivant l'initialisation du secteur *)

do
  (* arrivée d'un train *)
  {TOijk}   (Bijk=0) : ?INijk, Bijk := 1 |

  (* départ d'un train  *)
  {FROMijk} (Bijk=1) : !OUTijk, Bijk := 0
od ;
```

Chaque voie est composée de la mise en parallèle des tâches représentant les secteurs qui la constituent, connectées par des variables échangées internes à la tâche composée :

```
cotask TRACKij ;

  input INij ;
  output OUTij ;

task SECTIONij1 ;
  ---
  ---
task SECTIONijk ;

port Rij12, Rij23, ..., Rijk-1k,INij,OUTij ;
```

```
body
  SECTIONij1 (INij, Rij12) //
  SECTIONij2 (Rij12,Rij23) //
    ---
  SECTIONijk (Rijk-1k,OUTij) ;
```

La description du système est finalement la mise en parallèle des tâches représentant les six voies de notre réseau ferroviaire. Nous ne décrivons pas la direction qu'emprunte un train quittant une voie. Un tel train est susceptible d'aller sur le premier secteur libre d'une voie quelconque connectée au noeud qu'il vient d'atteindre. Ainsi, une variable émise par une voie TRACKij peut être reçue par n'importe quelle voie TRACKjl, la voie TRACKjl pouvant elle-même recevoir des variables provenant de n'importe quelle tâche TRACKij.

```
cotask RAILWAY ;

  cotask TRACK12 ;

    input IN12 ;
    output OUT12 ;

    task SECTION121 ;
      input IN121 ;
      output OUT121 ;
    declare
      B121 : 0..1 ;
    init
      B121 := 0 ;
    do
      {TO121}   (B121=0) : ?IN121, B121 := 1 |
      {FROM121} (B121=1) : !OUT121, B121 := 0
    od ;

    task SECTION122 ;
      input IN122 ;
      output OUT122 ;
    declare
      B122 : 0..1 ;
    init
      B122 := 0 ;
    do
      {TO122}   (B122=0) : ?IN122, B122 := 1 |
      {FROM122} (B122=1) : !OUT122, B122 := 0
    od ;

    port R1212,IN12,OUT12 ;

    body
      SECTION121(IN12,R1212) //
```

```
      SECTION122(R1212,OUT12) ;

  cotask TRACK23 ;

    input IN23 ;
    output OUT23 ;

    task SECTION231 ;
      input IN231 ;
      output OUT231 ;
    declare
      B231 : 0..1 ;
    init
      B231 := 1 ;
    do
      {TO231}   (B231=0) : ?IN231, B231 := 1 |
      {FROM231} (B231=1) : !OUT231, B231 := 0
    od ;

    port IN23,OUT23 ;

    body
      SECTION231(IN23,OUT23) ;

  cotask TRACK31 ;

    input IN31 ;
    output OUT31 ;

    task SECTION311 ;
      input IN311 ;
      output OUT311 ;
    declare
      B311 : 0..1 ;
    init
      B311 := 0 ;
    do
      {TO311}   (B311=0) : ?IN311, B311 := 1 |
      {FROM311} (B311=1) : !OUT311, B311 := 0
    od ;

    task SECTION312 ;
      input IN312 ;
      output OUT312 ;
    declare
      B312 : 0..1 ;
    init
      B312 := 0 ;
    do
      {TO312}   (B312=0) : ?IN312, B312 := 1 |
      {FROM312} (B312=1) : !OUT312, B312 := 0
```

```
  od ;

  port R3112,IN31,OUT31 ;

  body
    SECTION311(IN31,R3112) //
    SECTION312(R3112,OUT31) ;

cotask TRACK21 ;

  input IN21 ;
  output OUT21 ;

  task SECTION211 ;
    input IN211 ;
    output OUT211 ;
  declare
    B211 : 0..1 ;
  init
    B211 := 1 ;
  do
    {TO211}   (B211=0) : ?IN211, B211 := 1 |
    {FROM211} (B211=1) : !OUT211, B211 := 0
  od ;

  task SECTION212 ;
    input IN212 ;
    output OUT212 ;
  declare
    B212 : 0..1 ;
  init
    B212 := 0 ;
  do
    {TO212}   (B212=0) : ?IN212, B212 := 1 |
    {FROM212} (B212=1) : !OUT212, B212 := 0
  od ;

  port R2112,IN21,OUT21 ;

  body
    SECTION211(IN21,R2112) //
    SECTION212(R2112,OUT21) ;

cotask TRACK13 ;

  input IN13 ;
  output OUT13 ;

  task SECTION131 ;
    input IN131 ;
    output OUT131 ;
```

```
    declare
      B131 : 0..1 ;
    init
      B131 := 1 ;
    do
      {TO131}   (B131=0) : ?IN131, B131 := 1 |
      {FROM131} (B131=1) : !OUT131, B131 := 0
    od ;

    port IN13,OUT13 ;

    body
      SECTION131(IN13,OUT13) ;

  cotask TRACK32 ;

    input IN32 ;
    output OUT32 ;

    task SECTION321 ;
      input IN321 ;
      output OUT321 ;
    declare
      B321 : 0..1 ;
    init
      B321 := 0 ;
    do
      {TO321}   (B321=0) : ?IN321, B321 := 1 |
      {FROM321} (B321=1) : !OUT321, B321 := 0
    od ;

    task SECTION322 ;
      input IN322 ;
      output OUT322 ;
    declare
      B322 : 0..1 ;
    init
      B322 := 0 ;
    do
      {TO322}   (B322=0) : ?IN322, B322 := 1 |
      {FROM322} (B322=1) : !OUT322, B322 := 0
    od ;

    task SECTION323 ;
      input IN323 ;
      output OUT323 ;
    declare
      B323 : 0..1 ;
    init
      B323 := 1 ;
    do
```

```
    {TO323}   (B323=0) : ?IN323, B323 := 1 |
    {FROM323} (B323=1) : !OUT323, B323 := 0
  od ;

  port R3212,R3223,IN32,OUT32 ;

  body
    SECTION321(IN32,R3212)  //
    SECTION322(R3212,R3223) //
    SECTION323(R3223,OUT32) ;

port NOEUD1, NOEUD2, NOEUD3 ;

body
  TRACK12(NOEUD1,NOEUD2) //
  TRACK23(NOEUD2,NOEUD3) //
  TRACK31(NOEUD3,NOEUD1) //
  TRACK21(NOEUD2,NOEUD1) //
  TRACK13(NOEUD1,NOEUD3) //
  TRACK32(NOEUD3,NOEUD2).
```

2.10.5. TRANSMISSION SYNCHRONE ET ASYNCHRONE

Dans ce cinquième exemple, nous décrivons un objet comportant deux entrées et une sortie. La sortie et une des entrées envoient et reçoivent respectivement des paquets d'informations à des intervalles réguliers. La deuxième entrée est asynchrone, c.à.d. reçoit des données à des moments indéterminés. Les paquets qui arrivent à l'entrée synchrone peuvent être vides ou non, et l'objet envoie des données uniquement en transmettant celles de l'entrée synchrone, ou en remplissant les paquets d'informations vides par des paquets de l'entrée asynchrone. Tous les paquets doivent avoir la même taille.

Nous représentons le système ci-dessus par deux tâches. L'une, de nom OBJECT, décrit l'objet de transmission de paquets. L'autre, de nom SENDREC, servira à la fois d'émetteur et de récepteur des paquets d'informations.

Le contenu des paquets d'informations étant non significatif, nous supposerons que ces paquets sont formés uniquement d'une valeur booléenne indiquant si le paquet est vide ou non.

Bien qu'en pratique il soit peu vraisemblable que les trois portes de l'objet soient connectées au même instrument, nous pouvons cependant regrouper les émetteurs et le récepteur des paquets au sein d'une même tâche dont la seule fonction sera d'envoyer des paquets aux deux entrées de l'objet, et de recevoir les paquets émis par ce même objet.

Nous supposons par ailleurs que le fonctionnement de l'objet consiste à propager immédiatement les paquets de l'entrée synchrone vers la sortie synchrone, en substituant un paquet vide par le contenu du dernier paquet reçu à l'entrée asynchrone. Les paquets de l'entrée asynchrone ne seront pas mémorisés, l'arrivée d'un paquet à cette entrée écrasant le paquet précédent.

Pour représenter le caractère synchrone de deux des portes de l'objet, nous donnons à la tâche OBJECT un fonctionnement cyclique comportant deux états : le premier valide la réception par l'objet d'un paquet à l'entrée synchrone, le deuxième valide l'émission d'un paquet par la sortie synchrone. L'objet peut à tout moment recevoir un paquet à l'entrée asynchrone.

La description du système ainsi définie se traduit par le programme suivant :

```
cotask SYN_ASYNCHRON ;

  task SENDREC ;

    input IN ;
    output SOUT, AOUT ;

  do
    (* réception d'un paquet émis par l'objet *)
    true : ?IN |

    (* émission d'un paquet vers l'entrée syncrone *)
    true : !SOUT := 0 |
    true : !SOUT := 1 |

    (* émission d'un paquet vers l'entrée asyncrone *)
    true : !AOUT := 0 |
    true : !AOUT := 1
  od ;

  task OBJECT ;

    input INS, INA ;
    output OUT ;

  declare
    PACKA : 0..1 ; (* paquet de l'entrée asyncrone *)
    PACKS : 0..1 ; (* paquet de l'entrée syncrone *)
    CLOCK : 0..1 ; (* horloge *)
  init
    PACKA := 0, CLOCK := 0 ;

  do
    (* réception à l'entrée asynchrone *)
```

```
    {recasyn} true : PACKA :=?INA |

    (* réception à l'entrée synchrone *)
    {recsyn} CLOCK=0 : PACKS := ?INS, CLOCK := 1 |

    (* transmission du paquet non vide *)
    {sendpacks}
    (CLOCK=1) and (PACKS<>0) : !OUT := PACKS, CLOCK := 0 |

    (* substitution d'un paquet vide *)
    {sendpacka}
    (CLOCK=1) and (PACKS=0)  : !OUT := PACKA,
                               PACKA := 0, CLOCK := 0
  od ;

port SYNINPUT, SYNOUTPUT, ASYNINPUT ;

body
  SENDREC( SYNOUTPUT, SYNINPUT, ASYNINPUT ) //
  OBJECT ( SYNINPUT, ASYNINPUT, SYNOUTPUT ) .
```

2.10.6. RESEAU DE DISTRIBUTEURS DE BILLETS

Nous décrivons dans ce dernier exemple un réseau de distributeurs de billets de banque. Le système est composé d'un ensemble de distributeurs qui peuvent accéder à une base de données centrale, contenant les informations relatives aux comptes bancaires. L'utilisation d'un distributeur est classique : l'usager introduit une carte magnétique et tape un nombre qui est comparé avec un code figurant sur la carte. Si le nombre tapé n'est pas le bon, la carte est conservée par le distributeur, sinon l'usager est en mesure d'effectuer les trois opérations suivantes :

1) obtenir le montant disponible sur son compte ;

2) effectuer un retrait ;

3) demander l'envoi postal d'un extrait de compte.

Les informations relatives aux comptes sont détenues par une base de données qui peut être inaccessible (occupée ou hors d'usage). Ainsi l'opération 1) n'est pas toujours possible. Si la base est accessible, toute somme inférieure ou égale au montant du compte peut être retirée ; sinon l'utilisateur peut retirer jusqu'à 100 francs par jour (la somme retirée chaque jour doit donc être mémorisée sur la carte).

La ligne de transmission entre un distributeur et la base ne peut

envoyer des informations que dans un seul sens à un instant donné. Les transactions effectuées à un distributeur sont enregistrées localement, et envoyées à la base de données. Les transactions qui se produisent durant une transmission sont traitées comme si la base était inaccessible, et sont ajoutées à la liste des transactions locales.

Pour décrire le système, nous représentons la base de données centrale par une tâche de nom DATABASE, et chaque distributeur par une tâche DISTRIBUTEURi. Les lignes de transmission entre les distributeurs et la base ne sont pas représentées, les échanges s'effectuant directement entre les tâches.

Chaque tâche DISTRIBUTEURi est une tâche composée reflétant l'exécution parallèle de trois processus qui composent le distributeur :

- une tâche GUICHETi correspondant à l'interface entre l'utilisateur et le distributeur de billets (entrée et sortie d'une carte magnétique, clavier sur lequel sont tapées les commandes) ;
- une tâche APPAREILi qui décrit le fonctionnement du distributeur. Cette tâche reçoit des commandes de la tâche GUICHETi, communique avec la base centrale et exécute si possible les commandes, et envoie des comptes rendus à la tâche GUICHETi ;
- une tâche MEMORYi qui stocke les transactions locales, et les envoie dès que possible à la base de données.

La base est représentée par une tâche élémentaire qui communique avec les tâches DISTRIBUTEURi. Elle peut être inaccessible (et ne peut plus alors communiquer) ou en état de marche, dans ce cas elle répond aux différentes demandes provenant des distributeurs.

Nous décrivons le système pour un réseau composé d'une base et de deux distributeurs. Nous présentons en détail les différentes tâches, et précisons les hypothèses que nous avons dû faire sur le fonctionnement du système pour simplifier la description. Signalons que notre solution peut s'appliquer à un système comportant un nombre quelconque de distributeurs sans grandes modifications.

Une tâche GUICHETi représente l'interface entre l'usager et la machine. Son corps est composé de l'émission vers la tâche APPAREILi des actions effectuées par l'usager : introduction et retrait d'une carte, commandes 1, 2 ou 3 et leurs comptes rendus. La description de la tâche GUICHETi est la suivante :

```
task GUICHETi ;

  input APAWRONGCARDi, APAGOODCARDi,
        APATOTALi, APACOM1IMPOi ;
        (* code invalide, valide *)
        (* compte rendu de la commande 1 *)
        (* commande 1 impossible *)
```

```
  output APACARDi,APASTOPi,APADEMi1,APADEMi2,APADEMi3 ;
        (* carte in et out, commandes 1, 2 et 3 *)
  declare
    PRESENTi : 0..3 ;
        (* compteur ordinal de la tâche *)
  init
    PRESENTi := 0 ;
do
  (* entrée d'une carte *)
  {cardini} PRESENTi=0 : !APACARD, PRESENTi := 1 |
  (* code valide *)
  PRESENTi=1 : ?APAGOODCARDi, PRESENTi := 2 |
  (* code invalide *)
  PRESENTi=1 : ?APAWRONGCARDi, PRESENTi := 0 |
  (* commande 1 *)
  {commandei1} PRESENTi=2 : !APADEMi1, PRESENTi := 3 |
  (* commande 1 impossible *)
  PRESENTi=3 : ?APACOM1IMPOi, PRESENTi := 2 |
  (* compte-rendu commande 1 *)
  PRESENTi=3 : ?APATOTALi, PRESENTi := 2 |
  (* commande 2 *)
  {commandei2} PRESENTi=2 : !APADEMi2 |
  (* commande 3 *)
  {commandei3} PRESENTi=2 : !APADEMi3 |
  (* retrait d'une carte *)
  {cardouti} PRESENTi=2 : !APASTOPi, PRESENTi := 0
od ;
```

La tâche APAREILi décrit le fonctionnement interne d'un distributeur de billets.

Sa communication avec la tâche GUICHETi consiste à récupérer les commandes tapées par l'usager (et l'entrée et la sortie d'une carte), et à lui envoyer des comptes rendus.

La communication de la tâche avec la base de données est plus complexe. Lorsque la tâche désire dialoguer avec la base, elle lui envoie un message : si le message n'est pas reçu, alors la base est inaccessible ; sinon la connection est établie et l'échange d'informations peut s'effectuer. La connection de la base à un distributeur n'est pas interruptible (la base ne peut pas tomber hors d'usage à ce moment), et rend la base inaccessible aux autres distributeurs jusqu'à la fin de la connection.

La tâche APAREILi effectue des transactions locales, lorsque la base est inaccessible. Ces transactions sont envoyées à la tâche MEMORYi, qui se charge de les stocker. Nous ne nous intéressons pas au montant qui figure sur une carte magnétique pour limiter les retraits journaliers à 100 francs. Nous décrivons simplement les deux actions possibles de retrait :

- si la base est disponible, un retrait quelconque peut être effectué jusqu'au montant du compte ;
- sinon un retrait est possible (inférieur à 100 francs), ou aucun retrait n'est possible (forfait journalier épuisé).

Les demandes d'extraits de compte sont considérées comme des transactions locales, qui sont mémorisées et envoyées à la base dès que celle-ci est accessible.

Une tâche APAREILi est décrite de la façon suivante :

```
task APPAREILi ;

  input GUICARDi, GUISTOPi, GUIDEMi1, GUIDEMi2,
        GUIDEMi3, DATATOTALi ;
        (* carte in et out, commandes 1, 2, 3,
           montant du compte de l'usager *)
  output DATAACCi, DATAWITHDi, MEMOLOCWITHDi,
         MEMOSTATi, GUITOTALi, GUIWRONGCARDi,
         GUIGOODCARDi, GUICOM1IMPOi ;
        (* commandes 1, retrait, retrait local,
        extrait de compte, montant du compte,
        code invalide, valide, commande 1 impossible *)
  declare
    Pi : 0..8 ;
        (* compteur ordinal interne *)
  init
    Pi := 0 ;

do
  (* carte in *)

  Pi=0 : ?GUICARDi, Pi := 1 |
  (* code i invalide *)
  Pi=1 : !GUIWRONGCARDi, Pi := 0 |
  (* code i valide *)
  Pi=1 : !GUIGOODCARDi, Pi := 2 |

  (* commande 1 *)

  Pi=2 : ?GUIDEMi1, Pi := 3 |
  (* base indisponible *)
  Pi=3 : !GUICOM1IMPOi, Pi := 2 |
  (* base disponible *)
  {sendacci} Pi=3 : !DATAACCi, Pi := 4 |
  (* réception du montant du compte *)
  Pi=4 : ?DATATOTALi, Pi:= 5 |
  (* envoi du montant à GUICHETi *)
  Pi=5 : !GUITOTALi, Pi := 2 |

  (* commande 2 *)

  Pi=2 : ?GUIDEMi2, Pi := 6 |
  (* base indisponible, forfait journalier épuisé *)
  Pi=6 : Pi := 2 |
  (* base indisponible, retrait local mémorisé *)
```

```
  Pi=6 : !MEMOLOCWITHDi, Pi := 2 |
  (* base disponible, retrait direct *)
  {rectotali} Pi=6 : ?DATATOTALi, Pi := 7 |
  Pi=7 : !DATAWITHDi, Pi := 2 |

  (* commande 3 *)

  Pi=2 : ?GUIDEMi3, Pi := 8 |
  (* mémorisation de la demande d'extrait de compte *)
  Pi=8 : !MEMOSTATi, Pi := 2 |

  (* retrait d'une carte *)

  Pi=2 : ?GUISTOPi, Pi := 0
od ;
```

La tâche MEMORYi est chargée de stocker les transactions locales, et de les envoyer le plus tôt possible à la base de données. Nous ne représentons pas la file d'attente des transactions, mais uniquement les communications de cette tâche avec les tâches APAREILi ou DATABASE.

```
task MEMORYi ;

  input APALOCWITHDi, APASTATi ;
      (* retrait local, extrait de compte *)
  output DATALOCWITHDi, DATASTATi ;
      (* retrait local, extrait de compte *)
do
  (* réception du retrait local *)
  true : ?APALOCWITHDi |
  (* envoi du retrait *)
  true : !DATALOCWITHDi |
  (* réception d'une demande d'extrait *)
  true : ?APASTATi |
  (* envoi de la demande *)
  true : !DATASTATi
od ;
```

L'exécution parallèle de ces trois tâches est décrite par l'intermédiaire de la tâche composée DISTRIBUTEURi, qui définit par ailleurs les communications entre les tâches.

```
cotask DISTRIBUTEURi ;

  input ITOTALi ;
    (* montant d'un compte venant de la base *)
  output OACCi, OWITHDi, OLOCWITHDi, OSTATi ;
    (* demande de montant, retrait direct avec la base,
       retrait local, demande d'extrait de compte  *)
```

```
  port wrongcardi,goodcardi,totali,comlimpoi,cardi,
       stopi,demi1,demi2,demi3,locwithdi,stati,
       itotali,oacci,owithdi,olocwithdi,ostati ;

  body
    GUICHETi (wrongcardi,goodcardi,totali,comlimpoi,
              cardi,stopi,demi1,demi2,demi3) //
    APAREILi (cardi,stopi,demi1,demi2,demi3,
              ITOTALi,OACCi,OWITHDi,locwithdi,
              stati,totali,wrongcardi,goodcardi,
              comlimpo) //
    MEMORYi (locwithdi,stati,OLOCWITHDi,OSTATi) ;
```

La base de données est représentée par la tâche DATABASE. La base peut être inaccessible à un distributeur, soit parce qu'elle est hors d'usage, soit parce qu'elle est en communication avec un autre distributeur. La base peut tomber hors d'usage à un instant quelconque, mais pas au milieu d'une communication avec un distributeur. Nous décrivons son fonctionnement de la façon suivante :

```
task DATABASE ;

  input IACC1, IWITHD1, ILOCWITHD1, ISTAT1,
        IACC2, IWITHD2, ILOCWITHD2, ISTAT2 ;
        (* informations provenant des tâches
           DISTRIBUTEUR1 et DISTRIBUTEUR2 :
        demande du montant, retrait direct,
        retrait local, demande d'extrait de compte *)
  output OTOTAL1, OTOTAL2 ;
        (* envoi des montants aux deux tâches *)
  declare
    KO : 0..1 ;
    (* variable de disponibilité de la base *)
    LIBRE : 0..2 ;
    (* compteur ordinal interne *)
  init
    KO := 0, LIBRE := 0 ;

do
  (* mise hors d'usage de la base *)
  (KO=0) and (LIBRE=0) : KO := 1 |
  (* mise en service de la base *)
  KO=1 : KO := 0 |

  (* communication de la base avec
     le distributeur numéro 1 *)

  (* demande de montant *)
  (KO=0) and (LIBRE=0) : ?IACC1, LIBRE := 1 |
  LIBRE=1 : !OTOTAL1, LIBRE := 0 |
  (* retrait direct *)
```

```
  (KO=0) and (LIBRE=0) : !OTOTAL1, LIBRE := 2 |
  LIBRE=2 : ?IWITHD11, LIBRE := 0 |
  (* retrait local *)
  (KO=0) and (LIBRE=0) : ?ILOCWITHD1 |
  (* demande d'extrait de compte *)
  (KO=0) and (LIBRE=0) : ?ISTAT1 |

  (* communication de la base avec
     le distributeur numéro 2 *)

  (KO=0) and (LIBRE=0) : ?IACC2, LIBRE := 1 |
  LIBRE=1 : !OTOTAL2, LIBRE := 0 |
  (KO=0) and (LIBRE=0) : !OTOTAL2, LIBRE := 2 |
  LIBRE=2 : ?IWITHD2, LIBRE := 0 |
  (KO=0) and (LIBRE=0) : ?ILOCWITHD2 |
  (KO=0) and (LIBRE=0) : ?ISTAT2
od ;
```

Le système complet est représenté finalement par le programme de description suivant :

```
cotask TILLSANDDATABASE ;

  cotask DISTRIBUTEUR1 ;

    input ITOTAL1 ;
    output OACC1, OWITHD1, OLOCWITHD1, OSTAT1 ;

    task GUICHET1 ;

      input APAWRONGCARD1, APAGOODCARD1,
            APATOTAL1, APACOM1IMPO1 ;
      output APACARD1, APASTOP1, APADEM11,
             APADEM12, APADEM13 ;
      declare
        PRESENT1 : 0..3 ;
      init
        PRESENT1 := 0 ;
    do
      {cardin1} PRESENT1=0 : !APACARD1, PRESENT1 := 1 |
      PRESENT1=1 : ?APAGOODCARD1, PRESENT1 := 2 |
      PRESENT1=1 : ?APAWRONGCARD1, PRESENT1 := 0 |
      {commande11} PRESENT1=2 : !APADEM11, PRESENT1 := 3 |
      PRESENT1=3 : ?APACOM1IMPO1, PRESENT1 := 2 |
      PRESENT1=3 : ?APATOTAL1, PRESENT1 := 2 |
      {commande12} PRESENT1=2 : !APADEM12 |
      {commande13} PRESENT1=2 : !APADEM13 |
      {cardout1} PRESENT1=2 : !APASTOP1, PRESENT1 := 0
    od ;

    task APAREIL1 ;
```

```
      input GUICARD1, GUISTOP1, GUIDEM11, GUIDEM12,
            GUIDEM13, DATATOTAL1 ;
      output DATAACC1, DATAWITHD1, MEMOLOCWITHD1,
             MEMOSTAT1, GUITOTAL1, GUIWGRONGCARD1,
             GUIGOODCARD1, GUICOM1IMPO1 ;
      declare
        P1 : 0..8 ;
      init
        P1 := 0 ;

    do
      P1=0 : ?GUICARD1, P1 := 1 |
      P1=1 : !GUIWRONGCARD1, P1 := 0 |
      P1=1 : !GUIGOODCARD1, P1 := 2 |

      P1=2 : ?GUIDEM11, P1 := 3 |
      P1=3 : !GUICOM1IMPO1, P1 := 2 |
      {sendacc1} P1=3 : !DATAACC1, P1 := 4 |
      P1=4 : ?DATATOTAL1, P1:= 5 |
      P1=5 : !GUITOTAL1, P1 := 2 |

      P1=2 : ?GUIDEM12, P1 := 6 |
      P1=6 : P1 := 2 |
      P1=6 : !MEMOLOCWITHD1, P1 := 2 |
      {rectotal1} P1=6 : ?DATATOTAL1, P1 := 7 |
      P1=7 : !DATAWITHD1, P1 := 2 |

      P1=2 : ?GUIDEM13, P1 := 8 |
      P1=8 : !MEMOSTAT1, P1 := 2 |

      P1=2 : ?GUISTOP1, P1 := 0
    od ;

    task MEMORY1 ;

      input APALOCWITHD1, APASTAT1 ;
      output DATALOCWITHD1, DATASTAT1 ;
    do
      true : ?APALOCWITHD1 |
      true : !DATALOCWITHD1 |
      true : ?APASTAT1 |
      true : !DATASTAT1
    od ;

port wrongcard1,goodcard1,total1,com1impo1,card1,
     stop1,dem11,dem12,dem13,locwithd1,stat1,
     itotal1,oacc1,owithd1,olocwithd1,ostat1 ;

body
  GUICHET1 (wrongcard1,goodcard1,total1,com1impo1,
```

```
             card1,stop1,dem11,dem12,dem13) //
   APAREIL1 (card1,stop1,dem11,dem12,dem13,
             ITOTAL1,OACC1,OWITHD1,locwithd1,
             stat1,total1,wrongcard1,goodcard1,
             com1impo1) //
   MEMORY1 (locwithd1,stat1,OLOCWITHD1,OSTAT1) ;

 cotask DISTRIBUTEUR2 ;

   input ITOTAL2 ;
   output OACC2, OWITHD2, OLOCWITHD2, OSTAT2 ;

   task GUICHET2 ;

     input APAWRONGCARD2, APAGOODCARD2,
           APATOTAL2, APACOM1IMPO2;
     output APACARD2, APASTOP2, APADEM21,
            APADEM22, APADEM23 ;
     declare
       PRESENT2 : 0..3 ;
     init
       PRESENT2 := 0 ;
   do
     {cardin2} PRESENT2=0 : !APACARD, PRESENT2 := 1 |
     PRESENT2=1 : ?APAGOODCARD2, PRESENT2 := 2 |
     PRESENT2=1 : ?APAWRONGCARD2, PRESENT2 := 0 |
     {commande21} PRESENT2=2 : !APADEM21, PRESENT2 := 3 |
     PRESENT2=3 : ?APACOM1IMPO2, PRESENT2 := 2 |
     PRESENT2=3 : ?APATOTAL2, PRESENT2 := 2 |
     {commande22} PRESENT2=2 : !APADEM22 |
     {commande23} PRESENT2=2 : !APADEM23 |
     {cardout2} PRESENT2=2 : !APASTOP2, PRESENT2 := 0
   od ;

   task APAREIL2 ;

     input GUICARD2, GUISTOP2, GUIDEM21, GUIDEM22,
           GUIDEM23, DATATOTAL2 ;
     output DATAACC2, DATAWITHD2, MEMOLOCWITHD2,
            MEMOSTAT2, GUITOTAL2, GUIWRONGCARD2,
            GUIGOODCARD2, GUICOM1IMPO2 ;
     declare
       P2 : 0..8 ;
     init
       P2 := 0 ;
   do
     P2=0 : ?GUICARD2, P2 := 2 |
     P2=1 : !GUIWRONGCARD2, P2 := 0 |
     P2=1 : !GUIGOODCARD2, P2 := 2 |

     P2=2 : ?GUIDEM21, P2 := 3 |
```

```
      P2=3 : !GUICOM1IMPO2, P2 := 2 |
      {sendacc2} P2=3 : !DATAACC2, P2 := 4 |
      P2=4 : ?DATATOTAL2, P2:= 5 |
      P2=5 : !GUITOTAL2, P2 := 2 |

      P2=2 : ?GUIDEM22, P2 := 6 |
      P2=6 : P2 := 2 |
      P2=6 : !MEMOLOCWITHD2, P2 := 2 |
      {rectotal2} P2=6 : ?DATATOTAL2, P2 := 7 |
      P2=7 : !DATAWITHD2, P2 := 2 |

      P2=2 : ?GUIDEM23, P2 := 8 |
      P2=8 : !MEMOSTAT2, P2 := 2 |

      P2=2 : ?GUISTOP2, P2 := 0
    od ;

    task MEMORY2 ;

      input APALOCWITHD2, APASTAT2 ;
      output DATALOCWITHD2, DATASTAT2 ;
    do
      true : ?APALOCWITHD2 |
      true : !DATALOCWITHD2 |
      true : ?APASTAT2 |
      true : !DATASTAT2
    od ;

  port wrongcard2,goodcard2,total2,com1impo2,card2,
       stop2,dem21,dem22,dem23,locwithd2,stat2,
       itotal2,oacc2,owithd2,olocwithd2,ostat2 ;

  body
    GUICHET2 (wrongcard2,goodcard2,total2,com1impo2,
              card2,stop2,dem21,dem22,dem23) //
    APAREIL2 (card2,stop2,dem21,dem22,dem23,
              ITOTAL2,OACC2,OWITHD2,locwithd2,
              stat2,total2,wrongcard2,goodcard2,
              com1impo2) //
    MEMORY2 (locwithd2,stat2,OLOCWITHD2,OSTAT2) ;

  task DATABASE ;

    input IACC1, IWITHD1, ILOCWITHD1, ISTAT1,
          IACC2, IWITHD2, ILOCWITHD2, ISTAT2 ;
    output OTOTAL1, OTOTAL2 ;
    declare
      KO : 0..1 ;
      LIBRE : 0..2 ;
    init
      KO := 0, LIBRE := 0 ;
```

```
  do
    (KO=0) and (LIBRE=0) : KO := 1 |
    KO=1 : KO := 0 |

    (KO=0) and (LIBRE=0) : ?IACC1, LIBRE := 1 |
    LIBRE=1 : !OTOTAL1, LIBRE := 0 |
    (KO=0) and (LIBRE=0) : !OTOTAL1, LIBRE := 2 |
    LIBRE=2 : ?IWITHD1, LIBRE := 0 |

    (KO=0) and (LIBRE=0) : ?IACC2, LIBRE := 1 |
    LIBRE=1 : !OTOTAL2, LIBRE := 0 |
    (KO=0) and (LIBRE=0) : !OTOTAL2, LIBRE := 2 |
    LIBRE=2 : ?IWITHD2, LIBRE := 0 |

    (KO=0) and (LIBRE=0) : ?ILOCWITHD1 |
    (KO=0) and (LIBRE=0) : ?ISTAT1 |
    (KO=0) and (LIBRE=0) : ?ILOCWITHD2 |
    (KO=0) and (LIBRE=0) : ?ISTAT2
  od ;

port
  TOT1,TOT2,ACC1,ACC2,WITHD1,WITHD2,LOC1,LOC2,ST1,ST2 ;

body
  DISTRIBUTEUR1(TOT1,ACC1,WITHD1,LOC1,ST1) //
  DISTRIBUTEUR2(TOT2,ACC2,WITHD2,LOC2,ST2) //
  DATABASE(ACC1,WITHD1,LOC1,ST1,ACC2,WITHD2,LOC2,ST2,TOT1,TOT2).
```

3. TRADUCTION D'UN PROGRAMME DE DESCRIPTION

Nous présentons dans ce paragraphe la phase de traduction d'un programme de description dans le langage QUASAR, vers un programme PASCAL qui servira à générer le système de transitions représentant l'application décrite.

Cette opération s'effectue en deux temps. Une première phase consiste à générer les corps des tâches composées à partir des corps des tâches composantes par fusion des commandes gardées représentant l'émission et la réception d'une même variable échangée, de façon à exprimer la sémantique du rendez-vous. Cette opération, appelée opération de composition, produit une description de l'application sous forme d'un ensemble plat de commandes gardées dans lesquelles ne figurent plus d'échanges. La deuxième phase de la traduction consiste alors à transcrire cette nouvelle description en un programme PASCAL, qui sera utilisé pour générer le graphe d'états du système.

Nous allons décrire en détail ces deux opérations et nous appliquerons la méthode pour l'exemple de l'exclusion mutuelle de DECKER.

3.1. COMPOSITION DES TACHES

3.1.1. PRINCIPE DE LA COMPOSITION

Etant donné un ensemble de tâches formant une tâche composée, l'opération de composition permet d'obtenir, après substitution des paramètres échangés, la représentation de la tâche composée sous forme d'une tâche élémentaire dont le corps, construit à partir des corps des tâches composantes, exprime le fonctionnement en parallèle des tâches composantes. Le vecteur de variables de la tâche composée est l'union des vecteurs des tâches composantes, et l'initialisation de ces variables est la juxtaposition des instructions d'initialisation des tâches composantes.

L'opération de composition est basée sur la notion de fusion de commandes gardées, que nous allons expliciter ci-dessous.

Soient T1 et T2 deux tâches composant une tâche T, telles que T1 émet une variable échangée VE de T reçue par T2. Notons E1 une instruction d'émission de VE par la tâche T1, et E2 une instruction de réception de VE par la tâche T2. Pour simplifier, nous supposerons que E1 et E2 sont les instructions suivantes (après substitution des paramètres échangés par la variable VE) :

E1 = (C1 : !VE:=x1, A1) , E2 = (C2 : x2:=?VE, A2),

où C1 et C2 sont des conditions booléennes sur les variables internes de T1 et T2, A1 et A2 des affectations simultanées de ces variables, telles que A2 n'affecte pas la variable x2. La sémantique du rendez-vous est telle que l'échange de la variable VE par l'intermédiaire des instructions E1 et E2 n'est possible que si les conditions C1 et C2 sont vraies, et la réalisation du rendez-vous consiste en l'exécution simultanée des instructions A1 et A2 et de l'affectation à x2 de la valeur de x1.

La fusion des commandes gardées E1 et E2 consiste à remplacer ces deux commandes par une commande gardée unique

(C1 and C2 : x2:=x1, A1, A2),

étiquetée par l'union des étiquettes de E1 et de E2, qui exprime exactement le fonctionnement du rendez-vous (remarquons que les affectations A1 et A2 peuvent être simultanées, puisqu'elles portent sur des ensembles de variables disjoints).

La difficulté de l'opération de composition vient du fait que les tâches à composer comportent généralement plusieurs commandes gardées contenant l'émission (ou la réception) d'une même variable échangée. Un rendez-vous entre une tâche émettrice et une tâche réceptrice d'une variable VE peut avoir lieu lorsqu'une commande gardée quelconque comportant l'émission de VE est exécutée simultanément avec une commande gardée quelconque comportant la réception de la variable VE. On procédera donc à une fusion pour chacun des couples possibles.

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 :

- plusieurs tâches émettrices de la même variable (il suffit que l'une d'entre elles émette) ;
- plusieurs tâches réceptrices de la même variable, en mode PORT (l'une quelconque doit la recevoir) ou en mode BROAD (toutes doivent la recevoir).

Cela conduit à définir une opération de fusion assez complexe, de façon à pouvoir dans tous les cas se réserver, après la composition de deux tâches, la possibilité de composer à nouveau la tâche ainsi obtenue, suivant les modes et sens d'échanges de chacune des variables échangées communes aux tâches à composer. Lorsque la totalité de la tâche composée a été obtenue de façon incrémentale, il est alors nécessaire de supprimer les possibilités introduites pour permettre une composition ultérieure, afin de ramener les possibilités

de communication de la tâche composée à celles indiquées par ses paramètres échangés. Cette opération est appelée opération d' encapsulation.

Les opérations de composition et d'encapsulation de réseaux de Petri interprétés ont été énoncées dans [QUE82]. Nous présentons ces opérations, adaptées aux systèmes que nous étudions.

3.1.2. ALGORITHME DE COMPOSITION

Soient T1 et T2 deux tâches élémentaires à composer. Après avoir substitué dans les commandes gardées de T1 et T2 les paramètres échangés par des variables échangées de la tâche composée contenant T1 et T2, (qui peuvent être des paramètres échangés de cette tâche), nous pouvons définir les ensembles suivants :

- Xi le vecteur des variables internes de Ti ;
- XOUTi l'ensemble des variables échangées émises par Ti ;
- XINi l'ensemble des variables échangées reçues par Ti ;
- XIN1i l'ensemble des variables échangées en mode PORT reçues par Ti ;
- XIN*i l'ensemble des variables échangées en mode BROAD reçues par Ti (XIN1i et XIN*i forment une partition du vecteur XINi).

L'opération de composition appliquée à T1 et T2 produit une tâche élémentaire T dont les vecteurs de variables X, XIN et XOUT sont les ensembles (X1 u X2), (XIN1 u XIN2) et (XOUT1 u XOUT2).

L'opération de composition agit uniquement sur les commandes gardées comportant l'échange d'une variable échangée entre les deux tâches.

Le mode de réception d'une variable échangée ne dépend pas des tâches qui la reçoivent (à l'intérieur d'une tâche composée, le mode de réception est toujours le même pour une variable donnée). On a donc nécessairement :

pour tout i,j : XIN1i n XIN*j = 0

Par contre, chacune des intersections suivantes peut ne pas être vide :

- XOUT1 n XOUT2 : les deux tâches émettent la même variable ;
- XIN11 n XIN12 : les deux tâches reçoivent la même variable, échangée en mode PORT ;
- XIN*1 n XIN*2 : les deux tâches reçoivent la même variable, échangée en mode BROAD ;
- XOUTi n XIN1j (j<>i) : une des tâches émet une variable que reçoit l'autre en mode PORT ;

- XOUTi n XIN*j (j<>i) : une des tâches émet une variable que reçoit l'autre en mode BROAD.

Pour chaque variable de chacune de ces intersections, un traitement différent est réalisé afin d'effectuer la composition des deux tâches, d'une façon qui garantit la possibilité ultérieure de pouvoir composer la tâche T obtenue avec une autre de façon incrémentale.

Les cinq cas de rendez-vous sont récapitulés dans la tableau suivant :

T1	T2	Règle
XOUT	XOUT	1
XIN1	XIN1	2
XIN*	XIN*	3
XOUT	XIN1	4
XOUT	XIN*	5

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.

L'opération de composition est basée sur une opération de fusion entre deux commandes gardées, qui a pour effet de remplacer deux commandes gardées t1 et t2 par une commande gardée t, dont la garde est l'intersection booléenne des gardes de t1 et t2, et dont l'action est obtenue à partir des actions de t1 et t2, d'une façon qui dépend du cas de rendez-vous envisagé. L'ensemble des étiquettes associées à t est l'union des étiquettes associées à t1 et t2.

L'ensemble des commandes gardées de T est formé :

- des commandes gardées (recopie intégrale) de T1 et T2 qui ne comportent pas d'échange ;
- des commandes gardées de T1 (resp. T2) qui comportent l'échange d'une variable que n'échange pas T2 (resp. T1) ;
- des commandes gardées obtenues par les règles de composition (qui sont exposées ci-dessous).

Pour l'opération d'encapsulation, il est nécessaire de pouvoir identifier certaines commandes gardées créées par la composition ; nous supposerons donc que nous disposons d'une technique de marquage des commandes gardées (ces commandes gardées seront suivies du symbole (**)).

Dans la suite, nous utiliserons les notations suivantes :

- t1 (resp. t2) une commande gardée de T1 (resp. T2) ;
- x1 (resp. x2) une variable (ou un ensemble de variables) interne(s) de T1 (resp. T2) ;
- c1 (resp. c2) une condition sur les variables internes de T1 (resp. T2) ;
- e1 (resp. e2) une expression sur les variables internes de T1 (resp. T2) ;
- a1 (resp. a2) une affectation simultanée sur les variables internes de T1 (resp. T2), ne modifiant pas les variables de x1 (resp. x2).

Règle 1 : variable émise par les deux tâches

Dans ce cas, les tâches consommatrices (composées ultérieurement) effectuerons un rendez-vous soit avec T1, soit avec T2, sans qu'il y ait interaction entre T1 et T2.

Les commandes gardées comportant l'émission de ces variables ne sont donc pas affectées par la composition de T1 et T2. Elles sont donc recopiées comme commandes gardées de la composée de T1 et T2.

Règle 2 : variable reçue par T1 et T2 en mode PORT

Losque cette variable sera émise par une tâche composée ultérieurement, un rendez-vous aura lieu entre celle-ci et T1 ou T2, sans qu'il y ait d'interaction entre T1 et T2.

Les commandes gardées comportant la réception de cette variable ne sont donc pas affectées par l'opération de composition. Elles sont donc reproduites intégralement dans le corps de la tâche composée de T1 et T2.

Règle 3 : variable reçue par T1 et T2 en mode BROAD

Notons x cette variable. Lorsque x sera émise ultérieurement par une tâche composée, un rendez-vous aura lieu entre elle et les tâches T1 et T2 simultanément (ainsi que toutes les tâches recevant x composées ultérieurement).

Soit T?x1 (reps. T?x2) l'ensemble des commandes gardées de T1 (resp. T2) comportant une réception de x. Un rendez-vous pour l'échange de x est effectué avec une commande gardée quelconque de T?x1 et simultanément une commande gardée quelconque de T?x2. L'ensemble des rendez-vous possibles impliquant T1 et T2 est donc décrit par l'ensemble des couples de T?x1 X T?x2. L'algorithme de composition relatif à la variable échangée x est alors le suivant :

Pour chaque couple (t1 , t2) de T?x1 X T?x2, construire par fu-

sion de t1 et t2 la commande gardée t telle que :

t1 = c1 : x1 := ?x, a1
t2 = c2 : x2 := ?x, a2
t = c1 and c2 : x1, x2 := ?x, a1, a2

La commande gardée construite comporte la réception de x. Ceci permettra d'effectuer ultérieurement une composition avec 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).

Règle 4 : variable émise par T1 et reçue par T2 en mode PORT

Un rendez-vous peut avoir lieu entre T1 et T2 pour l'échange de la variable x, mais T1 peut également effectuer un rendez-vous avec une autre tâche recevant x (composée ultérieurement), ou T2 avec une autre tâche émettant x.

Soit T!x1 (resp. T?x2) l'ensemble des commandes gardées de T1 (resp. T2) comportant une émission (resp. réception) de x. L'ensemble des rendez-vous possibles de T1 est décrit par l'ensemble T!x1 X (T?x2 u {e}), les couples de T!x1 X T?x2 décrivant les rendez-vous entre T1 et T2, et les couples (t1, e) représentant les rendez-vous potentiels de T1 avec d'autres tâches que T2. De même, les rendez-vous de T2 sont décrits par l'ensemble (T!x1 u {e}) X T?x2, les couples (e, t2) représentant les rendez-vous potentiels de T2 avec d'autres tâches que T1. L'algorithme de composition relatif à la variable échangée x est alors le suivant :

- pour chaque couple (t1,t2) de T!x1 X T?x2, 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 and c2 : x1, x2 := e1, a1, a2 ;

- rajouter au corps de la composée de T1 et T2 les transitions de T!x1 et T?x2 telles quelles, pour représenter les communications potentielles de T1 ou T2 avec d'autres partenaires.

Cette opération est répétée pour chaque variable x de XOUT1 n XIN12. On procède de même (symétriquement) pour les variables de XIN11 n XOUT2.

Règle 5 : variable émise par T1 et reçue par T2 en mode BROAD

Un rendez-vous ne peut avoir lieu qu'entre T1 et l'ensemble des

tâches recevant x (dont T2), ou entre une tâche émettant x autre que T1 et cet ensemble.

Soit T!x1 (resp. T?x2) l'ensemble des commandes gardées de T1 (resp. T2) comportant une émission (resp. réception) de x. L'ensemble des rendez-vous possibles de T1 est décrit par l'ensemble T!x1 X T?x2 (puisque T2 participe toujours au rendez-vous). Par contre, l'ensemble des rendez-vous possibles de T2 (et l'ensemble des autres tâches recevant x) sont décrits par l'ensemble (T!x1 u {e}) X T?x2, les couples (e, t2) représentant les rendez-vous potentiels de T2 (et l'ensemble des autres tâches recevant x) avec d'autres tâches que T1. L'algorithme de composition relatif à la variable échangée x est le suivant :

- pour chaque couple (t1,t2) de T!x1 X T?x2 construire par fusion de t1 et t2 une commande gardée t telle que :

t1 = c1 : x1, !x := e1, a1
t2 = c2 : x2 := ?x, a2
t = c1 and c2 : x1, x2, !x := e1, a1, a2 (**) ;

cette commande gardée sera marquée, pour signifier qu'elle représente un rendez-vous, en mode BROAD, bien que comportant l'émission de x ;

- rajouter les commandes gardées de T?x2 sans les modifier, pour représenter les communications potentielles de T2 avec d'autres partenaires.

Remarquons que cette opération laisse un exemplaire non fusionné de t2 comportant la réception de x, alors que les commandes gardées obtenues par fusion comportent son émission. Ceci permettra d'effectuer ultérieurement une composition avec une tâche recevant ou émettant x.

Cette opération est répétée pour chaque variable x de XOUT1 n XIN*2. On procède de même (symétriquement) pour les variables de XIN*1 n XOUT2.

Remarque

Les cinq règles citées ne tiennent pas compte du fait que les vecteurs XIN et XOUT d'une tâche peuvent ne pas être disjoints. Ce cas peut se présenter en deux occasions :

- pour une tâche élémentaire, lorsqu'un paramètre formel d'entrée et un paramètre formel de sortie sont tous deux substitués par la même variable échangée (il ne faut évidemment pas fusionner les commandes gardées comportant son émission avec celles comportant sa réception, puisque les rendez-vous s'effectuent entre deux tâches distinctes) ;

- pour une tâche composée, après application des règles 4 ou 5 lors de la composition des tâches composantes.

Lorsqu'une tâche T1 de ce type est composée avec une autre tâche T2, plusieurs règles deviennent applicables pour une même variable échangée, comme le montre le tableau suivant :

cas	T1	T2	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

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 aux cas du tableau ci-dessus. Les notations sont les mêmes que celles employées plus haut. Pour tout terme aij, l'indice i identifie la tâche T1 ou T2.

<u>Règle 1'</u>

```
t11 = c11 : x11, !x := e11, a11
t12 = c12 : x12 := ?x, a12
t2 = c2 : x2, !x :=e2, a2
t = c12 and c2 : x12, x2 := e2, a12, a2 ;
```

<u>Règle 2'</u>

```
t11 = c11 : x11 := ?x, a11
t12 = c12 : x12, !x := e12, a12
t2 = c2 : x2 := ?x, a2
t = c12 and c2 : x12, x2 := e12, a12, a2 ;
```

<u>Règle 3'</u>

```
t11 = c11 : x11, !x := e11, a11
t12 = c12 : x12 := ?x, a12
t21 = c21 : x21 := ?x, a21
t22 = c22 : x22, !x := e22, a22
t  = c11 and c21 : x11, x21 := e11, a11, a21
t' = c12 and c22 : x12, x22 := e22, a12, a22 ;
```

Règle 4´

```
t11 = c11 : x11, !x := e11, a11
t12 = c12 : x12 := ?x, a12
t2 = c2 : x2, !x :=e2, a2
t = c12 and c2 : x12, x2, !x := e2, a12, a2  (**) ;
```

Règle 5´

```
t11 = c11 : x11, !x := e11, a11
t12 = c12 : x12 := ?x, a12
t2 = c2 : x2 := ?x, a2
t  = c11 and c2 : x11, x2, !x := e11, a11, a2  (**)
t´ = c12 and c2 : x12, x2 := ?x, a12, a2 ;
```

Règle 6´

```
t11 = c11 : x11, !x := e11, a11
t12 = c12 : x12 := ?x, a12
t21 = c21 : x21 := ?x, a21
t22 = c22 : x22, !x := e22, a22
t  = c11 and c21 : x11, x21, !x := e11, a11, a21  (**)
t´ = c12 and c22 : x12, x22, !x := e22, a12, a22  (**)
t´´= c21 and c12 : x21, x12 := ?x, a21, a12.
```

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 PORT par les deux tâches

-A1- pour chaque couple (t1,t2) de T!x1 X T?x2 construire une commande gardée 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?x1 X T!x2

-A3- rajouter telles quelles les commandes gardées de T!x1, T?x1, T!x2, T?x2 pour représenter les échanges potentiels avec d´autres partenaires

REGLE B : Variable émise et reçue par les deux tâches en mode BROAD

-B1- pour chaque couple (t1,t2) de T!x1 X T?x2 construire une

commande gardée t telle que :

t1 = c1 : x1, !x := e1, a1
t2 = c2 : x2 := ?x, a2
t = c1.c2 : x1, x2 := e1, a1, a2 (**)

-B2- procéder de même (symétriquement) pour chaque couple de T?x1 X T!x2

SI T?x1 et T?x2 sont tous les deux non vides :
-B3- pour chaque couple (t1,t2) de T?x1 X T?x2 construire une commande gardée t telle que :

t1 = c1 : x1 := ?x, a1
t2 = c2 : x2 := ?x, a2
t = c1.c2 : x1, x2 := ?x, a1, a2

SINON SI T?x1 est non vide et T?x2 est vide (ou l'inverse) :
-B4- rajouter les commandes gardées de T?x1 et celles de T!x1 telles quelles (ou celles de T?x2 et T!x2 dans le cas inverse)

SINON (dans ce cas T?x1 et T?x2 sont tous deux vides) :
-B5- prendre telles quelles les commandes gardées de T!x1 et T!x2

3.1.3. PROPRIETES DE LA COMPOSITION

1) L'opération de composition est commutative : cette propriété découle immédiatement des rôles symétriques joués par les tâches T1 et T2 dans l'algorithme de composition ;

2) L'opération de composition est associative, 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.

Cette restriction garantit qu'une même variable échangée l'est toujours selon le même mode. Les échanges de variables différentes n'interagissant pas, l'associativité découle immédiatement de l'associativité des deux règles A et B pour une variable donnée.

3.1.4. OPERATION D'ENCAPSULATION

L'opération d'encapsulation :

- a pour premier argument une tâche de vecteurs de variables

échangées XIN et XOUT (cette tâche ayant été produite par l'opération de composition, certaines de ses commandes gardées sont marquées, pour différencier les commandes gardées représentant les rendez-vous en mode BROAD des commandes gardées 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 une tâche ayant pour vecteurs de variables échangées les vecteurs XIN' et XOUT'.

Elle est effectuée de la façon suivante :

- toutes les commandes gardées comportant une réception de x pour toutes les variables x de XIN - XIN' sont supprimées ;

- toutes les commandes gardées marquées, comportant une émission de x pour toutes les variables x de XOUT - XOUT', sont transformées ainsi :

(cl : xl, !x := el, a) devient (cl : xl :=el, al) ;

- toutes les commandes gardées comportant une émission de x pour toutes les variables x de XOUT - XOUT' sont supprimées ;

- le marquage des commandes gardées est supprimé dans tous les cas.

Pour la construction du corps d'une tâche composée, on compose itérativement les différentes tâches composantes, puis la tâche ainsi obtenue est encapsulée de façon à conserver uniquement les vecteurs XIN' et XOUT' définis par la partie formelle de la tâche composée.

La structure arborescente d'un programme de description permet de définir inductivement le traitement de composition des tâches de la description : on effectue un parcours postfixé de l'arbre, en appliquant pour chaque tâche composée l'opération de composition des tâches dont elle est constituée, puis l'opération d'encapsulation au produit de cette composition.

On obtient finalement une représentation de l'application sous la forme d'une tâche élémentaire sans échanges, qui va servir de base à la construction du graphe des états du système décrit.

3.2. GENERATION D'UN PROGRAMME PASCAL

La phase de composition des tâches de la description de l'application a produit une représentation du système décrit comportant un ensemble fini X de variables entières bornées (dont certa-

ines d'entre elles sont initialisées par une affectation vectorielle INITX), et un ensemble fini non structuré de commandes gardées étiquetées dont les actions sont des affectations vectorielles des variables de X. Nous noterons cette représentation "système conditions-actions" de l'application, ou plus simplement CA-système.

Un système conditions-actions a la forme suivante :

```
X,INITX
do
  {étiquettes} cond1 : action1 |
   -- -- -- --
  {étiquettes} condn : actionn
od.
```

Un état s du CA-système est une valeur du vecteur X des variables. Une commande gardée ti = (condi : actioni) est dite validée pour un état s si l'expression booléenne condi est vraie pour l'état s. L'exécution de ti à partir d'un état s n'est possible que si s valide ti, et positionne le système à un état s' par application de l'action actioni au vecteur X des variables.

L'ensemble des états possibles du CA-système est le produit cartésien des domaines de définition des variables de X. L'ensemble des états effectivement atteints par le système dépend de l'initialisation des variables, et ne comporte pas obligatoirement tous les états possibles.

Pour générer le graphe des états du système conditions-actions, il faut construire l'ensemble des états initiaux possibles du système (par énumération exhaustive des valeurs possibles de l'ensemble des variables qui ne sont pas initialisées explicitement dans le programme de description), puis générer l'ensemble des états accessibles du système par exécution systématique de toutes les commandes gardées exécutables à partir de l'un quelconque des états initiaux possibles du système.

Pour mener à bien cette tâche, il est nécessaire de disposer d'un interpréteur de commandes gardées qui produit, à partir d'un état et d'une commande gardée validée pour cet état, l'état atteint après exécution de l'action de la commande gardée, et qui teste les éventuelles erreurs d'exécution telles que les divisions par zéro ou les dépassements de capacité lors de l'affectation de valeurs aux variables.

Il nous est apparu qu'une interprétation, programmée par nos soins, des commandes gardées se révélait assez lente, donc coûteuse. Nous avons alors choisi une solution plus efficace, qui consiste à traduire le CA-système en un programme PASCAL représentant l'application décrite.

Cette façon d'opérer présente de multiples avantages :

- la possibilité d'utilisation dans les expressions de tous les opérateurs et fonctions disponibles dans le langage PASCAL (par exemple, l'opérateur d'appartenance "in" pour les expressions booléennes) ;
- la vérification syntaxique des expressions est assurée par le compilateur PASCAL, de telle sorte que la phase d'analyse syntaxique d'un programme de description en QUASAR se trouve diminuée ;
- l'interprétation des commandes gardées est faite par appel de procédures PASCAL. Le gain de temps produit par la méthode est considérable. Lors de la génération du graphe d'états, le nombre d'interprétations de commandes gardées est en général très important, de telle sorte que les millièmes de secondes économisés pour chaque interprétation compensent finalement très largement le temps de compilation du programme PASCAL généré.

Nous présentons brièvement la structure des programmes PASCAL que nous générons. Nous expliciterons au prochain paragraphe l'implémentation de la méthode de traduction, en détaillant les opérations de composition, d'encapsulation et de génération de programmes PASCAL.

3.2.1. REPRESENTATION DES VARIABLES

Les variables du CA-système produit par l'opération de composition des tâches de la description sont toutes du type intervalle d'entiers borné. Pour des raisons de commodité, ces variables sont traduites dans le programme PASCAL par des variables du type entier universel. L'ordre d'apparition des variables dans le texte du programme de description engendre une numérotation implicite qui permet d'identifier les variables par un entier de [1..n], où n est le nombre total de variables de l'application. Les bornes des intervalles de définition des variables sont connues du programme PASCAL, de façon à pouvoir tester les éventuels débordements de capacité lors de l'exécution d'une commande gardée.

3.2.2. PROCEDURE D'INITIALISATION DES VARIABLES

Une procédure d'initialisation des variables du programme PASCAL est générée, pour permettre l'initialisation effective de ces variables au début de la construction du graphe des états du CA-système. La présence d'une telle procédure évite l'interprétation des expressions dans les instructions d'initialisation, tout en autorisant l'emploi des opérateurs et fonctions standard dans ces mêmes expres-

sions. La procédure effectue l'affectation des variables qui sont initialisées explicitement dans le programme de description, aux valeurs (littérales ou calculées) figurant dans l' instruction d'initialisation du système conditions-actions.

3.2.3. PROCEDURE D'EXECUTION DES COMMANDES GARDEES

Une procédure d'exécution de commandes gardées est générée pour calculer, pour un état et une commande gardée quelconque, l'état atteint après exécution, s'il y a lieu, de l'action de la commande gardée. La procédure doit donc tester dans un premier temps la valeur de la garde de la commande gardée. Si la garde est vraie pour l'état considéré, elle calcule alors l'action de la commande, en exécutant l'affectation vectorielle correspondante. Un test d'appartenance est effectué pour chaque affectation d'une variable, afin de détecter les dépassements.

Le principe de traduction d'une commande gardée en une instruction PASCAL est élémentaire. Il suffit de considérer le fait qu'une commande gardée

t = Condition : Affectation

prise individuellement est équivalente à l'instruction suivante :

IF Condition THEN Affectation

On réalise alors aisément que l'obtention de la procédure d'exécution des commandes gardées du CA-système se traduit finalement par de simples modifications syntaxiques du texte des commandes, après substitution des variables du système conditions-actions par les noms des variables du programme PASCAL.

3.3. REALISATION DE LA TRADUCTION

Le but de ce paragraphe est de montrer comment les outils théoriques que nous venons de présenter ont été utilisés dans la réalisation du système QUASAR. Nous ne souhaitons pas rentrer dans les détails de la programmation de l'analyseur ; nous préférons mettre en évidence la structure des données, et l'agencement des différents algorithmes.

Nous décrivons les étapes successives de la traduction d'un programme de description vers un programme PASCAL représentant l'application étudiée, en soulignant les points jugés importants pour une bonne compréhension du traitement effectué par l'analyseur.

3.3.1. COMPILATION DE LA DESCRIPTION

La compilation d'un programme de description comporte deux étapes. La première étape relève de la compilation classique : analyse syntaxique du programme source, construction de la table des identificateurs et génération d'un langage intermédiaire pour représenter les tâches élémentaires. La deuxième étape consiste à composer les tâches élémentaires pour obtenir une représentation, dans le langage intermédiaire, de l'application sous forme d'un ensemble plat de commandes gardées, qui sera ensuite utilisé pour générer le programme PASCAL.

1) Analyse syntaxique

Cette partie de la compilation ne comporte aucun trait d'originalité. Le programme de description est lu à partir d'un fichier texte, et analysé terme par terme par l'intermédiaire de procédures récursives descendantes. Les informations relatives aux variables de la description sont mémorisées pour permettre de faire ultérieurement le lien entre les parties description et analyse des systèmes étudiés.

Précisons que l'analyseur syntaxique ne remplit pas toutes les fonctions d'un analyseur classique. La syntaxe des expressions n'est pas vérifiée (ce travail sera réalisé plus tard par le compilateur PASCAL), et le traitement d'erreurs est sommaire. Toute erreur de syntaxe non triviale provoque l'interruption de la compilation, après impression d'un message d'erreur. Cet aspect peut sembler rude, mais il traduit correctement le fait que nous avons mis l'accent dans QUASAR sur les problèmes d'évaluation, et que le compilateur est avant tout un traducteur.

2) Production d'un langage intermédiaire

Parallèlement à l'analyse syntaxique, le compilateur génère pour chaque tâche élémentaire un fichier temporaire qui contient, sous forme codée, les commandes gardées qui composent le corps de la tâche.

Le format d'un fichier temporaire est le suivant :

```
e <RC>          (en-tête signifiant qu'il s'agit
                   d'une tâche élémentaire )
[commande gardée <RC>]+
```

Chaque commande gardée occupe une ligne du fichier et a le format suivant :

[{étiquette}]* condition :
    [échange [,affectation]*] _|

Les étiquettes sont telles qu'elles figurent dans le "source" en langage de description. Les conditions, échanges et affectations respectent la même syntaxe que dans le langage de description, mais les identificateurs sont remplacés par :

- $n qui désigne la n-ième variable interne déclarée depuis le début de la description (toutes tâches confondues) ;

- ^npm qui désigne le m-ième paramètre échangé de la n-ième tâche déclarée depuis le début de la description.

3) Composition des tâches

La composition permet de construire le corps d'une tâche composée en langage intermédiaire à partir des fichiers temporaires des tâches qui la composent.
Les tâches composées sont également représentées par des fichiers temporaires en langage intermédiaire dont l'en-tête est la lettre "c".

Avant d'effectuer la composition, les paramètres échangés par les tâches à composer sont substitués par les paramètres effectifs définis par les instances de tâches formant le corps de la tâche composée.

Les tâches sont ensuite composées itérativement deux à deux, et l'opération d'encapsulation est appliquée pour ramener les possibilités de communication de la tâche composée à celles définies par ses paramètres.

Le programme de composition et d'encapsulation est assez lent, car il effectue des parcours répétés de fichiers texte dont les accès ne peuvent être que séquentiels. Néanmoins, après application successive de la composition, on obtient un fichier unique en langage intermédiaire qui représente la totalité de l'application décrite par un seul système conditions-actions. Tous les autres fichiers temporaires sont alors détruits.

4) Traitement des étiquettes

Les étiquettes figurant dans le programme de description peuvent être utilisées dans les formules de spécification pour exprimer la possibilité d'exécuter une action, ou le contrôle d'une tâche de l'application étudiée par rapport à ses actions.

La traduction d'un programme de description doit donc générer un ensemble d'informations sur les étiquettes, pour permettre d'évaluer après la construction du graphe des états les prédicats qui font intervenir des noms d'actions.

Le traitement effectué par QUASAR lors de l'analyse syntaxique et de la traduction de la description d'un système réparti est le suivant :

- toute commande gardée est numérotée implicitement par son ordre d'apparition i dans le texte du programme ; si la transition est étiquetée, on associe à chacune des étiquettes ce même numéro i ;

- lors de la production du langage intermédiaire d'une tâche élémentaire, on ajoute à chacune de ses commandes gardées l'étiquette {#i}, où i est le numéro de la commande gardée ;

- pour chaque tâche de la description, on mémorise l'ensemble des commandes gardées dont elle est constituée : si la tâche est élémentaire, on effectue l'union des numéros des commandes gardées de son corps ; si la tâche est une tâche composée, on fait l'union des ensembles de ses tâches composantes ;

- lors de la composition des tâches, la fusion de deux commandes gardées effectue l'union des étiquettes associées aux deux commandes gardées initiales. Ainsi, dans le système conditions-actions composé représentant l'application dans sa totalité, toute commande gardée j comporte en plus des étiquettes figurant dans la description, l'ensemble des étiquettes {#i} de toutes les commandes gardées dont elle est issue.

Ces différentes opérations permettent, pour toute commande gardée du système conditions-actions composé, de repérer si la commande gardée fait intervenir une certaine action étiquetée de l'application. On construit alors pour toute étiquette de la description l'ensemble des commandes gardées du système auxquelles participe l'action étiquetée.

Par ailleurs, pour toute tâche de la description, on construit l'ensemble des commandes gardées du système qui font intervenir une action quelconque de cette tâche.

3.3.2. GENERATION D'UN PROGRAMME PASCAL

Le programme PASCAL généré à partir du fichier en langage intermédiaire représentant l'application décrite utilise pleinement les possibilités de compilation séparée offertes par le langage PASCAL.

Les variables de la description sont représentées par un tableau d'entiers T, chaque élément T[i] contenant la valeur de la variable i (un état du CA-système est ainsi caractérisé par une valeur de T), et les bornes des variables sont contenues dans deux tableaux INF et SUP, initialisés lors de la déclaration des variables dans la description. Ces différents tableaux étant des éléments externes au programme PASCAL généré, on peut ainsi les définir une fois pour toutes en leur donnant une taille fixe (ce qui limite néanmoins le nombre de variables possibles dans une description), et on peut aussi écrire une procédure de dépassement de capacité unique, commune à toutes les descriptions soumises à l'analyseur. Les deux procédures générées pour une application agiront alors uniquement sur le tableau T.

1) Procédure d'initialisation des variables

Cette procédure se contente d'initialiser le tableau T. Les variables initialisées explicitement dans le programme de description sont affectées par les valeurs des instructions d'initialisation de la description (ces valeurs étant éventuellement calculées), et les autres variables sont affectées par défaut à la valeur de la borne inférieure de leur intervalle de définition, diminuée de 1.

2) Procédure d'exécution des commandes gardées

Chaque commande gardée est identifiée par un numéro j, auquel correspond une procédure TRANSITIONj qui modifie le vecteur T selon l'action de la commande gardée. Les affectations simultanées sont en fait exécutées séquentiellement, mais une copie locale de la valeur du tableau T à l'entrée dans la procédure est utilisée pour calculer les expressions des parties droites des affectations, simulant ainsi parfaitement la simultanéité des différentes affectations. Chaque affectation est suivie d'un appel de la procédure de test de dépassement de capacité.

Une procédure unique, de nom SYSTEME et admettant pour paramètre d'entrée un numéro de commande gardée j, teste la condition de la commande pour l'état T, et si cette condition est vraie, exécute

l'action de la commande par appel de la procédure TRANSITIONj.

Deux paramètres résultat permettent de savoir d'une part si la condition de la commande j était vraie pour l'état T, et d'autre part si un débordement de capacité a eu lieu ou non.

Nous allons désormais présenter l'application de notre méthode de traduction pour le programme de description de l'exclusion mutuelle de DECKER.

3.4. UN EXEMPLE DE TRADUCTION

La description de l'algorithme d'exclusion mutuelle de DECKER est formée des trois tâches P1, P2 et CONTROLER s'exécutant en parallèle, et regroupées au sein d'une tâche composée DECKER (cf II.2.10.1). Nous donnons pour chacune des trois tâches élémentaires le contenu des fichiers temporaires générés. Nous présentons ensuite le CA-système obtenu après composition des tâches, et le programme PASCAL produit.

tâche P1

```
  ( $1 représente la variable y1 de P1 )
e
  {local_p1}{#1} $1=0 : $1:=1 |
  {dem_res1}{#2} $1=1 : !^1p2,$1:=2 |
  {aut_accès1}{#3} $1=2 : ?^1p1,$1:=3 |
  {libere_1}{#4} $1=3 : !^1p3,$1:=0 |
```

tâche P2

```
  ( $2 représente la variable y2 de P2 )
e
  {local_p2}{#5} $2=0 : $2:=1 |
  {dem_res2}{#6} $2=1 : !^2p2,$2:=2 |
  {aut_accès2}{#7} $2=2 : ?^2p1,$2:=3 |
  {libere_2}{#8} $2=3 : !^2p3,$2:=0 |
```

tâche CONTROLER

```
  ( $3,$4,$5,$6 et $7 représentent les
     variables c1,c2,x1,x2 et turn )
e
  {dem1}{#9} $5=0 : ?^3p1,$3:=0,$5:=1 |
  {acces1}{#10} ($5=1)and($4=1) : !^3p5,$5:=2 |
  {lib1}{#11} $5=2 : ?^3p3,$7:=0,$3:=1,$5:=0 |
  {a1}{#12} ($5=1)and($4=0)and($7=0) : $3:=1,$5:=3 |
  {a2}{#13} ($5=3)and($7=1) : $3:=0,$5:=1 |
```

```
{dem2}{#14} $6=0 : ?^3p2,$4:=0,$6:=1 |
{acces2}{#15} ($6=1)and($3=1) : !^3p6,$6:=2 |
{lib2}{#16} $6=2 : ?^3p4,$7:=1,$4:=1,$6:=0 |
{b1}{#17} ($6=1)and($3=0)and($7=1) : $4:=1,$6:=3 |
{b2}{#18} ($6=3)and($7=0) : $4:=0,$6:=1 |
```

CA-système composé

Le système conditions-actions obtenu après les opérations de composition et d'encapsulation appliquées aux trois fichiers ci-dessus est le suivant (nous avons modifié l'ordre des commandes gardées pour faciliter la comparaison avec le texte du programme de description) :

```
c
  {local_p1}{#1} $1=0 : $1:=1 |
  {dem1}{dem_res1}{#2}{#9}
    ($1=1)and($5=0) : $1:=2,$3:=0,$5:=1 |
  {acces1}{aut_acces1}{#3}{#10}
    ($1=2)and($5=1)and($4=1) : $1:=3,$5:=2 |
  {lib1}{libere_1}{#4}{#11}
    ($1=3)and($5=2) : $1:=0,$7:=0,$3:=1,$5:=0 |
  {a1}{#12} ($5=1)and($4=0)and($7=0) : $3:=1,$5:=3 |
  {a2}{#13} ($5=3)and($7=1) : $3:=0,$5:=1 |

  {local_p2}{#5} S2=0 : $2:=1 |
  {dem2}{dem_res2}{#6}{#14}
    ($2=1)and($6=0) : $2:=2,$4:=0,$6:=1 |
  {acces2}{aut_acces2}{#7}{#15}
    ($2=2)and($6=1)and($3=1) : $2:=3,$6:=2 |
  {lib2}{libere_2}{#8}{#16}
    ($2=3)and($6=2) : $2:=0,$7:=1,$4:=1,$6:=0 |
  {b1}{#17} ($6=1)and($3=0)and($7=1) : $4:=1,$6:=3 |
  {b2}{#18} ($6=3)and($7=0) : $4:=0,$6:=1 |
```

Programme PASCAL

Nous ne donnons pas le texte complet du programme, car les douze procédures correspondant aux douze commandes gardées sont similaires.

```
program decker(output);
$import
   'dcl(pascal)':T,TT;
   'cesar(pascal)':testborne;
   'cmd_sim(pascal)':debordement $
$export
   systeme,initsysteme $
const
```

```
   NBVAR=60;
type
   etat=array[1..NBVAR] of integer;
var
   T,TT:etat;

procedure DEBORDEMENT(num_ident,num_transi:integer;
                      var err:boolean);external;
procedure TESTBORNE(numid:integer;var err:boolean);
                      external;

procedure INITSYSTEME(var hb:boolean);
begin
   TT:=T;
   TT[1]:=0; testeborne(1,hb);
   TT[2]:=0; testeborne(2,hb);
   TT[3]:=1; testeborne(3,hb);
   TT[4]:=1; testeborne(4,hb);
   TT[7]:=1; testeborne(7,hb);
   TT[5]:=0; testeborne(5,hb);
   TT[6]:=0; testeborne(6,hb);
   T:=TT;
end;

procedure TRANSITION1(var ex,er:boolean);
begin
   {local_p1}{#1}
   if T[1]=0 then
   begin
      TT[1]:=1;
      debordement(1,1,er);
      ex:=true;
   end
   else ex:=false;
end;

- - - -
- - - -

procedure SYSTEME(i:integer;var ex,er:boolean);
begin
   TT:=T;
   er:=false;
   case i of
     1 : transition1(ex,er);
     2 : transition2(ex,er);
     - - - -
     - - - -
     12: transition12(ex,er);
   end;
   T:=TT;
```

```
end;

begin
end.
```

Remarquons que le nom des procédures PASCAL générées (SYSTEME ET INITSYSTEME) est indépendant de l'application décrite. Ceci est indispensable pour que notre programme de construction du graphe d'états, qui référence les deux procédures, puisse être accepté par le compilateur PASCAL. Un mécanisme d'ajout de noms (que nous expliquons en annexe), permet de distinguer les procédures des différentes applications soumises au programme d'analyse.

Après avoir produit un programme PASCAL, le traducteur appelle le compilateur PASCAL pour générer le code objet des procédures INIT-SYSTEME et SYSTEME. La phase de traduction d'un programme de description est alors achevée et tous les outils nécessaires à la construction du graphe des états du système sont en place.

4. CONCLUSION

Nous avons présenté dans ce chapitre le langage de description du système QUASAR et sa traduction vers un programme PASCAL représentant l'application décrite.

Le langage de description peut surprendre par sa pauvreté : toutes les possibilités syntaxiques et structurelles des langages de programmation classiques (PASCAL, PLI, ...) lui font défaut. De ce fait, le langage est d'un emploi peu commode, et demande à l'utilisateur un effort réel pour décrire des systèmes complexes. Nous pensons cependant que sa puissance de description est suffisante pour représenter, sous réserve d'un travail intermédiaire important, une application répartie quelconque.

Le point le plus faible du langage de description nous semble être l'absence de possibilité de définition de types structurés tels que des tableaux ou des enregistrements. Cette limitation apparaît parfois insurmontable, notamment pour l'emploi de variables échangées représentant des paquets d'informations ; l'utilisateur est alors obligé de restreindre la conformité de la description avec le fonctionnement de l'application décrite, ou d'utiliser des stratagèmes pour contourner certaines difficultés.

Les objectifs de QUASAR étant avant tout de prouver la faisabilité d'une méthode d'analyse, il ne semble pas nécessaire d'attacher une importance trop grande à la puissance de son langage de description. Celui-ci possède en fait la puissance que nos moyens nous ont permis de lui donner, dans l'élaboration de notre prototype d'analyseur. Il va de soi que des systèmes d'analyse utilitaires devront répondre à des critères plus exigeants.

CHAPITRE III

LE LANGAGE DE SPECIFICATION

1. Introduction
2. La logique temporelle CTL
3. Spécification de propriétés dans QUASAR
4. Conclusion

1. INTRODUCTION

L'objet de ce chapitre est de présenter le langage de spécification du système QUASAR. Ce langage permet de construire des formules qui expriment des propriétés de fonctionnement d'une application, décrite dans le langage de description de QUASAR. Ces formules se présentent comme des formules bien formées d'une logique temporelle, qui utilise, en plus des opérateurs booléens classiques, des opérateurs temporels permettant d'exprimer des modalités dans le temps.

Le lien entre le langage de spécification et la description d'une application est effectué au moyen d'une interprétation des formules de la logique temporelle dans le modèle généré par la traduction de la description. Le vocabulaire terminal du langage, formé d'un ensemble de variables propositionnelles, sera interprété en termes de prédicats booléens portant sur les variables du programme de description. Les opérateurs temporels seront interprétés sous forme de conditions sur les séquences d'exécution dans le graphe des états du système.

Nous nous intéressons dans un premier temps à la logique temporelle utilisée pour la construction des formules de spécification ; nous décrivons cette logique de façon formelle, et nous en donnons une procédure de décision. Nous présentons ensuite le langage de spécification de QUASAR, et son interprétation dans le graphe des états d'une application.

Nous traitons finalement un par un les exemples présentés au chapitre précédent. Nous décrivons pour chaque application un ensemble de propriétés comportementales qui doivent être vérifiées par la description fournie. Ces propriétés seront reprises ultérieurement dans le chapitre d'utilisation de QUASAR, et seront testées par notre système d'analyse.

2. LA LOGIQUE TEMPORELLE CTL

2.1. INTRODUCTION

La formulation de propriétés comportementales d'un système discret fait souvent appel à des notions temporelles telles que : il est possible que, il est toujours vrai que, il est inévitable que, sous telle condition il est impossible que, ... , qui permettent d'exprimer qu'une certaine propriété se trouve vérifiée à certains moments du fonctionnement du système.

L'utilisation d'une logique temporelle pour formuler les propriétés répond à ces besoins, et permet une description rigoureuse et non ambigüe du comportement d'un système. La spécification est abstraite, et donc indépendante de choix algorithmiques. Par ailleurs, la représentation des spécifications sous forme de formules permet une manipulation formelle des spécifications : si la logique temporelle comporte une procédure de décision, on peut d'une part découvrir une inconsistance formelle des formules, et d'autre part détecter des redondances dans les spécifications par comparaison des formules.

Les logiques temporelles sont classées en deux catégories, les logiques du temps arborescent et les logiques du temps linéaire, suivant que l'on considère l'ensemble des séquences d'exécution possibles ou une séquence particulière. Lamport [LAM80] a discuté l'utilité de ces deux approches, et conclut que la première est adaptée à l'étude du non déterminisme, tandis que la seconde convient à l'étude des problèmes de concurrence. Nous nous intéresserons donc aux logiques de la première catégorie.

Emerson et Halpern [EH82] ont décrit un ensemble de logiques du temps arborescent construites autour du sytème UB ("unified system of branching time" [BMP81]), et les ont classées selon leur puissance d'expression. Afin de permettre une spécification la plus complète possible, nous avons choisi la logique dont le pouvoir d'expression est le plus grand, la logique temporelle CTL ("Conditional branching Time Logic") [CE82] [QS82]. Nous en rappelons les principales caractéristiques.

2.2. SYNTAXE ET SEMANTIQUE DE CTL

Le modèle sous-jacent de la logique est l'arbre T de toutes les exécutions possibles d'un programme non déterministe. La logique CTL est construite à partir du calcul propositionnel par adjonction

de deux opérateurs temporels. Le vocabulaire terminal du langage est un ensemble F0 de formules atomiques (ou variables propositionnelles). L'ensemble F des formules de la logique est obtenu par application des règles suivantes :

1. F0 est inclus dans F ;

2. pour tout f de F, NOT f appartient à F ;

3. pour tout f1, f2 de F, les expressions f1 AND f2, f1 OR f2, f1 => f2 sont des éléments de F ;

4. pour tout f1, f2 de F, ALL[f1]f2 est un élément de F (ALL[f1]f2 est vrai à un noeud s de T, si pour toute branche de T partant de s, f2 est vrai jusqu'à ce que f1 devienne faux) ;

5. pour tout f1,f2 de F, SOME[f1]f2 est un élément de F (SOME[f1]f2 est vrai à un noeud s de T, si il existe au moins une branche de T partant de s telle que f2 est vrai jusqu'à ce que f1 devienne faux).

On notera par commodité POT et INEV les opérateurs duaux de ALL et SOME. Leurs significations sont les suivantes :

POT[f1]f2 = NOT ALL[f1] (NOT f2)
(POT[f1]f2 est vrai à un noeud s de T, si il existe une chemin de s à un noeud s' vérifiant f2, et tel que tous les noeuds du chemin - sauf s' éventuellement -, vérifient f1) ;

INEV[f1]f2 = NOT SOME[f1](NOT f2)
(INEV[f1]f2 est vrai à un noeud s de T, si pour toute branche b de T partant de s, on atteint un noeud s'(b) vérifiant f2, et tel que tous les noeuds du chemin - sauf s'(b) éventuellement -, vérifient f1).

Nous donnons désormais une définition plus formelle de la sémantique de CTL.

Un modèle T pour CTL est un triplet (S,P,R), où S est un ensemble d'états et P une affectation des variables propositionnelles pour chaque état. Pour un état s et une variable a, a est un élément de P(s) si et seulement si a est vrai à l'état s. R est une relation binaire entre les états qui définit la structure de T : s R t signifie que t est un successeur direct de s. Une s-branche b est un chemin b = (s=s_0,s1_,s_2,...) telle que s_i est un élément de S et que s_i R s_i+1.

Nous dirons qu'un état s de T satisfait une formule f (et nous noterons T,s |= f, ou s |= f si T est sous entendu) dans les cas suivants :

```
1. pour toute variable a, s |= a
   ssi a est élément de P(s) ;

2. s |= NOT a    ssi  s |= a  est faux ;

3. s |= p OR q   ssi  s |= p  ou  s |= q ;

4. s |= p AND q  ssi  s |= p  et  s |= q ;

5. s |= p => q   ssi  s |= NOT p  ou  s |= q ;

6. s |= ALL[p]q
                 ssi  pout toute s-branche b,
                      et pour tout k >= 0

                      ( b=(s=s_0,s_1,...,s_k,...) et
                       s_0 |= p et ... et s_k-1 |= p )
                            => s_k |= q ;

7. s |= POT[p]q
                 ssi  il existe une s-branche b,
                      et il existe un k >= 0

                      b=(s=s_0,s_1,...,s_k,...) et
                      s_0 |= p et ... et s_k-1 |= p
                      et  s_k |= q ;

8. s |= SOME[p]q
                 ssi  il existe une s-branche b,
                      et pour tout k >= 0

                      ( b=(s=s_0,s_1,...,s_k,...) et
                       s_0 |= p et ... et s_k-1 |= p )
                            => s_k |= q ;

9. s |= INEV[p]q
                 ssi  pout toute s-branche b,
                      il existe un k >= 0

                      b=(s=s_0,sl,...,s_k,...) et
                      s_0 |= p et ... et s_k-1 |= p
                      et s_k |= q.
```

Une formule p sera dite satisfaisable s'il existe un modèle T et un état s de T tel que T,s |= p.

Une formule p sera dite valide pour le modèle T si pour tout état s de T, on a T,s |= p.

Une formule sera dite valide si elle est valide pour n'importe quel modèle. Une telle formule sera appelée théorème de la logique CTL.

Nous allons décrire au prochain paragraphe une procédure qui permet de décider si une formule est satisfaisable ou non, et que nous avons utilisée pour programmer un démonstrateur de théorèmes de la logique temporelle CTL.

2.3. PROCEDURE DE DECISION

Nous présentons dans ce paragraphe une procédure de décision de la logique temporelle CTL, basée sur la méthode des tableaux sémantiques introduite par Hughes et Cresswell [HC68]. Un algorithme de décision a été proposé par Manna et Pnueli pour le système UB [BMP81], revu et corrigé par Ben-Ari [BA82]. Nous avons repris cet algorithme, en l´adaptant à la logique CTL.

Les objectifs de la procédure de décision sont de déterminer si une formule quelconque f de la logique est satisfaisable, c.à.d. s´il existe un modèle T=(S,P,R) et un état s de S tel que T,s |= f.

Le principe de la méthode consiste à tenter de construire progressivement un modèle pour f, en essayant à chaque étape de satisfaire les impératifs exprimés par les opérateurs temporels de la formule. Ces impératifs peuvent s´avérer incompatibles entre eux, et rendre ainsi impossible l´obtention d´un modèle pour f ; la formule est alors dite non satisfaisable. Inversement, les impératifs peuvent être toujours compatibles ; la formule est alors satisfaisable, et la méthode produit un modèle pour f.

Pour présenter la méthode de décision, nous introduisons un opérateur, noté PRE, tel que pour toute formule f, un état s d´un modèle (S,P,R) satisfait PRE f ssi il existe un successeur direct s´ de s tel que s´ satisfait f :

s |= PRE f ssi il existe s´, tel que s R s´ et s´ |= f.

L´opérateur PRE n´est pas un opérateur de la logique CTL, et ne peut donc pas être employé dans les formules à décider. Il est introduit uniquement pour la présentation de la procédure de décision. Nous notons PRETILDA l´opérateur dual de PRE, défini par :

s |= PRETILDA f ssi pour tout s´, (s R s´ => s´ |= f).

L´opérateur PRE (resp. PRETILDA) permet d´exprimer, pour un état s, qu´un au moins des successeurs directs de s satisfait (resp. que tous les successeurs directs de s satisfont) une formule. Ces opérateurs sont utilisés pour définir la satisfaction d´une formule temporelle par un état, en fonction uniquement de la satisfaction d´un ensemble de formules par l´état et par ses successeurs directs.

Prenons pour exemple l'atteignabilité conditionnelle exprimée par l'opérateur POT[p]q. Un état s satisfait POT[p]q s'il existe un chemin menant à un état satisfaisant q, dont tous les états intermédiaires satisfont p. Cette condition s'exprime comme l'union de deux conditions : soit s satisfait q, soit s satisfait (p AND PRE POT[p]q). On peut ainsi définir "récursivement" la satisfaction d'une formule temporelle par un état :

Soient T = (S,P,R) un modèle, s un état de T,
p et q deux formules de CTL,

T,s |= ALL[p]q ssi T,s |= (q AND (NOT p OR PRETILDA ALL[p]q))

s doit satisfaire q, et si s satisfait de plus p, alors tous les successeurs directs de s doivent satisfaire ALL[p]q

T,s |= SOME[p]q ssi T,s |= (q AND (NOT p OR PRE SOME[p]q))

s doit satisfaire q, et si s satisfait de plus p, alors un au moins des successeurs directs de s doit satisfaire SOME[p]q

T,s |= POT[p]q ssi T,s |= (q OR (p AND PRE POT[p]q))

s satisfait q, ou bien s satisfait p et un au moins de ses successeurs directs satisfait POT[p]q

T,s |= INEV[p]q ssi T,s |= (q OR (p AND PRETILDA INEV[p]q))

s satisfait q, ou bien s satisfait p et tous ses successeurs directs satisfont INEV[p]q

Nous présentons désormais la méthode de décision, en trois étapes. Nous introduisons tout d'abord les structures de Hintikka, nous montrons ensuite que la satisfaisabilité d'une formule est équivalente à l'existence d'une structure de Hintikka pour la formule, et nous décrivons finalement l'algorithme de décision de la logique temporelle CTL.

2.3.1. STRUCTURE DE HINTIKKA

Une structure est un triplet (S,P,R) où S est un ensemble d'états, P une affectation d'un ensemble de formules temporelles pour chaque état, et R une relation binaire entre états. Une structure diffère d'un modèle par le fait que l'affectation des formules pour un état n'est pas obligatoirement complète ; pour une formule p et un état s, P(s) peut ne contenir aucune des deux formules p et NOT p.

Une formule est appelée formule de propagation si son opérateur principal est PRE ou PRETILDA. Une formule est dite élémentaire si elle est une variable propositionnelle, la négation d'un variable propositionnelle ou une formule de propagation.

Par commodité, nous simplifierons les doubles négations présentes dans les formules et nous remplacerons les opérateurs par leurs duaux pour que les formules manipulées soient non négatives.

Nous définissons deux catégories de formules, les a-formules a qui expriment la conjonction de deux formules a1 et a2, et les b-formules b qui expriment la disjonction de deux formules b1 et b2. Ces deux types de formules sont énumérés par le tableau suivant, et utilisent les résultats décrits plus haut (cf III.2.3) :

a	a1	a2
p AND q	p	q
ALL[p]q	q	NOT p OR PRETILDA ALL[p]q
SOME[p]q	q	NOT p OR PRE SOME[p]q

b	b1	b2
p OR q	p	q
POT[p]q	q	p AND PRE POT[p]q
INEV[p]q	q	p AND PRETILDA INEV[p]q

Une structure de Hintikka est une structure dans laquelle l'ensemble des affectations des formules aux états est consistante, pour la notion de satisfaisabilité définie pour les modèles. Plus formellement :

une structure H = (S,P,R) est appelée structure de Hintikka si et seulement si pour tout état s de S, les propriétés suivantes sont vérifées :

H1. NOT p est dans P(s) => p n'est pas dans P(s)
(H est consistant) ;

H2. une a-formule a est dans P(s) => a1 et a2 sont dans P(s) ;

H3. une b-formule b est dans P(s) => b1 ou b2 est dans P(s) ;

H4a. PRETILDA p est dans P(s) => pour tout t tel que s R t , p est dans P(t) ;

H4b. PRE p est dans P(s) => il existe t, tel que s R t et p est dans P(t) ;

H4c. POT[p]q est dans P(s) =>
il existe une s-branche b,
et il existe un $k >= 0$ tel que

$b=(s=s_0,s_1,\ldots,s_k,\ldots)$ et
p est dans $P(s_0)$ et ... et p est dans $P(s_{k-1})$
et q est dans $P(s_k)$;

H4d. INEV[p]q est dans P(s) =>
pour toute s-branche b,
il existe un $k >= 0$ tel que

$b=(s=s_0,s_1,\ldots,s_k,\ldots)$ et
p est dans $P(s_0)$ et ... et p est dans $P(s_{k-1})$
et q est dans $P(s_k)$.

Une structure H = (S,P,R) est appelée structure de Hintikka pour une formule p si pour un état s de S, p appartient à P(s).

Remarquons qu'un modèle pour une formule p est une structure de Hintikka pour p. Inversement, on peut construire à partir d'une structure de Hintikka (S,P,R) pour p, un modèle (S,P',R) pour p tel que P(s) est inclu dans P'(s) pour tout état s de S.
Nous pouvons alors énoncer le théorème suivant, dont la démonstration est immédiate :

<u>Théorème 3.1 :</u>

Une formule p est satisfaisable si est seulement si il existe une structure de Hintikka pour p.

Nous présentons désormais la méthode des tableaux sémantiques et nous montrerons qu'il existe une procédure de décision pour la logique temporelle CTL.

2.3.2. METHODE DES TABLEAUX SEMANTIQUES

Notre méthode des tableaux sémantiques consiste en la recherche d'une structure de Hintikka pour une formule p0 de la logique CTL, et se décompose en trois étapes :

- construction d'un tableau sémantique pour la formule p0 ;
- application d'un algorithme de marquage, qui détermine si le tableau sémantique pour p0 peut être transformé en une structure de Hintikka pour p0 ; si tel est le cas, l'algorithme produit une structure pour p0, qui n'est pas encore nécessairement une structure de Hintikka pour p0 ;
- transformation de la structure produite par l'étape précédente

(s'il y a lieu) en une structure de Hintikka pour p0, par application d'un algorithme de déroulement d'une structure.

Construction d'un tableau sémantique :

Un tableau sémantique T(p0) pour une formule p0 est un graphe orienté dont les noeuds sont étiquetés par des formules dérivées de la formule initiale p0. On note n un noeud du graphe, U_n l'ensemble des formules étiquetant n, et p E n pour "p appartient à U_n".

Soit p0 une formule de la logique temporelle CTL, et soit n0 la racine du tableau T(p0) étiquetée par U_n0 = {p0}. T(p0) est construit inductivement par application des règles suivantes aux noeuds n qui sont des feuilles de T(p0) :

Ra : si a E n (a une a-formule), créer un fils n1 de n tel que

U_n1 = (U_n-{a}) u {a1,a2} ;

La règle Ra crée un noeud n1 fils de n, identique à n, mais tel que la a-formule a est substituée par les deux formules a1 et a2 de la conjonction exprimée par a.

Rb : si b E n (b une b-formule), créer deux fils n1 et n2 de n tel que

U_ni = (U_n-{b}) u {bi} , i = 1,2 ;

La règle Rb est appliquée pour une b-formule b qui exprime la disjonction de deux formules b1 et b2. Elle crée deux noeuds n1 et n2 fils de n, chacun exprimant une des deux alternatives de la formule b.

Les règles Ra et Rb ont pour but de remplacer un noeud feuille de T(p0) par un ensemble de noeuds contenant uniquement des formules élémentaires.

Rx : si toutes les formules de U_n sont des formules élémentaires, considérons l'ensemble suivant :

V_n = { PRE p1, ..., PRE pk } u
{ PRETILDA q1, ..., PRETILDA ql }
l'ensemble des formules de propagation de n ;

Construire k fils ni, i=1,...,k de n tels que

U_ni = {pi, q1, ..., ql}.

La règle Rx crée, pour un noeud dont toutes les formules sont élémentaires, un ensemble de noeuds contenant tous les formules "obligatoires" exprimées par les formules PRETILDA, et chacun une des

formules d'atteignabilité exprimées par les formules PRE.

Un noeud n sera noté a-noeud, b-noeud ou x-noeud selon que la règle Ra, Rb ou Rx lui a été appliquée.

Le nombre de formules différentes qui peuvent constituer l'ensemble U_n d'un noeud n de T(p0) est fini et borné par une constante liée à la longueur de la formule. Aussi les deux règles de terminaison suivantes permettent-elles d'assurer que la construction du tableau T(p0) est finie :

1) si p E n et NOT p E n , le noeud n est dit fermé et aucune règle ne lui est appliquée ;

2) si un noeud m est sur le point d'être crée comme fils de n, et si il existe un noeud n' de T(p0) tel que $U_n'=U_m$, ne pas créer m mais rattacher n à n'.

Le tableau sémantique ainsi construit peut comporter des noeuds fermés. Un noeud fermé correspond à une inconsistance, c.à.d. à l'impossibilité de satisfaire simultanément des impératifs exprimés par un ensemble de formules temporelles. Il faut, pour chaque noeud fermé, déterminer quel est l'ensemble de formules incompatibles qui est à la base de l'inconsistance. Pour cela, il faut propager les inconsistances, selon un ensemble de règles que nous décrivons ci-dessous. Si l'inconsistance est propagée jusqu'à la racine du tableau, il n'est pas possible de construire une structure de Hintikka pour la formule p0.

Nous décrivons maintenant un algorithme de marquage, énoncé par Ben-Ari [BA82], qui permet de déterminer si le tableau sémantique construit peut être transformé en une structure de Hintikka pour la formule p0. L'algorithme est décrit en deux parties : premièrement, nous présentons un algorithme simple qui s'avèrera incomplet, et nous décrirons ensuite les modifications à apporter pour le compléter.

<u>Algorithme (provisoire) de marquage :</u>

L'algorithme de marquage répercute les inconsistances vers le haut, c.à.d. parcourt le tableau en sens inverse du sens de construction, en marquant au fur et à mesure les noeuds fermés :

Appliquer jusqu'à stabilisation les règles suivantes :

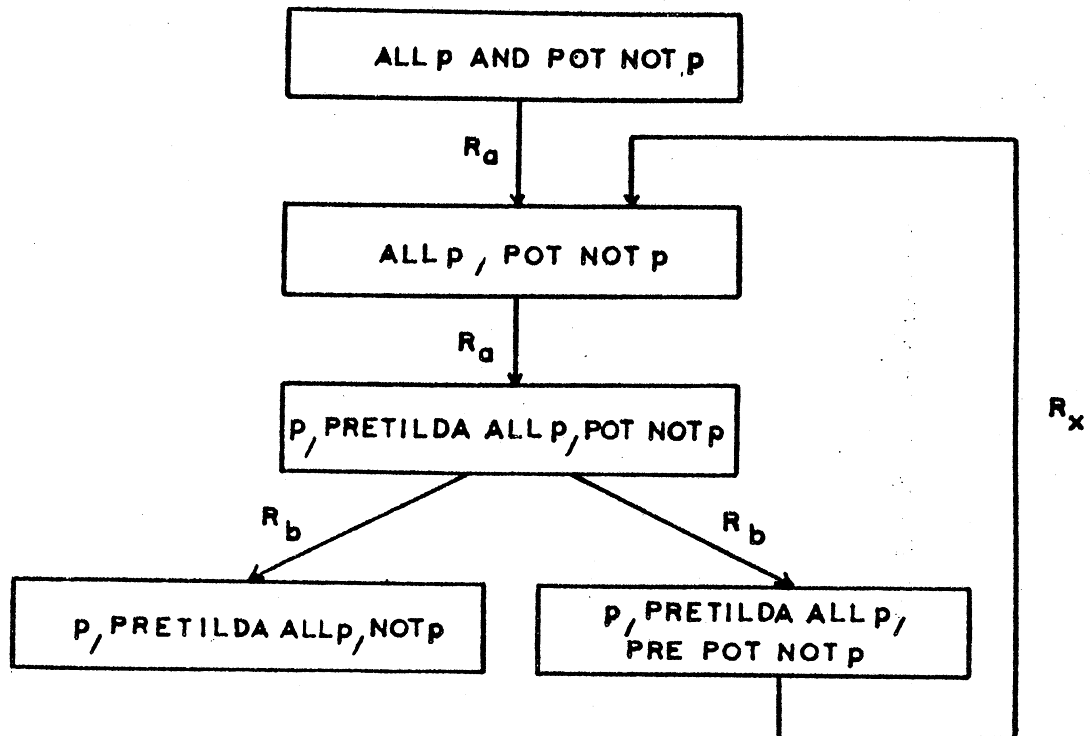
M1. marquer chaque noeud fermé
(noeud contenant à la fois p et NOT p) ;
M2. si n est un a-noeud et si son fils n1 est marqué,
alors marquer n ;
M3. si n est un b-noeud et si ses deux fils n1 et n2

sont marqués, alors marquer n ;
M4. si n est un x-noeud et si l'un quelconque de ses fils ni est marqué, alors marquer n.

Définissons une structure H(p0)=(S,P,R) de la façon suivante. Soit S, l'ensemble des états, formé des x-noeuds non marqués de T(p0). Soit R la relation construite par : s R t si il existe un chemin de s à t dont les noeuds intermédiaires sont tous des a-noeuds ou des b-noeuds. P est défini par : a appartient à P(s) si et seulement si a E s.

Il est facile de constater que la structure ainsi définie vérifie les propriétés H1, H2, H3 et H4.a-b (cf 3.2.3.1), mais pas obligatoirement H4.c-d. Prenons pour exemple le tableau construit pour la formule ALL[true]p AND POT[true] NOT p, que nous noterons plus simplement ALLp AND POT NOT p.

figure 3.1



Toute branche finie est fermée car la tentative d'affecter la valeur "faux" à p est inconsistante avec la formule ALLp. Il existe cependant une branche ouverte, qui correspond au fait de toujours renvoyer à plus tard la tâche de satisfaire la formule NOT p.

L'algorithme de marquage laisse donc en place des états s de H(p0) qui contiennent des formules de la forme INEV[p]q, PRETILDA

INEV[p]q, POT[p]q ou PRE POT[p]q (appelées formules futures), tels que les impératifs temporels exprimés par les formules ne sont pas vérifiés. Nous allons décrire un algorithme d'évaluation qui détecte les formules futures non satisfaites, et nous modifierons ensuite l'algorithme de marquage pour éliminer les états de H(p0) qui comportent de telles formules.

Algorithme d'évaluation des formules futures :

L'algorithme d'évaluation est appliqué au tableau T(p0). Il ne concerne que les noeuds non fermés du tableau.

Le principe de l'algorithme consiste, pour toute formule future p, à affecter à chaque x-noeud n de T(p0) qui la contient une valeur booléenne RK(p,n), qui indique si les contraintes temporelles de la formule sont vérifiées (RK(p,n)=1) ou non (RK(p,n)=0) par la structure H(p0) à partir de l'état s engendré par le x-noeud n.

L'affectation se fait de façon inductive, en partant des x-noeuds qui vérifient la formule argument de p, par parcours inverse des chemins de T(p0) qui mènent à ces noeuds. L'algorithme effectue ainsi une évaluation exhaustive des états s de la structure H(p0) qui satisfont une formule future p, et détecte ceux qui sont censés la vérifier et qui ne la vérifient pas.

L'algorithme d'évaluation est décrit par l'application itérée (jusqu'à stabilisation) des six règles suivantes :

Soient n un noeud de T(p0), n1, n2, ... ses fils éventuels ;
soit r E n une formule future qui n'a pas été affectée,

R1. Si n est un b-noeud pour une b-formule r de {INEV[p]q,POT[p]q}
    et si q E n1, alors RK(r,n)=1 ;

R2. Si n est un b-noeud pour une b-formule r de {INEV[p]q,POT[p]q}
    et si RK(1,n2)=1, alors RK(r,n)=1
    (1=PRETILDA INEV[p]q / PRE POT[p]q respectivement) ;

(R1 et R2 évaluent les deux composantes d'une formule disjonctive, et affectent la valeur 1 au noeud n pour r si l'une au moins des deux composantes possède la valeur 1.

R3. si n est un b-noeud, mais pas pour r,
    et si RK(r,ni)=1 pour un i, i=1,2 alors RK(r,n)=1 ;

(la formule disjonctive du noeud n n'est pas r, on effectue l'union sur les fils de n)

R4. si n est un a-noeud et si RK(r,n1)=1,
    alors RK(r,n)=1 ;

(affectation simple pour un a-noeud)

R5. si n est un x-noeud, r=PRETILDA INEV[p]q
et RK(INEV[p]q,ni)=1 pour tout ni, alors RK(r,n)=1 ;

(tous les fils d'un x-noeud doivent satisfaire les formules PRETILDA)

R6. si n est un x-noeud, r=PRE POT[p]q
et pour un certain ni,
POT[p]q E ni et RK(POT[p]q,ni)=1, alors RK(r,n)=1.

(un fils au moins doit satisfaire une formule PRE)

Puisque le nombre de formules futures dans T(p0) est fini, l'algorithme se termine.

Nous présentons maintenant la version corrigée de l'algorithme de marquage.

<u>Algorithme (définitif) de marquage :</u>

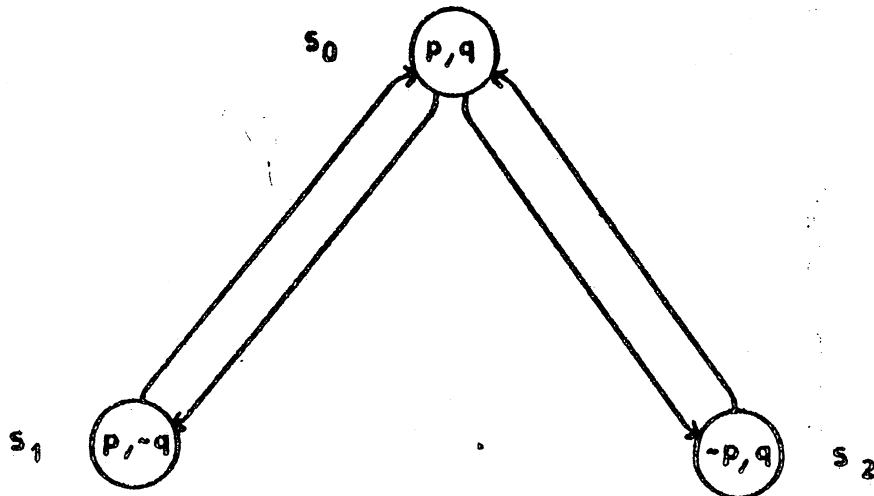
1) Appliquer les règles M1-M4 autant que possible ; les x-noeuds non marqués de T(p0) forment alors une structure H(p0) définie comme précédemment ;

2) M5. Appliquer l'algorithme d'évaluation à T(p0) ;
Marquer chaque noeud n contenant une formule future r telle que RK(r,n)<>1.

En fait, l'application de M5 peut modifier la structure H(p0). Il faut alors réappliquer les règles 1) et 2). Comme chaque étape marque au moins un noeud de H(p0), l'algorithme de marquage se termine obligatoirement.

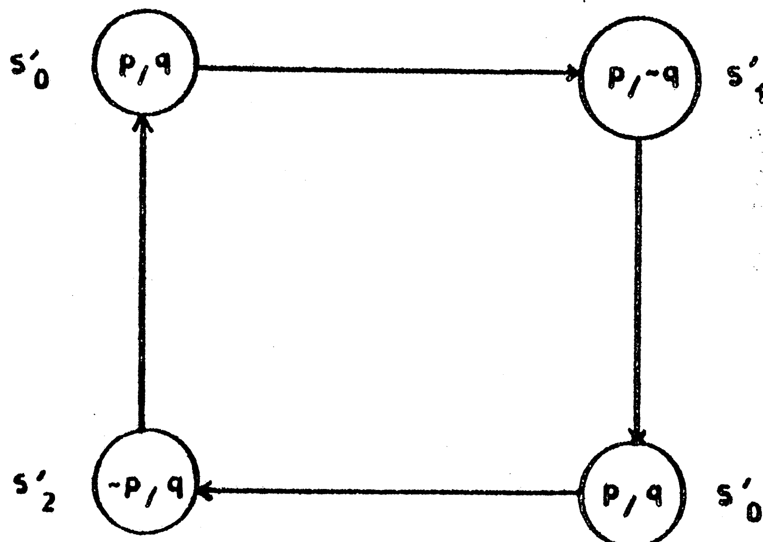
Plaçons nous dans le cas où la racine n0 de T(p0) n'est pas marquée. La structure H(p0) obtenue après application de l'algorithme de marquage n'est pas encore nécessairement une structure de Hintikka. Le problème peut être montré par la formule p0 = p AND q AND INEV(p AND NOTq) AND INEV(NOTp AND q). La structure H(p0) est la suivante :

figure 3.2



La branche s0,s1,s0,s1,... ne satisfait pas la condition INEV(NOTp AND q). La raison est que les alternatives offertes par les b-noeuds de T(p0) donnent parfois une trop grande liberté. On peut remédier à ce problème en déroulant la structure comme sur la figure suivante, c.à.d. en forçant chaque branche infinie à passer par les deux fils d'un b-noeud.

figure 3.3



Nous donnons une méthode de déroulement d'une structure, par construction d'un tableau T'(p0) dont les noeuds sont des instances des noeuds de T(p0). Les instances d'un noeud n de T(p0) seront notées

n´, n´´, ... Le terme noeud s´appliquera désormais uniquement aux noeuds non marqués. PI(n´i,n´j) représentera un chemin de n´i vers n´j dans T´(p0), ne contenant pas n´j, et un noeud alternatif sera un b-noeud dont les deux fils sont non marqués.

Algorithme de déroulement d´une structure :

n´0 est la racine de T´(p0) ;

si n´ est une feuille de T´(p0), alors prolonger T´(p0) de la façon suivante :

W1. si n est un noeud alternatif, alors pour chaque fils ni de n, on construit un fils n´i de n´ dans T´(p0) ;

W2. si n est un noeud alternatif, soit k le nombre d´instances de n dans PI(n´0,n´) ;

1. si k=1, alors n´1 (arbitrairement) est le fils de n´ dans T´(p0) ;
2. si k>1, alors si n´´1 (n´´2) était le fils de n´´ choisi pour la k-1 ème instance n´´ de n, alors n´2 (n´1) est le fils de n´ dans T´(p0) ;

On alterne ainsi les choix ;

W3. cependant, si n a une instance antérieure n´´ dans PI(n´´,n´) et si chaque noeud alternatif qui possède une instance dans PI(n´´,n´) possède au moins deux instances dans ce même chemin, on identifie n´ à n´´.

On constate que l´algorithme de déroulement se termine, puisqu´il y a un nombre fini de noeuds alternatifs.

Les résultats précédents permettent d´énoncer la propriété qui conclut notre étude :

la racine n0 de T(p0) est non marquée par l´algorithme de marquage si et seulement si il existe une structure de Hintikka (finie) pour la formule p0 de n0. En conséquence, par application du théorème 3.1, il existe une procédure de décision pour CTL, et CTL possède la propriété des modèles finis.

La démonstration de cette propriété est omise. Nous avons montré comment on peut construire une structure de Hintikka à partir d´un tableau sémantique dont la racine est non marquée. Inversement, la complétude de CTL permet de prouver que pour tout noeud marqué du tableau, la négation de la conjonction des formules étiquetant ce noeud est un théorème de la logique CTL.

Nous présentons désormais l´algorithme de décision de la logique

temporelle CTL.

2.3.3. ALGORITHME DE DECISION

Soit p une formule de la logique CTL. Pour prouver p, on va tenter de construire un modèle pour la formule NOT p.
S'il n'existe pas de modèle pour NOT p, la formule p est donc vraie pour tout état de tout modèle. P est alors un théorème de CTL.
S'il existe un modèle pour NOT p, la formule p n'est pas valide, et le modèle pour NOT p permet de comprendre pourquoi la formule p n'est pas valide.

Les différents résultats des précédents paragraphes sont regroupés pour former l'algorithme de décision suivant :

1. construire le tableau sémantique T(p) pour la formule NOT p ;

2. appliquer l'algorithme de marquage à T(p) pour obtenir une structure H(p) ;

3. si la racine de H(p) est marquée, p est un théorème de CTL, sinon il existe un modèle pour NOT p construit par l'algorithme de déroulement d'une structure, puis par transformation de cette structure en un modèle.

L'algorithme de décision présenté ci-dessus opère en un temps exponentiel avec la taille de la formule. Emerson [EH82] a prouvé l'équivalence des trois propositions suivantes :

(a) p est satisfaisable ;

(b) il existe une structure de Hintikka pour p de taille <= n.8**n, où n est la taille de la formule p ;

(c) il existe un modèle pour p de taille <= n.8**n ;

et affirme que la procédure de décision s'exécute en un temps borné par 2**c.n, où c est une constante dépendant principalement de l'implémentation de l'algorithme. Il n'est pas possible, conclut-il, d'élaborer des algorithmes d'ordre inférieur, et la méthode des tableaux sémantiques s'avère en pratique satisfaisante.

2.4. APPLICATION DE LA PROCEDURE DE DECISION

Il nous a semblé utile de munir notre système d'analyse d'un démonstrateur de théorèmes de la logique CTL. En effet, si la compréhension intuitive des formules peu complexes (POTp, INEVp , ALL[p] INEVq, ...) ne pose aucun problème, il n'est pas aisé de percevoir avec exactitude le sens d'une formule plus élaborée. La procédure de décision permet de prouver la validité de formules, et la comparaison des formules par implication ou par équivalence.
On peut ainsi examiner la correspondance entre deux formules par une approche progressive : on teste l'équivalence de deux formules f1 et f2, et si celle-ci est non valide, on étudie successivement les deux implications f1 => f2 et f2 => f1. Pour chaque implication non valide, le modèle produit par la procédure de décision permet de se faire une idée explicite de ce qui sépare f1 et f2.

Nous pensons que l'intégration d'un démonstrateur de théorèmes dans le système d'analyse QUASAR se traduit par l'apport d'un outil efficace d'aide à la spécification des systèmes, en permettant à l'utilisateur de mieux comprendre un langage dont le poids des mots n'est pas toujours perceptible.

La réalisation d'un programme de décision dans QUASAR a été effectuée par application stricte des algorithmes que nous avons décrits dans les précédents paragraphes. Néanmoins, quelques simplifications ont été introduites ; les a-noeuds ne sont pas représentés, car la substitution des a-formules s'effectue directement, sans création d'un nouveau noeud ; les formules de propagation ne sont pas représentées explicitement, mais par un marquage particulier des formules dont elles proviennent. Les temps de calcul sont jugés très corrects, dans la mesure où les formules soumises au programme sont de taille raisonnable.

Nous présentons les différents résultats intermédiaires produits par le programme pour la formule

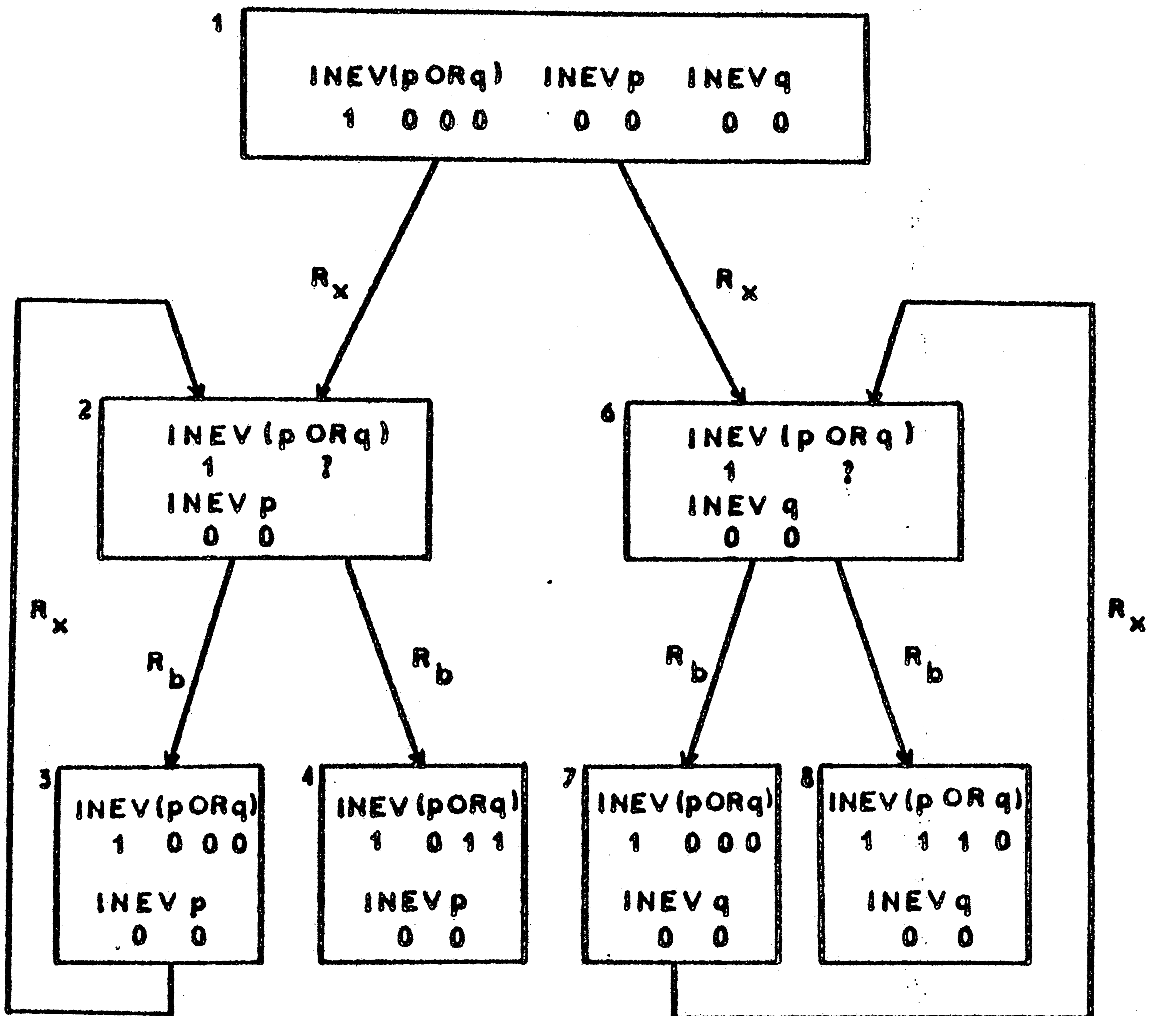
f = INEV (p OR q) => INEV p OR INEV q.

La formule choisie est de petite taille, mais souligne parfaitement les différentes étapes de la procédure de décision.

Construction du tableau sémantique T(f) :

Le tableau généré par le programme est celui représenté sur la figure ci-dessous :

figure 3.4



Le tableau sémantique est construit pour la négation de la formule f. Il est commode de représenter les formules en leur associant une valeur (1 pour vrai, 0 pour faux), et de représenter ainsi une formule NOT g par la formule g ayant pour valeur 0.

Le noeud 1 contient la formule f de valeur 0, c.à.d. la conjonction des formules INEV(p OR q) de valeur 1, INEVp de valeur 0 et INEVq de valeur 0. Les valeurs de p et q sont obtenues par applications de la règle Ra.

Les noeuds 2 et 6 sont construits par application de la règle Rx au noeud 1. Ils contiennent tous deux la formule "obligatoire" INEV(p OR q) qui n'a pas encore été satisfaite, et chacun d'eux une formule INEVp ou INEV q de valeur 0.

Les noeuds 3 et 4 sont produits par application de la règle Rb au

noeud 2, pour envisager les deux valeurs possibles de la formule p OR q du noeud 2. Il en est de même pour les noeuds 7 et 8 par rapport au noeud 6.

Les noeuds 4 et 8 sont des feuilles, car ils ne contiennent aucune formule de propagation. L'application de la règle Rx aux noeuds 3 et 7 génère deux boucles dans le graphe. Les x-noeuds du tableau T(f) sont finalement les noeuds 1, 3, 4, 7 et 8.

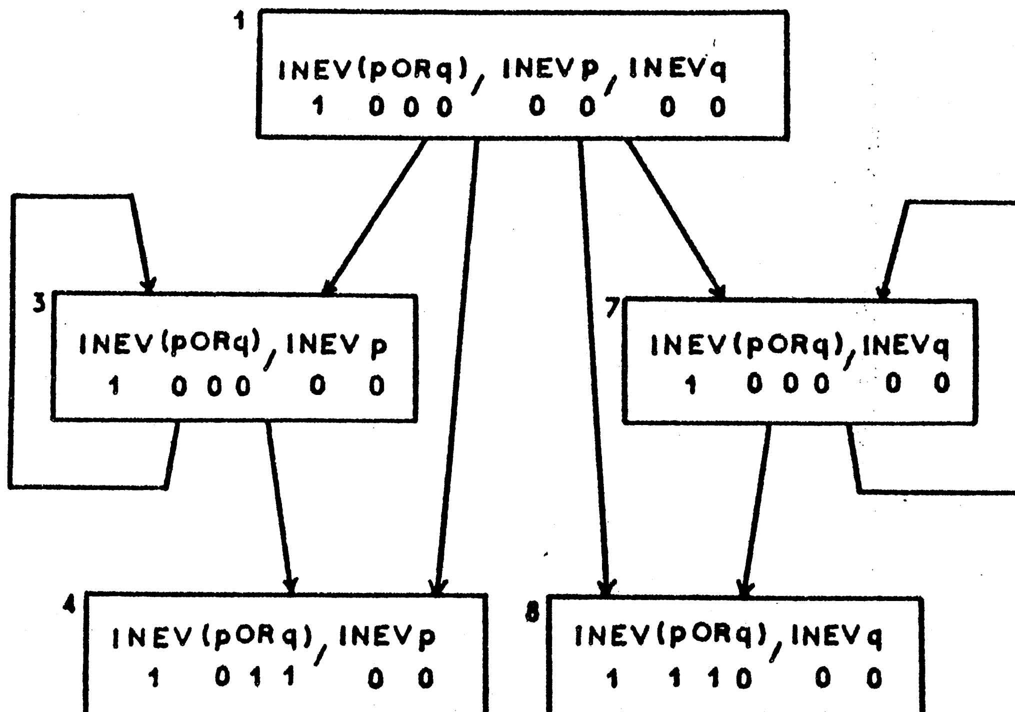
Application de l'algorithme de marquage :

Les étapes M1-M4 de l'algorithme de marquage n'ont marqué aucun noeud du graphe. Le programme applique donc l'algorithme d'évaluation qui ne modifie pas le tableau. La formule n'est donc pas un théorème de la logique CTL. Nous exhibons alors un modèle pour la formule NOT f en deux étapes :

1) Construction de la structure H(f)

La structure H(f) construite à partir du tableau T(f) est formée des x-noeuds non marqués, à savoir les noeuds 1, 3, 4, 7 et 8. Elle est représentée par la figure suivante :

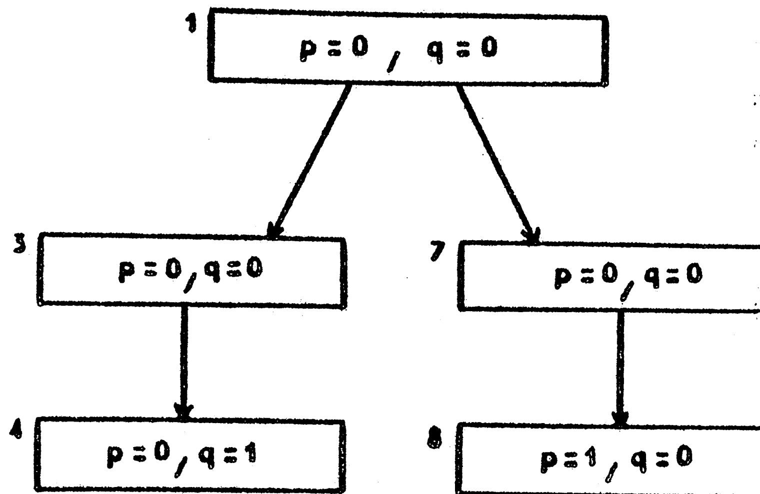
figure 3.5



La structure ainsi construite n'est pas encore une structure de Hintikka pour la formule NOT f. L'application de l'algorithme de déroulement va produire le modèle suivant :

2) Construction d'un modèle

figure 3.6



Le modèle exhibé nous montre de façon explicite pourquoi la formule f n'est pas un théorème de CTL, par l'intermédiaire d'un modèle pour NOT f.

La formule INEV(p OR q) est satisfaite par l'état 1 car les branches (1,3,4) et (1,7,8) aboutissent respectivement aux états 4 et 8 qui satisfont tous deux p OR q. Par contre, la branche (1,3,4) ne comporte aucun état satisfaisant p ; la formule INEVp n'est donc pas satisfaite par l'état 1. Symétriquement, l'état 1 ne satisfait pas la formule INEVq car la branche (1,7,8) ne comporte aucun état satisfaisant q.

L'état 1 satisfait ainsi la formule INEV(p OR q) AND NOT INEVp AND NOT INEVq. La négation de cette formule, à savoir la formule f = INEV(p OR q) => INEVp OR INEVq n'est pas un théorème de la logique temporelle CTL.

Remarquons que l'algorithme de décision produit un modèle particulier, parmi l'ensemble des modèles possibles pour NOT f. Ce modèle n'est pas obligatoirement le plus petit, comme le montre notre exemple. Les noeuds 3 et 7 peuvent être détruits, car l'ensemble des trois noeuds 1, 4 et 8 forme à lui seul un modèle pour NOT f.

3. SPECIFICATION DE PROPRIETES DANS QUASAR

Nous présentons dans ce paragraphe de quelle façon est utilisée la logique temporelle CTL que nous venons de décrire, au sein du système QUASAR, pour spécifier une application répartie.

La logique temporelle CTL permet de construire des formules à partir d'un ensemble de variables propositionnelles. Pour que ces formules expriment des propriétés comportementales d'une application, il est nécessaire d'effectuer une liaison entre la description de l'application et les formules. Cette liaison est faite par l'utilisation dans les formules de variables propositionnelles en relation avec le fonctionnement de l'application étudiée :

- des prédicats sur les variables de l'application ;
- des variables propositionnelles exprimant le positionnement d'une tâche par rapport à une action.

Les formules temporelles permettent de caractériser le comportement dans le temps de tels prédicats, et d'exprimer ainsi des propriétés qui traduisent des fonctionnements (possibles ou souhaités) d'une application.

La notion de satisfaisabilité d'une formule, étudiée au paragraphe précédent, ne peut pas être exploitée directement pour la spécification des systèmes. Il nous importe généralement peu de savoir si une formule de spécification est un théorème de la logique CTL ou non. Ce qui est intéressant, par contre, est de pouvoir déterminer si une formule de spécification est vérifiée ou non par une application donnée. Pour cela nous sommes amenés à définir une interprétation des formules de la logique sous forme de l'ensemble des états de l'application qui satisfont la formule. Si cet ensemble comporte tous les états de l'application, la formule est satisfaite à tout moment par l'application ; la formule est alors valide pour l'application étudiée. S'il existe un état qui ne satisfait pas la formule, la formule est dite non valide pour l'application.

Pour présenter le langage de spécification de QUASAR, nous définissons pour commencer les règles d'interprétation des formules. Nous présentons ensuite la syntaxe exacte du langage, puis sa sémantique par la définition des interprétations des éléments du vocabulaire du langage. Nous étudions ensuite l'ensemble des propriétés que permettent d'exprimer les formules du langage, et nous présentons le problème de l'équité soulevé par le non déterminisme présent dans les systèmes décrits dans QUASAR ; nous proposons une solution formelle à ce problème.

Nous spécifions finalement les six exemples que nous avons présentés au chapitre précédent. Pour chaque application, nous donnons un ensemble de propriétés qui doivent être vérifiées par la

description que nous avons faite de l'application.

3.1. INTERPRETATION DES FORMULES DE LA LOGIQUE

Nous définissons dans ce paragraphe l'interprétation des formules temporelles, à l'aide des notions présentées au paragraphe III.2 (modèle, satisfaction d'une formule par un état).

Un système conditions-actions (produit par la traduction d'un programme de description) peut être considéré comme un modèle (S,P,R) pour la logique temporelle CTL (cf III.2.2), construit de la façon suivante :

- un état s du modèle est une valeur du vecteur X des variables du CA-système ;
- la relation d'accessibilité R du modèle est définie par : s1 R s2 ssi il existe une commande gardée du CA-système, validée pour l'état s1, et dont l'exécution mène à l'état s2 ;
- S est l'ensemble de tous les états atteignables à partir de l'initialisation des variables du système conditions-actions ;
- P est une affectation d'un ensemble de variables propositionnelles pour chaque état de S (ces variables sont en relation avec la description d'une application, et seront définies ultérieurement).

Le modèle étant constitué d'un ensemble d'états et d'un ensemble de transitions entre les états, nous utiliserons souvent par la suite le terme "système de transitions" pour désigner le modèle. Les s-branches $b = (s=s_0, s_1, \ldots)$ du modèle représentent des suites d'exécutions de transitions, et seront dénommées séquences d'exécutions à partir de l'état s. On notera EXs l'ensemble des séquences d'exécutions possibles à partir d'un état s de S.

Etant donné un système de transitions T=(S,P,R) issu de la traduction d'un programme de description, on définit l'interprétation |f| d'une formule temporelle f de CTL comme étant l'ensemble des états du système qui satisfont la formule :

$$|f| = \{ s \text{ de } S \mid T,s \models f \}$$

L'ensemble d'états |f| est aussi appelé ensemble caractéristique de la formule f pour le système T.

Nous donnons l'interprétation des éléments du vocabulaire des formules de CTL, qui découle immédiatement de la satisfaction d'une formule par un état :

- pour toute variable propositionnelle f,

```
|f| = { s de S | f appartient à P(s) } ;
```

- pour tout couple de formules (f1,f2) :

```
|NOT f1| = S - |f1|
|f1 OR f2| = |f1| u |f2|
|f1 AND f2| = |f1| n |f2|
```

- pour tout couple de formules (f1,f2) :

```
|ALL[f1]f2| = { s de S |
   pour toute séquence b de EXs, pour tout k entier,
   ( b = (s=s_0,s_1,...,s_k) et pour tout i de 0 à k-1,
     s_i appartient à |f1| => s_k appartient à |f2| ) }

|SOME[f1]f2| = { s de S |
   il existe une séquence b de EXs telle que,
   pour tout k entier,
   ( b = (s=s_0,s_1,...,s_k) et pour tout i de 0 à k-1,
     s_i appartient à |f1| => s_k appartient à |f2| ) }
```

3.2. SYNTAXE DU LANGAGE DE SPECIFICATION

Le vocabulaire du langage de spécification comporte un certain nombre d'opérateurs qui ne figurent pas dans la logique CTL (opérateurs _[_], ALW, WPOT, SONT, OBL, FAIR et la possibilité d'utiliser tous les opérateurs temporels avec un seul argument). Ces opérateurs sont des cas particuliers des opérateurs de CTL ; il sont introduits car ils correspondent à des types précis de propriétés. Leur signification sera étudiée ultérieurement.

Les formules de spécification reconnues par le système QUASAR sont définies par la grammaire suivante, dont l'axiome est <formule>. Les notations utilisées sont celles du chapitre II pour le langage de description.

```
<formule> ::= <terme> <> <terme>  |
              <terme> => <terme>  |
              <terme> [ AND <terme> ]*  |
              <terme> [ OR <terme> ]*  |
              <terme>

<terme> ::= [ <opérateur> ]* <prédicat>

<opérateur> ::= NOT  |
                _[_]  |
                <opérateur temporel> [ _[ <formule> _] ]
```

<opérateur temporel> ::= ALL |
POT |
SOME |
INEV |
OBL |
SONT |
ALW |
WPOT |
FAIR

<prédicat> ::= <prédicat sur les variables> |
<prédicat prédéfini> |
(<formule>)

<prédicat prédéfini> ::= INIT |
ENABLE |
ENABLE <liste d'actions> |
ENABLE <liste de tâches> |

AFTER <liste d'actions> |
SINK

<liste d'actions> ::= (<nom d'étiquette du programme >
[, <nom d'étiquette du programme >]*)

<liste de tâches> ::= (<nom de tâche du programme>
[, <nom de tâche du programme>]*)

<prédicat sur les variables> ::= <identificateur>

Les noms d'étiquettes ou de tâches du programme sont des identificateurs qui désignent respectivement des étiquettes ou des tâches du programme de description.

3.3. SEMANTIQUE DU LANGAGE DE SPECIFICATION

Nous présentons dans ce paragraphe la sémantique du langage de spécification de QUASAR, en définissant l'interprétation des variables propositionnelles qui sont utilisées dans les formules.

L'interprétation des opérateurs temporels a déjà été étudiée ; nous ne la rappelons pas, mais nous définissons un certain nombre d'opérateurs qui sont des cas particuliers des opérateurs de la logique CTL, et qui correspondent à des classes de propriétés précises.

3.3.1. LES VARIABLES PROPOSITIONNELLES

Les variables propositionnelles figurant dans les formules de spécification sont en relation avec la description d'une application, et peuvent être de deux types :

- des prédicats sur les variables du programme de description ;
- des variables propositionnelles associées aux actions étiquetées du programme de description.

Nous définisons l'interprétation de chacun des deux types de variables propositionnelles.

3.3.1.1. PREDICATS SUR LES VARIABLES DE L'APPLICATION

L'utilisateur peut employer comme variable propositionnelle tout prédicat exprimé par une condition (selon la syntaxe du langage de description) sur les variables du programme de description. Chaque prédicat de ce type est une condition booléenne dont la valeur est déterminée pour chaque état du système de transitions par la valeur des variables de l'application pour cet état.

L'interprétation d'un prédicat sur les variables de l'application est l'ensemble des états du système de transitions pour lesquels la condition exprimée par le prédicat est vraie.

Trois variables propositionnelles prédéfinies INIT, ENABLE et SINK peuvent par ailleurs être utilisées dans les formules de spécification. Elles caractérisent respectivement les états initiaux possibles du système, la possibilité d'évolution du système et son blocage définitif :

- l'interprétation de INIT est l'ensemble des états initiaux possibles définis par la valeur initiale de chaque variable initialisée dans le programme de description, et par une valeur quelconque dans leurs intervalles de définition des variables non initialisées ;
- l'interprétation de ENABLE est l'ensemble des états s de S tel qu'il existe un état s' de S tel que s R s', c.à.d. l'ensemble des états à partir desquels le système peut évoluer ;
- l'interprétation de SINK est définie par $|SINK| = S - |ENABLE|$, et contient les états puits du système, c.à.d. les états à partir desquels le système est définitivement bloqué.

3.3.1.2. VARIABLES PROPOSITIONNELLES ASSOCIEES AUX ACTIONS

Pour chaque action étiquetée définie dans le programme de description, le système QUASAR reconnaît des variables propositionnelles :

- caractérisant la possibilité d'exécuter cette action ;
- caractérisant la position relative (immédiatement après) du contrôle d'une tâche par rapport à cette action.

3.3.1.2.1. EXECUTABILITE D'UNE ACTION

Pour chaque action a étiquetée dans le programme de description par un nom ea, le système QUASAR reconnaît la variable propositionnelle :

ENABLE (ea)

signifiant que l'action a est exécutable.

Une étiquette ea associée à une action a du programme de description peut étiqueter, à l'issue de l'opération de composition des tâches de l'application, plusieurs commandes gardées du système conditions-actions composé. L'action a est réputée exécutable lorsqu'une transition quelconque étiquetée par ea est exécutable, c.à.d. pour tout état qui vérifie la condition associée à une de ces transitions.

L'interprétation de la variable ENABLE (ea) est donc l'ensemble des états du système de transitions qui valident au moins une transition étiquetée par ea.

Par extension, la variable propositionnelle :

ENABLE (ea1, ea2, ..., ean)

est définie comme l'union des variables propositionnelles ENABLE (eai). Elle caractérise le fait que l'une au moins des actions ai est exécutable.

Par ailleurs, pour toute tâche Ta de l'application, le système QUASAR reconnaît la variable propositionnelle :

ENABLE (Ta)

signifiant que la tâche Ta peut encore évoluer.

Si Ta est une tâche élémentaire, ENABLE (Ta) est vraie lorsque l'une quelconque des commandes gardées de la tâche Ta est exécutable. ENABLE (Ta) prend donc en compte toutes les actions de Ta, qu'elles soient étiquetées ou non.

Si Ta est une tâche composée, ENABLE (Ta) est définie comme l´union des variables propositionnelles ENABLE (Tai) pour toutes les tâches Tai composant Ta.

On définit de même la variable propositionnelle :
ENABLE (Ta1, Ta2, ..., Tan)
comme l´union des variables propositionnelles ENABLE (Tai). Elle caractérise le fait que l´une des tâches Tai au moins peut évoluer.

3.3.1.2.2. CONFIRMATION DE L´EXECUTION D´UNE ACTION

Pout toute action a étiquetée par un nom ea, le système QUASAR reconnaît la variable propositionnelle :
AFTER (ea)
caractérisant le fait que le contrôle de la tâche Ta qui contient l´action a se situe immédiatement après l´exécution de a. AFTER (ea) est vraie lorsque Ta vient d´exécuter l´action a, et reste vraie tant que cette tâche n´a pas exécuté une action autre que a. AFTER (ea) reste ainsi valide tant que le système évolue sans exécuter une action de la tâche Ta, à moins que l´action exécutée par Ta ne soit a elle-même.

L´interprétation de la variable propositionnelle AFTER (ea) est donc l´ensemble des états du système de transitions pour lesquels le contrôle de la tâche contenant l´action a étiquetée par ea, se situe immédiatement après l´exécution d´une transition quelconque étiquetée par ea.

On définit par extension la variable propositionnelle :
AFTER (ea1, ea2, ..., ean)
comme l´union des variables propositionnelles AFTER (eai). Elle caractérise l´ensemble des états qui vérifient au moins l´une des variables propositionnelles AFTER (eai).

3.3.2. LES OPERATEURS TEMPORELS

Nous définissons les opérateurs temporels [], ALW, WPOT, SONT et OBL de la façon suivante :

|[]f| = |INIT => ALL f|

L´opérateur [] est utilisé très souvent dans les formules de spécification, car la majeure partie des propriétés s´écrivent sous cette forme, pour exprimer qu´un prédicat ou une formule demeure valide si le système est initialisé correctement (conformément à la

description).

|ALW[f1]f2| = |ALL[f1](f2 AND ENABLE)|
|SONT[f1]f2| = |SOME[f1](f2 AND ENABLE)|

Les opérateurs ALW et SONT sont des restrictions des opérateurs ALL et SOME, qui ne prennent en compte que les séquences d'exécutions infinies. Ils sont généralement utilisés pour étudier les comportements cycliques des systèmes.

|WPOT[f1]f2| = |POT[f1](f2 OR SINK)|
|OBL[f1]f2| = |INEV[f1](f2 OR SINK)|

Les opérateurs WPOT et OBL sont utilisés pour exprimer des notions d'atteignabilité, potentielle ou obligatoire, qui prennent en compte le blocage éventuel du système.

Les opérateurs temporels (autres que []) définis jusqu'à présent sont tous des opérateurs à deux arguments (opérateurs binaires) de la forme OP[f1]f2. On peut définir pour chacun d'eux un opérateur à un argument (opérateur unaire) OPf2, tel que |OPf2| = |OP[true]f2|. L'opérateur unaire est un cas particulier de OP[f1]f2, obtenu en prenant f2 = true.

Les opérateurs binaires sont souvent dénommés opérateurs conditionnels, car ils expriment des notions d'invariance, de trajectoire ou d'atteignabilité (potentielle ou inévitable) d'une formule f2 sous la condition f1. Les opérateurs unaires sont obtenus en forçant la condition à la valeur true, et sont appelés opérateurs non conditionnels. Ils permettent d'exprimer une nombre très grand de propriétés des applications.

3.4. EXPRESSION DE PROPRIETES

Nous présentons dans cette partie l'utilisation des opérateurs temporels pour exprimer des propriétés comportementales des applications. Nous n'établissons pas une classification rigide des propriétés ; nous décrivons pour chaque opérateur (et pour son opérateur dual) un ensemble de propriétés qu'il permet d'exprimer. L'énumération des propriétés n'est pas obligatoirement exhaustive, mais recouvre une large majorité des propriétés usuellement employées pour la spécification des systèmes. Nous commençons par les opérateurs unaires.

Utilisations des opérateurs POT et ALL

a) Expression de propriétés invariantes

Ce sont des propriétés qui doivent être vérifiées en permanence dans le système décrit. Elles peuvent être exprimées par des formules de la forme :

INIT => ALL f

En fait, toutes les propriétés sont de cette forme, avec une formule f plus ou moins compliquée ; c'est la raison pour laquelle la notation [] a été introduite :

[] 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 :

[] (NOMBRE_UTILISATEURS in [0..N])

exprimant que le nombre d'utilisateurs utilisant une ressource est borné par N.

* propriété localement invariante

[] (p => f)

où p est un prédicat sur les variables propositionnelles associées aux actions étiquetées de la description, ou bien un prédicat sur des variables jouant le rôle de compteurs ordinaux (ce qui permet d'exprimer une position du contrôle du système décrit) et f un prédicat sur les variables du système.

Exemples :

[] (IN_ECRITURE => NB_LECTEURS = 0)

exprimant le fait qu'une écriture (caractérisée par le prédicat IN_ECRITURE sur les variables compteurs ordinaux) ne peut avoir lieu que lorsque les lecteurs n'utilisent pas la ressource.

[] (AFTER(ECRITURE) => NB_ELEMENTS >= 1)

exprimant le fait qu'après une écriture, un buffer ne peut pas être vide.

* exclusion dirigée

[] (p1 => NOT p2)

où p1 et p2 sont des prédicats exprimant des positions.

Exemple :

[] (RESS_UTIL_1 => NOT RESS_UTIL_2)
exprimant le fait que si le processus P1 utilise une ressource, alors P2 ne peut pas l'utiliser en même temps.

* exclusion mutuelle

[] NOT (p1 AND p2)
où p1 et p2 sont des prédicats exprimant des positions.

b) Vivacité

On peut exprimer qu'une action (représentée par une commande gardée) est vivante (au sens usuellement employé pour les transitions des réseaux de Petri [SIF79]) par une formule de la forme suivante :

INIT => ALL POT ENABLE(e)
où e est une étiquette de cette action. Cette formule signifie que, quelle que soit l'évolution ultérieure du système décrit. on pourra toujours rendre l'action considérée 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 considérée ne sera jamais plus exécutable. Un tel invariant est geréalement appelé verrou (en anglais "deadlock") pour l'action considérée.

On peut étudier de la même façon la vivacité d'un rendez-vous en remplaçant ENABLE(e) par ENABLE(e1, e2, ..., en) où e1, e2, ..., en sont des étiquettes de toutes les commandes gardées participant à ce rendez-vous.

Le fait que le système n'est jamais totalement bloqué s'exprime par
INIT => NOT POT SINK
ce qui est équivalent à
INIT => ALL ENABLE
c.à.d. [] ENABLE

Utilisations des opérateurs INEV et SOME

On peut obtenir des variantes intéressantes des propriétés données précédemment en remplacant l'opérateur POT par l'opérateur INEV, ou l'opérateur ALL par l'opérateur SOME. De plus, les deux types de propriétés suivants s'expriment grâce à l'opérateur INEV.

a) Atteignabilité inévitable

Ce sont des propriétés qui expriment qu'un certain prédicat doit être atteint. Elles se traduisent par une formule de la forme :
INIT => INEV f

Ainsi, la correction d'un programme s'exprime par le fait
- que le programme se termine ;
- qu'un certain résultat (caractérisé par un prédicat f) est alors atteint.

Cela s'exprime par la formule :
INIT => INEV (END AND f)
où END est un prédicat sur les variables jouant le rôle des compteurs ordinaux, qui caractérise la terminaison du système décrit.

Exemple :

La formule
f = (T1<=T2) AND (T2<=T3) ... AND (Tn-1<=Tn)
permet d'exprimer la correction d'un programme de tri sur un tableau T de longueur n (représenté par les variables T1 à Tn).

b) Réponse à une action

Il s'agit de spécifier les situations où une action a1 en provoque une autre a2. Elles s'expriment par des formules de la forme :
[] (p1 => INEV p2)
où p1 et p2 sont des prédicats indiquant une position par rapport à a1 et a2.

Exemple :

[] (AFTER(SEND) => INEV ENABLE(RECEIVE))
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 INEV.

Utilisations des opérateurs OBL et SONT

On peut obtenir certaines variantes intéressantes des propriétés données précédemment, en remplacant les opérateurs POT ou INEV par l'opérateur OBL, ou les opérateurs ALL et SOME par l'opérateur SONT. De plus, la propriété suivante s'exprime au moyen de l'opérateur OBL.

Absence de famine

Nous dirons qu'il y a famine pour une action, si le système décrit peut toujours évoluer sans jamais valider (c.à.d. rendre exécutable) cette action. Ceci s'exprime par le fait qu'il existe une trajectoire sans fin dans NOT ENABLE(e) (e étant une étiquette de cette action), atteignable depuis l'état initial. La propriété exprimant qu'il n'y a pas possiblité de famine pour l'action est donc :

INIT => NOT POT SONT NOT ENABLE(e)

c.à.d. par dualisation :

INIT => ALL OBL ENABLE(e)

ou [] OBL ENABLE(e)

Exemple :

[] OBL ENABLE(i_EAT)

où l'action i_EAT caractérise le fait que le philosophe numéro i est en train de manger.

Utilisations des opérateurs WPOT et ALW

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 globalemant 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)

Utilisations 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 AND c) => ALL[c] f

où c est la condition exprimant le fonctionnement normal et f est un prédicat quelconque.

Exemples :

Avec c = (NB_REQUETES_EN_ATTENTE < 1000)

et f = ENABLE(ALLOCATEUR)

cette formule exprime le fait qu'un allocateur de ressources ne se

bloque pas (prédicat ENABLE(ALLOCATEUR)) tant qu'il n'est pas saturé.

Ce type de formule peut être généralisé pour exprimer une propriété conditionnelle quelconque, c.à.d. une propriété exprimée par une formule temporelle qui est vraie tant qu'une certaine condition est vérifiée.

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 à une action peuvent être exprimées en utilisant les opérateurs conditionnels.

Supposons qu'une action a1 admet pour réponse l'action a2. Nous pouvons exprimer la situation suivante : l'action a1, après avoir été exécutée une fois, ne peut être répétée tant que la réponse a2 n'a pas été effectuée. Cela s'exprime par une formule de la forme :

[] AFTER(e1) => ALL[NOT AFTER(e2)] NOT ENABLE(e1)

où e1 et e2 sont des étiquettes des actions a1 et a2.

Remarquons que si l'exécution de l'action a2 valide immédiatement l'action a1, il faut utiliser une formule plus précise :

[] AFTER(e1) => ALL[NOT AFTER(e2)]
(NOT ENABLE(e1) OR AFTER(e2))

c.à.d.

[] AFTER(e1) => ALL[NOT AFTER(e2)] ENABLE(e1) => AFTER(e2)

Ces propriétés ne garantissent pas que la réponse soit effectivement exécutée, mais dans ce cas, a1 ne sera jamais plus exécutée. Si l'on désire avoir la propriété précédente, tout en imposant que la réponse a2 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) exprime 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.

Les différents opérateurs temporels que nous venons d'utiliser permettent de formuler un grand nombre de propriétés comportemen-

tales d'un système. Nous avons donné une interprétation de ces opérateurs sous forme de conditions sur des séquences d'exécutions, en tenant compte uniquement du non déterminisme entre actions. Il s'avère cependant que le non déterminisme, s'il permet effectivement de représenter un ensemble d'actions possibles à un instant précis, ne donne pas toujours une image fidèle du fonctionnement souhaité du système décrit, comme nous allons le montrer ci-dessous.

Soit s un état d'un système et (a1 , a2) un couple d'actions exécutables à l'état s. Du point de vue description, le non déterminisme entre a1 et a2 représente correctement les deux évolutions possibles du système à partir de l'état s. Celui-ci autorise cependant qu'une séquence d'exécution qui passe une infinité de fois par l'état s, n'exécute jamais l'une des deux actions a1 ou a2. Ce comportement est non équitable, puisqu'il privilégie une des deux actions par rapport à l'autre, et peut entrainer un fonctionnement qui ne répond pas aux impératifs du système.

Nous allons expliciter le problème de l'équité, et nous montrerons comment nous caratériserons les fonctionnements équitables d'un système.

3.5. LE PROBLEME DE L'EQUITE

Nous allons illustrer le problème à l'aide de deux exemples de systèmes répartis.

a) Considérons une ligne (entre un émetteur et un récepteur) qui peut perdre des messages. Cette ligne peut être représentée par une tâche dont le corps est le suivant :

```
do
  {get}      CO=1 : MES:=?M, CO:=2  |
  {transmit} CO=2 : !MM:=MES, CO:=1  |
  {lose}     CO=2 : CO:=1
od ;
```

Chaque fois qu'un message est reçu par la ligne, celle-ci peut transmettre le message ou le perdre. Le choix étant non déterministe, il est possible que la ligne perde tous les messages qu'elle reçoit, et donc qu'aucun ne soit transmis au récepteur. Un tel comportement est non équitable, car il favorise l'action de perte de message ({lose}). En fait, tout comportement dans lequel un nombre infini de messages est reçu par la ligne, et un nombre fini seulement est transmis, est non équitable, car il existe un instant à partir duquel la ligne ne transmet plus rien.

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 messages.

Il apparait donc que certains comportements non équitables ne présentent pas d'inconvénients, mais que d'autres portent préjudice à la finalité même du système, qui consiste à transmettre des messages en un temps fini.

b) Considérons le système formé des trois processus P1, P2 et CONTROLER, où P1 et P2 sont deux utilisateurs d'une ressource en exclusion mutuelle, et CONTROLER un processus gestionnaire de la ressource ayant le fonctionnement d'un sémaphore. Les processus peuvent être représentés par des tâches dont les corps sont :

```
P1 : do
        CO1=1 : !P1, CO1:=2  |
        CO1=2 : !V1, CO1:=1
     od ;

P2 : do
        CO2=1 : !P2, CO2:=2  |
        CO2=2 : !V2, CO2:=1
     od ;

CONTROLER :
     do
        SEMAPHORE=1 : ?CP, SEMAPHORE:=SEMAPHORE+1  |
        TRUE        : ?CV, SEMAPHORE:=SEMAPHORE-1
     od ;
PORT P, V ;
BODY
   P1 ( P , V ) //
   P2 ( P , V ) //
   CONTROLER ( P , V )
```

Le gestionnaire effectue à chaque échange de la variable P un choix non déterministe entre le rendez-vous avec P1 ou celui avec P2, choix qui peut toujours être résolu en faveur de P1. P2 peut donc ne jamais entrer en section critique, alors qu'il est 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 qu'un nombre fini de fois à la ressource.

Diverses interprétations de la notion d'équité ont déjà été formulées dans la littérature informatique. Nous utilisons celle définie par Queille et Sifakis [QS82], selon laquelle l'accessibilité équitable est définie par rapport à un ensemble d'états d'un système.

Définition :

Une séquence d'exécution infinie b = (s_0, s_1, ..., s_k,...) est non équitable si et seulement si il existe un état s_i figurant une infinité de fois dans b tel que :
il existe un état s', successeur direct de s_i (s_i R s') tel que la sous-séquence s_i,s' ne figure pas un nombre infini de fois dans b.

On notera EXFs l'ensemble des séquences équitables à partir de s. Pour un système de transitions (S,P,R) ayant un nombre fini d'états, pour tout état s de S, EXFs n'est jamais vide. La démonstration de cette propriété figure dans [QS82].

L'opérateur d'atteignabilité équitable FAIR

Tous les opérateurs temporels que nous avons introduits jusqu'ici sont définis en considérant l'ensemble EXs de toutes les séquences d'exécution à partir de l'état s, qu'elles soient équitables ou non. Ils ne prennent donc pas en compte l'éventuelle équité du système décrit. Nous sommes donc amenés à introduire un opérateur défini en considérant uniquement l'ensemble EXFs des séquences équitables à partir de s.

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| = { s de S |
pour toute séquence d'exécution b de EXFs,
il existe un k entier tel que :
b = (s=s_0, s_1, ..., s_k) et s_k appartient à |f| }.

L'opérateur FAIR est donc équivalent à l'opérateur INEV, mais on ne considère pour l'évaluer que les comportements équitables du système décrit.

On définit également l'opérateur conditionnel FAIR de la façon suivante :

|FAIR[f1]f2| = { s de S |
pour toute séquence d'exécution b de EXFs,
il existe un k entier tel que :
b = (s_0=s, s_1, ..., s_k) et s_k appartient à |f2|
et pour tout i de 0 à k-1 :
s_i appartient à |f1| }.

Caractérisation de l'opérateur FAIR

Il n'est pas possible d'exprimer l'opérateur unaire FAIR à l'aide des autres opérateurs temporels unaires (et des opérateurs booléens). On peut cependant en donner une approximation par les deux formules suivantes :

pour tout f,

(INEV f OR ALL POT f) => FAIR f
et FAIR f => (INEV f OR SOME POT f).

La caractérisation exacte de l'opérateur FAIR est la suivante : pour tout f,f1,f2

FAIR f = ALL[NOT f] POT f
FAIR [f1] f2 = ALL [NOT f2] POT [f1] f2

(ces différentes propriétés ont été démontrées dans [QUE82]).

3.6. EXEMPLES DE SPECIFICATION

Nous traitons dans ce paragraphe les six exemples que nous avons présentés dans le chapitre précédent. Nous exprimons diverses propriétés qui doivent être vérifiées par les systèmes ; l'énumération des propriétés ne sera pas exhaustive, et pourra comporter des redondances. La recherche d'un ensemble minimal et complet de propriétés caractérisant le fonctionnement d'un système quelconque est un problème que nous n'avons pas abordé. Nous avons cependant classé les propriétés en 3 catégories :

- propriétés de vivacité ;
- propriétés invariantes ;
- propriétés de réponse à une action.

Le principe de classification des propriétés en trois catégories permet de couvrir un nombre important de propriétés, et peut donc servir de méthode de spécification d'une application.

3.6.1. EXCLUSION MUTUELLE DE DECKER

* propriétés de vivacité

a) absence de blocage

La proriété d'absence de blocage du système s'exprime par la formule :

[] ENABLE

b) vivacité des accès à la ressource

La finalité de l'algorithme est d'assurer l'exclusion mutuelle entre les accès des deux processus à la ressource, et de garantir que chaque processus peut effectivement utiliser cette ressource. La vivacité des actions d'accès s'exprime par :

pour P1
  [] POT ENABLE (aut_acces1)
ou encore
  [] POT (Y1=2) AND (X1=1) AND (C2=1)

pour P2
  [] POT ENABLE (aut_acces2)
ou encore
  [] POT (Y2=2) AND (X2=1) AND (C1=1)

* <u>propriétés invariantes</u>

a) exclusion mutuelle

L'exclusion mutuelle est exprimée par la formule suivante :

[] mutuelle_exclusion

où mutuelle_exclusion est le prédicat X1<>2 OR X2<>2, qui garantit que les deux processus n'accèdent pas simultanément à la ressource.

b) la ressource est toujours libérée

Aucun des deux processus ne peut garder indéfiniment la ressource. Cela s'exprime par :

  INIT => NOT POT ALL AFTER(aut_acces1)
ou
  INIT => ALL NOT ALL AFTER(aut_acces1)

  INIT => NOT POT ALL AFTER(aut_acces2)
ou
  INIT => ALL NOT ALL AFTER(aut_acces2)

* <u>réponse à une action</u>

L'exemple étudié permet de souligner l'intérêt de l'opérateur temporel FAIR. Pour montrer qu'une demande d'accès à la ressource

est satisfaite en un temps fini, nous pouvons écrire :

[] AFTER(dem_res1) => INEV AFTER(aut_acces1)
[] AFTER(dem_res2) => INEV AFTER(aut_acces2)

Ces formules ne sont cependant pas vérifiées par la description du système que nous avons proposée, pour la raison suivante. Il existe une séquence d'exécution, à partir d'un état qui satisfait AFTER(dem_res1), dans laquelle le processus P2 uniquement évolue. Ainsi, l'accès à la ressource par P1 est possible pour tout état de la séquence, mais l'action n'est jamais exécutée. La séquence est donc non équitable pour l'action considérée.

Si l'on considère uniquement les évolutions équitables du système, les séquences de ce type sont éliminées, et l'action d'accès à la ressource par P1 devient inévitable.

Les formules exactes qui caractérisent la satisfaction en un temps fini des demandes d'accès sont alors :

[] AFTER(dem_res1) => FAIR AFTER(aut_acces1)
[] AFTER(dem_res2) => FAIR AFTER(aut_acces2)

3.6.2. PROTOCOLE D'ALLOCATION DE BUS

* propriété de vivacité

a) absence de blocage et vivacité des tâches

[] ENABLE

[] POT ENABLE(P1)
[] POT ENABLE(P2)
[] POT ENABLE(P3)

b) vivacité des accès à la ressource

Les actions d'accès à la ressource de chacun des trois processus sont vivantes. Cela s'exprime par les formules suivantes :

[] POT ENABLE(rec_jeton_a_1,rec_jeton_b_1,rec_jeton_c_1)
[] POT ENABLE(rec_jeton_a_2,rec_jeton_b_2,rec_jeton_c_2)
[] POT ENABLE(rec_jeton_a_3,rec_jeton_b_3,rec_jeton_c_3)

* propriétés invariantes

a) un processus ne peut pas garder indéfiniment le jeton de contrôle

```
[] NOT ALL AFTER(rec_jeton_a_1) AND NOT ALL AFTER(rec_jeton_b_1)
                                AND NOT ALL AFTER(rec_jeton_c_1)
[] NOT ALL AFTER(rec_jeton_a_2) AND NOT ALL AFTER(rec_jeton_b_2)
                                AND NOT ALL AFTER(rec_jeton_c_2)
[] NOT ALL AFTER(rec_jeton_a_3) AND NOT ALL AFTER(rec_jeton_b_3)
                                AND NOT ALL AFTER(rec_jeton_c_3)
```

b) un processus ne peut pas être exclu indéfiniment de l'anneau

```
[] INEV AFTER(rec_jeton_a_1, rec_jeton_b_1, rec_jeton_c_1)
[] INEV AFTER(rec_jeton_a_2, rec_jeton_b_2, rec_jeton_c_2)
[] INEV AFTER(rec_jeton_a_3, rec_jeton_b_3, rec_jeton_c_3)
```

c) le jeton de contrôle appartient à un processus au plus à un instant donné

(soit util1, util2 et util3 les prédicats exprimant
l'utilisation de la ressource par P1, P2 et P3)

```
[] AFTER(rec_jeton_a_1, rec_jeton_b_1, rec_jeton_c_1)
        => NOT
   AFTER(rec_jeton_a_2, rec_jeton_b_2, rec_jeton_c_2)

[] AFTER(rec_jeton_a_2, rec_jeton_b_2, rec_jeton_c_2)
        => NOT
   AFTER(rec_jeton_a_3, rec_jeton_b_3, rec_jeton_c_3)

[] AFTER(rec_jeton_a_1, rec_jeton_b_1, rec_jeton_c_1)
        => NOT
   AFTER(rec_jeton_a_3, rec_jeton_b_3, rec_jeton_c_3)
```

ou plus simplement

```
[] util1 => NOT util2
[] util2 => NOT util3
[] util1 => NOT util3
```

d) si le jeton de contrôle se perd, il est inévitablement régénéré

```
[] NOT(util1 OR util2 OR util3) =>
                  INEV (util1 OR util2 OR util3)
```

3.6.3. TRANSMISSION AVEC DECONNECTION

Pour spécifier le système, nous utilisons les prédicats prédéfinis suivants sur les variables de l'application :

```
fina = (disca=1) ;   (* déconnection de 'a' *)
```

```
finb = (discb=1) ;    (* déconnection de 'b' *)
fin  = (fina) and (finb) ;
pasvideab = (countab>0) ;  (* buffer a->b non vide *)
paspleinab = (countab<2) ; (* buffer a->b non plein *)
pasvideba = (countba>0) ;  (* buffer b->a non vide *)
paspleinba = (countba<2) ; (* buffer b->a non plein *)
```

* <u>propriétés d'atteignabilité</u>

a) la déconnection des deux sites 'a' et 'b' est possible à partir de l'initialisation du système. Ces propriétés sont valides et s'expriment par les formules :

```
INIT => POT fina (* déconnection de 'a' *)
INIT => POT finb (* déconnection de 'b' *)
INIT => POT fin  (* déconnection de 'a' et de 'b' *)
```

b) la déconnection des sites 'a' et 'b' est inévitable à partir de l'état initial du système. Ces propriétés sont non valides, et s'expriment par les formules :

```
INIT => INEV fina (* déconnection de 'a' *)
INIT => INEV finb (* déconnection de 'b' *)
INIT => INEV fin  (* déconnection de 'a' et de 'b' *)
```

Le non déterminisme entre les actions d'un site est tel que le site peut ne jamais se déconnecter (en n'exécutant jamais les commandes gardées "senddisa" ou "senddisb"). Les formules ci-dessus ne sont donc pas vérifiées par la description que nous avons donnée du système.

c) par contre, si nous supposons un fonctionnement équitable du système, les actions de déconnection des sites 'a' et 'b' finiront par être exécutées. Ces propriétés s'expriment par les formules :

```
INIT => FAIR fina (* déconnection de 'a' *)
INIT => FAIR finb (* déconnection de 'b' *)
INIT => FAIR fin  (* déconnection de 'a' et de 'b' *)
```

* <u>propriétés de vivacité</u>

Nous pouvons exprimer la vivacité des trois tâches ENDA, ENDB et LINE par les formules :

```
[] POT ENABLE (ENBA)
[] POT ENABLE (ENDB)
[] POT ENABLE (LINE)
```

Les deux premières propriétés sont valides, car les commandes gardées correspondant aux traitements internes des sites 'a' et 'b' sont toujours exécutables, indépendamment de l'état de la ligne de

transmission. Par contre, la tâche LINE peut être définitivement bloquée, lorsque les deux sites sont déconnectés ; la troisième propriété n'est donc pas valide. Remarquons que l'absence de blocage du système est garantie par le fait que les deux tâches ENDA et ENDB sont non seulement vivantes, mais vérifient de plus :

[] ENABLE(ENDA)
[] ENABLE(ENDB)

ce qui implique la validité de la formule

[] ENABLE

* <u>propriétés invariantes</u>

a) nous pouvons exprimer le fait qu'un processus, une fois déconnecté, ne peut plus communiquer avec la ligne de transmission, et qu'il reste indéfiniment dans cet état :

[] fina => ALL fina
[] fina => ALL NOT ENABLE(sendal)
[] fina => ALL NOT ENABLE(recla)

[] finb => ALL finb
[] finb => ALL NOT ENABLE(sendbl)
[] finb => ALL NOT ENABLE(reclb)

b) les actions d'émission et de réception de messages par la tâche LINE pour un site connecté sont possibles, dès lors que les contraintes de stockage sont favorables :

[] (pasvideab AND NOT finb) => ENABLE(sendlb)
[] (paspleinab AND NOT fina) => ENABLE(recal)

[] (pasvideba AND NOT fina) => ENABLE(sendla)
[] (paspleinba AND NOT finb) => ENABLE(recbl)

* <u>propriétés de réponse aux actions</u>

a) l'envoi d'un message de déconnection provoque la déconnection effective du site :

[] AFTER(senddisa) => fina
[] AFTER(senddisb) => finb

b) l'envoi d'un message par un site provoque le stockage du message, et valide l'envoi d'un message par la ligne à l'autre site :

[] AFTER(recal) => pasvideab
[] (AFTER(recal) AND NOT finb) => ENABLE(sendlb)

```
[] AFTER(recbl) => pasvideba
[] (AFTER(recbl) AND NOT fina) => ENABLE(sendla)
```

c) l'envoi d'un message par un site entraîne sa réception par l'autre site. Cette propriété n'est pas valide, pour deux raisons. Premièrement, il est possible que l'autre site se déconnecte avant de recevoir le message. Deuxièmement, l'autre site peut ne pas recevoir le message, si l'action de réception n'est jamais exécutée :

```
[] AFTER(recal) => INEV AFTER(sendlb)
[] AFTER(recbl) => INEV AFTER(sendla)

[] AFTER(recal) => INEV (finb OR AFTER(sendlb))
[] AFTER(recbl) => INEV (fina OR AFTER(sendla))
```

Les propriétés suivantes sont cependant valides :

```
[] AFTER(recal) => FAIR (finb OR AFTER(sendlb))
[] AFTER(recbl) => FAIR (fina OR AFTER(sendla))
```

3.6.4. RESEAU DE TRAFIC FERROVIAIRE

L'ensemble des spécifications que l'on peut formuler sur le système que nous avons décrit n'est pas très important. Nous pouvons cependant énoncer quelques caractéristiques générales du système.

Nous notons ainsi les prédicats pré-définis suivants :

```
sigma4 = (B121 + ... + B323 = 4) ;
(* nombre de trains dans le réseau = 4 *)

occup121 = (B121=1) ;
(* occupation du secteur 121 *)
---
occup323 = (B323=1) ;

full = (occup121) AND ... AND (occup323) ;
(* le réseau est plein *)
```

* propriétés de vivacité

a) le système ne peut pas se bloquer. Cette propriété s'exprime par la formule :

```
[] ENABLE
```

b) les actions d'accès à un secteur quelconque (et celles de

départ) sont vivantes, à condition que le réseau ne soit pas saturé à l'initialisation du système. Ces propriétés sont valides et s'expriment par les formules :

(INIT AND NOT full) => ALL POT ENABLE(TOijk)

(INIT AND NOT full) => ALL POT ENABLE(FROMijk)

pour tout secteur (i,j,k) du réseau.

* propriétés invariantes

a) le système que nous proposons est un système fermé comportant une configuration initiale telle que le réseau n'est pas saturé. Aussi les propriétés suivantes sont elles vérifiées :

INIT => NOT full
(* le réseau n'est pas saturé au départ *)

(INIT AND NOT full) => ALL NOT full
(* le réseau ne sera jamais saturé *)

[] sigma4
(* le nombre de trains est constant dans le réseau *)

b) deux trains ne peuvent pas occuper le même secteur à un instant donné. Cette propriété s'exprime par la formule :

```
[]      B121 in [0..1]
    AND B122 in [0..1]
        - - - - - -
    AND B323 in [0..1]
```

3.6.5. TRANSMISSION SYNCHRONE ET ASYNCHRONE

Le système que nous avons décrit ne présente pas un grand intérêt du point de vue de la spécification. Il nous permet cependant de mettre en évidence l'utilité des opérateurs temporels conditionnels, et de souligner les nuances entre des formules intuitivement très proches.

* propriétés de vivacité

[] ENABLE
[] POT ENABLE(SENDREC)
[] POT ENABLE(OBJECT)

[] POT ENABLE(action)

où action est une étiquette associée à une commande gardée quelconque de la description.

* propriétés de réponse à une action

Pour vérifier le caractère synchrone (plus précisément cyclique dans notre description) d'une des entrées et de la sortie de l'objet décrit, nous pouvons exprimer le fait que la tâche OBJECT, après avoir reçu un paquet à l'entrée synchrone, ne peut pas recevoir un autre paquet à cette même entrée tant qu'elle n'a pas envoyé un paquet à la sortie synchrone :

(nous notons AFTERSEND le prédicat
AFTER(sendpacka,sendpacks))

f1 = [] AFTER(recsyn) => ALL[NOT AFTERSEND]
ENABLE(recsyn) => AFTERSEND

Cette formule est à distinguer de la formule suivante :

f2 = [] AFTER(recsyn) => ALL[NOT AFTERSEND]
NOT ENABLE(recsyn)

car les deux formules n'ont pas le même domaine de validité.

En effet, plaçons nous à un état du système dans AFTER(recsyn), et suivons une séquence b qui mène dans (NOT AFTERSEND) à l'exécution de l'action étiquetée par "sendpacks" ou "sendpacka". Considérons l'état s de b qui précède l'action. Cet état appartient à l'ensemble

| POT [NOT AFTERSEND] AFTERSEND |,

mais pas à l'ensemble

| POT [NOT AFTERSEND] NOT ENABLE(recsyn) |,

car l'exécution de l'action valide immédiatement l'action étiquetée par "recsyn", de telle sorte que l'état s' successeur de s dans b vérifie AFTERSEND mais aussi ENABLE(recsyn).

La formule qui exprime la propriété que nous avons énoncée est donc f1, et nous vérifierons au chapitre V que f1 est une formule valide, tandis que f2 ne l'est pas.

Pour exprimer de plus que la réception d'un paquet à l'entrée synchrone provoque effectivement l'émission d'un paquet à la sortie synchrone, nous utilisons la formule

[] AFTER(recsyn) => INEV[NOT ENABLE(recsyn)] AFTERSEND

Cette formule n'est pas valide pour la description fournie, car la tâche OBJECT peut évoluer indéfiniment en recevant uniquement des paquets de l'entrée asynchrone. Si l'on substitue l'opérateur INEV par l'opérateur FAIR, la formule devient valide.

3.6.6. RESEAU DE DISTRIBUTEURS DE BILLETS

De nombreuses propriétés intéressantes du système concernent les retraits d'argent. Notre description ne tient pas compte du facteur "sommes d'argent", et ne nous permet donc pas de spécifier et de vérifier que le système respecte les impératifs fixés.

Nous pouvons cependant exprimer des propriétés comportementales qui ne dépendent pas du facteur "sommes retirées", et qui caractérisent le fonctionnement général du réseau de distributeurs de billets.

Nous notons ainsi les prédicats pré-définis suivants :

base_out = (KO=1) ;
(* base hors d'usage *)

base_busy = (LIBRE>=1) ;
(* base en cours de communication *)

dis_busy_i = (PRESENTi>0) ;
(* distributeur i occupé *)

* <u>propriétés de vivacité</u>

a) le système ne peut jamais se bloquer complètement. Cette propriété s'exprime par la formule :

[] ENABLE

b) toutes les actions décrites dans le système sont vivantes. Cela s'exprime par des formules du type :

[] POT ENABLE(action)

où action est une étiquette associée à une transition quelconque.

Ainsi, nous pouvons exprimer que les actions d'insertion et de retrait d'une carte magnétique, ou de commandes (1, 2 ou 3) sont toujours possibles :

[] POT ENABLE(cardini) (i = 1,2)
(* entrée d'une carte *)
[] POT ENABLE(commandei1)

```
[] POT ENABLE(commandei2)
[] POT ENABLE(commandei3)
[] POT ENABLE(cardouti)
   (* vivacité des commandes 1,2 ou 3,
            et de la sortie d'une carte *)
```

c) les différentes actions ne sont cependant pas inévitables, car le non déterminisme de la description permet d'éviter l'exécution de certaines commandes gardées du système. Ainsi les formules :

```
[] INEV AFTER(action)
```

ne sont pas toutes valides. On peut par exemple ne jamais retirer une carte magnétique d'un distributeur :

```
[] INEV AFTER(cardouti)
```

est une formule non valide, pour deux raisons. Il existe premièrement des évolutions du système dans lesquelles un distributeur n'intervient pas. Deuxièmement, l'introduction d'un carte magnétique dans l'appareil n'impose nullement son retrait, la carte pouvant rester indéfiniment dans le distributeur.

d) par contre, toutes les actions décrites sont inévitablement atteignables sous l'hypothèse de l'équité du fonctionnement du système, ce qui se traduit par :

```
[] FAIR AFTER(action)
```

pour toute étiquette action associée à une commande gardée quelconque.

Pour l'action de retrait d'une carte magnétique, nous exprimons ainsi trois propriétés :

```
[] POT ENABLE(cardouti)  (* valide *)
[] INEV AFTER(cardouti)  (* non valide *)
[] FAIR AFTER(cardouti)  (* valide *)
```

* <u>Propriétés invariantes</u>

La base de données peut à tout instant tomber hors d'usage. Par contre les propriétés suivantes sont vérifiées :

a) nous pouvons exprimer le fait que la base ne peut pas tomber hors d'usage durant une communication avec un distributeur. Cette propriété est valide et s'exprime par la formule :

```
[] base_out => NOT base_busy
```

b) deux distributeurs ne peuvent pas communiquer simultanément avec

la base de données. Pour exprimer cette propriété, nous utilisons les formules suivantes :

```
[] AFTER(sendacc1,rectotal1) => NOT AFTER(sendacc2,rectotal2)
[] AFTER(sendacc2,rectotal2) => NOT AFTER(sendacc1,rectotal1)
```

c) deux usagers ne peuvent pas utiliser simultanément le même distributeur. Cette propriété s'exprime par la formule suivante :

```
[] dis_busy_1 => NOT ENABLE(cardin1)
[] dis_busy_2 => NOT ENABLE(cardin2)
```

4. CONCLUSION

Dans ce chapitre, nous avons présenté l'utilisation d'une logique temporelle pour la spécification des applications réparties.

Le choix d'une logique du temps arborescent présente deux avantages :

- il permet de distinguer les modalités d'atteignabilité potentielle (POT) et inévitable (INEV), alors que les logiques du temps linéaire ne le permettent pas ;
- il donne la possibilité de raisonner en termes d'ensembles d'états plutôt qu'en termes de séquences, et permet de résoudre le problème de l'équité posé par le non déterminisme des modèles utilisés pour représenter les systèmes répartis.

L'emploi d'une logique temporelle pour spécifier et prouver les propriétés comportementales des systèmes distribués fait actuellement l'objet de nombreux travaux. Nous pensons que nos recherches ont permis de mettre au point une méthode de spécification fiable et originale :

- elle utilise une logique temporelle (de grande expressibilité) pour spécifier les propriétés, en relation avec un langage de description des systèmes répartis ;
- la logique employée est décidable, ce qui permet de disposer d'une procédure de décision pour étudier et comparer formellement les formules de spécification ;
- elle propose une solution formelle pour caractériser les comportements équitables des systèmes étudiés.

CHAPITRE IV

```
**********************************
*                                *
*   PREUVE PAR EVALUATION DES    *
*   FORMULES DE SPECIFICATION    *
*                                *
**********************************
```

1. Introduction
2. Principe de l'évaluation
3. Evaluation dans le graphe des états
4. Le cas particulier de la logique S4
5. Evaluation dans le treillis des prédicats
6. Programmation et comparaison des méthodes
7. Conclusion

1. INTRODUCTION

L'objet de ce chapitre est de présenter une méthode d'analyse d'un système parallèle par évaluation de formules de spécification. Le système, décrit initialement sous forme algorithmique, a été traduit en un système conditions-actions plat ne comportant plus d'échanges. Les formules de spécification sont des formules de la logique temporelle CTL dont les variables propositionnelles sont des prédicats booléens non temporels sur les variables du programme de description. Nous avons défini dans le chapitre précédent une interprétation d'une formule de spécification dans le système de transitions construit à partir du CA-système, sous forme de l'ensemble des états du système de transitions qui vérifient la formule.

La méthode d'analyse consiste à évaluer, pour une formule de spécification donnée, l'ensemble des états qui satisfont cette formule. Si cet ensemble coïncide avec l'ensemble de tous les états du système, la formule est vérifiée à tout instant par le système, pour toute initialisation et pour toute évolution ; la formule est alors dite valide. S'il existe un état qui ne satisfait pas la formule, la formule est dite non valide.

Nous présentons deux méthodes d'analyse distinctes qui ont été toutes deux programmées.

La première consiste à construire explicitement le système de transitions repésentant l'application (par un graphe d'états), et à définir l'évaluation des formules par des calculs d'ensembles d'états dans ce système.

La deuxième méthode consiste à analyser directement le système conditions-actions, sans générer le système de transitions. La présence de variables entières implique l'utilisation de méthodes de démonstration automatique pour les entiers. Pour éviter cela, nous pouvons nous ramener au cas des CA-systèmes à variables booléennes, pour lesquels il existe des résultats exploitables. Il suffit de savoir transformer un CA-système entier en un CA-système booléen qui lui est équivalent. Dans le CA-système booléen les états du système sont représentés par des expressions booléennes sur des variables booléennes, les formules de spécification sont caractérisées par un prédicat booléen exprimant le domaine de validité de la formule, et l'évaluation des formules se ramène à un calcul booléen.

Nous présentons successivement ces deux méthodes d'analyse d'un système conditions-actions borné. La première est celle qui est utilisée dans QUASAR, et comporte des simplifications lorsque

l'ensemble des formules de spécification est construit uniquement sur le sous ensemble S4 de la logique temporelle CTL. La deuxième méthode, bien qu'ayant été expérimentée, n'a pas été retenue dans QUASAR. Nous détaillons en fin de chapitre les raisons de notre choix.

2. PRINCIPE DE L'EVALUATION

L'approche suivie est due à J. Sifakis [SIF79], et consiste à caractériser les opérateurs temporels comme des points fixes de transformateurs de prédicats monotones. Elle fournit par ailleurs une méthode d'évaluation par calcul itératif de ces opérateurs.

Soit (S,P,R) un système de transitions. L'ensemble des prédicats sur S, muni des opérations d'union (u), d'intersection (n) et de complémentation (^) peut être identifié avec le treillis PS des parties de S.

Un transformateur de prédicats est une application de PS 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 tel que :

(f u g)(X) = f(X) u g(X)
(f n g)(X) = f(X) n g(X)
^f(X) = ^(f(X))
^f^(X) = ^f(^X)
Id(X) = X

et pour tout k>=0,

[f]k+1(X) = f([f]k(X))
avec [f]0(X) = Id

On introduit également les notations suivantes :

[f]* = Id u f u [f]2 u ... [f]k u ...
[f]x = Id n f n [f]2 n ... [f]k n ...

2.1. LE TRANSFORMATEUR DE PREDICATS PRE

On définit le transformateur de prédicats pre tel que, pour tout X de PS, et pour tout s de S

pre(X)(s) = il existe s' de S tel que s R s' et X(s')

c.à.d. PRE(X) est l'ensemble des états de S à partir desquels il est possible d'atteindre un état vérifiant X par exécution d'une transition.

On définit le tranformateur de prédicats pretilda, dual de pre, par pretilda = ^pre^. Pretilda(X) est l'ensemble des états à partir

desquels il n'est pas possible d'atteindre un état de ~X par exécution d'une transition.

On note ainsi les prédicats sur S suivants :

ENABLE (s) = il existe s' tel que s R s'
SINK (s) = ~ENABLE(s)

Les transformateurs de prédicats pre et pretilda vérifient les propriétés suivantes, qui ont été démontrées dans [SIF79].

Pour tout système de transitions (S,P,R) :

a) pre (∅) = ∅
pretilda (S) = S ;

b) pre (S) = ENABLE
pretilda (∅) = SINK ;

c) pour tout X :
pre (X) est inclus dans ENABLE
pretilda (X) contient SINK ;

d) pour tout X :
(pre n pretilda) (X) = pretilda (X) n ENABLE
(pre u pretilda) (X) = pre (X) u SINK ;

e) pre est distributif par rapport à l'union
pretilda est distributif par rapport à l'intersection ;

f) pre et pretilda sont monotones.

2.2 INVARIANTS ET TRAJECTOIRES

Les notions d'invariants et de trajectoires présentées au chapitre précédent peuvent être définies ainsi :

Etant donné un système de transitions (S,P,R) et C un prédicat sur S,

- un invariant conditionnel (sous la condition C) est un prédicat X sur S tel que, pour tout s,s' de S :

X(s) et C(s) et s R s' implique X(s') ;

- une trajectoire conditionnelle (sous la condition C) est un prédicat W sur S tel que, pour tout s de S :

W(s) et C(s) et ENABLE(s) implique
il existe s´ de S tel que (s R s´ et W(s´)).

Un invariant (resp. trajectoire) est un invariant conditionnel (resp. trajectoire conditionnelle) dont la condition est le prédicat identiquement vrai. 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.

Etant donné un système de transitions (S,P,R) et C un prédicat sur S :

a) X est un invariant ssi
X = X n pretilda (X),

b) Y est un invariant sans blocage ssi
Y = Y n (pre n pretilda) (Y),

c) W est une trajectoire ssi
W = W n (pre u pretilda) (W),

d) V est une trajectoire sans fin ssi
V = V n pre (V),

e) X est un invariant conditionnel (sous la condition C) ssi
X = X n (^C u pretilda (X)),

f) Y est un invariant conditionnel sans blocage (sous la condition C) ssi
Y = Y n (^C u (pre n pretilda) (Y)),

g) W est une trajectoire conditionnelle (sous la condition C) ssi
W = W n (^C u (pre u pretilda) (W)),

h) V est une trajectoire conditionnelle sans fin (sous la condition C) ssi
V = V n (^C n pre (V)).

2.3. RAPPEL SUR LES POINTS FIXES DES FONCTIONS MONOTONES

Nous rappelons quelques résultats connus sur les points fixes des fonctions monotones ([TAR55] [PAR69] [SIF79]).

Définitions :

Une fonction monotone f dans un treillis est dite u-continue si

elle préserve les limites des suites croissantes, c.à.d. si P1, P2, ... est une suite croissante (Pi est inclus dans Pi+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 PS, et X0 un élément de PS :

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.

2.4. CALCUL ITERATIF DES OPERATEURS TEMPORELS

Nous avons vu que pre (resp. pretilda) est distributif par rapport à l'union (resp. l'intersection) et donc pre (resp. pretilda) est u-continu (resp. n-continu). Sifakis [SIF79] a montré de plus que pre (resp. pretilda) est n-continu (resp. u-continu) ssi la relation R est à image finie, c.à.d. pour chaque s de S, il existe k entier tel que

$$| \{ s' \mid s R s' \} | <= k.$$

Les systèmes conditions-actions manipulés par QUASAR sont dans ce cas, d'où les résultats suivants :

Etant donné un système de transitions (S,P,R) et un prédicat C sur S :

a) [Id n pretilda]x (X0) = [pretilda]x (X0) est le plus grand invariant contenu dans X0,

b) [Id n pre n pretilda]x (Y0) est le plus grand invariant sans blocage contenu dans Y0,

c) [Id n (pre u pretilda)]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 pretilda)]x (X0) = [^C u pretilda]x (X0) est le plus grand invariant conditionnel (sous la condition C) contenu dans X0,

f) [Id n (^C u pre n pretilda)]x (Y0) est le plus grand invariant conditionnel sans blocage (sous la condition C) contenu dans Y0,

g) [Id n (^C u pre u pretilda)]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.

Etant donné un système de transitions (S,P,R) dont la relation R est à image finie, f et g deux formules temporelles de la logique CTL :

```
|ALL(f)|        = [pretilda]x (|f|)
|ALW(f)|        = [Id n pre n pretilda]x (|f|)
|SOME(f)|       = [Id n (pre u pretilda)]x (|f|)
|SONT(f)|       = [Id n pre]x (|f|)
|ALL[g](f)|     = [|^g| u pretilda]x (|f|)
|ALW[g](f)|     = [Id n (|^g| u pre n pretilda)]x (|f|)
|SOME[g](f)|    = [Id n (|^g| u pre u pretilda)]x (|f|)
|SONT[g](f)|    = [Id n (|^g| u pre)]x (|f|)

|POT(f)|        = [pre]* (|f|)
|WPOT(f)|       = [Id u pre u pretilda]* (|f|)
|INEV(f)|       = [Id u (pre n pretilda)]* (|f|)
|OBL(f)|        = [Id u pretilda]* (|f|)
|POT[g](f)|     = [|g| n pre]* (|f|)
|WPOT[g](f)|    = [Id u |g| n (pre u pretilda)]* (|f|)
|INEV[g](f)|    = [Id u |g| n pre n pretilda)]* (|f|)
|OBL[g](f)|     = [Id u |g| n pretilda]* (|f|)
```

Les résultats précédents permettent de calculer itérativement l'interprétation des opérateurs temporels comme la limite des suites suivantes :

```
ALL(f)       : Xk+1 = Xk n pretilda (Xk)
ALW(f)       : Xk+1 = Xk n (pre n pretilda) (Xk)
SOME(f)      : Xk+1 = Xk n (pre u pretilda) (Xk)
SONT(f)      : Xk+1 = Xk n pre (Xk)
ALL[g](f)    : Xk+1 = Xk n (|^g| u pretilda) (Xk)
ALW[g](f)    : Xk+1 = Xk n (|^g| u (pre n pretilda)) (Xk)
SOME[g](f)   : Xk+1 = Xk n (|^g| u pre u pretilda) (Xk)
SONT[g](f)   : Xk+1 = Xk n (|^g| u pre) (Xk)

POT(f)       : Xk+1 = Xk u pre (Xk)
WPOT(f)      : Xk+1 = Xk u (pre u pretilda) (Xk)
INEV(f)      : Xk+1 = Xk u (pre n pretilda) (Xk)
```

```
OBL(f)        : Xk+1 = Xk u pretilda (Xk)
POT[g](f)     : Xk+1 = Xk u (|g| n pre) (Xk)
WPOT[g](f)    : Xk+1 = Xk u (|g| n (pre u pretilda)) (Xk)
INEV[g](f)    : Xk+1 = Xk u (|g| n pre n pretilda) (Xk)
OBL[g](f)     : Xk+1 = Xk u (|g| n pretilda) (Xk)
```

en initialisant avec X0 = |f|.

Ces résultats fournissent une méthode de calcul des opérateurs temporels dans tous les modèles où il existe une technique de calcul de la fonction pre.

3. EVALUATION DANS LE GRAPHE DES ETATS

3.1. PRINCIPE DE LA METHODE

La méthode d'analyse que nous allons présenter consiste à construire explicitement le système de transitions sous la forme d'un graphe des états, et à définir des règles de calcul qui permettent de produire l'ensemble caractéristique d'une formule de spécification quelconque.

Nous présentons la méthode en trois étapes : construction du graphe des états, évaluation des ensembles caractéristiques des prédicats non temporels du système (qui seront les variables propositionnelles des formules à analyser), et évaluation de l'ensemble caractéristique d'une formule quelconque.

3.2. CONSTRUCTION DU GRAPHE DES ETATS

Le système de transitions est représenté par un graphe orienté, dont les noeuds sont les états du CA-système et les arcs la relation d'accessibilité entre les états. Chaque état est défini par un ensemble complètement spécifié de valeurs des variables, et ses successeurs sont les états que l'on atteint par exécution d'une commande gardée validée pour cet état.

Les procédures PASCAL générées à partir du programme de description de l'application sont utilisées pour construire le graphe :

- la procédure INITSYSTEME produit les valeurs initiales des variables initialisées, et indique quelles sont celles qui ne le sont pas. Soit XINIT l'ensemble des variables initialisées. Une initialisation du système est caractérisée par les valeurs initiales des variables de XINIT, et par une valeur quelconque de chacune des autres variables dans son domaine de définition. L'ensemble des états initiaux possibles du système est donc construit par un parcours exhaustif du produit cartésien des domaines de définition des variables qui n'appartiennent pas à XINIT ;
- l'ensemble des états du système de transitions est construit par la mise à feu systématique de toutes les commandes gardées exécutables à partir de tous les états initiaux possibles du système. On obtient ainsi de nouveaux états auxquels on applique, s'il n'ont pas déjà été rencontrés, le même traitement. Pour un état s, on construit l'ensemble succ(s) des états successeurs directs possibles de s, c'est à dire l'ensemble des états s' tels qu'il existe une commande gardée validée pour l'état s et dont l'exécution conduit à l'état s'. L'ensemble succ(s) est calculé

itérativement par appel de la procédure SYSTEME pour l'état s et chacune des transitions.

La construction du graphe est accompagnée du calcul des ensembles ENAB(a) et AFT(a), pour toute commande gardée a figurant dans le programme de description. ENAB(a) est l'ensemble des états à partir desquels on peut exécuter a, et AFT(a) est l'ensemble des états pour lesquels le contrôle de la tâche élémentaire contenant a se situe immédiatement après l'exécution de a. Les ensembles sont obtenus progressivement par application des règles suivantes :

- un état s appartient à ENAB(a) ssi il existe au mois une commande gardée t validée pour s, telle que a participe à t (c.à.d. telle que t=a, ou telle que t a été obtenue par fusion d'un ensemble de commandes gardées contenant a) ;

- pour toute commande gardée a, aucun des états initiaux n'appartient à AFT(a) ;

- pour toute commande gardée t validée pour un état s (dont l'exécution à partir de s mène à l'état s'), et pour toute commande gardée a figurant dans le corps d'une tâche élémentaire Ta :
  - si s appartient à AFT(a), et si aucune des commandes gardées participant à t ne figure dans Ta, alors s' appartient à AFT(a) ;
  - si a participe à t, alors s' appartient à AFT(a) ;

- pour tout état s' obtenu par appel de la procédure SYSTEME, s' ne peut être confondu avec un état s'' déjà traité que si s' et s'' correspondent aux mêmes valeurs des variables de l'application, et si les ensembles :
{ a | s' appartient à AFT(a) }
et { a | s'' appartient à AFT(a) }
sont égaux. Si la deuxième condition n'est pas remplie, les deux états sont considérés comme distincts, bien qu'ils correspondent au même état des variables de l'application (ce cas se produit par exemple lorsque l'exécution d'une transition conduit à une valeur des variables déjà représentée par un état de l'ensemble INIT des états initiaux possibles du système). Ce procédé est indispensable pour distinguer les états que l'on peut atteindre par des transitions différentes. En pratique, pour limiter la multiplication des états, on ne s'intéresse qu'aux commandes gardées étiquetées dans la description.

Au cours de la génération du graphe, les états s tels que succ(s) est l'ensemble vide sont rangés dans l'ensemble SINK des états puits du système. Cet ensemble peut contenir un état particulier, appelé état d'erreur, qui est atteint lorsqu'une commande gardée produit une opération interdite sur une des variables, division par zéro ou affectation d'une valeur en dehors des bornes du domaine de définition.

L'ensemble TOUT de tous les états du système est déterminé à la fin de la phase de construction du graphe, et l'on définit l'ensemble ENABLE des états à partir desquels le système peut évoluer, par la relation :

ENABLE = TOUT - SINK = ^SINK.

Du point de vue pratique, il est évident que la construction du graphe des états peut être faite une fois pour toutes, le système de transitions restant valide tant que l'on ne modifie pas le programme de description.

Pour calculer les interprétations des opérateurs, la relation d'accessibilité dans le graphe sous la forme de succession ne présente aucune utilité. C'est pourquoi le graphe est en fait représenté par un ensemble d'états et par la relation de précédence entre états.

3.3. EVALUATION DES PREDICATS NON TEMPORELS

3.3.1 VARIABLES PROPOSITIONNELLES ASSOCIEES AUX ACTIONS

L'évaluation des variables propositionnelles associées aux actions du système est faite à partir des ensembles ENAB(a) et AFT(a), calculés lors de la phase de construction du graphe d'états.

Pour toute tâche élémentaire Ta de l'application, la variable propositionnelle ENABLE(Ta) est calculée par union des ensembles d'états ENAB(a), pour toutes les commandes gardées a (étiquetées ou non) figurant dans le corps de Ta. Pour toute tâche composée Ta, ENABLE(Ta) est l'union des ensembles ENABLE(Tai) des tâches Tai composant Ta.

Pour toute étiquette ea, associée à une commande gardée a du programme de description, les ensembles caractéristiques des variables propositionnelles ENABLE(ea) et AFTER(ea) sont respectivement égaux à ENAB(a) et AFT(a).

3.3.2 PREDICATS SUR LES VARIABLES DE L'APPLICATION

Pour faciliter l'écriture des formules de spécification, on demande à l'utilisateur de définir d'un seul tenant un ensemble de prédicats sur les variables de l'application. Chaque prédicat est désigné par un identificateur de prédicat :

<identificateur> = <prédicat> ;

et seul l'identificateur de prédicat peut apparaître dans les formules.

L'évaluation de ces prédicats consiste à calculer leurs ensembles caractéristiques par un parcours exhaustif des états du système.

Chaque prédicat est une expression booléenne portant sur les variables de l'application, dont la valeur est déterminée pour chaque état du système de transitions. On obtient donc l'ensemble caractéristique d'un prédicat non temporel en testant la valeur du prédicat à chaque état.

Les prédicats de ce type définis par l'utilisateur sont en fait traduits vers une fonction booléenne PASCAL de nom PREDICAT, qui produit pour un état et un numéro de prédicat donnés la valeur du prédicat pour cet état. De cette façon, l'interprétation des expressions est faite automatiquement par le compilateur PASCAL, et la syntaxe des expressions est celle admise par ce langage, ce qui permet d'utiliser avec aisance les appartenances ensemblistes qui s'avèrent fort utiles pour vérifier les domaines parcourus par les variables.

L'évaluation des prédicats est alors faite par appel, pour chaque état du graphe, de la procédure PREDICAT pour chacun des prédicats.

3.4. EVALUATION DES FORMULES

Nous donnons tout d'abord une représentation des formules de la logique sous forme d'un graphe. Nous définissons ensuite les règles d'évaluation de l'ensemble caractéristique d'une formule f par le calcul progressif des ensembles caractéristiques des sous formules étiquetant les noeuds de la représentation de f.

3.4.1. REPRESENTATION DES FORMULES

A toute formule f de la logique temporelle CTL est associée sa représentation naturelle sous forme de graphe. Les noeuds du graphe sont étiquetés par les éléments du vocabulaire de la formule ; les feuilles sont étiquetées par les variables propositionnelles, les noeuds internes par les opérateurs présents dans la formule.

Pour tout noeud n, on note f(n) la sous formule de f représentée par le sous graphe de racine n. Soit n un noeud interne du graphe, étiqueté par un opérateur OP(arg1,arg2). Les fils gauche (ng) et

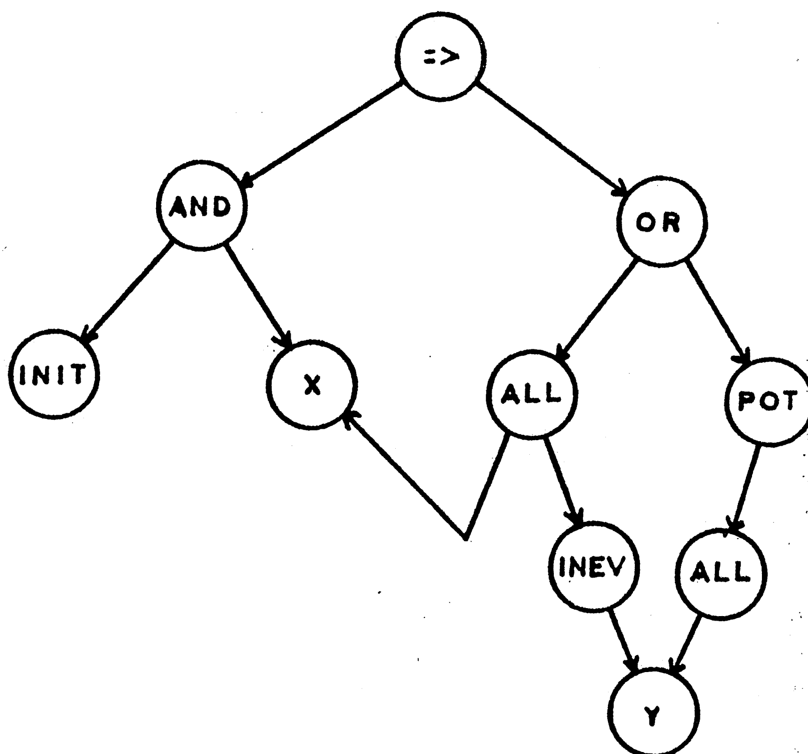
droit (nd) de n représentent les sous formules f1=f(ng) et f2=f(nd) de f, telles que f(n)=OP(f1,f2).

Il est possible de regrouper les sous formules qui sont présentes plusieurs fois dans la formule f, de façon à réduire au maximum la taille du graphe. Le graphe ainsi obtenu est un graphe acyclique orienté que nous notons DAG(f).

Nous donnons à titre d'exemple le graphe DAG(f), pour la formule

f = (INIT AND X) => (ALL [X] INEV Y OR POT ALL Y)

figure 4.1



3.4.2. EVALUATION DES FORMULES

Soit f une formule de la logique représentée par son graphe DAG(f).

L'évaluation de l'ensemble caractéristique de f consiste à calculer progressivement, pour tout noeud n de DAG(f), l'ensemble caractéristique S(n) de la formule f(n) à partir des ensembles caractéristiques S(ng) et S(nd) des formules f(ng) et f(nd) étiquetant les noeuds ng et nd, fils de n.

Les feuilles du graphe DAG(f) sont étiquetées par les prédicats non temporels de l'application, dont les ensembles caractéristiques ont été précalculés. L'évaluation de la formule f consiste en un parcours postfixé de DAG(f), chaque étape permettant de constituer l'ensemble caractéristique d'un noeud n en fonction des ensembles caractéristiques de ses fils ng et nd. On obtient ainsi à la fin du parcours l'ensemble caractéristique S(n0) de la formule étiquetant la racine de DAG(f), à savoir la formule f.

Nous présentons désormais les règles de calcul des ensembles caractéristiques des formules temporelles.

3.4.3. EVALUATION DES OPERATEURS

Soit f une formule et n un noeud du graphe DAG(f). Nous avons vu comment sont calculés les ensembles caractéristiques des variables propositionnelles étiquetant les feuilles du graphe. Nous supposons que n est un noeud interne du graphe, étiqueté donc par un opérateur de la logique. Nous présentons les règles de calcul de l'ensemble caractéristique S(n) en fonction des ensembles S(ng) et S(nd) des noeuds ng et nd, fils de n (ng est étiqueté par f1, nd par f2).

Le cas des opérateurs booléens ne présente aucune difficulté, car le calcul s'effectue immédiatement par des opérations ensemblistes élémentaires.

(les notations utilisées sont :

- u pour l'union d'ensembles
- n pour l'intersection
- ^ pour la complémentation : ^S = TOUT - S)

```
f = NOT f1          S(n) = ^S(ng)
f = f1 AND f2       S(n) = S(ng) n S(nd)
f = f1 OR f2        S(n) = S(ng) u S(nd)
f = f1 => f2        S(n) = ^S(ng) u S(nd).
```

Le cas des opérateurs temporels est plus complexe, car l'obtention de S(n) n'est pas immédiate. Les résultats du paragraphe IV.2 fournissent une méthode de calcul des opérateurs temporels, et sont utilisés pour calculer itérativement l'ensemble S(n).

Ce calcul suppose connue la fonction d'ensembles pre, qui associe à chaque ensemble d'états S l'ensemble des états du système de transitions qui sont des prédécesseurs directs des états de S. Cette fonction est la relation inverse d'accessibilité du graphe des états du système de transitions, qui a été calculée lors de la génération du graphe. La fonction pretilda se déduit de pre par dualisation.

Pour une formule f(n) dont l´opérateur principal est un opérateur temporel, S(n) est la limite d´une suite monotone (Xk) d´ensembles d´états. S(n) est calculé à partir de la donnée de l´ensemble caractéristique S(ng) = X0, et d´une relation de récurence qui produit Xk+1 en fonction de Xk, S(nd), pre et pretilda. Nous formulons cette relation, pour tout opérateur temporel de CTL :

```
ALL(f)       : Xk+1 = Xk n pretilda (Xk)
ALW(f)       : Xk+1 = Xk n (pre n pretilda) (Xk)
SOME(f)      : Xk+1 = Xk n (pre u pretilda) (Xk)
SONT(f)      : Xk+1 = Xk n pre (Xk)
ALL[g](f)    : Xk+1 = Xk n (^S(nd) u pretilda) (Xk)
ALW[g](f)    : Xk+1 = Xk n (^S(nd) u (pre n pretilda)) (Xk)
SOME[g](f)   : Xk+1 = Xk n (^S(nd) u pre u pretilda) (Xk)
SONT[g](f)   : Xk+1 = Xk n (^S(nd) u pre) (Xk)

POT(f)       : Xk+1 = Xk u pre (Xk)
WPOT(f)      : Xk+1 = Xk u (pre u pretilda) (Xk)
INEV(f)      : Xk+1 = Xk u (pre n pretilda) (Xk)
OBL(f)       : Xk+1 = Xk u pretilda (Xk)
POT[g](f)    : Xk+1 = Xk u (S(nd) n pre) (Xk)
WPOT[g](f)   : Xk+1 = Xk u (S(nd) n (pre u pretilda)) (Xk)
INEV[g](f)   : Xk+1 = Xk u (S(nd) n pre n pretilda) (Xk)
OBL[g](f)    : Xk+1 = Xk u (S(nd) n pretilda) (Xk).
```

Le calcul de S(n) s´effectue donc itérativement à partir de S(ng). Le nombre d´itérations nécessaires pour l´obtention de S(n) est borné, car le nombre d´états du graphe est fini. Plus précisement, pour les huit premiers opérateurs de la liste ci-dessus, la suite (Xk) est décroissante ; le nombre de pas de calcul est donc borné par le cardinal de X0 = S(ng). Pour les autres opérateurs temporels, la suite (Xk) est croissante, et atteint sa limite avant un nombre d´itérations égal au plus au cardinal de ^S(ng).

3.5. UN EXEMPLE D´EVALUATION

Nous illustrons dans ce paragraphe les différents principes que nous venons d´énoncer, en traitant l´exemple de l´exclusion mutuelle de DECKER (cf II.2.10.1). Nous présentons le graphe des états de l´application, nous indiquons les ensembles caractéristiques des prédicats non temporels associés aux actions étiquetées du programme de description, et nous détaillons le cheminement de l´évaluation d´une formule donnée.

1) Graphe des états

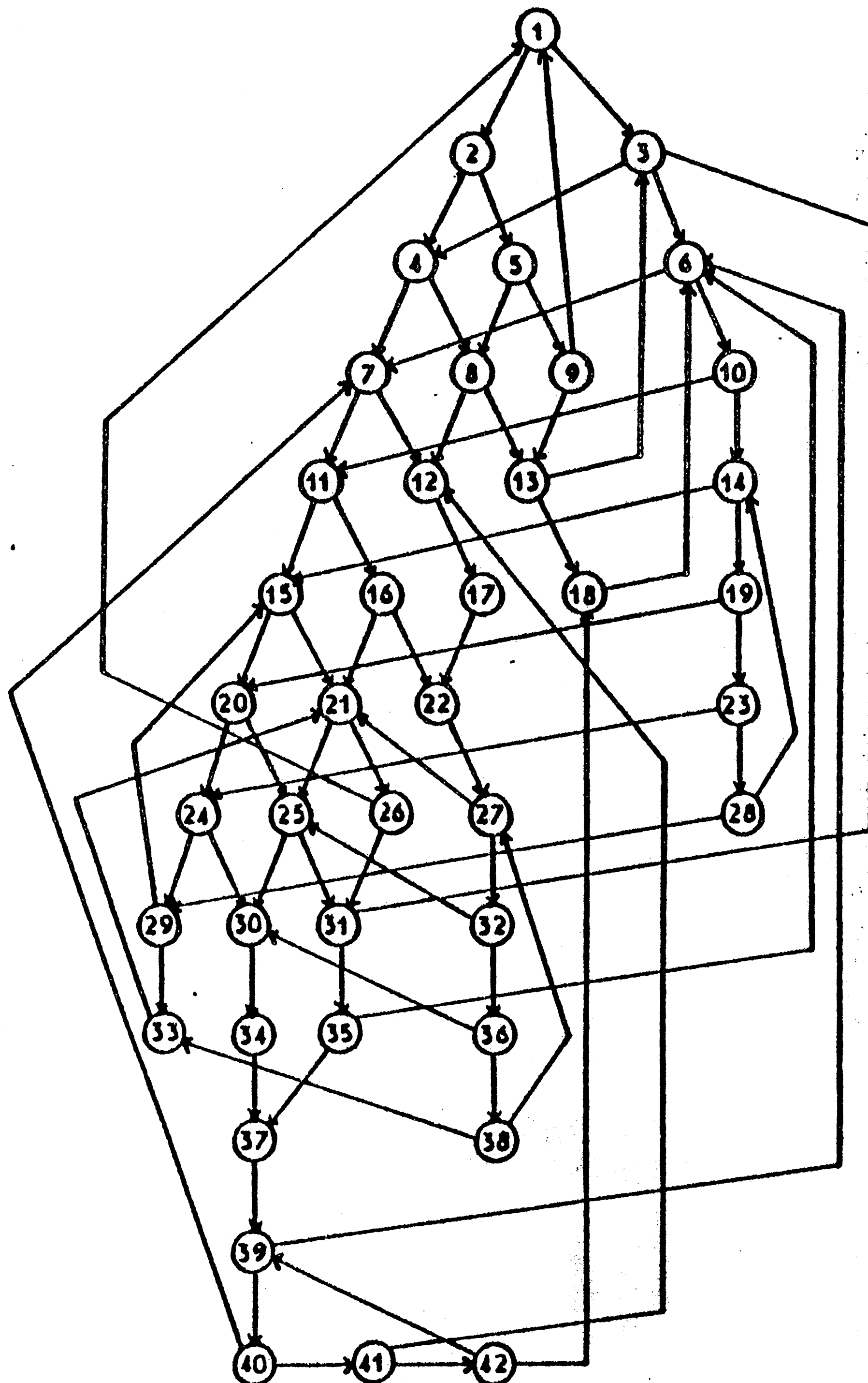
Le graphe d'états du système étudié est de petite taille. Il comporte 42 états, ce qui nous permet d'en faire une représentation graphique. Nous donnons tout d'abord la liste des états du graphe, en précisant pour chacun d'eux la valeur des variables de la description.

Etats initiaux du système : 1

état	y1	y2	c1	c2	x1	x2	turn	successeurs
1	0	0	1	1	0	0	1	2, 3
2	0	1	1	1	0	0	1	4, 5
3	1	0	1	1	0	0	1	4, 6
4	1	1	1	1	0	0	1	7, 8
5	0	2	1	0	0	1	1	8, 9
6	2	0	0	1	1	0	1	7, 10
7	2	1	0	1	1	0	1	11, 12
8	1	2	1	0	0	1	1	12, 13
9	0	3	1	0	0	2	1	1, 13
10	3	0	0	1	2	0	1	11, 14
11	3	1	0	1	2	0	1	15, 16
12	2	2	0	0	1	1	1	17
13	1	3	1	0	0	2	1	3, 18
14	0	0	1	1	0	0	0	15, 19
15	0	1	1	1	0	0	0	20, 21
16	3	2	0	0	2	1	1	21, 22
17	2	2	0	1	1	3	2	22
18	2	3	0	0	1	2	1	6
19	1	0	1	1	0	0	0	20, 23
20	1	1	1	1	0	0	0	24, 25
21	0	2	1	0	0	1	0	25, 26
22	3	2	0	1	2	3	1	27
23	2	0	0	1	1	0	0	24, 28
24	2	1	0	1	1	0	0	29, 30
25	1	2	1	0	0	1	0	30, 31
26	0	3	1	0	0	2	0	1, 31
27	0	2	1	1	0	3	0	21, 32
28	3	0	0	1	2	0	0	14, 29
29	3	1	0	1	2	0	0	15, 33
30	2	2	0	0	1	1	0	34
31	1	3	1	0	0	2	0	3, 35
32	1	2	1	1	0	3	0	25, 36
33	3	2	0	0	2	1	0	21
34	2	2	1	0	3	1	0	37
35	2	3	0	0	1	2	0	6, 37
36	2	2	0	1	1	3	0	30, 38
37	2	3	1	0	3	2	0	39
38	3	2	0	1	2	3	0	27, 33
39	2	0	1	1	3	0	1	6, 40
40	2	1	1	1	3	0	1	7, 41
41	2	2	1	0	3	1	1	12, 42
42	2	3	1	0	3	2	1	18, 39

Le graphe d'états est représenté par la figure suivante :

figure 4.2



2) Prédicats associés aux actions étiquetées :

Nous donnons pour chaque prédicat prédéfini de l'application l'ensemble des états qui vérifient le prédicat :

TOUT = [1..42]
ENABLE = [1..42] = TOUT
SINK = Ø (* pas d'état puits *)

ENABLE(P1) = [1..11,13..17,19..29,31..33,36,38]
ENABLE(P2) = [1..11,13..15,18..21,23..26]
u [28,29,31,34,35,37,39..42]
ENABLE(CONTROLER) = [2..13,15..42]
ENABLE(DECKER) = [1..42] = TOUT

ENABLE(dem_res1) = [3,4,8,13,19,20,25,31,32]
ENABLE(dem_res2) = [2,4,7,11,15,20,24,29,40]
ENABLE(aut_acces1) = [6,7,17,23,24,36]
ENABLE(aut_acces2) = [5,8,21,25,34,41]

AFTER(dem_res1) = [6,7,12,17,18,23,24,30,34..37,39..42]
AFTER(dem_res2) = [5,8,12,16,17,21,22,25,27,30]
u [32..34,36,38,41]
AFTER(aut_acces1) = [10,11,16,22,28,29,33,38]
AFTER(aut_acces2) = [9,13,18,26,31,35,37,42]

3) Evaluation d'une formule :

Nous présentons les différentes étapes de l'évaluation de la formule de spécification f =

[] AFTER(dem_res1) => INEV AFTER(aut_acces1)

qui exprime le fait qu'une demande d'accès à la ressource émise par le processus P1 est inévitablement satisfaite en un temps fini.

Les prédicats AFTER(dem_res1) et AFTER(aut_acces1) valent respectivement :

A = |AFTER(dem_res1)|
= [6,7,12,17,18,23,24,30,34..37,39..42]
B = |AFTER(aut_acces1)|
= [10,11,16,22,28,29,33,38]

calcul de INEV AFTER(aut acces1)

S0 = B = [10,11,16,22,28,29,33,38]
S1 = S0 u (pre n pretilda) (S0)
= S0 u [17]
S2 = S1 u [12]
S3 = S2 u [7]

S4 = S3 u [6]
S5 = S4 u [18]
S6 = S5

finalement,

C = |INEV AFTER(aut_acces1)|
= [6,7,10,11,12,16,17,18,22,28,29,33,38]

<u>calcul de g = AFTER(dem_res1) => INEV AFTER(aut_acces1)</u>

D = |AFTER(dem_res1) => INEV AFTER(aut_ACCES1)|
= (TOUT - A) u C
= [1..42] - [6,7,12,17,18,23,24,30,34..37,39..42]
u [6,7,10,11,12,16,17,18,22,28,29,33,38]
= [1..22,25..29,31..33,38]

<u>calcul de ALL(g)</u>

S0 = D = [1..22,25..29,31..33,38]
S1 = S0 n pretilda (S0)
= S0 - [19,20,25,31,32] = [1..18,21,22,26..29,33,38]
S2 = S1 - [14,15,21,26,27] = [1..13,16..18,22,28,29,33,38]
S3 = S2 - [10,11,16,22,28,29,33,38] = [1..9,12,13,17,18]
S4 = S3 - [7,17] = [1..6,8,9,12,13,18]
S5 = S4 - [4,6,12] = [1..3,5,8,9,13,18]
S6 = S5 - [2,3,8,18] = [1,5,9,13]
S7 = S6 - [1,5,13] = [9]
S8 = ∅

finalement,

E = |ALL(g)| = ∅

<u>calcul de la formule f</u>

F = |INIT => ALL(g)|
= (TOUT-INIT) u E
= [1..42] -[1] u ∅
= [2..42]

L'ensemble caractéristique F de la formule proposée f est différent de l'ensemble TOUT de tous les états du système étudié. La formule f n'est donc pas valide pour l'application décrite.

Nous reprenons le même calcul en remplaçant l'opérateur INEV par l'opérateur FAIR :

f = [] AFTER(dem_res1) => FAIR AFTER(aut_acces1)

et nous montrons que cette nouvelle formule est valide pour la des-

cription fournie.

Les prédicats AFTER(dem_res1) et AFTER(aut_acces1) valent respectivement :

```
A = |AFTER(dem_res1)|
  = [6,7,12,17,18,23,24,30,34..37,39..42]
B = |AFTER(aut_acces1)|
  = [10,11,16,22,28,29,33,38]
```

calcul de g = FAIR AFTER(aut acces1)

```
C = |FAIR AFTER(aut_acces1)|
  = |ALL[NOT AFTER(aut_acces1)] POT AFTER(aut_acces1)|
```

a) calcul de POT AFTER(aut acces1)

```
C1 = |POT AFTER(aut_acces1)|

S0 = |AFTER(aut_acces1) = B
   = [10,11,16,22,28,29,33,38]
S1 = S0 u pre (S0)
   = S0 u [6,7,17,23,24,36]
S2 = S1 u [3,4,12,18,19,20,32,35,39,40]
S3 = S2 u [1,2,8,13,14,15,27,31,37,41,42]
S4 = S3 u [5,9,25,26,34]
S5 = S4 u [21,30] = TOUT
```

finalement,

```
C1 = TOUT
```

b) calcul de ALL[NOT AFTER(aut acces1)] POT AFTER(aut acces1)

La formule argument de l'opérateur ALL ayant pour ensemble caractéristique l'ensemble TOUT, on en déduit immédiatement que la formule g a elle aussi pour ensemble caractéristique l'ensemble TOUT :

```
C = TOUT
```

Calcul de la formule f

L'obtention de l'ensemble des états qui vérifient la formule f est désormais élémentaire :

```
D = |AFTER(dem_res1) => FAIR AFTER(aut_acces1)|
  = (TOUT - A) u C
  = TOUT

E = |ALL(AFTER(dem_res1) => FAIR AFTER(aut_acces1))|
  = D = TOUT
```

F = |[] AFTER(dem_res1) => FAIR AFTER(aut_acces1)|
  = (TOUT - INIT) u E
  = TOUT

Le domaine de validité de la formule f est l'ensemble de tous les états du système, et par conséquent la formule est vérifiée par la description fournie.

Les calculs que nous venons de présenter confirment ainsi les résultats "intuitifs" que nous avions énoncés au chapitre III. La satisfaction d'une demande d'accès à la ressource par un processus n'est effective que si l'on considère les séquences d'exécution équitables de l'application.

4. LE CAS PARTICULIER DE LA LOGIQUE S4

4.1. PRESENTATION DU PROBLEME

Un problème permanent que nous avons rencontré dans l'élaboration de notre analyseur, a été la limitation mémoire imposée par notre calculateur lors de la construction du graphe des états d'une application. Il ne nous a jamais été possible de traiter des exmples convaincants, en partie pour des raisons de temps de calcul élevés, mais surtout pour des questions de saturation mémoire. Nous nous sommes alors penchés sur les possibilités de réduction du graphe d'états d'une application, en recherchant quelles étaient les règles de réduction applicables qui préservaient la validité des formules de spécification.

Nos conclusions n'ont pas été à la mesure de nos espérances, car nos recherches nous ont montré les points suivants :

- la réduction du graphe ne présente un intérêt que si elle peut se faire en cours de construction, afin d'économiser de la place mémoire, et de pouvoir travailler sur les graphes de systèmes plus importants ;
- l'évaluation d'une formule ne doit pas entraîner la reconstruction d'un graphe, même réduit, sous peine de remplacer les contraintes de place par des problèmes de temps ;
- des règles de réduction peuvent être exhibées, mais elles sont généralement propres à un opérateur donné, et ne préservent pas la validité des formules utilisant d'autres opérateurs. Elles s'avèrent donc inutilisables en pratique.

Nous avons cependant pu isoler le cas particulier des formules de la logique S4, qui permettent d'exprimer un ensemble important de formules de spécification.

La logique S4 est un sous ensemble de la logique CTL, construit à l'aide des opérateurs temporels unaires POT et ALL. Elle permet d'exprimer des propriétés telles que l'invariance ou l'atteignabilité, faible ou non.

La particularité de cette logique est que la valeur d'une formule temporelle ALLf ou POTf est la même pour tous les noeuds d'un graphe des états, qui figurent sur un circuit. Cette propriété, évidente intuitivement, est due au fait que tous les états d'un circuit sont équivalents modulo l'atteignabilité, ce qui entraine l'uniformité de la valeur d'une formule ALLf ou POTf sur le circuit. Malheureusement, cette propriété n'est pas vérifiée par un prédicat non temporel, dont la valeur peut être distincte d'un point à l'autre d'un même circuit.

Ainsi la possibilité de réduction du graphe d'états par compac-

tage des circuits, suggérée par la propriété des formules ALL et POT, ne peut pas être appliquée directement puisque les prédicats non temporels n'ont pas de valeur uniforme sur un circuit. Nous avons cependant trouvé un moyen de contourner cette difficulté, de telle sorte que l'évaluation des formules de S4 puisse être effectuée dans le graphe réduit des états du système de transitions.

La méthode est basée sur l'élaboration d'une interprétation trivaluée des formules de la logique S4, que nous allons expliciter.

Le graphe réduit du système est composé de deux sortes de noeuds : les états proprement dits (S), et les noeuds qui proviennent du compactage d'un circuit (S'). Pour une formule temporelle f de S4, on définit trois ensembles d'états :

V(f) = { s de S, tel que f(s) }
    u { s de S', tel que pour tout s' du circuit s, f(s') } ;

F(f) = V(^f) ;

M(f) = { s de S', tel que
        - il existe s' de s, f(s')
        - il existe s'' de s, ^f(s'') }.

Autrement dit, pour toute formule f de S4, nous partitionnons l'ensemble des états du graphe réduit en trois ensembles : le premier est constitué des états qui vérifient uniformément f, le second des états qui vérifient uniformément ^f, et le troisième est formé des états provenant d'un compactage tels que un des états du circuit au moins vérifie f, et un au moins vérifie ^f.

La donnée d'un ensemble de prédicats non temporels {p}, et la connaissance des ensembles V(p) et M(p). dans le graphe réduit d'un système de transitions, (obtenu par simple compactage des circuits du graphe d'états initial), est suffisante pour évaluer, dans le graphe réduit, toute formule de S4 formée à partir des prédicats non temporels p, et qui ne comporte pas d'opération booléenne entre deux prédicats non temporels.

La méthode d'évaluation des formules de spécification construites à partir de la logique temporelle S4 se décompose en deux étapes : définition d'un ensemble de prédicats non temporels et construction du graphe réduit des états comportant l'évaluation des ensembles V(p) et M(p) pour tout prédicat non temporel p, puis évaluation des formules. Nous présentons successivement ces deux étapes, et nous illustrons la méthode par un exemple.

4.2. CONSTRUCTION DU GRAPHE REDUIT DES ETATS

La construction du graphe réduit des états est liée à la définition d'un ensemble de prédicats non temporels utilisés dans les formules de spécification. En effet, les ensembles V(p), F(p) et M(p) sont constitués lors de la construction du graphe, ce qui impose de reconstruire le graphe dès que l'on augmente l'ensemble des prédicats non temporels.

Le graphe réduit des états est obtenu à partir du graphe des états par compactage des états qui figurent sur un circuit.

Soit C= { s1,...,sp } un circuit. On substitue l'ensemble C par un état c tel que pre (c) = pre (s1) u ... u pre (sp).

Pour tout prédicat non temporel p, on met à jour les ensembles V(p), F(p) et M(p) par application des règles suivantes :

- si p est uniformément vrai (resp. faux) en tout point du circuit, on remplace dans V(p) (resp. F(p)) l'ensemble C par l'état c ;
- sinon, on retranche à V(p) et F(p) les éléments de C qui y figurent, et l'on ajoute l'état c à l'ensemble M(p).

La difficulté majeure provient du fait que pour gagner de la place mémoire, la réduction doit être faite au fur et à mesure de la constitution de circuits lors de la génération du graphe. Cela implique une recherche permanente des composantes fortement connexes nouvellement formées, et l'application d'une phase de récupération de la place mémoire occupée par les états compactés. L'algorithme de construction se révèle ainsi assez lent, mais permet une augmentation importante de capacité de l'analyseur, car la majorité des systèmes étudiés sont constitués de processus cycliques dont la composition contient elle aussi de nombreux circuits.

4.3. EVALUATION DES FORMULES

La méthode d'évaluation des formules de spécification dans le graphe réduit des états du système de transitions est semblable à celle que nous avons présentée au paragraphe précédent. Seul le calcul des opérateurs, booléens ou temporels, se trouve modifié par l'interprétation trivaluée des formules.

Les formules f de la logique S4 sont représentées par leur graphe DAG(f) et leur validité est décidée par comparaison de leur ensemble V(f) avec l'ensemble TOUT de tous les états du graphe réduit. Si

V(f) est égal à TOUT, la formule f est dite valide dans le système, elle est dite non valide sinon.

Le principe d'évaluation d'une formule f consiste à effectuer un parcours postfixé du graphe DAG(f), en calculant pour chaque noeud n les deux ensembles V(f(n)) et M(f(n)), à partir des ensembles V(f(ng)), V(f(nd)), M(f(ng)) et M(f(nd)) associés aux noeuds ng et nd fils de n, et de la relation d'accessibilité inverse pre du graphe réduit des états. Nous rappelons l'évaluation des prédicats non temporels, et nous présentons pour chaque opérateur de la logique la règle de calcul itératif de l'opérateur.

4.3.1. EVALUATION DES PREDICATS NON TEMPORELS

L'évaluation des prédicats non temporels du système est faite lors de la génération du graphe réduit des états, par l'intermédiaire du calcul, pour chaque prédicat p, des deux ensembles V(p) et M(p).

Les prédicats non temporels prédéfinis vérifient les propriétés suivantes :

(S est l'ensemble des états proprement dits,
S' est l'ensemble des états provenant d'un compactage,
TOUT est l'ensemble de tous les états du graphe réduit)

TOUT = S u S'
M(TOUT) = ∅
V(TOUT) = S u S' ;

M(SINK) = ∅
V(SINK) inclus dans S ;

M(ENABLE) = ∅
V(ENABLE) contient S'
V(SINK) u V(ENABLE) = TOUT.

4.3.2. CALCUL ITERATIF DES OPERATEURS

Le calcul des opérateurs booléens se fait à l'aide d'opérations ensemblistes :

f = NOT g :
V(f) = V(^g) = TOUT - V(g) - M(g)
M(f) = M(g) ;

f = g AND h :
  V(f) = V(g) n V(h)
  M(f) = V(g) n M(h) u M(g) n V(h) ;

f = g OR h :
  V(f) = V(g) u V(h)
  M(f) = V(^g) n M(h) u M(g) n V(^h) ;

f = g => h :
  V(f) = V(^g) u V(h)
  M(f) = V(g) n M(h) u M(g) n V(^h).

Les règles ci-dessus ne sont valables que lorsque les arguments des opérateurs binaires ne sont pas tous les deux des prédicats non temporels. Le calcul de M(f) est possible dès que g (ou h) est une formule temporelle, car l'ensemble M(g) (ou M(h)) est alors vide. On comprend très aisément qu'il est impossible, par exemple, de calculer pour deux prédicats non temporels p1 et p2 l'ensemble M(p1 OR p2) a l'aide de M(p1) et M(p2), puisqu'on a perdu la valeur des états qui ont été compactés.

On impose ainsi à l'utilisateur de ne pas employer d'opération booléenne directe entre deux prédicats non temporels. L'utilisateur peut remédier à cette contrainte en définissant explicitement dans son ensemble de prédicats non temporels, le prédicat produit de l'opération.

Les règles concernant les deux opérateurs temporels POT et ALL sont les suivantes :

V(POT f) = [pre]* (V(f) u M(f))
M(POT f) = Ø ;

V(ALL f) = [pretilda]x (V(f))
M(ALL f) = Ø .

Les ensembles V(POT f) et V(ALL f) sont calculés itérativement comme les limites des suites monotones suivantes :

POT f : Xk+1 = Xk u pre Xk

en initialisant X0 = V(f) u M(f)
= TOUT - V(^f) ;

ALL f : Xk+1 = Xk n pretilda Xk

en initialisant X0 = V(f).

Le nombre d'itérations est borné car le nombre d'états du graphe réduit est fini. Pour l'opérateur ALL, la suite (Xk) est décroissante et le nombre de pas de calcul est borné par le cardinal de l'ensemble V(f), pour l'opérateur POT, la suite (Xk) est croissante et le nombre d'itérations est borné par le cardinal de l'ensemble V(^f).

Nous allons désormais appliquer cette méthode d'analyse à l'exemple que nous avons déjà traité au paragraphe précédent, afin de comparer les résultats produits par les deux formes d'évaluation.

4.4. UN EXEMPLE D'EVALUATION

Nous reprenons l'exemple de l'exclusion mutuelle de DECKER, décrit au chapitre II de la thèse.

Le graphe des états de l'application, présenté au paragraphe précédent, comporte 42 états et possède la particularité d'être un graphe fortement connexe. Le graphe réduit des états n'est donc formé que d'un seul état, que nous notons E, provenant du compactage des 42 états du graphe initial :

TOUT = E
E = [1..42]
pre(E) = pretilda(E) = E.

Les prédicats prédéfinis du système sont caractérisés par les égalités suivantes :

TOUT = E
V(ENABLE) = E, M(ENABLE) = Ø
V(SINK) = Ø, M(SINK) = Ø
V(INIT) = Ø, M(INIT) = E

V(ENABLE(P1)) = V(ENABLE(P2)) = V(ENABLE(CONTROLER) = Ø
M(ENABLE(P1)) = M(ENABLE(P2)) = M(ENABLE(CONTROLER) = E

V(mutuelle_exclusion) = E

pour chaque étiquette ea élément de
{dem_res1, dem_res2, aut_acces1, aut_acces2},

V(ENABLE(ea)) = V(AFTER(ea) = Ø
M(ENABLE(ea)) = M(AFTER(ea) = E

Il ne nous est pas possible d'évaluer les deux formules traitées au chapitre précédent, car elles utilisent des opérateurs qui n'appartiennent pas à la logique S4. Nous pouvons cependant prouver

la validité des formules suivantes :

```
[] ENABLE
[] POT ENABLE(aut_acces1)
[] POT ENABLE(aut_acces2)
[] mutuelle_exclusion
```

<u>Calcul de [] ENABLE</u>

a) <u>calcul de ALL ENABLE</u>

```
S0 = V(ENABLE) = E
S1 = S0 n pretilda S1 = E
   = S0
```

b) <u>calcul de [] ENABLE</u>

```
V([] ENABLE) = V(^INIT) u V(ALL ENABLE)
             = Ø u E
             = TOUT
```

L'ensemble caractéristique de la formule [] ENABLE est l'ensemble de tous les états du graphe réduit ; la formule [] ENABLE est donc valide pour la description fournie.

<u>Calcul de [] POT ENABLE(aut acces1)</u>

a) <u>calcul de POT ENABLE(aut acces1)</u>

```
S0 = V(ENABLE(aut_acces1)) u M(ENABLE(aut_acces1))
   = E
S1 = pre S0 u S0 = E
   = S0
```

finalement,

```
V(POT ENABLE(aut_acces1) = TOUT
```

b) <u>calcul de [] POT ENABLE(aut acces1)</u>

```
V(ALL POT ENABLE(aut_acces1) = TOUT
V( [] POT ENABLE(aut_acces1) = Ø u TOUT = TOUT
```

La formule [] POT ENABLE(aut_acces1) est donc valide pour la description fournie.

<u>Calcul de [] mutuelle exclusion</u>

```
V(mutuelle_exclusion) = E = TOUT
V(ALL mutuelle_exclusion) = E = TOUT
V( [] mutuelle_exclusion) = TOUT
```

La formule [] mutuelle_exclusion est valide pour l'application étudiée, ce qui garantit que les deux processus n'accèdent pas simultanément à la ressource.

Remarque :

L'exemple étudié est tel que les calculs d'évaluation dans le graphe réduit sont triviaux, et c'est en partie pour cette raison que nous l'avons choisi. Il traduit néanmoins très bien le fait que de nombreuses applications que l'on peut analyser par QUASAR fonctionnent de façon cyclique, et possèdent un graphe réduit dont la taille peut être très petite devant celle du graphe d'états initial. La différence, notre exemple le prouve, est parfois considérable, et permet de faire converger les calculs itératifs bien plus rapidement.

La puissance de spécification offerte est cependant trop faible pour donner un crédit suffisant à la méthode.

5. EVALUATION DANS LE TREILLIS DES PREDICATS

5.1. PRESENTATION DE LA METHODE

La méthode d'évaluation de formules de spécification que nous allons présenter diffère de la précédente par le fait que l'ensemble des états effectivement atteints par le système n'est pas connu. Le système de transitions n'est pas généré, et l'analyse porte directement sur le CA-système produit par la traduction d'un programme de description. L'interprétation d'une formule temporelle est celle qui a été définie précédemment, mais son ensemble caractéristique est représenté par une fonction booléenne dont le domaine de validité est l'ensemble des états du système qui vérifient la formule.

La première phase de l'analyse consiste à coder le CA-système à variables entières bornées en un CA-système à variables booléennes qui exprime un fonctionnement identique de l'application décrite. Les prédicats non temporels du système peuvent alors s'exprimer sous forme de fonctions booléennes des variables du CA-système booléen. La donnée du transformateur de prédicats pre permet ensuite de calculer itérativement les opérateurs temporels. La validité d'une formule de spécification sera decidée par comparaison de sa fonction caractéristique avec la fonction identiquement vraie.

Nous présentons d'abord la phase de codage d'un CA-système booléen, énoncée dans [MAY81]. Nous rappelons ensuite quelques résultats du calcul booléen qui serviront de support à la méthode d'évaluation. Nous explicitons finalement les règles de calcul des opérateurs temporels, et appliquons la méthode à un exemple.

5.2. CODAGE D'UN CA-SYSTEME ENTIER

5.2.1. CHOIX DU CODAGE

L'application à étudier est décrite par un CA-système borné plat, comportant un ensemble fini de transitions et un ensemble fini de variables entières bornées. Notre but est de traduire ce CA-système vers un CA-système qui lui est équivalent, mais dont les variables sont de type booléen. Cette transformation - ou codage du CA-système - revient à définir une représentation des variables entières à l'aide de variables booléennes, puis à donner les règles de traduction des transitions du CA-système entier en transitions du CA-système booléen.

Parmi les différents codages possibles d'une variable entière bornée, deux d'entre eux ont retenu notre attention : le codage 1 parmi n, et le codage logarithmique - ou codage optimal -.

Le codage 1 parmi n consiste à représenter une variable X de domaine de définition I(X) = (X1,X2,...,Xn) = [Xinf..Xsup] à l'aide de n variables booléennes x1,x2,...,xn, chaque variable xi correspondant à la valeur possible Xi de la variable entière X. Ce codage est manifestement coûteux, mais présente l'avantage non négligeable d'être très simple.

Le codage optimal consiste à représenter la même variable X par un nombre minimum de variables booléennes, ce nombre valant [log2(n)] ou [log2(n)]+1 suivant que n est une puissance de 2 ou non. Ce codage économise un nombre important de variables, mais présente deux inconvénients majeurs. Il est très peu lisible et il complique considérablement l'écriture des expressions booléennes. Prenons pour exemple l'expression élémentaire X=Xk. Le codage 1 parmi n permet de représenter cette expression par la variable xk, tandis que le codage logarithmique représente la condition par (k1.x1). ... .(kp.xp), où (k1,...,kp) et (x1,...,xp) sont les codages respectifs de Xk et de X.

Nous avons opté pour la simplicité et avons choisi le codage 1 parmi n. Nous définissons le codage par un homomorphisme F=(Fx,Fi,Fc,Fa) que nous représenterons schématiquement ainsi :

```
  | X          |     | Fx(X)             |
  | init(X)    |     | Fi(init(X))       |
F | do         |  =  | do                |
  |   Ci : Ai  |     |   Fc(Ci) : Fa(Ai) |
  | od         |     | od                |
```

5.2.2. RESTRICTIONS ENGENDREES PAR LE CODAGE

Les règles de codage que nous allons présenter ne permettent pas de traduire n'importe quelle condition booléenne sur des variables entières. Nous devons nous restreindre aux CA-systèmes dont les conditions et actions vérifient la syntaxe suivante :

(les notations utilisées sont :

- OU pour la disjonction de deux expressions booléennes,
- ET pour la conjonction de deux expressions booléennes)

- toute condition Ci s'écrit OU ET Cij , où Cij est l'un des termes suivants :

(x,y sont des variables de X , k est une constante entière)

x>=k , x<=k , x=k ,
x>=y , x<=y , x=y ,
x>=y+k , x<=y+k , x=y+k ,
x>=y-k , x<=y-k , x=y-k.

- toute action Ai est de la forme Ai = Ai1,...,Aip , où Aij est l'un des termes suivants :

(x,y,z sont des variables de X , k et n des constantes entières)

x:=k ,
x:=y ,
x:=y+k , x:=y-k ,
x:=y+z , x:=y-z ,
x:=(y+k) mod n , x:=(y-k) mod n.

Ces restrictions sont bien évidemment très contreignantes, mais elles sont obligatoires puisque nous ne savons pas traduire de façon simple les termes plus complexes. Nous présentons désormais l'homomorphisme F, par l'intermédiaire de ses quatre composantes Fx, Fi, Fc et Fa.

5.2.3. CODAGE DES VARIABLES

Chaque variable entière X de domaine de définition [X_inf..X_sup] est représentée par X_sup-X_inf+1 variables booléennes x_inf, x_inf+1, ... , x_sup.

Remarquons que le codage entraine une perte d'information, dans la mesure où l'on oublie que la variable X ne peut avoir qu'une seule valeur à un instant donné. Nous pouvons récupérer cette information en intégrant au CA-système booléen ce qu'on appelle un invariant de processus, la relation qui exprime le fait qu'une seule des variables booléennes de {x_inf,...,x_sup} vaut 1 quelque soit l'état dans lequel le système se trouve. Cette condition s'écrit ainsi :

$$\underset{(i=inf,..,sup)}{\text{OU}} \; x_i \text{ ET } (\underset{(j<>i)}{\text{ET}} \; \hat{} x_j) = 1.$$

L'utilisation des invariants de processus dans l'évaluation des formules de spécification n'est pas indispensable, mais elle s'avère très utile en pratique car elle permet de faire converger les algorithmes de calcul de fonctions booléennes beaucoup plus rapidement.

5.2.4. CODAGE DES CONDITIONS

Soit Ci = OU ET Cij une condition sur les variables du CA-système entier. Fc est défini de telle sorte que Fc(OU ET Cij) = OU ET Fc(Cij), ce qui permet de limiter la définition de Fc aux conditions élémentaires Cij. Nous donnons successivement pour chaque terme Cij la valeur de Fc(Cij).

codage de x>=k

```
si k<=x_inf          ,  Fc(x>=k) = vrai

si x_inf<k<=x_sup    ,
   Fc(x>=k) = x_k OU x_k+1 OU ... OU x_sup

si x_sup<k           ,  Fc(x>=k) = faux ;
```

codage de x<=k

```
si k>=x_sup          ,  Fc(x<=k) = vrai

si x_inf<=k<x_sup    ,
   Fc(x<=k) = x_inf OU x_inf+1 OU ... OU x_k

si x_inf>k           ,  Fc(x<=k) = faux ;
```

codage de x=k

```
si k<x_inf ou k>x_sup  ,  Fc(x=k) = faux
sinon                  ,  Fc(x=k) = x_k ;
```

codage de x>=y

```
si x_sup<y_inf          ,  Fc(x>=y) = faux
sinon si x_inf>=y_sup   ,  Fc(x>=y) = vrai
sinon
  Fc(x>=y) =

     x_max(x_inf,y_inf)
       ET (y_inf OU...OU y_max(x_inf,y_inf)

 OU  x_[max(x_inf,y_inf)+1]
       ET (y_inf OU...OU y_[max(x_inf,y_inf)+1])
 ...
 ...
 OU  x_sup
```

```
        ET (y_inf OU...OU y_min(x_sup,y_sup)) ;
```

<u>codage de x<=y</u>

```
si x_sup<=y_inf          , Fc(x<=y) = vrai
sinon si y_sup<x_inf     , Fc(x<=y) = faux
sinon
  Fc(x<=y) =

     x_max(x_inf,y_inf)
      ET (y_max(x_inf,y_inf) OU...OU y_sup)

 OU  x_[max(x_inf,y_inf)+1]
      ET (y_[max(x_inf,y_inf)+1] OU...OU y_sup)
 ...
 ...
 OU  x_min(x_sup,y_sup)
      ET (y_min(x_sup,y_sup) OU...OU y_sup) ;
```

<u>codage de x=y</u>

```
si x_sup<y_inf ou y_sup<x_inf   , Fc(x=y) = faux
sinon
  Fc(x=y) =      x_max(x_inf,y_inf)
                   ET  y_max(x_inf,y_inf)

             OU  x_[max(x_inf,y_inf)+1]
                   ET  y_[max(x_inf,y_inf)+1]
             ...
             ...
             OU  x_min(x_sup,y_sup)
                   ET  y_min(x_sup,y_sup).
```

Le codage des conditions x>=y+k , x>=y-k , x<=y+k , x<=y-k , x=y+k et x=y-k est obtenu à l'aide des règles précédentes en considérant les variables y+k de domaine de définition [y_inf+k..y_sup+k], et y-k de domaine [y_inf-k..y_sup-k].

5.2.5. <u>CODAGE DES ACTIONS</u>

On définit l'homomorphisme Fa de telle sorte que l'on puisse calculer l'image d'une action composée à partir des images des actions élémentaires :

Fa(Ai1, ..., Ain) = Fa(Ai1), ..., Fa(Ain).

On introduit une variable booléenne erreur, mise à 1 lorsqu'un débordement d'une variable, non détectable statiquement, se produit. Nous donnons successivement les règles de codage pour chaque type d'action élémentaire :

codage de x:=k

```
si k=x_c dans [x_inf..x_sup]
      Fa(x:=k) =
    x_c:=1
    pour tout i de [inf..sup] , i<>c , x_i:=0 ;
```

codage de x:=y

```
si I(y) est inclu(egal) dans I(x)
      Fa(x:=y) =
    pour tout i de I(y)       , x_i:=y_i
    pour tout i de I(x)-I(y) , x_i:=0
sinon
      F(c : x:=y ) =
    Fc(c) ET Fc(y<x_inf OU y>x_sup)
             : erreur:=1
    Fc(c) ET Fc(y>=x_inf ET y<=x_sup)
    : pour tout k de I(x) ET I(y)       , x_k:=y_k
      pour tout k hors de I(x) ET I(y) , x_k:=0.
```

Le codage des actions x:=y+k, x:=y-k s'effectue à partir du précédent en posant z=y+k ou z=y-k.

codage de x:=y+z

Deux cas sont à envisager :

1) [y_inf+z_inf..y_sup+z_sup] est inclu (égal) dans I(x)

Pour traiter ce cas, nous nous ramenons au cas précédent en supposant que z est une constante. Nous sommes amenés à construire z_sup-z_inf+1 commandes gardées pour examiner toutes les valeurs possibles de z. Remarquons qu'il est avantageux de choisir pour constante, parmi y ou z, la variable dont le domaine de définition est le plus petit, afin de rajouter le moins possible de commandes gardées. Nous supposerons qu'il s'agit de z.

```
F( c : a,x:=y+z ) =
```

```
Fc(c) ET z_inf
        : a , pour tout i de I(x) ET I(y)
                 x_[i+z_inf]:=y_i ,
              pour tout i hors de I(x) ET I(y)
                 x_i:=0
| ...
| ... |
Fc(c) ET z_sup
        : a , pour tout i de I(x) ET I(y)
                 x_[i+z_sup]:=y_i ,
              pour tout i hors de I(x) ET I(y)
                 x_i:=0 ;
```

2) [y_inf+z_inf..y_sup+z_sup] non inclu dans I(x)

```
F( c : a,x:=y+z ) =

    Fc(c) ET z_inf
         ET Fc(y<=x_sup-z_inf ET y>=x_inf-z_inf)
       : a , pour tout i de I(x) ET I(y)
                x_[i+z_inf]:=y_i
             pour tout i hors de I(x) ET I(y)
                x_i:=0                                    |
    Fc(c) ET z_inf
         ET Fc(y>x_sup-z_inf  OU y<x_inf-z_inf)
       : erreur:=1
    |...
    |...|
    Fc(c) ET z_sup
         ET Fc(y<=x_sup-z_inf ET y>=x_inf-z_inf)
       : a , pour tout i de I(x) ET I(y)
                x_[i+z_sup]:=y_i
             pour tout i hors de I(x) ET I(y)
                x_i:=0                                    |
    Fc(c) ET z_sup
         ET Fc(y>x_sup-z_inf  OU y<x_inf-z_inf)
       :  erreur:=1 ;
```

<u>codage de x:=(y+k) mod n</u>

On considère que k<n. Si k>=n, ce qui se détermine statiquement, on applique la règle (y+k) mod n = (y+(k mod n)) mod n.

```
si x_inf<=0 ET x_sup>=n-1
   Fa(x:=(y+k) mod n) =

       pour tout i de [0,n-k[ , x_[i+k]:=y_i
       pour tout i de [-k,0[  , x_[i+k]:=y_[i+n]
       pour tout i hors de [0,n[ , x_i:=0
```

```
sinon
    ( posons A =   (y>x_sup-k ET y<n-k)
                OU (y>x_sup-k+n ET y>=n-k)
                OU (y<x_inf-k+n ET y>=n-k)
                OU ( y<x_inf-k ET y<n-k) )

  Fa(x:=(y+k) mod n) =

  Fc(c) ET Fc(A)  : erreur:=1                    |
  Fc(c) ET ^Fc(A) :
    pour tout i de [0,n-k[ , x_[i+k]:=y_i
    pour tout i de [-k,0[  , x_[i+k]:=y_[i+n]
    pour tout i hors de [0,n[ , x_i:=0.
```

Nous n'avons pas décrit explicitement l'homomorphisme Fi. On remarque que Fi s'exprime directement à partir de Fa car l'instruction d'initialisation n'est qu'un cas particulier des actions.

La phase de codage d'un CA-système entier en un CA-système booléen est complexe, coûteuse et produit des systèmes de taille importante. Elle est néanmoins indispensable pour mener à bien l'analyse du système. Nous présentons maintenant la méthode d'évaluation proprement dite.

5.3. RAPPELS SUR LE CALCUL BOOLEEN

Le but de ce paragraphe n'est pas de présenter formellement des résultats, mais simplement d'énoncer quelques rappels de notations et de théorèmes qui faciliterons la compréhension des prochains paragraphes.

Soit (x1,x2,...,xn) n variables booléennes, et soit f une fonction booléenne des variables x1,...,xn. Le domaine de validité de f est l'ensemble des valeurs (y1,...,yn) du vecteur x1,...,xn pour lesquelles la fonction est vraie.

On représente généralement le domaine de définition de f par un hypercube de dimension n, dont chaque sommet est une valeur du vecteur x1,...,xn ; le domaine de validité de f est alors formé de tous les sommets pour lesquels la fonction est vraie.

La façon la plus élémentaire de représenter f est de considérer la somme des monômes élémentaires formés par les points de l'hypercube où f est vraie. Cette écriture est cependant très volumineuse et peut se simplifier par regroupement des points adjacents du cube. Différentes méthodes de simplification peuvent être appliquées, mais elles ne conduisent pas forcément à un résultat unique. Il est pourtant souhaitable, si l'on désire comparer des fonctions booléennes sans énumérer leurs domaines de validité, de disposer d'une représentation unique des fonctions.

Cette représentation existe, et s'appelle forme canonique des fonctions booléennes. La transformation d'une fonction en sa forme canonique s'obtient par la règle de double complémentation que nous énonçons sous forme d'algorithme :

1) complémenter f ;
2) simplifier f autant que possible par la règle d'absorbtion
   a.x + a -> a ;
3) complémenter f ;
4) simplifier f autant que possible par la règle
   a.x + a -> a.

Les règles usuelles de calcul seront utilisées : le produit (.), la somme (+) et la complémentation (^) de monômes, qui vérifient les propriétés suivantes :

^^m = m (m, m1, m2, m3 monômes)
m+m = m
m+^m = vrai
m.^m = faux
m1+m2 = m2+m1
m1.m2 = m2.m1
m1.(m2+m3) = m1.m2+m1.m3
^(m1+m2) = ^m1.^m2
^(m1.m2) = ^m1+^m2.

5.4. EVALUATION DES PREDICATS NON TEMPORELS

Un prédicat non temporel est une condition booléenne sur les variables de la description du système. En codant le CA-système entier, les variables du programme de description ont perdu leur signification propre puisqu'elles ont été codées. Il faut donc, pour exprimer les prédicats non temporels au sein du CA-système booléen, traduire les conditions sur ces variables en des conditions sur les variables introduites par le codage. Les expressions ainsi obtenues seront des fonctions booléennes de variables booléennes que l'on pourra représenter par leur forme canonique sous forme de somme de monômes.

La traduction d'une condition sur les variables entières a été vue lors du codage d'un CA-système entier.

On note TOUT,INIT,SINK,ENABLE les prédicats prédéfinis qui caractérisent respectivement les ensembles suivants :

- l'ensemble de toutes les valuations possibles des variables booléennes (TOUT est le prédicat identiquement vrai) ;

- l'ensemble de toutes les valuations initiales possibles des variables booléennes ;

Soit X1:=X1_k, ..., Xp:=Xp_k l'instruction facultative d'initialisation des variables du CA-système entier. Initialiser une variable Xi correspond dans le CA-système booléen à affecter entièrement l'ensemble des variables xi_j du codage de Xi : l'une d'entre elles (xi_k) prend la valeur 1, toutes les autres sont mises à 0. Initialiser Xi revient donc à restreindre l'ensemble des valuations possibles par la contrainte (xi_k:=1, et pour tout j<>k , xi_j:=0). On en déduit immédiatement la valeur du prédicat INIT :

$$INIT = \bigcap_{Xi \text{ initialisée}} (xi_k \cap \bigcap_{j <> k} \neg xi_j) ;$$

Remarquons que si aucune variable n'est initialisée, alors INIT est égal au prédicat identiquement vrai TOUT ;

- l'ensemble des valuations des variables booléenes à partir desquelles le système ne peut plus évoluer, c'est à dire à partir desquelles aucune commande gardée n'est éxécutable. Ce blocage s'exprime immédiatement par le prédicat

$$SINK = \bigcap_{ti = (ci:ai)} \neg ci ;$$

- l'ensemble des valuations des variables booléennes qui valident au moins une commande gardée, c'est à dire l'ensemble des états à partir desquels le système peut évoluer. On note que ENABLE = ^SINK.

Nous allons désormais définir les règles de calcul des opérateurs temporels par calcul itératif de prédicats.

5.5. EVALUATION DES FORMULES TEMPORELLES

La méthode d'évaluation des formules est analogue à celle qui est effectuée dans le graphe des états, mais l'interprétation d'une formule est représentée par une fonction booléenne dont le domaine de validité est l'ensemble des états qui vérifient la formule.

Les formules f sont à nouveau représentées par leur graphe DAG(f), et leur évaluation consiste en un parcours postfixé du graphe DAG(f). Pour tout noeud n, on construit la fonction booléenne p(n) caractérisant la sous formule f(n) de f, à partir des fonctions p(ng) et p(nd) caractérisant les formules f(ng) et f(nd) associées aux noeuds ng et nd, fils de n.

La particularité de la méthode repose sur le calcul du transformateur pre. L'expression de pre a été présentée dans [SIF79]. Nous en rappelons la formulation, avant d'énoncer les règles de calcul des opérateurs de CTL.

5.5.1. CALCUL DU TRANSFORMATEUR DE PREDICATS PRE

Soit p(y) un prédicat booléen sur les variables du CA-système booléen, de domaine de validité S(p) (S(p) est l'ensemble des états s du CA-système qui vérifient le prédicat : p(s) = vrai). Soit s un état. On définit pr(s) l'ensemble des états prédécesseurs directs possibles de s. On définit par extension la fonction d'ensembles pr(S) qui associe à un ensemble d'états S l'ensemble des états prédécesseurs directs des éléments de S.

Le transformateur de prédicats pre associe à un prédicat booléen p(y) le prédicat p'(y) = pre(p) dont le domaine de validité est S(p') = pr (S(p)).

Pour déterminer la valeur de pre, il suffit d'examiner attentivement les effets de la mise à feu des commandes gardées du CA-système :

soit ti = (ci,ai) une transition du CA-système, et soit s une valeur du vecteur X des variables :

- ti est exécutable à partir de l'état s si et seulement si la condition ci(s) est vraie ;

- l'action effectuée lors de la mise à feu de ti se traduit par une modification de la valeur de X, et peut être représentée par une affectation vectorielle de X :

X := ai(X).

Un prédicat p(y) peut devenir vrai du fait de la mise à feu de ti si et seulement si ci(y) est vraie et p(ai(y)) également. D'où l'expression du transformateur de prédicats pre :

$$pre(p(y)) = \bigcup_{ti = (ci : ai)} (ci(y) \cap p(ai(y))).$$

Nous allons expliciter désormais, pour un prédicat p(y) et une action ai, le calcul du prédicat p(ai(y)).

Soit p(y) un prédicat booléen. Nous supposons que p est mis sous forme de somme de monômes p = m1+m2+ ... +mk.
La distributivité de pre par rapport à l'union nous permet de

calculer pre(p) sous la forme pre(m1)+ ... +pre(mk). Aussi limiterons nous notre énoncé au calcul du prédicat m(ai), où m est un monôme.

Soit m un monôme. Le domaine de validité S(m´) de m´=m(ai) est l´ensemble des états s´ tels que ai(s´) appartient au domaine de validité S(m) de m. L´ensemble S(m´) est donc égal à l´image inverse de S(m) par la fonction ai, et nous noterons S(m´) = (ai)-1 (S(m)).

Un monôme m s´écrit :

$$m = x1.x2. \ \ldots \ .xj \ . \ \hat{}y1.\hat{}y2. \ \ldots \ .\hat{}yk,$$

où les xi et yi sont des variables booléennes du CA-système ; m+ = {x1,...,xj} est l´ensemble des variables positives, et m- = {y1,...,yk} est l´ensemble des variables négatives du monôme m.

L´obtention de m´ s´effectue à partir de m par substitution dans l´écriture de m de chaque variable x de m+ u m- par (ai)-1(x).

Nous énoncons les différentes règles de calcul des valeurs (ai)-1(x).

Soit ti = (ci,ai) une commande gardée du CA-système booléen. L´action ai consiste en un ensemble d´affectations simultanées de la forme Xj:=0, Xj:=1 ou Xj:=Xk.

<u>On note :</u>

- VG = l´ensemble des variables des parties gauches de ai ;
- M1 = l´ensemble des variables mises à 1 par ai ;
- M0 = l´ensemble des variables mises à 0 par ai ;
- MG = l´ensemble des variables affectées par d´autres variables dans ai ;
- MD = l´ensemble des variables affectées à d´autres variables dans ai ;
- VM = l´ensemble des variables présentes dans le monôme (VM = m+ u m-).

L´obtention du prédicat m´ = m(ai) est déterminée par l´expression suivante :

Si (VG n VM) est vide alors m(ai) = m

sinon

si (M0 n m+ est non vide)
ou (M1 n m- est non vide)
ou (MD n m- est non vide)
alors m(ai) = faux

sinon m(ai) = (m1+,m1-)

(où ml+ = (m+ - M1) [MG/MD]
et ml- = (m- - M0) [MG/MD]).

Pour calculer ml+ (ou ml-), on retranche à m+ (ou m-) l'ensemble M1 (ou M0) et l' on substitue dans ce nouvel ensemble toute occurrence d'une variable de MG par la variable correspondante dans MD.

Prenons pour exemple le monôme m = x1.x2.^x3.

- pour ai = x1:=1, on a m(ai) = x2.^x3 ;
- pour ai = x1:=0, on a m(ai) = faux, car x1 est présente positive dans m ;
- pour ai = x1:=1,x2:=1,x3:=0, on a m(ai) = vrai ;
- pour ai = x1:=x4,x3:=x5, on a m(ai) = x4.x2.^x5.

A partir des règles ci-dessus, il suffit d'effectuer des opérations d'union et d'intersection pour obtenir la valeur pre(p) pour un prédicat p quelconque.
En pratique, le transformateur pre sera représenté par une procédure, admettant pour paramètre d'entrée un prédicat f, et qui produit pour résultat le prédicat pre(p). Remarquons que pretilda, dual de pre, s'obtient aisément à partir de pre puisque pretilda(p) = ^pre(^p).

Munis d'une méthode de calcul de la fonction pre, nous pouvons désormais énoncer les règles d'évaluation des formules de la logique CTL.

5.5.2. CALCUL ITERATIF DES OPERATEURS

Les règles d'évaluation des opérateurs sont identiques à celles présentées pour la méthode d'évaluation dans le graphe des états d'une application (cf IV.3.4.3). Il suffit de remplacer les opérations ensemblistes d'union, d'intersection et de complémentation par les opérations booléennes de somme, de produit et de négation.

Lors de l'évaluation d'un opérateur, il est nécessaire de comparer à chaque pas d'itération k le prédicat booléen Xk avec le prédicat Xk-1, pour déterminer à quel moment la suite (Xi) atteint sa limite.

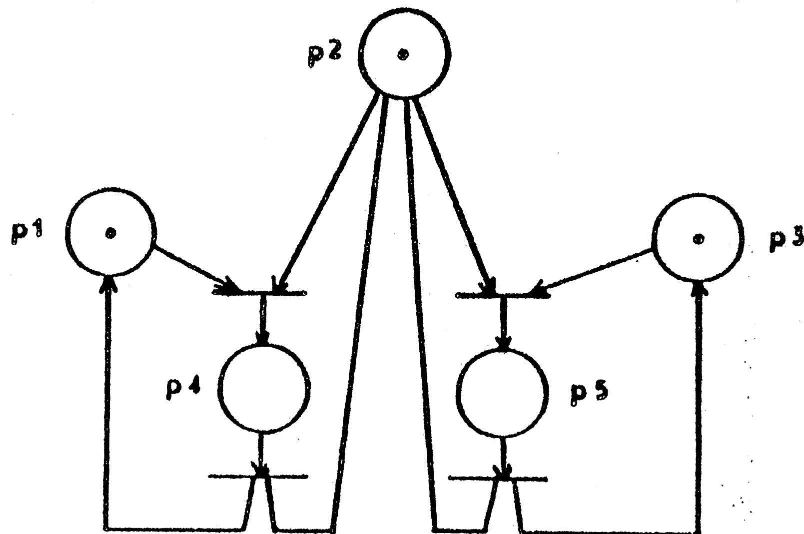
Pour vérifier l'égalité de deux fonctions booléennes sans énumérer exhaustivement leurs domaines de validité, il faut comparer les formes canoniques de ces fonctions. L'algorithme d'évaluation doit donc comporter une procédure utilitaire qui transforme une fonction booléenne, représentée par une somme de monômes, en sa forme canonique.

En pratique, le programme d'évaluation effectue des appels répétés à cette procédure, ce qui a pour conséquence d'accroître de façon sensible les temps de calcul.

5.6. UN EXEMPLE D'EVALUATION

L'exemple choisi a été présenté dans [CRO75]. Il s'agit d'un exemple simple d'exclusion mutuelle. Deux processus P1 et P2 partagent une ressource R à laquelle ils ne peuvent pas accéder simultanément. Le système se représente très simplement à l'aide du réseau de Petri suivant :

figure 4.3



- la place p1 (p3) est marquée lorsque le processus P1 (P2) n'utilise pas la ressource R ;
- la place p2 est marquée lorsque la ressource est libre ;
- la place p4 (p5) est marquée lorsque le processus P1 (P2) utilise la ressource R.

Le système est décrit par le CA-système booléen suivant :

```
SYSTEME MUTEX :

   var p1,p2,p3,p4,p5 ;
   init p1:=1,p2:=1,p3:=1,p4:=0,p5:=0 ;
   do
     p1.p2 : p1 := 0 , p2 := 0 , p4 := 1          |
```

```
           -- ressource libre prise par P1
p2.p3 : p2 := 0 , p3 := 0 , p5 := 1          |
           -- ressource libre prise par P2
p4    : p4 := 0 , p1 := 1 , p2 := 1          |
           -- ressource libérée par P1
p5    : p5 := 0 , p3 := 1 , p2 := 1
           -- ressource libérée par P2
od.
```

Nous allons vérifier deux types de propriétés :

- d'une part que les deux processus ne peuvent pas utiliser simultanément la ressource ;
- d'autre part que le processus P1 peut toujours utiliser la ressource.

5.6.1. VERIFICATION DE L'EXCLUSION MUTUELLE

La formule qui caractérise l'exclusion mutuelle est p4.p5 = 0. Soumettons la formule INIT => not POT (p4.p5) à l'analyseur, et examinons le cheminement de l'évaluation de la formule.

(les calculs itératifs d'opérateurs sont effectués à l'aide de la suite de prédicats booléens p_0, p_1, ... , p_k)

calcul de POT (p4.p5)

p_0=p4.p5
p_1=p4.p5 + p1.p2.p5 + p2.p3.p4
p_2=p_1 ;

calcul de la formule

* La formule INIT => not POT (p4.p5) vaut
  not INIT OR not POT (p4.p5)

  et se réécrit
  not (INIT AND POT (p4.p5))

* Le prédicat INIT vaut p1.p2.p3.^p4.^p5

* Le prédicat INIT AND POT p4.p5 vaut alors
  p1.p2.p3.^p4.^p5 .
  (p1.p2.p5 + p2.p3.p4 + p4.p5)
  = faux.

Le prédicat caractérisant la formule soumise à l'analyseur est le prédicat identiquement vrai. La formule est donc valide pour le système.

Remarquons que si l'initialisation du système est p1,p2,^p3,^p4,p5 alors le prédicat associé à la formule de spécification vaut ^INIT. Ce prédicat n'étant pas identiquement vrai, la formule n'est pas valide pour le système. On vérifie aisément que cette initialisation permet de marquer simultanément les places p4 et p5 du réseau de Petri.

5.6.2. VIVACITE DES ACCES A LA RESSOURCE

La formule de spécification qui caractérise la vivacité de l'action d'accès à la ressource par le processus P1 est INIT => ALL POT p4. Cette expression signifie que l'initialisation du système garantit que le processus P1 pourra toujours être prêt à utiliser la ressource.

calcul de POT p4

p_0=p4
p_1=p4 + p1.p2
p_2=p4 + p1.p2 + p1.p5
p_3=p_2

donc POT p4 = p4 + p1.p2 + p1.p5 ;

calcul de ALL POT p4

* la formule ALL POT p4 vaut ^ POT ^ POT p4

* ^POT p4 = ^p1.^p4 + ^p2.^p4.^p5

* calculons POT ^ POT p4

p_0=^p1.^p4 + ^p2.^p4.^p5
p_1=p_0

donc POT ^ POT p4 = ^ POT p4
et par conséquent, ALL POT p4 = POT p4 ;

calcul de la formule

* la formule INIT => ALL POT p4 s'écrit
^INIT OR ALL POT p4
et vaut
(sachant que ^INIT vaut ^p1+^p2+^p3+p4+p5)

^p1+^p2+^p3+p4+p5 + p4+p1.p2+p1.p5
= ^p1 + ^p2 + p1.p2 + ...
= vrai.

On constate ainsi que la formule est valide pour le système, ce qui prouve la vivacité du processus P1. Un calcul symétrique peut s'effectuer pour prouver la vivacité du processus P2.

On notera que l'exemple choisi est de très petite taille. Nous discuterons ultérieurement l'efficacité de cette méthode, mais nous devons préciser qu'il ne nous a pas été possible de traiter des exemples de taille convaincante, essentiellement pour des raisons de temps de calculs élevés, qui produisent des coûts trop importants.

6. PROGRAMMATION ET COMPARAISON DES METHODES

Le but de ce paragraphe n'est pas de présenter en détail la programmation de l'évaluation, ou d'effectuer des comparaisons chiffrées entre les méthodes d'analyse que nous avons décrites. Nous souhaitons simplement donner une idée de l'organisation des programmes, et des structures de données retenues, pour permettre d'apprécier les avantages et les points faibles des trois méthodes. Nous comparerons alors l'efficacité des algorithmes sur des plans divers, tels que leurs complexités, leurs coûts et leurs capacités.

6.1. SCHEMA DE PROGRAMMATION DE L'EVALUATION

Nous présentons d'abord les structures de données utilisées pour représenter le graphe des états, les ensembles d'états et les prédicats booléens. Nous décrirons ensuite schématiquement l'algorithme d'évaluation des formules de spécification.

6.1.1. STRUCTURES DE DONNEES

Les programmes qui composent le système d'analyse QUASAR ont été écrits dans le langage PASCAL ; la représentation des données a donc été établie uniquement en fonction des possibilités offertes par ce langage dans le domaine de la définition de types.

Le graphe des états

Un état est représenté par un tableau d'entiers, dont chaque élément contient la valeur d'une variable du programme de description de l'application. Une numérotation implicite, liée à l'ordre d'apparition des variables dans la description, permet d'identifier une variable avec un entier i de l'intervalle [1..n], où n est le nombre total de variables du système.

Un tableau des états permet de stocker les états rencontrés lors de la construction du graphe. Ceux-ci sont numérotés, de telle sorte qu'un ensemble d'états consiste en un ensemble d'entiers de l'intervalle [1..N], où N est le nombre total d'états du système.

A chaque état s est associé l'ensemble pre(s) des états prédécesseurs directs de s, construit lors de la génération du graphe. Le calcul d'un ensemble pre(S) se fait par union des ensembles pre(s), pour tout élément s de S.

Les ensembles d'états

Pour représenter un ensemble d'états, il est possible d'utiliser une liste chainée d'enregistrements contenant les numéros des états qui appartiennent à l'ensemble. Cette solution, peu gourmande en place mémoire, ne permet pas d'accéder aux éléments de l'ensemble sans effectuer un parcours complet de la liste chainée.

Le langage PASCAL offre la possibilité d'utiliser un type "ensemble d'entiers de [0..k]", dont la représentation mémoire est une chaine de k+1 bits, le bit i valant 1 si l'entier i est présent dans l'ensemble. La taille maximale d'un ensemble étant p, l'utilisation d'ensembles plus grands impose l'emploi de tableaux d'ensembles ; l'élément j du tableau représentera le sous ensemble [j*p..(j+1)*p-1].

Cette forme de représentation des ensembles a été retenue dans QUASAR. Elle est assez coûteuse en place mémoire, car les ensembles sont souvent creux, mais permet d'accéder directement aux éléments d'un ensemble par l'opérateur d'appartenance IN, et autorise l'utilisation des opérations ensemblistes PASCAL, telles que l'union, l'intersection et la différence. Ces opérations sont utilisées pour élaborer des procédures qui effectuent le même traitement sur des tableaux d'ensembles.

Les prédicats booléens

Les fonctions booléennes sont toutes représentées sous forme de somme de monômes, chaque monôme étant un élément d'une liste chainée d'enregistrements, déterminé par les ensembles des variables postives et négatives. Ces deux ensembles possèdent la structure définie ci-dessus.

Des procédures utilitaires permettent d'effectuer les opérations entre fonctions booléennes, addition, produit, complémentation et mise sous forme canonique. Ces différentes opérations imposent des parcours exhaustifs et répétés des listes chainées.

6.1.2. ALGORITHME D'EVALUATION

L'évaluation des formules est faite par un algorithme de faible complexité, composé d'un ensemble de procédures récursives descendantes qui vérifient d'une part la cohérence syntaxique d'une formule f, et qui calculent d'autre part les ensembles caractéristiques des sous formules de f.

A chaque opérateur OP(a,b) de la logique CTL correspond une procédure qui calcule, pour une formule f dont l'opérateur principal est OP(g,h), l'ensemble caractéristique de f à partir des ensembles

caractéristiques de g et h, selon les méthodes que nous avons explicitées antérieurement.

La représentation d'une formule f par un graphe acyclique orienté DAG(f) a été introduite pour clarifier la présentation des méthodes d'évaluation. Elle n'est pas utilisée en pratique ; les formules sont lues terme par terme, à chaque terme correspond un appel de procédure qui produit l'ensemble caractéristique de la formule pointée par ce terme. L'arbre des appels pour une formule f réalise un parcours postfixé du graphe fictif DAG(f).

Nous donnons à titre d'exemple la procédure qui produit l'ensemble caractéristique S d'une formule f dont l'opérateur principal est POT[a]b ; la formule f s'écrit f = POT[g]h, les ensembles caractéristiques des formules g et h sont notés Sg et Sh, et ont été produits par des procédures du même type.

```
Procedure POTCONDITIONNEL ( entrée Sg,Sh ; résultat S );

   var A : ensemble d'états -- variable de travail

begin

   S := Sh ;
   A := vide ;
   while S<>A do
   begin
      A := S ;
      S := ( PRE(S) n Sg )  u  A ;
   end ;

end ;
```

Nous avons présenté la procédure pour la méthode d'évaluation dans le graphe des états d'une application. Pour la méthode d'évaluation dans le treillis des prédicats, il suffit de remplacer les ensembles par des prédicats booléens, et les opérations d'union et d'intersection par la somme et le produit de fonctions booléennes.

Cet exemple souligne le caractère itératif de l'évaluation des formules de spécification, et met en évidence la faible complexité des procédures utilisées par l'algorithme d'évaluation.

Nous allons désormais discuter l'efficacité des différentes méthodes.

6.2. COMPARAISON DES METHODES

Il ne nous est pas possible de comparer explicitement les méthodes en les appliquant à des exemples précis. En effet, le système d'analyse dans le graphe des états a été réalisé entièrement, alors que la méthode par calcul itératif de prédicats booléens n'a été programmée que partiellement ; la partie "codage" n'a pas été effectuée, les exemples traités ont été préliminairement traduits à la main. On ne peut donc pas comparer les programmes, puisqu'ils n'acceptent pas les mêmes entrées. Nous avons cependant pu tirer des enseignements très utiles de nos travaux, notamment sur les avantages et les inconvénients des diverses méthodes.

Nous ne nous intéressons pas aux problèmes d'implémentation des algorithmes, qui peuvent être considérés comme indépendants de la valeur intrinsèque des méthodes d'évaluation, et nous jugeons les résultats produits par nos algorithmes, tout en ayant conscience que les programmes ne sont pas obligatoirement optimisés.

Evaluation par calcul booléen

La caractéristique principale du programme est sa lenteur d'exécution. Nous n'avons pu traiter que des exemples très élémentaires de petite taille, car des applications plus complexes entrainaient des temps de calcul trop élevés. La raison principale de cette lenteur est l'utilisation permanente dans le calcul booléen de procédures de traitement des expressions booléennes (mise sous forme canonique, somme, produit, complémentation,...) qui parcourent inlassablement des listes chainées d'enregistrements et des ensembles. Nous ne prétendons pas avoir optimisé le rendement de la méthode, mais nous pensons pouvoir affirmer qu'il n'est pas possible d'obtenir des résultats satisfaisants de cette manière. C'est pourquoi nous avons rapidement délaissé cette forme d'évaluation des formules, sans mettre au point le codage des CA-systèmes, et réalisé la méthode d'évaluation par calcul dans le graphe des états, bien plus performante en pratique.

Evaluation dans le graphe des états

La méthode d'évaluation des formules de spécification en termes de sous ensembles du graphe des états est caractérisée par une vitesse de calcul très satisfaisante. Les phases de construction du graphe des états et d'évaluation des prédicats non temporels sont assez lentes, mais ne doivent en principe être exécutées qu'une seule fois ; seule la modification du programme de description, ou l'adjonction de nouveaux prédicats non temporels entraine une réexécution de ces deux opérations. Le temps d'évaluation d'une formule de spécification est généralement assez court, même pour des formules longues et complexes, car la représentation des ensembles d'états à l'aide d'ensembles PASCAL, et le calcul préalable de la

fonction pre, permettent des calculs rapides.

Le principal problème soulevé par la méthode est la limitation du nombre des états d'un graphe. Nous n'avons pu étudier que des systèmes qui comportaient moins de 3000 états, ce qui interdit en pratique l'analyse de systèmes complexes. L'introduction d'un graphe réduit des états permet d'augmenter très sensiblement la capacité des programmes, tout en réduisant les temps d'évaluation, mais n'autorise pas une puissance de spécification suffisante.

Les exemples que nous avons traités possédaient des graphes d'états limités à 2000 états. Les temps de construction du graphe ne dépassaient pas la soixantaine de secondes, et les temps d'évaluation des formules de spécification étaient de l'ordre de la seconde. Ces résultats nous ont semblé satisfaisants, notamment si on les compare avec ceux qui avaient été obtenus pour la méthode d'évaluation par calcul de prédicats booléens.

7. CONCLUSION

Nous avons présenté deux méthodes d'analyse de systèmes conditions-actions par évaluation itérative de formules de spécification. Le principe d'analyse, retenu dans le système QUASAR, peut s'appliquer à d'autres modèles de représentation des systèmes parallèles (par exemple les réseaux de Petri), sous la seule condition que ces modèles permettent la définition d'une fonction pre. Il faut donc voir en QUASAR une simple application pratique d'une méthode d'analyse qui peut être utilisée sous des formes différentes.

La caractérisation en termes de points fixes des opérateurs temporels a permis d'élaborer une méthode d'analyse fiable des systèmes répartis ; la vérification des propriétés est effectuée automatiquement par la machine sur un modèle lui même généré automatiquement, par opposition à certaines techniques manuelles qui reposent uniquement sur l'intuition du "prouveur".

L'implémentation de la méthode d'évaluation nous a fait prendre conscience de l'étendue du fossé qui sépare l'élaboration d'une théorie bien définie, de sa réalisation. Les contraintes matérielles s'avèrent très rapidement insurmontables, et limitent inéxorablement la capacité des systèmes d'analyse. Il faut peut-être voir là aussi une certaine limitation de la méthode d'analyse elle-même.

Notre analyseur constitue cependant une preuve de la faisabilité de la méthode. Sa réalisation nous a montré que des investissements plus importants dans la constitution des programmes et dans la représentation des données doivent permettre la mise au point sur de gros systèmes informatiques, de systèmes d'analyse performants et de grande capacité.

CHAPITRE V

```
*****************************
*                           *
*   UTILISATION DE QUASAR   *
*                           *
*****************************
```

1. Introduction
2. Rappels sur le système MULTICS
3. Présentation du système QUASAR
4. Liste des commandes utilisateur
5. Schéma d'une session
6. Quelques exemples complets d'analyse
7. Conclusion

1. INTRODUCTION

L'objet de ce chapitre est de présenter une notice d'emploi du programmme qui réalise le système QUASAR. Nous ne souhaitons pas décrire le programme en détail, bien qu'il faille par moments donner des précisions sur les méthodes qu'il utilise, mais expliciter de quelle façon un utilisateur peut se servir du programme pour analyser le comportement des systèmes parallèles.

Notre exposé est divisé en trois parties :

- une première partie décrit l'environnement d'implémentation du système d'analyse ; nous présentons brièvement le système MULTICS et les langages de programmation que nous avons utilisés, pour permettre de mieux comprendre l'organisation des programmes, et pour définir avec précision les possibilités et les restrictions de QUASAR ;

- une deuxième partie décrit le mode d'emploi du système d'analyse ; nous présentons en détail les commandes qui sont à la disposition de l'utilisateur, les opérations qu'elles effectuent et les informations qu'elles engendrent pour permettre à l'utilisateur d'effectuer des contrôles permanents sur le déroulement des différents calculs ;

- la troisième partie décrit le schéma d'une session d'utilisation de QUASAR ; nous présentons le déroulement d'une phase d'analyse, en soulignant la succession naturelle des opérations que doit effectuer l'utilisateur au cours de la session.

Nous illustrons pour terminer l'utilisation du système QUASAR par l'analyse complète de quelques-uns des exemples que nous avons présentés dans les précédents chapitres.

2. RAPPELS SUR LE SYSTEME MULTICS

Le système QUASAR a été réalisé sur un ordinateur HB68 de la compagnie CII-Honeywell Bull, fonctionnant sous le système d'exploitation MULTICS. Les programmes ont été écrits en grande majorité dans le langage PASCAL, mais quelques utilitaires ont nécessité l'emploi du langage de programmation PLI. Nous décrivons brièvement les principales caractéristiques du système MULTICS, qui ont dirigé l'élaboration de notre analyseur.

Le système MULTICS est un système d'exploitation multi-utilisateurs implémenté sur un calculateur multi-processeurs. Il fournit aux utilisateurs une grande aisance de programmation, grâce notamment aux points suivants :

- un grand nombre de langages de programmation sont disponibles, tels les langages usuels BASIC, FORTRAN, PASCAL ou PLI ;

- le système fonctionne en temps partagé, ce qui implique des temps de réponse élevés lorsque le nombre d'utilisateurs est grand, mais permet à un programme de disposer de la totalité de la place mémoire d'un processeur ;

- des utilitaires du système permettent de faciliter la mise au point des programmes ;

- la majeure partie des fonctions du système sont à la disposition des utilisateurs, sous forme de procédures utilisables par programme.

Parmi les points cités ci-dessus, le dernier a constitué un apport important dans la réalisation du système QUASAR. QUASAR est un système d'analyse interactif qui se présente pour l'utilisateur comme un système indépendant, comportant un ensemble de fonctions de calcul pour l'analyse, mais aussi un ensemble de fonctions utilitaires pour permettre à l'utilisateur de gérer les fichiers manipulés par l'analyse. L'écriture de QUASAR a donc imposé la mise au point de telles fonctions, et cette tâche a été simplifée par la possibilité offerte d'accéder aux fonctions du système MULTICS.

Par ailleurs, pour améliorer le confort des utilisateurs, toutes les commandes MULTICS sont utilisables au sein même de QUASAR.

3. PRESENTATION DU SYSTEME QUASAR

Le système QUASAR se présente comme un système d´exploitation interactif autonome, comportant un ensemble de commandes d´exécution des fonctions de description et d´analyse de systèmes répartis. L´interaction entre le système et l´utilisateur est classique :

- le système au repos est en attente d´une commande ;
- l´envoi d´une commande par l´utilisateur provoque son interprétation par exécution du code correspondant, le système affiche alors un éventuel compte rendu du déroulement des opérations (résultats, erreurs...), puis se met en attente d´une nouvelle commande.

Les objets manipulés par le système QUASAR sont de trois types :

- des fichiers "texte" (formés uniquement d´une suite de caractères) qui contiennent la représentation de l´application à étudier. Ces fichiers constituent les entrées du système d´analyse, et sont créés puis gérés par l´utilisateur pour la description ou la spécification d´une application ;

- des fichiers de travail générés par le système d´analyse pour la sauvegarde des informations internes utilisées par les fonctions d´analyse (langage intermédiaire lors de la compilation, informations sur les étiquettes, graphe des états, ...). Ces fichiers sont en principe des suites d´enregistrements, et s´avèrent donc inaccessibles à l´utilisateur ;

- des fichiers "texte" générés par le système d´analyse qui constituent les sorties de l´analyseur. Ces fichiers contiennent sous forme lisible les informations produites par le système (graphe des états, ...) et peuvent être exploités par l´utilisateur pour contrôler le déroulement de l´étude d´une application.

Tous les fichiers générés par QUASAR sont sauvegardés, et peuvent donc être réutilisés lors d´une phase d´analyse ultérieure.

Pour permettre à l´utilisateur de constituer une bibliothèque d´applications, et pour toujours pouvoir reprendre sans frais une étude interrompue, les fichiers générés par l´analyseur pour des applications distinctes sont eux aussi distincts. La donnée du nom d´un système à étudier permet de préfixer les fichiers relatifs à ce système par ce nom, de telle sorte que seule une nouvelle définition d´une application de même nom provoque l´écrasement des fichiers.

Capacité d'analyse de QUASAR :

La maquette réalisée comporte un certain nombre de limitations, imposées par des problèmes d'implémentation. Les applications étudiées doivent respecter les contraintes suivantes :

- le programme de description doit comporter au maximum 50 tâches, 60 variables, 50 étiquettes et 250 commandes gardées, toutes tâches confondues ;

- le système conditions-actions composé peut contenir au plus 250 commandes gardées ;

- le graphe d'états de l'application ne doit pas dépasser 3000 états ;

- le nombre de prédicats sur les variables de la description ne doit pas être supérieur à 50.

Toutes ces limitations sont contrôlées systématiquement lors des différentes phases de l'analyse ; un dépassement de capacité provoque dans tous les cas l'affichage d'un message d'erreur de QUASAR, et interrompt l'exécution de la commande QUASAR en cours.

Invocation de QUASAR :

Pour pouvoir utiliser le système QUASAR, il faut établir un lien avec le programme au moyen de la commande MULTICS :

```
lk >udd>LANGAGE>Mamar>ANALY>Cesar>objet>ancas
```

Une fois ce lien établi (une fois pour toutes), l'environnement QUASAR peut être invoqué en tapant simplement le nom du programme :

```
ancas
```

Après un message de bienvenue, le système QUASAR imprime le sigle "QUASAR ?" chaque fois qu'il est en attente d'une commande.

Nous allons désormais présenter l'ensemble des commandes que QUASAR met à la disposition des utilisateurs pour décrire, spécifier et analyser un système distribué.

4. LISTE DES COMMANDES UTILISATEUR

Nous donnons la liste exhaustive des commandes reconnues par le système QUASAR, en suivant l'ordre naturel des opérations à effec-

tuer pour l'analyse d'une application. Nous décrivons pour chaque commande le traitement réalisé par le programme, en précisant les entrées requises par la fonction correspondante et les informations générées en cours de traitement.

Les commandes suivantes sont reconnues par le système QUASAR :

cas
: définition du nom de l'application que l'on va étudier ;

comp
: compilation de la description ;

sim
: génération du graphe des états du système décrit ;

def
: définition des prédicats non temporels utilisés par l'analyse, et évaluation de ceux-ci ;

eval
: évaluation d'une formule temporelle ;

decide
: appel de la procédure de décision de CTL ;

del
: destruction des fichiers relatifs à une application ;

e
: exécution d'une ligne de commande MULTICS ;

help
: listage des commandes QUASAR ;

listcas
: listage de toutes les applications disponibles ;

print
: impression du graphe d'états ;

q
: sortie de l'environnement QUASAR ;

@
: indirection sur un fichier de commandes ;

Commande cas

format : cas nom

Cette commande permet de définir le nom de l'application que l'on va étudier. Elle effectue un certain nombre d'initialisations, et de ce fait, elle est obligatoire avant la première commande "comp", "sim", "def", "eval" ou "print".

Pour des raisons liées au compilateur PASCAL, la longueur du nom ne doit pas dépasser 8 caractères.

Commande comp

format : comp [option]*

Cette commande lance la compilation de la description fournie dans le fichier de nom "nom.cas", où "nom" a été défini par une commande "cas" préalable. Elle admet plusieurs options (qui doivent apparaître dans l'ordre décrit) :

** 1ère option : "-sy", "-comp" ou "-all" :

-sy : génération des procédures PASCAL représentant le système décrit ; pour pouvoir générer le graphe des états, ces procédures devront ultérieurement être compilées par le compilateur PASCAL (au moyen de l'option "-qomp") ; cette option génère quatre fichiers de noms respectifs :

_"nom.cas.des" qui contient la table des identificateurs sous forme codée ;

_"nom.cas.obj.pascal" qui contient les procédures PASCAL générées ;

_"nom.cas.etiq" qui contient toutes les informations relatives aux étiquettes figurant dans le programme de description ;

_"noml.compo", ou "noml" est le nom de la tâche la plus englobante du système, qui contient le système dans sa totalité en langage intermédiaire ;

De plus, la composition utilise un nombre variable de fichiers de travail, de suffixe ".temp", qui sont normalement détruits à la fin de cette opération ;

-comp : invocation du compilateur PASCAL sur les procédures PASCAL générées pour représenter le système décrit ; ces procédures

doivent avoir été générées auparavant (au moyen de l'option "-sy") ; cette option génère un fichier de nom "nom.cas.obj" (objet produit par le compilateur PASCAL) ;

-all : exécution successive des options "-sy" et "-comp" ; l'option "-all" est l'option par défaut de la commande "comp" ;

** autres options (dans l'ordre possible d'apparition) :

-id : génération de la description de toutes les variables figurant dans le programme de description, dans un fichier de nom "nom.cas.tab" ; cette option doit obligatoirement être précédée de l'option "-sy" ou "-all" ;

-dl : destruction du fichier de nom "nom.cas.obj.pascal" après la compilation PASCAL ; cette option doit obligatoirement être précédée de l'option "-comp" ou "-all" ;

** toutes les options du compilateur PASCAL

(uniquement si l'option "-comp" ou "-all" est présente) ; les options "-ns -no_iow -aen" sont toujours présentes par défaut.

<u>Commande sim</u>

format : sim [option]*

Cette commande permet de générer le graphe d'états (ainsi que le graphe inverse qui est exploité pour l'évaluation des formules temporelles). Pour pouvoir exécuter cette commande, un nom d'application a du être fourni par une commande "cas", et le système a du être compilé au cours de la présente session ou durant une précédente invocation de QUASAR.

Cette commande génère quatre fichiers de noms respectifs :

_"nom.cas.valeur" qui contient, sous forme codée, la valeur des variables pour chacun des états du système ;

_"nom.cas.copy" qui contient, sous forme codée, le graphe d'états inverse utilisé pour l'analyse des formules temporelles ;

_"nom.cas.enafter" qui contient, sous forme codée, les ensembles caractéristiques des variables propositionnelles ENABLE(Ta), ENABLE(ea), pour toute tâche Ta et toute étiquette ea figurant dans le programme de description ;

_"nom.cas.nb" qui contient le nombre d'états du système.

Le graphe d'états d'une application est construit par défaut en

considérant que le programme de description ne comporte pas d'étiquettes. Tous les états qui correspondent à une même valeur des variables sont identifiés, même s'ils sont atteints par exécution de transitions distinctes.

Si l'utilisateur souhaite employer dans les formules de spécification des variables propositionnelles AFTER(ea) (où ea désigne une étiquette de la description), il doit faire suivre la commande "sim" d'une des deux options suivantes :

-af : les seules étiquettes prises en compte pour la génération du graphe sont celles qui sont citées dans un fichier de nom "nom.cas.after". Ce fichier est un fichier texte, créé par l'utilisateur, dont chaque ligne comporte le nom d'une étiquette du programme de description. L'algorithme de construction du graphe distingue alors les états correspondant à une même valeur des variables, et qui sont atteints par des transitions étiquetées distinctes dont les noms apparaissent dans le fichier "nom.cas.after".

-afall : toutes les étiquettes d'actions figurant dans le programme de description sont prises en compte lors de la génération du graphe.

Dans les deux cas, les ensembles d'états AFTER(ea) des étiquettes mentionnées sont sauvegardés dans le fichier de nom "nom.cas.enafter".

La commande admet par ailleurs les options suivantes, qui permettent de générer des fichiers contenant différentes descriptions "lisibles" de l'espace d'états du système :

-gr : description du graphe des états (direct) dans le fichier de nom "nom.cas.gra" ;

-gri : description du graphe inverse des états dans le fichier de nom "nom.cas.grai" ;

-st : liste et valeurs des états du système, dans le fichier de nom "nom.cas.etat" ;

-all : génération des trois fichiers précédents.

Lors de la génération du graphe d'états, l'erreur "Attempt to reference through null pointer" peut se produire, lorsque la taille du graphe d'états dépasse celle de la zone de travail allouée pour les enregistrements PASCAL. Il est possible de remédier à cela en augmentant la taille de cette zone, mais l'instance du programme en cours d'exécution ne peut être toutefois relancée. Une manière d'opérer peut être la suivante :

rl

(pour libérer la pile d'exécution de MULTICS)

pascal_area -reset -size nnn
(libération de la zone de travail prise par les enregistrements, puis définition d'une nouvelle taille de nnn pages mémoire. La valeur par défaut est 255 pages)

ancas
.....
(reprise du traitement interrompu)

Commande def

format : def

Cette commande provoque l'évaluation des prédicats sur les variables du programme de description, définis dans le fichier de nom "nom.cas.predicat", où nom a été défini au moyen d'une commande "cas" préalable. Pour pouvoir exécuter cette commande, la description contenue dans le fichier "nom.cas" a du être compilée et le graphe des états a du être généré, au cours de la session présente, ou durant une session précédente.

Cette commande génère les fichiers suivants :

_"nom.cas.predicat.des" qui contient la table des identificateurs des prédicats (sous forme codée) ;

_"nom.cas.predicat.obj.pascal" qui contient la procédure PASCAL générée pour permettre l'évaluation des prédicats ;

_"nom.cas.predicat.obj" qui contient le code généré par compilation PASCAL du fichier précédent ;

_"nom.cas.predicat.copy" qui contient, sous forme codée, la liste des états qui vérifient chacun des prédicats ;

_"nom.cas.predicat.nb" qui contient le nombre de prédicats.

Commande eval

format : eval formule

Cette commande provoque l'évaluation de la formule temporelle fournie. Pour pouvoir exécuter cette commande, un nom d'application doit être préalablement défini au moyen d'une commande "cas", ce système a du être compilé au moyen de la commande "comp" (avec l'option "-all", ou avec les options "-sy" et "-comp" successivement), le graphe d'états a du être généré au moyen de la commande "sim",

et les prédicats ont du être évalués au moyen de la commande "def". Ces différentes commandes ont pu être exécutées au cours de la présente invocation de QUASAR, ou dans une session précédente.

A l'issue du calcul, le programme indique le nombre d'états qui vérifient la formule, et précise si la formule est valide pour le système étudié ou non.

Commande decide

format : decide formule

Cette commande provoque l'exécution de la procédure de décision de la logique CTL pour la formule donnée. Les variables propositionnelles qui figurent dans la formule temporelle sont muettes, et n'ont ainsi aucun rapport avec les noms des identificateurs de prédicats qui peuvent être utilisés pour l'évaluation.

La procédure de décision tente de construire un modèle satisfaisant la négation de la formule proposée. Si la recherche est positive, le modèle est exhibé sous forme d'un graphe orienté, sinon la formule est un théorème de la logique.

Commande del

format : del nom [option]* | del * [option]*

Cette commande provoque la destruction de tous les fichiers de nom "nom.cas.**" et "nom.compo" (ou "*.cas.**" et "*.compo" dans le second cas). Elle admet toutes les options de la commande "dl" de MULTICS.

Commande e

format : e ligne

Cette commande provoque l'exécution de "ligne", pris comme une commande MULTICS. Il est recommandé d'éviter des manipulations explicites des fichiers de travail de QUASAR par ce moyen.

Commande help

format : help [commande]

Cette commande permet d'obtenir la liste des commandes disponibles si aucun argument n'est fourni, ou des informations sur l'utilisation d'une commande particulière (syntaxe, fichiers

d'entrée et de sortie, ...) dont le nom figure en argument de la commande "help".

Par ailleurs, la commande "help limit" affiche les limites actuelles (capacités du programme) liées à l'implémentation du système d'analyse QUASAR.

Commande listcas

format : listcas

Cette commande liste les noms de toutes les applications disponibles. Elle est équivalente à la commande "e ls *.cas".

Commande print

format : print option

Cette commande provoque l'impression d'un fichier (ou de tous les fichiers) parmi ceux générés par la commande "sim", suivant l'option indiquée :

-gr : impression du graphe d'états (fichier de nom "nom.cas.gra") ;

-gri : impression du graphe inverse des états (fichier de nom "nom.cas.grai") ;

-st : impression de la liste des états du système (fichier de nom "nom.cas.etat") ;

-all : impression des trois fichiers ci-dessus.

Ces fichiers doivent avoir été générés auparavant au moyen de la commande "sim", munie des options correspondantes.

Commande q

format : q

Cette commande fait sortir de l'environnement QUASAR.

Commande @

format : @nom

Cette commande provoque la lecture des commandes QUASAR qui figurent dans le fichier de nom "nom.ancas", comme si elles provenaient du terminal. Si le fichier ne comporte pas de commande "q" à la fin du fichier, QUASAR rend la main au terminal à la fin du fichier de commande.

Il est également possible de provoquer la lecture d'un fichier de commandes lors de l'invocation du système QUASAR, en tapant :

ancas nom

Ce type d'invocation de QUASAR est équivalent à l'invocation de QUASAR, suivie de la commande "@nom".

5. SCHEMA D'UNE SESSION

Nous présentons le schéma de déroulement de l'analyse complète d'un système. Ce schéma décrit la succession logique des différentes opérations à exécuter pour traiter une application.

Une session pour l'analyse complète d'un système décrit et spécifié se compose usuellement des commandes suivantes :

* utilisation de l'éditeur "ted" de MULTICS pour créer le fichier "nom.cas" contenant le programme de description de l'application à étudier ;

* utilisation de l'éditeur "ted" de MULTICS pour créer le fichier "nom.cas.predicat" contenant les prédicats non temporels qui seront utilisés dans les formules temporelles à évaluer ;

ancas
cas nom
(définition du nom de l'application étudiée)
comp -sy
(première partie de la compilation, permettant de corriger les erreurs de syntaxe faites dans le fichier de description ; s'il y a des erreurs :
e ted -pn nom.cas
...
(correction des erreurs sous l'éditeur)
w

q
   (sortie de l'éditeur)
comp -sy
   (recompilation : s'il n'y a plus d'erreur, on peut poursuivre)
comp -comp
   (compilation du PASCAL généré)
sim
   (génération du graphe des états, éventuellement avec options)
def
   (définition et évaluation des prédicats : s'il y a des erreurs de syntaxe dans la définition des prédicats, on les corrige sous éditeur)
eval formule
...
...
eval formule
   (évaluation des formules une par une)
q
   (sortie du moniteur ancas ; fin de l'interprétation QUASAR).

Il est tout à fait possible, lors d'une invocation de QUASAR, d'interrompre le traitement lié à l'étude d'une application, soit pour sortir de l'environnement "ancas", soit pour passer à l'étude d'une autre application. Les fichiers générés par les commandes de QUASAR sont permanents, et donc réutilisables lors d'une prochaine invocation du système d'analyse. La seule obligation de l'utilisateur est d'exécuter une commande "cas" à chaque fois qu'il souhaite changer de système. Les commandes qui ont été effectuées antérieurement n'ont pas besoin d'être réexécutées, car QUASAR se charge de récupérer au moment voulu les informations dont il a besoin ; dans la mesure où l'utilisateur n'a pas détruit explicitement les fichiers contenant ces informations, l'analyse se poursuit correctement.

6. QUELQUES EXEMPLES COMPLETS D'ANALYSE

Nous présentons dans ce paragraphe l'étude détaillée de l'analyse des exemples de systèmes répartis que nous avons décrits dans les chapitres précédents.

Chaque exemple fait l'objet d'une analyse que nous pensons exhaustive, de telle sorte que notre façon d'opérer peut servir de méthode d'utilisation du système QUASAR. Nous détaillons les différentes phases d'analyse, de la conception du programme de description d'une application jusqu'à l'évaluation des formules de spécification. Pour le cinquième exemple, (et pour cet exemple uniquement), nous présentons par ailleurs la session d'analyse en entier, telle qu'elle s'est passée devant un terminal d'ordinateur ; nous montrons à cette occasion l'emploi de la procédure de décision de la logique temporelle CTL.

Pour chacun des exemples, la démarche que nous adoptons est la suivante :

- nous ne rappelons pas le programme de description, qui a été présenté au chapitre II ; ce programme est légèrement modifié, par l'abandon des étiquettes non utilisées dans les formules de spécification ;

- nous effectuons une brève présentation du système de transitions construit à partir de la description de l'application ;

  - indication de la taille du graphe des états, et comparaison avec le graphe généré lorsque le programme de description ne comporte pas d'étiquettes ;
  - relevé des temps d'exécution nécessaires pour effectuer les différentes opérations ; compilation de la description, compilation PASCAL du système conditions-actions composé, génération du graphe d'états et calcul des ensembles caractéristiques des prédicats prédéfinis ENABLE() et AFTER() ; ces mesures permettent de juger le "rendement" de QUASAR ;

- nous évaluons les différentes formules de spécification présentées au chapitre III ; nous nous assurons que certaines formules, que nous avons déclarées valides intuitivement, sont effectivement vérifiées par le système décrit ;

- nous discutons finalement, s'il y a lieu, de la conformité de notre description avec la présentation du système à étudier.

Cette dernière partie constitue assurément une critique du système QUASAR, et souligne avec précision les limites théoriques (nous considérons que les "performances" de QUASAR n'interviennent

pas dans ce jugement) actuelles de notre système d'analyse.

6.1. EXCLUSION MUTUELLE DE DECKER

La description de l'algorithme d'exclusion mutuelle de DECKER est formée des trois tâches P1, P2 et CONTROLER décrivant le fonctionnement de deux processus cycliques P1 et P2 utilisant une ressource commune. L'exclusion mutuelle entre les accès des deux processus à cette ressource est assurée par la tâche CONTROLER, qui alterne par ailleurs les possibilités d'accès.

La phase de compilation de la description a produit un système conditions-actions comportant 12 commandes gardées (cf II.3.4), et dont l'espace d'états comporte 42 états (cf IV.3.5). Les temps de calcul des différentes opérations sont 10'' pour la compilation (dont 2'' pour la compilation PASCAL) et 2'' pour la construction du graphe des états. L'exemple est assez particulier, car la présence de 4 étiquettes dans la description n'a pas modifié la taille de l'espace d'états.

L'evaluation des formules de spécification a prouvé la validité des formules suivantes :

```
[] ENABLE
[] POT ENABLE(P1)
[] POT ENABLE(P2)
[] POT ENABLE(CONTROLER)
[] POT ENABLE(aut_acces1)
[] POT ENABLE(aut_acces2)
```

caractérisant l'absence de blocage du système, la vivacité des trois tâches et des actions d'accès à la ressource des deux processus ;

```
[] mutuelle_exclusion
[] NOT ALL AFTER(aut_acces1)
[] NOT ALL AFTER(aut_acces2)
```

exprimant que les processus n'accèdent pas simultanément à la ressource, et qu'un processus ne peut jamais garder la ressource indéfiniment ;

```
[] AFTER(dem_res1) => FAIR AFTER(aut_acces1)
[] AFTER(dem_res1) => FAIR AFTER(aut_acces2)
```

exprimant le fait que sous l'hypothèse de l'équité du système, les demandes d'accès à la ressource sont satisfaites en un temps fini.

Les formules suivantes ont été prouvées non valides par notre an-

alyseur :

```
ENABLE(P1), ENABLE(P2), ENABLE(CONTROLER)
[] AFTER(dem_res1) => INEV AFTER(aut_acces1)
[] AFTER(dem_res1) => INEV AFTER(aut_acces2)
```

exprimant que dans la description fournie les tâches ne peuvent pas évoluer à tout instant, et le fait que les demandes d'accès peuvent ne jamais être satisfaites.

Ces résultats, que la faible complexité du système nous permettait de prévoir avant même d'évaluer les formules, répondent en grande partie aux impératifs exprimés par l'algorithme de DECKER.

La description que nous avons proposée garantit l'utilisation de la ressource en exclusion mutuelle, la vivacité des actions d'accès, et la libération de la ressource par un processus en un temps fini.

Le problème principal soulevé par notre analyse est la représentation de l'exécution parallèle des processus P1 et P2. Ces deux processus, dans l'algorithme de DECKER, évoluent simultanément parce qu'ils s'exécutent sur des machines distinctes ou sur une machine multi-tâches. Notre description ne traduit pas fidèlement cette simultanéité, car il existe des séquences d'exécution infinies dans lesquelles un des processus ne fonctionne jamais.

Ce problème est lié en fait non pas à la description de l'algorithme de DECKER que nous avons faite, mais à la façon dont sont construites les séquences d'exécution du système conditions-actions représentant l'application. En effet, lors de la construction du graphe des états, nous prenons en compte toutes les séquences d'exécutions possibles du système, certaines d'entre elles pouvant ne comporter l'activation d'aucune action d'une tâche donnée de la description, même si celle-ci peut évoluer en tous points de ces séquences.

Une solution à ce problème a été étudiée dans [SCH81] pour les réseaux de Petri interprétés, et peut être appliquée aux systèmes conditions-actions.

Le principe de cette méthode consiste à définir une notion de pas d'exécution, au cours duquel chaque processus de l'application exécute obligatoirement, s'il en a la possibilité, une action exactement. On simule ainsi l'exécution parallèle des processus, en obligeant les processus qui peuvent évoluer à exécuter une action.

On peut intégrer cette méthode au système d'analyse QUASAR, par adjonction à la phase de compilation d'une application, d'une étape de synchronisation des tâches formant une tâche composée. Pour toute tâche composée T, formée des tâches T1, ..., Tn, on ajoute à l'ensemble {T1, ..., Tn} une tâche de synchronisation TS qui distri-

bue à chaque pas d'exécution un pouvoir d'exécuter une action à chaque tâche Ti, par l'échange en mode "BROAD" d'une variable de synchronisation.

On obtient finalement un système conditions-actions représentant la totalité de l'application décrite, dans lequel les tâches de la description évoluent dès qu'elles le peuvent. L'ensemble des séquences d'exécution construites à partir de ce système est obtenu de la même façon, mais toutes les séquences comportent l'évolution obligatoire des tâches qui peuvent exécuter des actions.

Une autre façon de résoudre ce problème consiste à ne considérer que l'ensemble des séquences d'exécution équitables d'un système. Il s'agit d'un problème différent, mais on impose d'exécuter les actions infiniment souvent possibles, et l'on interdit ainsi dans les séquences l'exclusion d'un processus qui peut évoluer.

6.2. PROTOCOLE D'ALLOCATION DE BUS

La description du protocole d'allocation d'un bus est composée des trois tâches P1,P2 et P3 qui représentent chacune un processus de l'anneau virtuel. La compilation du programme de description a produit un système conditions-actions comportant 102 commandes gardées et dont le graphe des états contient 114 états. Les temps d'exécution pour les différentes opérations ont été 55'' pour la compilation (dont près de 40'' pour la compilation PASCAL) et 6'' pour la construction du graphe des états.

L'évaluation des formules de spécification a démontré les formules suivantes :

```
[] ENABLE

[] POT ENABLE(P1)
[] POT ENABLE(P2)
[] POT ENABLE(P3)

[] POT ENABLE(rec_jeton_a_1,rec_jeton_b_1,rec_jeton_c_1)
[] POT ENABLE(rec_jeton_a_2,rec_jeton_b_2,rec_jeton_c_2)
[] POT ENABLE(rec_jeton_a_3,rec_jeton_b_3,rec_jeton_c_3)
```

exprimant l'absence de blocage du système, la vivacité des trois processus et celle de leur accès à la ressource ;

```
[] NOT ALL AFTER(rec_jeton_a_1) AND NOT ALL AFTER(rec_jeton_b_1)
                                AND NOT ALL AFTER(rec_jeton_c_1)
[] NOT ALL AFTER(rec_jeton_a_2) AND NOT ALL AFTER(rec_jeton_b_2)
                                AND NOT ALL AFTER(rec_jeton_c_2)
```

```
[] NOT ALL AFTER(rec_jeton_a_3) AND NOT ALL AFTER(rec_jeton_b_3)
                                AND NOT ALL AFTER(rec_jeton_c_3)

[] INEV AFTER(rec_jeton_a_1,rec_jeton_b_1,rec_jeton_c_1)
[] INEV AFTER(rec_jeton_a_2,rec_jeton_b_2,rec_jeton_c_2)
[] INEV AFTER(rec_jeton_a_3,rec_jeton_b_3,rec_jeton_c_3)
```

exprimant qu'un processus ne peut pas garder la ressource indéfiniment, et qu'il ne peut pas être exclu définitivement de l'anneau ;

```
[] util1 => NOT util2
[] util2 => NOT util3
[] util1 => NOT util3

[] NOT(util1 OR util2 OR util3) =>
                  INEV (util1 OR util2 OR util3)
```

exprimant le fait que le jeton de contrôle appartient à un processus au plus à un instant donné (ce qui garantit l'exclusion mutuelle), et que si le jeton de contrôle se perd, alors il est inévitablement régénéré en un temps fini.

La description que nous avons proposée du protocole d'allocation d'un bus répond ainsi pleinement aux exigences fixées par le problème. Tous les processus de l'anneau accèdent à la ressource, la ressource ne peut être utilisée que par un seul processus à la fois, et aucun processus ne peut être exclu définitivement de l'anneau.

Ce deuxième exemple met en évidence l'utilité de pouvoir déclarer dans les programmes de description des types de tâche, afin de caractériser certains fonctionnements et de définir des tâches de ces types. La notion de type de tâche, présente dans CESAR, n'a pas été introduite dans QUASAR. L'utilisateur est ainsi obligé de reproduire le texte de la description des tâches d'un même type, ce qui entraîne un travail fastidieux et une augmentation du temps de compilation d'une application.

Nous devrons à l'avenir faire un effort particulier pour rendre la description des systèmes répartis dans QUASAR la plus simple et la plus agréable possible. La définition des types est très certainement un des points les plus faibles de notre système, et nécessite des améliorations sensibles pour parvenir à un véritable langage de description.

6.3. TRANSMISSION AVEC DECONNECTION

La description du système de transmission avec déconnection est composée des trois tâches ENDA, ENDB et LINE. La compilation de ce système a produit un système conditions-actions comportant 8 commandes gardées, dont le graphe d´états comporte, pour des buffers à deux éléments, 1537 états. Les temps de calcul sont 6´´ pour la compilation et 1´ pour la génération du graphe.

Cet exemple est un cas intéressant, car il souligne la différence importante du nombre des états que peut comporter le graphe lorsque le programme de description comporte des étiquettes. La description analysée comporte 10 actions étiquetées et son graphe d´états possède 1537 états. Le même programme de description, mais ne comportant pas d´étiquette, génère un graphe de 144 états, en un temps de 6´´.

La différence est grande, et souligne la nécessité d´utiliser les étiquettes de façon très économique, afin d´éviter un accroissement exponentiel des temps de calcul et la saturation mémoire du calculateur. Il est beaucoup plus avantageux, sinon indispensable parfois, d´exprimer les prédicats associés aux actions par l´intermédiaire de prédicats sur les variables de la description jouant le rôle de compteurs ordinaux. On peut ainsi exprimer les mêmes propriétés, en faisant d´importantes économies.

L´évaluation des formules de spécification a prouvé les formules suivantes :

```
INIT => (POT fina) AND (POT finb) AND (POT fin)
INIT => (FAIR fina) AND (FAIR finb) AND (FAIR fin)
```

exprimant la déconnection possible, et inévitable sous l´hypothèse d´équité, des deux sites ´a´ et ´b´ ;

```
[] POT ENABLE (ENDA)
[] POT ENABLE (ENDB)
```

caractérisant la vivacité des deux sites ´a´ et ´b´ ;

```
[] fina => ALL fina
[] fina => ALL NOT ENABLE(sendal)
[] fina => ALL NOT ENABLE(recla)
[] finb => ALL finb
[] finb => ALL NOT ENABLE(sendbl)
[] finb => ALL NOT ENABLE(reclb)

[] (pasvideab AND NOT finb)  => ENABLE(sendlb)
[] (paspleinab AND NOT fina) => ENABLE(recal)
```

```
[] (pasvideba AND NOT fina)  => ENABLE(sendla)
[] (paspleinba AND NOT finb) => ENABLE(recbl)
```

exprimant que la déconnection d'un site est définitive et interrompt ses communications avec la ligne, et que les échanges de messages sont possibles pour un site connecté dès lors que les contraintes de stockage sont favorables ;

```
[] AFTER(senddisa) => fina
[] AFTER(senddisb) => finb

[] AFTER(recal) => pasvideab
[] (AFTER(recal) AND NOT finb) => ENABLE(sendbl)
[] AFTER(recbl) => pasvideba
[] (AFTER(recbl) AND NOT fina) => ENABLE(sendal)

[] AFTER(recal) => FAIR(finb OR AFTER(sendlb))
[] AFTER(recbl) => FAIR(fina OR AFTER(sendla))
```

exprimant le fait que les ordres de déconnection provoquent la déconnection effective des sites, que les messages envoyés sont stockés par la ligne et valident l'envoi d'un message par la ligne à l'autre site, et que sous l'hypothèse d'équité, l'envoi d'un message par un site provoque la réception d'un message par l'autre site, sauf s'il se déconnecte avant.

L'analyseur a montré que les formules suivantes ne sont pas valides pour la description fournie :

```
[] POT ENABLE(LINE)

INIT => INEV fina
INIT => INEV finb
INIT => INEV fin

[] AFTER(recal) => INEV AFTER(sendlb)
[] AFTER(recbl) => INEV AFTER(sendla)

[] AFTER(recal) => INEV (finb OR AFTER(sendbl))
[] AFTER(recbl) => INEV (fina OR AFTER(sendal))
```

ce qui exprime le fait que la ligne peut être définitivement inactive (lorsque les deux sites sont déconnectés), que la déconnection d'un site n'est pas inévitable, et que l'envoi d'un message par un site n'implique pas sa réception par l'autre site, même si ce site reste indéfiniment connecté.

La description que nous avons proposée du système de transmission avec déconnection répond correctement aux objectifs fixés par l'énoncé du système à réaliser.

La ligne échange des messages avec les sites dans les deux sens, jusqu´à réception d´un ordre de déconnection venant d´un site, après quoi elle ne communique plus avec ce site. L´ordre des messages est effectivement préservé, grâce à la présence de buffers de stockage. De plus, sous l´hypothèse de l´équité, les messages envoyés par un site finissent par être reçus par l´autre site, à moins que celui-ci ne se déconnecte entre temps.

Nous pourrions décrire le fonctionnement des sites ´a´ et ´b´ de façon différente, de telle sorte qu´au moment où un site attend un message, il ne puisse pas exécuter une autre action avant d´avoir reçu le message. Dans une telle description, il n´est pas nécessaire de se limiter aux séquences équitables pour prouver que les messages émis parviennent à destination, sauf si le site récepteur se déconnecte entre temps. Nous avons préféré simplifier la description des sites, pour ne décrire que leurs possibilités de communication avec la ligne.

Le choix de la taille des buffers (fixée à 2) a été imposé pour des raisons de capacité du système QUASAR. Ce nombre n´est pas significatif pour la description du système étudié, et l´on admet volontiers que la validité des propriétés comportementales est indépendante de la capacité des buffers. On constate cependant que le temps nécessaire au calcul du graphe d´états est important (environ 1´), et qu´un système du même type qui nécessite pour une raison quelconque des buffers de taille supérieure ne peut pas être analysé par QUASAR.

Il s´agit là sans nul doute d´une sérieuse limitation du système d´analyse, qui s´avère en pratique incapable de traiter des applications de grande taille. Fort heureusement, de nombreux systèmes peuvent être simplifiés, de façon à restreindre leurs espaces d´états. Il s´avère cependant que les limites liées à l´implémentation de QUASAR, même si elle peuvent être reculées par une amélioration des techniques de programmation ou par une augmentation de la capacité du calculateur, correspondent malgré tout à une certaine impraticabilité d´une théorie pourtant bien définie.

6.4. RESEAU DE TRAFIC FERROVIAIRE

Le système de trafic ferroviaire est constitué des six tâches composées TRACKij (i,j=1..3, i<>j) représentant chacune une voie. Chaque tâche TRACKij est formée d´un ensemble de tâches élémentaires SECTIONijk représentant les secteurs successifs dont est composée la voie.

La compilation de la description du système a produit un système conditions-actions comportant 17 commandes gardées, et dont l´espace

d'états contient 456 états. Les temps d'exécution ont été 22'' pour la compilation et 20'' pour la génération du graphe d'états.

La description analysée ne comporte que les deux étiquettes associées aux actions de la tâche SECTION121 et implique un graphe de 456 états. La même description, mais ne comportant pas d'étiquette, produit un graphe de 330 états. On constate une fois de plus que l'utilisation d'étiquettes dans la description accroît de façon sensible le nombre d'états, et qu'il s'avère impossible d'étiqueter les actions de toutes les tâches élémentaires du système.

L'évaluation des formules de spécification a confirmé les résultats que nous avions annoncés au chapitre III, à savoir la validité dans le système des formules suivantes :

```
[] ENABLE

[] (INIT AND NOT full) => ALL POT ENABLE(to121)
[] (INIT AND NOT full) => ALL POT ENABLE(from121)
```

exprimant que le système ne peut pas se bloquer, et que les actions d'arrivée et de départ d'un train pour le secteur 1 de la voie 12 sont vivantes (il en est évidemment de même pour les autres secteurs) ;

```
INIT => NOT full
(INIT AND NOT full) => ALL NOT full

[] sigma4

[]     B121 in [0..1]
   AND (B122 in [0..1])
         - - - - -
   AND (B323 in [0..1])
```

exprimant le fait que le réseau n'est pas saturé au départ et qu'il ne le deviendra jamais, que le nombre de trains est constant dans le réseau et que chaque secteur contient exactement 0 ou 1 train, ce qui implique que deux trains ne peuvent pas figurer en même temps sur un secteur donné.

La description que nous avons proposée respecte fidèlement les principes du système à modéliser. Les trains peuvent changer de voie uniquement aux noeuds du réseau, et ne peuvent pénétrer que sur un secteur ne contenant pas déjà un train. L'initialisation du système garantit par ailleurs qu'un secteur ne contient pas deux trains à un instant donné.

Le principe que nous avons adopté est de plus indépendant du nombre de trains dans le réseau et du nombre de secteurs par voies.

La description du système peut être enrichie par l'adjonction d'une tâche qui injecte des trains dans le réseau, ou qui en retire. Certaines des propriétés ne sont alors plus vérifiées, par exemple la constance du nombre de trains dans le réseau.

L'approche que nous avons suivie (exécution parallèle de tâches représentant les voies, décomposition d'une voie en un ensemble de secteurs au sein d'une tâche composée, passage d'un train d'un secteur à un autre par échange d'une variable) a été adoptée pour illustrer les possibilités de description modulaire de QUASAR, et la conception d'une application de façon hiérarchisée à partir des modules. Bien que le système décrit relève effectivement des applications réparties, il semble plus naturel de décrire immédiatement le réseau par un système conditions-actions sans échange, celui qui est produit par l'opération de composition des tâches de notre description.

6.5. TRANSMISSION SYNCHRONE ET ASYNCHRONE

Nous présentons pour cet exemple la session d'analyse telle qu'elle s'est déroulée face à l'ordinateur ; nous insérons dans le texte des parties du listing de la session, et commentons les opérations effectuées.

La session débute par l'appel du programme "ancas", et par la définition du nom du système étudié (ce nom est "ex5").

```
ancas

QUASAR ?cas ex5
QUASAR ?e pr ex5.cas

cotask ex5;

  task sendrec;
    input in;
    output sout,aout;
  do
    true : ?in |
    true : !sout:=0 |
    true : !sout:=1 |
    true : !aout:=0 |
    true : !aout:=1
  od ;

  task object;
    input ins,ina;
    output out;
  declare
    packa : 0..1 ;
    packs : 0..1 ;
    clock : 0..1 ;
  init
    packa:=0,clock:=0 ;
  do
    {recsyn} true : packa:=?ina |
    {recsyn} clock=0 : packs:=?ins, clock:=1 |
    {sendpacks} (clock=1) and (packs<>0) :
                !out:=packs, clock:=0 |
    {sendpacka} (clock=1) and (packs=0) :
                !out:=packa,packa:=0,clock:=0
  od;

port syninput,synoutput,asynoutput ;
body
    SENDREC(synoutput,syninput,asyninput) //
    OBJECT(syninput,asyninput,synoutput).
```

Nous effectuons désormais la première phase de la compilation du programme de description (qui se trouve dans le fichier "ex5.cas") ; analyse syntaxique et génération du système conditions-actions représentant l'application. Nous donnons le contenu des fichiers "ex5.cas.tab" et "ex5.compo" décrivant respectivement la table des identificateurs et le système conditions-actions.

```
QUASAR ?comp -sy -id
QUASAR ?e pr ex5.cas.tab
```

TABLE D'IDENTIFICATEURS DU CAS : ex5

No identificateur	Nom identificateur	borne sup	borne inf
1	packa	1	0
2	packs	1	0
3	clock	1	0

```
QUASAR ?e pr ex5.compo

C
{#9}{sendpacka}{#1} ($3=1) and ($2=0) : $1:=0,$3:=0 |
{#8}{sendpacks}{#1} ($3=!) and ($2<>0) : $3:=0 |
{#3}{#7}{recsyn} $3=0 : $2:=1,$3:=1 |
{#2}{#7}{recsyn} $3=0 : $2:=0,$3:=1 |
{#5}{#6}{recasyn} true : $1:=1 |
{#4}{#6}{recasyn} true : $1:=0 |
```

Le programme contenu dans le fichier "ex5.cas.obj.pascal" est alors analysé par le compilateur PASCAL.

```
QUASAR ?comp -comp
PASCAL 7.01
QUASAR ?
```

Le graphe d'états est construit par exécution de la commande sim. Nous utilisons l'option "-gr", pour générer un fichier "ex5.cas.gr" contenant une description lisible du graphe d'états.

```
QUASAR ?sim -gr -af
generation du graphe
le nombre d'etats du graphe est : 17
sauvegarde du graphe
calcul et sauvegarde des variables enable et after
QUASAR ?
```

Nous évaluons maintenant les prédicats sur les variables de l'application, qui sont décrits dans le fichier "ex5.cas.predicat". Les formules de spécification que nous évaluerons ultérieurement n'utilisent pas de prédicats sur les variables de l'application. Nous définissons cependant deux prédicats C1 et C0 pour montrer la déclaration et l'évaluation des prédicats.

```
QUASAR ?e pr ex5.cas.predicat

C1 = clock=1 ;
C0 = clock=0 ;

QUASAR ?def
PASCAL 7.01
le predicat c1          est verifie par : 8 etat(s)
le predicat c0          est verifie par : 9 etat(s)
```

Nous avons éxecuté toutes les commandes nécessaires pour pouvoir évaluer les formules de spécification. Nous soumettons désormais au système d'analyse les formules une par une, au moyen de la commande "eval".

```
QUASAR ?eval [] enable
formule valide pour le systeme

QUASAR ?eval [] pot enable(sendrec)
formule valide pour le systeme

QUASAR ?eval [] pot enable(object)
formule valide pour le systeme
```

L'analyse prouve ainsi l'absence de blocage du système et la vivacité des tâches SENDREC et OBJECT.

Nous allons désormais vérifier que la tâche OBJECT, après avoir reçu un paquet à l'entrée synchrone, ne peut pas recevoir un nouveau paquet à cette même entrée, tant qu'elle n'a pas émis le paquet à la sortie synchrone.

Au premier abord, la formule suivante permet d'exprimer cette propriété :

```
[] AFTER(recsyn) => ALL[NOT AFTERSEND] NOT ENABLE(recsyn)
```

Si nous soumettons la formule à l'analyseur, nous nous apercevons qu'elle n'est pas vérifiée par la description fournie.

```
QUASAR ?eval []( after(recsyn)=>all[not after(sendpacka,sendpacks)]
                          not enable(recsyn) )
formule non valide
formule verifiee par : 13 etat(s) sur 17 etats.
```

Une étude plus approfondie nous montre que la formule n'est pas valide, car les deux preddicats AFTERSEND et ENABLE(recsyn) peuvent être vrais simultanément. La formule exacte qui caractérise la propriété proposée est la suivante :

```
[] AFTER(recsyn) => ALL[NOT AFTERSEND]
                   ENABLE(recsyn) => AFTERSEND
```

Cette nouvelle propriété est vérifiée par notre description, comme le prouve l'analyseur :

```
QUASAR ?eval []( after(recsyn)=>all[not after(sendpacka,sendpacks)]
                   (enable(recsyn)=>after(sendpacka,sendpacks)) )
formule valide pour le systeme
```

Nous pouvons étudier formellement ce qui sépare les deux formules, en utilisant la procédure de décision de la logique temporelle CTL.

```
( nous notons
    f1 :  a => all [not b] c => b
    f2 :  a => all [not b] not c    )
```

```
QUASAR ?decide (a=>all[not b](c=>b))<>(a=>all[not b] not c)

**  CTL decision system  **


Votre formule n'est pas valide dans CTL :

( a => ALL[ NOT b](c => b)) <>(a => ALL[ NOT b] NOT c)

Voici un modele qui invalide la formule :


monde     3
  a          = 1
  b          = 1
  c          = 1
```

Le démonstrateur de théorèmes confirme ainsi que les formules ne sont pas équivalentes. Pour tenter de classer les deux formules, nous proposons au démonstrateur les deux implications (f1=>f2) et (f2=>f1).

```
QUASAR ?decide (a=>all[not b](c=>b))=>(a=>all[not b] not c)

**  CTL decision system  **


Votre formule n'est pas valide dans CTL :

( a => ALL[ NOT b](c => b)) =>(a => ALL[ NOT b] NOT c)

Voici un modele qui invalide la formule :


monde     2
  a         = 1
  b         = 1
  c         = 1


QUASAR ?decide (a=>all[not b] not c)=>(a=>all[not b](c=>b))

**  CTL decision system  **


Votre formule est un theoreme de CTL :

(a => ALL[ NOT b] NOT c) =>( a => ALL[ NOT b](c => b))
```

Nous constatons que la formule (f1=>f2) n'est pas valide, le modèle éxhibé ne comportant qu'un état pour lequel les variables propositionnelles b et c valent 1 simultanément.

Ces résultats confirment le fait que la deuxième propriété peut être valide pour notre système, quand la première ne l'est pas.

<u>Remarque</u>

L'utilisation de la procédure de décision pour les deux formules proposées n'était pas indispensable. Il était facile de mesurer la distance entre les formules, uniquement par la différence entre les sous formules arguments de l'opérateur conditionnel ALL.

Nous vérifions cependant que l'expression d'une propriété par une formule de la logique n'est pas une chose évidente, car les formules construites ne traduisent pas toujours ce que l'on souhaite qu'elles expriment. L'utilisation d'une procédure de décision peut contribuer à aider la spécification, en comparant et en classant les formules.

Nous allons vérifier maintenant que la tâche OBJECT, après avoir reçu un paquet à l'entrée synchrone, émet effectivement ce paquet à la sortie synchrone.

```
QUASAR ?eval [](  after(recsyn)=>inev[not enable(recsyn)]
                                  after(sendpacka,sendpacks) )
formule non valide
formule verifiee par : 13 etat(s) sur 17 etats.

QUASAR ?eval [](  after(recsyn)=>fair[not enable(recsyn)]
                                  after(sendpacka,sendpacks) )
formule valide pour le systeme
```

Nous constatons que la propriété n'est vérifiée que si l'on considère uniquement les fonctionnements équitables du système.

La présentation du système à modéliser étant assez peu précise, nous pouvons difficilement juger la conformité de notre description avec les impératifs de l'application.

Nous avons décrit un objet possédant une entrée asynchrone, une entrée et une sortie synchrones. Les paquets ont tous la même taille, et la sortie réémet les paquets de l'entrée synchrone, en les substituant lorsqu'ils sont vides par les paquets de l'entrée asynchrone. Nous avons donc répondu aux exigences du système, même si nous avons simplifié la description par l'abandon du stockage des paquets de l'entrée asynchrone. Le principe de mémorisation d'objets par des buffers circulaires a d'ailleurs déjà été utilisé dans le troisième exemple.

L'étude de cet exemple est intéressante à deux points de vue.

Elle confirme la nécessité d'un démonstrateur de théorèmes de la logique temporelle utilisée pour les formules de spécification, afin de comparer formellement les formules et de bien situer ce qui sépare deux propriétés intuitivement proches. C'est un outil indispensable pour apprendre à spécifier, pour comprendre avec précision le vocabulaire de la logique temporelle. Il s'avère de plus très utile dans les phases de spécification, même pour les personnes possédant une longue expérience de manipulation des formules temporelles.

Le système décrit soulève le problème de la représentation des événements synchrones dans QUASAR. Aucun travail n'a été fait dans cette direction, car l'analyseur se contente d'évaluer des propriétés statiques des systèmes. L'utilisateur est donc livré à lui-même, et doit imaginer des descriptions qui lui permettent de simuler par divers moyens le "caractère synchrone" de certaines actions.

Les extensions possibles de QUASAR que nous avons introduites pour l'exemple de l'exclusion mutuelle de DECKER (notion de pas d'exécution, fonctionnement parallèle forcé des tâches) peuvent

servir de base à l'introduction d'une notion de temps dans la description des applications.

L'utilisateur définit dans la description des durées d'actions. Une durée est un nombre entier positif (qui vaut 1 par défaut), la durée 1 correspondant à un pas d'exécution du système. Chaque action de durée n est décomposée en n actions successives de durée 1, la première correspondant au traitement à effectuer, les (n-1) suivantes ayant pour but de faire attendre la tâche pour que l'action initiale dure n pas d'exécution.

L'évolution du système par pas d'exécution simule alors un fonctionnement "temporisé" de l'application. Munis d'une fonction "temps" commune à toutes les tâches du système, nous pouvons décrire des applications dans lesquelles interviennent le temps.

Le principe de temporisation des modèles asynchrones a été étudié dans [SIF79] et [SCH81]. Les méthodes présentées dans [SCH81] peuvent être utilisées pour enrichir le système d'analyse QUASAR, et ouvrir ses portes à un grand nombre d'applications qu'il n'est pas susceptible de traiter actuellement.

6.6. RESEAU DE DISTRIBUTEURS DE BILLETS

La description que nous avons présentée (cf II.2.10.6) de ce système n'a pas pu être soumise à l'analyseur, car le programme de description dépasse les limites imposées par la capacité de QUASAR. Aussi avons nous testé une version plus simple de ce programme, ne comportant qu'un seul distributeur de billets.

La compilation de cette description a produit un système conditions-actions comportant 22 commandes gardées et dont le graphe d'états contient 22 états. Les temps d'exécution ont été 14'' pour la compilation et 2'' pour la construction du graphe.

Les formules suivantes ont été prouvées valides par notre analyseur :

[] ENABLE

[] POT ENABLE(cardin1)
[] POT ENABLE(cardout1)
[] POT ENABLE(commandeli) (i = 1,2,3)

exprimant l'absence de blocage du système, et la vivacité des fonctions du distributeur ;

[] base_out => NOT base_busy

[] dis_busy_1 => NOT ENABLE(cardinl)

exprimant le fait que la base ne peut pas tomber hors d'usage durant une communication, et que deux utilisateurs ne peuvent pas se servir simultanément du distributeur.

Ce dernier exemple situe sans ambiguité les possibilités réelles d'analyse offertes par QUASAR. En partant d'un énoncé sans grandes difficultés, nous avons été amenés dans un premier temps à décrire le système de façon incomplète, par l'abandon de tout ce qui concerne les problèmes liés aux sommes d'argent. Cette simplification n'a cependant pas été suffisante, puisque la description ainsi obtenue ne peut toujours pas être analysée. Cet exemple constitue donc un sérieux échec, que nous n'avons pas voulu dissimuler, et qui doit être la source d'une définition précise du domaine d'application de QUASAR.

L'ensemble des applications que QUASAR est capable de traiter est assez limité, et ne sous semble pas susceptible d'être accrû de façon sensible, même par des efforts importants. On peut très certainement enrichir le langage de description, augmenter la taille des graphes d'états, mais on ne parviendra jamais à faire de QUASAR un système ouvert à toutes les applications réparties réelles, car le principe d'évaluation par couverture exhaustive de l'espace d'états a montré ses limites pratiques.

Nous croyons néanmoins volontiers que des procédés nouveaux peuvent dans un proche avenir supprimer une grande part des problèmes de limitation de QUASAR, par une augmentation importante des mémoires vives des calculateurs. Il n'est pas impossible qu'un jour, les ordinateurs possèdent des capacités 100 ou 1000 fois plus grandes qu'actuellement. Ce jour là, un système d'analyse tel que QUASAR pourra ouvrir ses portes à un nombre très grand d'applications. Ce seul espoir justifie entièrement à nos yeux la persistance que nous manifestons à démontrer l'utilité de notre principe d'analyse. Ce n'est pas la faible capacité de la maquette réalisée qui ternit de quelque façon cette motivation.

7. CONCLUSION

Nous avons présenté dans ce chapitre le mode d'emploi du programme qui réalise le système d'analyse QUASAR. La constitution des programmes a été menée dans le but de satisfaire un compromis acceptable entre les points suivants :

- une utilisation aisée et agréable de l'analyseur, par l'intermédiaire d'un langage de commandes simplifié et facilement assimilable ;
- la présence d'un environnement QUASAR, muni de ses fonctions de gestion, pour permettre à l'utilisateur de créer des bibliothèques de systèmes répartis ;
- un interface programme-utilisateur le plus proche possible de celui employé par le système d'exploitation MULTICS, pour faciliter la tâche de l'utilisateur ;
- une possibilité permanente pour l'utilisateur de contrôler les opérations qu'il a effectuées, grâce à la génération automatique d'un grand nombre de fichiers de vérification exploitables ;
- une utilisation calculée (bien que non obligatoirement optimisée) des possibilités offertes par les langages de programmation PASCAL et PLI pour la définition des types, afin d'augmenter au maximum la capacité de l'analyseur, et d'accroître la rapidité des algorithmes de calcul.

Il n'est pas aisé de répondre pleinement aux impératifs que nous avons énumérés ci-dessus, car les facilités d'utilisation, bien que légitimes, provoquent très souvent une diminution de rendement des programmes ; ainsi par exemple les différentes sauvegardes d'informations sont la source d'une certaine lenteur d'exécution, sans que leur principe puisse être remis en cause.

Nous avons pour notre part tenté de trouver un équilibre satisfaisant entre l'"utile" et l'"agréable", mais en cherchant avant tout à atteindre notre objectif d'élaborer un prototype de système d'analyse totalement achevé. Nous avons conscience que notre but a été atteint en partie au détriment de l'efficacité de notre produit.

CHAPITRE VI

```
****************************
*                          *
*        CONCLUSION        *
*   BILAN ET PERSPECTIVES  *
*                          *
****************************
```

1. Bilan de l'expérience
2. Propositions d'améliorations
3. Conclusion

1. BILAN DE L'EXPERIENCE

La réalisation du système QUASAR a été effectuée dans le cadre du projet CESAR, développé au sein de l'équipe "LANGAGES" de l'IMAG. De nombreuses personnes ont contribué à ce projet, tant sur le plan théorique de la conception (S. GRAF, J.P QUEILLE, J. SIFAKIS) que sur celui de la programmation des algorithmes (J.P. CHARBONNIER, P. MARTINET, J.C. MARTY, J.P. QUEILLE, J.P. SCHWARTZ).

Le système CESAR utilise des techniques diverses dont certaines sont récentes et peu classiques (calcul itératif de prédicats, logique temporelle). Il nous a donc semblé indispensable de tester la validité de l'approche et la faisabilité des algorithmes. Avant de concevoir QUASAR, plusieurs parties de CESAR ont fait l'objet d'une étude programmée :

- une version expérimentale du compilateur de CESAR a été réalisée : analyse syntaxique et génération des réseaux de Petri interprétés correspondant aux tâches élémentaires [RS81], composition de ces tâches et élimination des variables non significatives [CHA82]. Bien que simplifié (sous langage de CESAR, traitement d'erreurs sommaire), le compilateur est assez évolué pour être démonstratif.

- deux méthodes d'analyse des formules de spécification ont été testées ;

* la première admet des systèmes conditions-actions booléens et calcule les opérateurs ALL, SOME et SONT non conditionnels, par calcul itératif sur des expressions booléennes [MAY81]. Le codage des réseaux de Petri interprétés en systèmes conditions-actions booléens a été étudié, mais non réalisé ;

* la deuxième est celle utilisée dans l'analyseur QUASAR, admettant des systèmes conditions-actions à variables entières bornées, basée sur la construction du graphe des états d'une application et l'évaluation des formules sous forme de calcul de l'ensemble des états qui vérifient la formule.

L'ensemble des programmes réalisés a permis de prouver la faisabilité des algorithmes utilisés par CESAR. Les programmes mis au point sont peu performants, mais justifient pleinement la démarche adoptée. Nous avons constaté tout au long du développement du projet que des moyens plus importants doivent permettre d'optimiser les programmes établis, ce que nous n'avons pas tenté de faire.

2. PROPOSITIONS D'AMELIORATIONS

Le système QUASAR est une version expérimentale du projet CESAR, mise au point pour prouver la faisabilité des algorithmes impliqués par CESAR. Bien qu'ayant constaté un certain nombre de limitations pratiques de la méthode, nous avons établi que des investissements plus conséquents en temps et en argent permettent de réaliser une maquette très proche du projet initial CESAR.

Le langage de description utilisé par QUASAR est très pauvre, et peut sans peine être enrichi de nombreuses fioritures syntaxiques pour améliorer l'aisance de description des applications. Deux efforts particuliers nous semblent nécessaires.

Premièrement, une possibilité de définition de types plus élaborés doit être offerte, pour permettre la construction de types structurés et la déclaration de variables, échangées ou non, de ces types.

Deuxièmement, la présence d'instructions composées itératives, alternatives et de séquencement doit être adjointe au non déterminisme pour permettre de construire les tâches élémentaires de façon structurée, afin d'éviter le séquencement des actions par emploi de compteurs ordinaux.

On peut ainsi réaliser un langage de description qui respecte les principes de CESAR (notions de tâches élémentaires et composées, variables échangées selon deux modes), et dont la description des processus élémentaires est proche d'un langage tel que CSP.

Le langage de spécification doit être enrichi pour lui conférer un formalisme moins mathématique et plus abordable à l'utilisateur. De plus, il serait souhaitable de réaliser un véritable système d'aide à la spécification qui permet de manipuler, de transformer, de comparer et de stocker les formules de spécification.

La méthode d'évaluation utilisée par QUASAR ne semble pas devoir être remise en cause (elle a donné de bons résultats), mais elle peut être améliorée par l'emploi de techniques de réduction du graphes d'états : réduction en cours de construction (identique à celle employée pour les formules de la logique S4), ou en vue de la démonstration d'une formule particulière. Nous pensons cependant que ce dernier point est sujet à caution, de par son principe même.

Une version expérimentale de CESAR ainsi faite, soigneusement implémentée sur un calculateur de forte capacité, constituerait indéniablement un outil de travail utile et agréable.

3. CONCLUSION

Le coté positif du projet est certainement de fournir une preuve du bien fondé et de la faisabilité des principes du système CESAR. Il constitue une véritable méthode de preuve des applications qui font intervenir le parallélisme, et cette méthode est automatisable lorsqu'on se limite aux systèmes dont l'espace d'états est fini.

Sa réalisation représente une expérience enrichissante pour le programmeur dans le domaine de la conception de systèmes informatiques, car la diversité des principes employés dans la maquette QUASAR recouvre une large part des techniques usuellement utilisées (compilation, monitoring, gestion d'environnements utilisateur, ...). De nombreuses autres techniques ont par ailleurs été étudiées (calcul de points fixes d'opérateurs monotones, construction et parcours de graphes, ...), et l'élaboration des algorithmes a fourni des enseignements précieux sur l'éventail des procédés de programmation. Ce travail impose cependant un lien étroit avec le système d'exploitation du calculateur sur lequel est implémenté le projet, et rend en pratique non translatables les programmes qui le constituent.

L'aspect négatif de nos travaux est de n'avoir pas pu réaliser un projet aussi élaboré que nous le souhaitions. Un manque de moyens évident a largement contribué à ce fait, et les difficultés matérielles chroniques rencontrées dans la mise au point des programmes ont constitué un facteur déterminant de perte de temps et de désorganisation de la programmation.

Le travail effectué justifie néanmoins pleinement l'étude et la réalisation d'un outil plus développé, vraisemblablement dans le milieu industriel où les moyens engagés doivent permettre d'aboutir à un véritable système d'aide à la conception et à l'intégration des applications réparties.

ANNEXE : ORGANISATION DES PROGRAMMES

1. Généralités
2. La structure du programme QUASAR
3. Les structures de données
4. Les modules de programme

1. GENERALITES

Le programme qui réalise le système d'analyse QUASAR a été mis au point sur un ordinateur HB68 de la firme CII Honeywell Bull, fonctionnant sous le système d'exploitation MULTICS.

L'écriture des programmes s'est faite dans le langage de programmation PASCAL. Le choix du langage PASCAL n'est pas le fruit d'une étude comparée des différents langages disponibles sur MULTICS ; il est du uniquement au fait qu'il s'agit du langage de haut niveau avec lequel les personnes qui ont contribué à la réalisation de QUASAR étaient le mieux familiarisées.
Quelques sous programmes ont été écrits dans le langage PLI pour parvenir à utiliser dans le programme les procédures du système MULTICS.

L'ensemble des programmes réalisés représente quelques 10000 lignes de PASCAL, auxquelles s'ajoutent une vingtaine de procédures PLI. Pour des raisons évidentes d'efficacité, la programmation a été effectuée de façon modulaire, en utilisant pleinement les possibilités de compilation séparée offertes par le langage PASCAL. Le programme QUASAR est ainsi constitué d'un programme principal et d'un ensemble de sous programmes, chacun d'eux correspondant à un traitement déterminé de l'analyse. Chaque sous programme est lui-même formé d'un ensemble plat de procédures PASCAL qui exécutent des parties du traitement effectué par le sous programme. En fait, la notion de sous programme n'est pas réellement présente dans QUASAR, car toutes les procédures sont au même niveau. Elles sont uniquement regroupées au sein de modules géographiques, pour assurer la segmentation du programme et la proximité des procédures faisant partie d'un même traitement.

2. LA STRUCTURE DU PROGRAMME QUASAR

Avant de présenter de façon plus détaillée la structure du programme d'analyse, et les différents modules qui le constituent, nous rappelons brièvement les principes de la compilation séparée du langage PASCAL, qui ont dirigé la réalisation du programme.

2.1 LA COMPILATION SEPAREE PASCAL

Un programme PASCAL est constitué généralement d'un programme principal faisant appel à un ensemble de sous programmes, chaque

sous programme étant lui-même composé d'un programme principal et d'un ensemble de procédures. Le langage offre la possibilité de répartir ces sous programmes dans des segments physiques distincts, et de compiler les segments séparément. Les avantages sont doubles : la programmation est modulaire et structurée, et les modifications d'un sous programme n'imposent pas de recompiler le programme dans sa totalité.

La liaison entre les différents segments s'effectue par l'intermédiaire de variables dites "externes", qui sont partagées par les modules. Les procédures figurant dans un segment peuvent être exportées et utilisées par d'autres segments. L'édition de liens est faite dynamiquement au cours de l'exécution du programme.

2.2 LES CONSTITUANTS DU PROGRAMME QUASAR

Le programme d'analyse est constitué de 40 segments de programme, répartis de la façon suivante :

- un segment "dcltype.incl.pascal" qui contient la déclaration de tous les types structurés utilisés dans les segments de programme. Ce segment est inclus dans les modules lors de la compilation, pour éviter de reproduire le même texte dans chaque segment ;

- un segment "dcl.pascal" qui contient uniquement la déclaration des variables externes du programme. Pour des raisons liées à MULTICS, les fichiers ne sont pas déclarés dans ce module (les fichiers externes doivent figurer dans le programme principal) ;

- 24 segments de programme PASCAL, contenant les procédures de traitement du système d'analyse. Ces modules sont formés uniquement d'un ensemble plat de procédures PASCAL, dont certaines sont déclarées externes pour pouvoir être utilisées par les procédures des autres segments. Un de ces modules, de nom "ancas.pascal", contient de plus le programme principal et la déclaration des fichiers externes ;

- 14 segments contenant chacun une procédure PLI. Ces procédures effectuent des appels à des fonctions du système MULTICS, qui ne peuvent généralement pas être utilisées directement par les programmes PASCAL. Les fonctions de MULTICS servent principalement à la gestion programmée des fichiers manipulés par QUASAR (vérification de l'existence ou de l'ouverture d'un fichier, fermeture ou desctruction d'un fichier), et à l'exécution de commandes MULTICS au sein de l'environnement QUASAR.

Nous ne souhaitons pas présenter en détail les différents modules

de QUASAR. Nous nous contentons de définir les structures de données retenues pour l'analyse, et de donner un bref aperçu du traitement effectué par chaque module.

3. LES STRUCTURES DE DONNEES

Nous présentons dans ce paragraphe les structures de données qui sont utilisées par le programme d'analyse. Nous expliquons les raisons de nos choix, ou les contraintes qui ont été imposées par le langage de programmation ou par son implémentation. Nous analysons successivement la définition des types et la gestion des fichiers maniés par l'analyse.

3.1 LA DEFINITION DES TYPES

Nous ne nous intéressons pas à la représentation de tous les objets manipulés par QUASAR, mais uniquement à ceux qui jouent un rôle important pour les performances et la capacité du programme. Ils sont au nombre de deux, et concernent la représentation du graphe des états d'une application : les états, les ensembles d'états et la relation d'accessibilité du graphe.

a) La définition d'un type "état" par un tableau d'entiers nous a semblé naturelle et commode. Elle présente cependant deux inconvénients majeurs :

- le type tableau doit avoir des bornes fixes, indépendamment de l'application traitée, ce qui impose de limiter une fois pour toutes le nombre de variables que peut comporter un programme de description ;
- l'occupation mémoire d'un état est importante (puisque les éléments du tableau sont du type entier universel), alors que les variables du programme de description possèdent généralement des intervalles de définition de petites tailles.

Il ne nous parait pas possible de représenter les états de façon plus condensée, sans imposer une taille maximale aux domaines de définition des variables d'une description.

Le rangement des états au fur et à mesure de la construction du graphe des états pose un problème de place mémoire lié au système MULTICS. La taille totale occupée par une variable ne pouvant pas excéder un certain seuil, il n'est pas possible de regrouper les états dans un tableau d'états, qui dépasserait la limite de taille autorisée. Nous avons choisi de stocker chaque état dans un enre-

gistrement PASCAL, et d'accéder aux états par un tableau de pointeurs vers les enregistrements. Ainsi le tableau possède une taille raisonnable et les états sont stockés dans la zone de travail de PASCAL dont la taille peut être définie par l'utilisateur.

b) La représentation des ensembles d'états par un type du langage PASCAL ne peut être résolue de façon triviale. Le langage propose certes un type pour les ensembles, mais la taille des ensembles est limitée à 278 éléments. Pour utiliser des ensembles plus grands, il faut construire un type "tableau d'ensembles". Nous avons adopté cette solution, mais nous avons conscience de n'avoir peut être pas fait le meilleur choix.

La représentation mémoire d'un tel ensemble est indépendante du nombre d'éléments dans l'ensemble, et occupe la place de l'ensemble d'états tout entier. Les ensembles d'états sont souvent creux, d'où une perte considérable de place. Cette solution offre cependant des avantages non négligeables, car elle facilite l'écriture des programmes de calcul sur les ensembles d'états, et produit des calculs rapides.

Une solution moins gourmande en place mémoire consiste à représenter les ensembles d'états par des listes chainées d'enregistrements, chaque enregistrement contenant un numéro d'état. Les calculs d'ensembles s'effectuent alors par des parcours de listes chainées. Nous n'avons pas testé cette représentation, car il nous parait certain qu'elle substitue les problèmes de place mémoire par des problèmes de temps d'exécution.

La relation d'accessibilité du graphe d'états est représentée dans QUASAR par la matrice de précédence du graphe. A chaque état est associé l'ensemble de ses états prédécesseurs directs dans le graphe. Ainsi le calcul de la fonction "pre" lors de l'évaluation des formules de spécification est immédiat.

Pour les raisons que nous avons explicitées auparavant, les ensembles d'états (pour la relation de précédence du graphe, et pour les ensembles caractéristiques des prédicats prédéfinis) sont stockés dans des enregistrements auxquels on accède par des tableaux de pointeurs.

Les choix que nous avons effectués pour la définition des types ont été motivés par le fait qu'ils simplifiaient l'écriture des programmes, et que nos expériences précédentes de calculs à partir de listes chainées (pour la méthode d'évaluation dans le treillis des prédicats) s'étaient avérées coûteuses et peu concluantes.

3.2 GESTION DES FICHIERS DE L'ANALYSE

Le programme d'analyse manipule un ensemble de fichiers qui peuvent être des fichiers d'entrée créés par l'utilisateur, des fichiers de travail ou de sauvegarde de l'analyseur, ou des fichiers de sortie destinés à l'utilisateur.

Le langage PASCAL propose des instructions de création, d'ouverture et de fermeture d'un fichier, ainsi que des opérations de lecture et d'écriture dans les fichiers. Il ne comporte cependant pas d'instructions qui permettent d'obtenir des renseignements sur un fichier physique (existence, longueur), et n'autorise aucune opération sur un fichier (renommage, destruction). Ainsi la tentative d'ouverture d'un fichier qui n'existe pas provoque une erreur d'exécution, et l'arrêt du programme en cours. Il en est de même pour la fermeture d'un fichier qui n'est pas ouvert. Afin d'éviter de tels problèmes, nous avons du munir notre analyseur de quelques fonctions de vérification des fichiers utilisés par QUASAR.

Le système MULTICS comporte un ensemble de fonctions de gestion de fichiers qui sont utilisables par programme, moyennant une procédure PLI intermédiaire. Nous avons écrit un ensemble de procédures de ce type, qui permettent de :

- tester l'existence d'un fichier, afin d'éviter les erreurs d'exécution par ouverture d'un fichier inexistant ;
- tester l'ouverture d'un fichier, pour ne pas fermer un fichier déjà clos, ou ne pas ouvrir un fichier ouvert ;
- tester si le fichier est vide (sa longueur vaut alors 0) ;
- détruire un fichier existant ; c'est le cas des fichiers temporaires créés par QUASAR lors de la traduction d'un programme de decription ;
- ajouter ou supprimer un nom à un fichier existant ; ce mécanisme est employé par QUASAR pour permettre à l'utilisateur de faire coexister plusieurs applications réparties :

  - l'analyseur génère pour chaque application étudiée trois procédures PASCAL dont les noms INITSYSTEME, SYSTEME et PREDICAT sont les mêmes pour toutes les applications ; ceci est indispensable pour que le programme QUASAR, qui utilise ces procédures, soit accepté par le compilateur PASCAL ;
  - or, il n'est pas possible sous le système MULTICS d'avoir plusieurs fichiers ayant le même nom ; QUASAR doit donc veiller à ce qu'une seule application à la fois puisse posséder un de ces noms ;
  - ainsi, avant de générer ces procédures (par appel programmé du compilateur PASCAL) pour l'application courante, QUASAR détruit le nom des procédures INITSYSTEME, SYSTEME ou PREDICAT existant ; de même, lorsque QUASAR appèle une de ces trois

procédures pour une application, il enlève le nom à la procédure qui le possède, pour le rajouter à la procédure de l'application.

Ces différentes procédures apportent une certaine sécurité dans l'emploi de QUASAR. Rien n'empêche cependant l'utilisateur de modifier les fichiers manuellement (par des commandes MULTICS, ou sous éditeur), dans quel cas il s'expose à d'éventuels problèmes lors d'une prochaine manipulation de ces fichiers par QUASAR.

4. LES MODULES DE PROGRAMME

Nous énumérons dans ce paragraphe l'ensemble des segments de programme qui forment le programme d'analyse. Nous précisons pour chacun d'eux le traitement effectué par les procédures qu'il contient.

ancas.pascal :

Ce module contient le programme principal, exécuté lors de l'invocation de QUASAR. Il effectue un certain nombre d'initialisations, et comporte les procédures "système" de QUASAR : impression des messages d'erreurs, lecture des commandes tapées par l'utilisateur et appel des procédures correspondantes.

cesar.pascal

Ce segment contient les programmes de traduction d'une description vers un programme PASCAL représentant l'application : analyse syntaxique, impression des erreurs de syntaxe, traduction des tâches élémentaires en langage intermédiaire, traduction du système vers un programme PASCAL (contenant les procédures INITSYSTEME et SYSTEME), et compilation de ce programme.

Les procédures de composition et d'encapsulation ne figurent pas dans ce segment, car elles ont été écrites indépendamment de QUASAR, avant d'être adaptées à ce système. Elles sont contenues dans les segments qlire.pascal, qrenum.pascal, qchainer.pascal, qrecopie.pascal, qtraiter.pascal, qpremierephase.pascal, qphase2.pascal, qphase3.pascal, qcompo.pascal, qfermeture.pascal.

cmd sim.pascal

Ce module contient les procédures de génération du graphe d'états d'une application : construction de l'ensemble des états initiaux, génération progressive du graphe par appel de la procédure SYSTEME pour chaque état, constitution éventuelle des fichiers contenant les descriptions lisibles du graphe d'états, construction des prédicats prédéfinis ENABLE(T), ENABLE(ea) et AFTER(ea).

<u>cmd predi.pascal</u>

Ce segment contient les procédures d'évaluation des prédicats sur les variables de l'application : analyse syntaxique des prédicats, traduction des prédicats vers un programme PASCAL (contenant la fonction PREDICAT), compilation de ce programme et calcul des ensembles caractéristiques des prédicats.

<u>cmd eval.pascal</u>

Ce module contient les procédures d'évaluation des formules de spécification : analyse syntaxique des formules temporelles et calcul des ensembles caractéristiques des formules.

<u>initialisation.pascal</u>

Ce segment contient un ensemble de procédures d'initialisation (de fichiers ou de variables), qui sont appelées par les procédures des segments déjà cités.

<u>gestion table.pascal</u>

Ce module contient les procédures utilitaires de gestion des variables figurant dans un programme de description : construction de la table des identificateurs, rangement d'un nouvel identificateur, procédure HCODE pour accélérer l'accès aux identificateurs, et recherche d'un identificateur dans la table.

<u>ss4.pascal</u>

Ce segment contient la procédure de décision de la logique temporelle CTL : analyse syntaxique des formules, recherche d'un modèle pour la négation de la formule, et impression éventuelle de ce modèle.

Les segments suivants contiennent une seule procédure :

<u>cmd print.pascal</u> : impression des descriptions lisibles du graphe d'états.

<u>cmd listcas.pascal</u> : liste des programmes de description disponibles.

<u>cmd cas.pascal</u> : définition d'un nom d'application, et exécution de quelques initialisations.

<u>commult.pascal</u> : envoi d'une ligne de commande MULTICS.

auxi.pascal : liste des commandes QUASAR, et pour chaque commande, impression de la syntaxe et du traitement effectué par la commande.

multauxi.pascal : procédure système de QUASAR ; initialisation de la zone contenant les enregistrements PASCAL, définition ou modification des règles de recherche (commandes "asr" et "dsr" de MULTICS).

existe file.pl1 : recherche de l'existence d'un fichier.

find file.pl1 : recherche de l'ouverture d'un ficher.

ajoutsuffixbis.pl1, ajoutsuffix.pl1 : concaténation de noms, puis recherche de l'existence d'un segment.

add nom.pl1 : ajout d'un nom à un segment.

del path.pl1 : retrait d'un nom à un segment.

del nom.pl1 : destruction d'un segment.

com mult.pl1 : exécution d'un commande MULTICS.

name.pl1, expand.pl1 : production du nom absolu d'un segment.

erreur pascal.pl1 : production du nombre d'erreurs rencontrées lors de la compilation PASCAL d'un programme.

traiter code.pl1 : impression d'un message d'erreur MULTICS.

statusb.pl1 : obtention de tous les renseignements concernant un segment ; cette procédure est utilisée par quelques unes des procédures PLI citées précédemment.

termine.pl1 : fermeture et détachement d'un segment.

REFERENCES

_ [ADA80] : "Reference manual for the ADA programming language" CII Honeywell Bull 1980

_ [BA82] : BEN-ARI M. "Complexity of proofs and models in programming logics" Tel-Aviv University 1981

_ [BMP81] : BEN-ARI M. & MANNA Z. & PNUELI A. "The temporal logic of branching time". POPL pp 164-176 1981

_ [CE82] : CLARKE E.M. & EMERSON E.A. "Design and synthesis of synchronisation skeletons using branching time temporal logic". Ed D.Kozen. Springer Verlag 1982

_ [CHA82] : CHARBONNIER J.P. "Composition des réseaux de Petri interprétés dans le cadre du projet CESAR". Rapport de projet ENSIMAG 1982

_ [COSY83] : Problem set for the workshop on the analasys of concurrent systems. Cambridge 1983

_ [CRO75] : CROCUS "Systèmes d'exploitation des ordinateurs" Ed. Dunod 1975

_ [DIJ68] : DIJKSTRA E.W. "Co-operating sequential processes in programming languages". NATO Advanced Study Institute. F.Genuys(ed.) pp 43-112 Academic Press New York 1968

_ [DIJ75] : DIJKSTRA E.W. "Guarded commands, nondeterminacy and formal derivation of programs". Communicaton of ACM 18-8 pp 453-457 1975

_ [EH82] : EMERSON E.A. & HALPERN J.Y. "Decision procedure and expressiveness in the temporal logic of branching time". 14 symposium on theory of computing. 1982

_ [GUI82] : GUIDACCI DA SILVEIRA G.F. "Sur la preuve des systèmes parallèles et distribués par la logique temporelle". Thèse d'état Université de Toulouse 1982

_ [HC68] : HUGHES G.E. & CRESWELL M.J. "An introduction to modal logic". Methuen & Co Ltd, London 1968

_ [HOA78] : HOARE. "Communicating sequential processes". CACM

vol.21 n.8 pp 666-677 1978

_ [LAM80] : LAMPORT L. "Sometimes is sometimes not never". 7 ACM Symposium on principles of programming languages pp 174-185 1980

_ [LEL81] : "Ditributed systems-Towards a formal approach". INRIA 1981

_ [MAY81] : MARTY J.C. "Analyse des systèmes conditions-actions par calcul itératif de prédicats". Rapport de DEA INP Grenoble 1981

_ [MM79] : MILNE G. & MILNER R. "Concurrent processes and their syntax". Journal of ACM 26-2 pp 302-321 79

_ [PAR69] : PARK D. "Fixpoint induction and proofs of program properties". Machine Intelligence 5, pp 59-78

_ [QUE81] : 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, computer society press pp 149-161 1981

_ [QUE82] : QUEILLE J.P. "Le système CESAR : description, spécification et évaluation des applications réparties". Thèse USMG 1982

_ [QS82] : 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 1982

_ [QSS83] : QUEILLE J.P. & SCHWARTZ J.P. & SIFAKIS J. "Description, Specification and Analysis in CESAR : four examples". Proceedings workshop on the analysis of concurrent systems. A paraitre dans "Lecture notes of computer science"

_ [RS81] : ROSIER M. & SCHWARTZ J.P. "Réalisation d'un traducteur du langage de description, "DADA", en réseaux de Petri interprétés". Rapport de projet ENSIMAG 1981

_ [SCH81] : SCHWARTZ J.P. "Expression de la sémantique des commandes temps-réel d'ADA en termes de réseaux de Petri interprétés" Rapport de DEA INP Grenoble 1981

_ [SIF79] : SIFAKIS J. "Le contrôle des systèmes asynchrones : concepts, propriétés et analyse statique". Thèse d'état USM Grenoble 1979

_ [TAR55] : TARSKI A. "A lattice-theorical fixpoint theorem and its applications". Pacific Journal of Mathematics 5, pp 285-305

AUTORISATION DE SOUTENANCE

VU les dispositions de l'article 3 de l'arrêté du 16 avril 1974,

VU les rapports de présentation de Messieurs
. PH JORRAND, Directeur de recherche et J. SIFAKIS, Chargé de recherche
. P. AZEMA, Maître de recherche

Monsieur SCHWARTZ Jean-Philippe

est autorisé à présenter une thèse en soutenance pour l'obtention du diplôme de DOCTEUR-INGENIEUR, spécialité "Informatique".

Fait à Grenoble, le 16 novembre 1983

Le Président de l'INP-G

D. BLOCH
Président
de l'Institut National Polytechnique
de Grenoble

P.O. le Vice-Président,