Spécification de la synchronisation par contraintes

Traian Muntean

THESE

présentée à

Université Scientifique et Médicale de Grenoble

Institut National Polytechnique de Grenoble

pour obtenir le grade de

DOCTEUR DE TROISIEME CYCLE

Spécialité : Informatique

par

Traian MUNTEAN

SPECIFICATION DE LA SYNCHRONISATION PAR CONTRAINTES

Thèse soutenue le 19 juin 1978 devant la Commission d'Examen :

Président : L. BOLLIET

Examinateurs : C. BOKSENBAUM
C. GIRAULT
S. KRAKOWIAK
M. SINTZOFF
J.P. VERJUS

UNIVERSITE SCIENTIFIQUE
ET MEDICALE DE GRENOBLE

Monsieur Gabriel CAU : Président
Monsieur Pierre JULLIEN : Vice Président

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

PROFESSEURS TITULAIRES

MM.	AMBLARD Pierre	Clinique de dermatologie
	ARNAUD Paul	Chimie
	ARVIEU Robert	I.S.N.
	AUBERT Guy	Physique
	AYANT Yves	Physique approfondie
Mme.	BARBIER Marie-Jeanne	Electrochimie
MM.	BARBIER Jean-Claude	Physique expérimentale
	BARBIER Reynold	Géologie appliquée
	BARJON Robert	Physique nucléaire
	BARNOU Fernand	Biosynthèse de la cellulose
	BARRA Jean-René	Statistiques
	BARRIE Joseph	Clinique chirurgicale
	BEAUDOING André	Clinique de pédiatrie et puériculture
	BELORIZKY Elie	Physique
	BERNARD Alain	Mathématiques pures
Mme.	BERTRANDIAS Françoise	Mathématiques pures
MM.	BERTRANDIAS Jean-Paul	Mathématiques pures
	BEZEZ Henri	Pathologie chirurgicale
	BLAMBERT Maurice	Mathématiques pures
	BOLLIET Louis	Informatique (IUT B)
	BONNET Jean-Louis	Clinique ophtalmologique
	BONNET-EYMARD Joseph	Clinique gastro-entérologique
Mme.	BONNIER Marie-Jeanne	Chimie générale
MM.	BOUCHERLE André	Chimie et toxicologie
	BOUCHEZ Robert	Physique nucléaire
	BOUSSARD Jean-Claude	Mathématiques appliquées
	BOUTET DE MONVEL Louis	Mathématiques pures
	BRAVARD Yves	Géographie
	CABANEL Guy	Clinique rhumatologique et hydrologique
	CALAS François	Anatomie
	CARLIER Georges	Biologie végétale
	CARRAZ Gilbert	Biologie animale et pharmocodynamie
	CAU Gabriel	Médecine légale et toxicologie
	CAUQUIS Georges	Chimie organique
	CHABAUTY Claude	Mathématiques pures
	CHARACHON Robert	Clinique oto-rhino-laryngologique
	CHATEAU Robert	Clinique de neurologie
	CHIBON Pierre	Biologie animale
	COEUR André	Pharmacie chimique et chimie analytique
	CONTAMIN Robert	Clinique gynécologique
	COUDERC Pierre	Anatomie pathologique

Mme.	DEBELMAS Anne-Marie	Matière médicale
MM.	DEBELMAS Jacques	Géologie générale
	DEGRANGE Charles	Zoologie
	DELORMAS Pierre	Pneumophtisiologie
	DEPORTES Charles	Chimie minérale
	DESRE Pierre	Métallurgie
	DESSAUX Georges	Physiologie animale
	DODU Jacques	Mécanique appliquée (IUT I)
	DOLIQUE Jean-Michel	Physique des plasmas
	DREYFUS Bernard	Thermodynamique
	DUCROS Pierre	Cristallographie
	GAGNAIRE Didier	Chimie physique
	GALVANI Octave	Mathématiques pures
	GASTINEL Noël	Analyse numérique
	GAVEND Michel	Pharmacologie
	GEINDRE Michel	Electroradiologie
	GERBER Robert	Mathématiques pures
	GERMAIN Jean-Pierre	Mécanique
	GIRAUD Pierre	Géologie
	JANIN Bernard	Géographie
	KAHANE André	Physique générale
	KOSZUL Jean-Louis	Mathématiques pures
	KLEIN Joseph	Mathématiques pures
	KRAVTCHENKO Julien	Mécanique
	KUNTZMANN Jean	Mathématiques appliquées
	LACAZE Albert	Thermodynamique
	LACHARME Jean	Biologie végétale
Mme.	LAJZEROWICZ Janine	Physique
MM.	LAJZEROWICZ Joseph	Physique
	LATREILLE René	Chirurgie générale
	LATURAZE Jean	Biochimie pharmaceutique
	LAURENT Pierre-Jean	Mathématiques Appliquées
	LEDRU Jean	Clinique médicale B
	LE ROY Philippe	Mécanique (IUT I)
	LLIBOUTRY Louis	Géophysique
	LOISEAUX Pierre	Sciences nucléaires
	LONGEQUEUE Jean-Pierre	Physique nucléaire
	LOUP Jean	Géographie
Melle	LUTZ Elisabeth	Mathématiques pures
MM.	MALINAS Yves	Clinique obstétricale
	MARTIN-NOEL Pierre	Clinique cardiologique
	MAZARE Yves	Clinique médicale A
	MICHEL Robert	Minéralogie et pétrographie
	MICOUD Max	Clinique maladies infectieuses
	MOURIQUAND Claude	Histologie
	MOUSSA André	Chimie nucléaire
	NOZIERES Philippe	Spectrométrie physique
	OZENDA Paul	Botanique
	PAYAN Jean-Jacques	Mathématiques pures
	PEBAY-PEYROULA Jean-Claude	Physique
	PERRET Jean	Semeiologie médicale (Neurologie)
	RASSAT André	Chimie systématique
	RENARD Michel	Thermodynamique
	REVOL Michel	Urologie
	RINALDI Renaud	Physique
	DE ROUGEMONT Jacques	Neuro-chirurgie
	SEIGNEURIN Raymond	Microbiologie et Hygiène
	SENGEL Philippe	Zoologie
	SIBILLE Robert	Construction mécanique (IUT I)

MM.	SOUTIF Michel	Physique générale
	TANCHE Maurice	Physiologie
	TRAYNARD Philippe	Chimie générale
	VAILLANT François	Zoologie
	VALENTIN Jacques	Physique nucléaire
	VAUQUOIS Bernard	Calcul électronique
Mme.	VERAIN Alice	Pharmacie galénique
MM.	VERAIN André	Physique
	VEYRET Paul	Géographie
	VIGNAIS Pierre	Biochimie médicale

PROFESSEURS ASSOCIES

MM.	CRABBE Pierre	CERMO
	DEMBICKI Eugéniuz	Mécanique
	JOHNSON Thomas	Mathématiques appliquées
	PENNEY Thomas	Physique

PROFESSEURS SANS CHAIRE

Melle	AGNIUS-DELORD Claudine	Physique pharmaceutique
	ALARY Josette	Chimie analytique
MM.	AMBROISE-THOMAS Pierre	Parasitologie
	ARMAND Gilbert	Géographie
	BENZAKEN Claude	Mathématiques appliquées
	BIAREZ Jean-Pierre	Mécanique
	BILLET Jean	Géographie
	BOUCHET Yves	Anatomie
	BRUGEL Lucien	Energétique (IUT I)
	BUISSON René	Physique (IUT I)
	BUTEL Jean	Orthopédie
	COHEN ADDAD Pierre	Spectrométrie physique
	COLOMB Maurice	Biochimie
	CONTE René	Physique (IUT I)
	DELOBEL Claude	M.I.A.G.
	DEPASSEL Roger	Mécanique des fluides
	FONTAINE Jean-Marc	Mathématiques pures
	GAUTRON René	Chimie
	GIDON Paul	Géologie et minéralogie
	GLENAT René	Chimie organique
	GROULADE Joseph	Biologie médicale
	HACQUES Gérard	Calcul numérique
	HOLLARD Daniel	Hématologie
	HUGONOT Robert	Hygiène et médecine préventive
	IDELMAN Simon	Physiologie animale
	JOLY Jean-René	Mathématiques pures
	JULLIEN Pierre	Mathématiques appliquées
Mme.	KAHANE Josette	Physique
MM.	KRAKOWIACK Sacha	Mathématiques appliquées
	KUHN Gérard	Physique (IUT I)
	LUU DUC Cuong	Chimie organique
	MAYNARD Roger	Physique du solide
Mme.	MINIER Colette	Physique (IUT I)
MM.	PELMONT Jean	Biochimie
	PERRIAUX Jean-Jacques	Géologie et minéralogie
	PFISTER Jean-Claude	Physique du solide
Melle	PIERY Yvette	Physiologie animale

MM.	RAYNAUD Hervé	M.I.A.G.
	REBECQ Jacques	Biologie (CUS)
	REYMOND Jean-Charles	Chirurgie générale
	RICHARD Lucien	Biologie végétale
Mme.	RINAUDO Marguerite	Chimie macromoléculaire
MM.	ROBERT André	Chimie papetière
	SARRAZIN Roger	Anatomie et chirurgie
	SARROT-REYNAULD Jean	Géologie
	SIROT Louis	Chirurgie générale
Mme.	SOUTIF Jeanne	Physique générale
MM.	STIEGLITZ Paul	Anesthésiologie
	VIALON Pierre	Géologie
	VAN CUTSEM Bernard	Mathématiques appliquées

MAITRES DE CONFERENCES ET MAITRES DE CONFERENCES AGREGES

MM.	ARMAND Yves	Chimie (IUT I)
	BACHELOT Yvan	Endocrinologie
	BARGE Michel	Neuro-chirurgie
	BEGUIN Claude	Chimie organique
Mme	BERIEL Hélène	Pharmacodynamie
MM.	BOST Michel	Pédiatrie
	BOUCHARLAT Jacques	Psychiatrie adultes
Mme.	BOUCHE Liane	Mathématiques (CUS)
MM.	BRODEAU François	Mathématiques (IUT B) (Personne étrangère habilitée à être directeur de thèse)
	CHAMBAZ Edmond	Biochimie médicale
	CHAMPETIER Jean	Anatomie et organogénèse
	CHARDON Michel	Géographie
	CHERADAME Hervé	Chimie papetière
	CHIAVERINA Jean	Biologie appliquée (EFP)
	CONTAMIN Charles	Chirurgie thoracique et cardio-vasculaire
	CORDONNIER Daniel	Néphrologie
	COULOMB Max	Radiologie
	CROUZET Guy	Radiologie
	CYROT Michel	Physique du solide
	DENIS Bernard	Cardiologie
	DOUCE Roland	Physiologie végétale
	DUSSAUD René	Mathématiques (CUS)
Mme.	ETERRADOSSI Jacqueline	Physiologie
MM.	FAURE Jacques	Médecine légale
	FAURE Gilbert	Urologie
	GAUTIER Robert	Chirurgie générale
	GIDON Maurice	Géologie
	GROS Yves	Physique (IUT I)
	GUIGNIER Michel	Thérapeutique
	GUITTON Jacques	Chimie
	HICTER Pierre	Chimie
	JALBERT Pierre	Histologie
	JULIEN-LAVILLAVROY Claude	O.R.L.
	KOLODIE Lucien	Hématologie
	LE NOC Pierre	Bactériologie-virologie
	MACHE Régis	Physiologie végétale
	MAGNIN Robert	Hygiène et médecine préventive
	MAILLION Jean-Michel	Médecine du travail
	MARECHAL Jean	Mécanique (IUT I)
	MARTIN-BOUYER Michel	Chimie (CUS)
	MICHOULIER Jean	Physique (IUT I)

MM.	NEGRE Robert	Mécanique (IUT I)
	NEMOZ Alain	Thermodynamique
	NOUGARET Marcel	Automatique (IUT I)
	PARAMELLE Bernard	Pneumologie
	PECCOUD François	Analyse (IUT B) (Personnalité étrangère habilité à être directeur de thèse)
	PEFFEN René	Métallurgie (IUT I)
	PERRIER Guy	Géophysique-Glaciologie
	PHELIP Xavier	Rhumatologie
	RACHAIL Michel	Médecine interne
	RACINET Claude	Gynécologie et obstétrique
	RAMBAUD André	Hygiène et hydrologie (Pharmacie)
	RAMBAUD Pierre	Pédiatrie
	RAPHAEL Bernard	Stomatologie
Mme.	RENAUDET Jacqueline	Bactériologie (Pharmacie)
MM.	ROBERT Jean-Bernard	Chimie physique
	Romier Guy	Mathématiques (IUT B) (Personnalité étrangère habilité à être directeur de thèse)
	SCHAERER René	Cancérologie
	SHOM Jean-Claude	Chimie générale
	STOEBNER Pierre	Anatomie pathologie
	VROUSOS Constantin	Radiologie

MAITRES DE CONFERENCES ASSOCIES

MM.	DEVINE Roderick	Spectro physique
	HODGES Christopher	Transition de phases

Fait à SAINT MARTIN D'HERES, NOVEMBRE 1976.

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Monsieur Philippe TRAYNARD : Président

Monsieur Pierre-Jean LAURENT : Vice Président

PROFESSEURS TITULAIRES

MM.	BENOIT Jean	Radioélectricité
	BESSON Jean	Electrochimie
	BLOCH Daniel	Physique du solide
	BONNETAIN Lucien	Chimie minérale
	BONNIER Etienne	Electrochimie et électrométallurgie
	BOUDOURIS Georges	Radioélectricité
	BRISSONNEAU Pierre	Physique du solide
	BUYLE-BODIN Maurice	Electronique
	COUMES André	Radioélectricité
	DURAND Francis	Métallurgie
	FELICI Noël	Electrostatique
	FOULARD Claude	Automatique
	LESPINARD Georges	Mécanique
	MOREAU René	Mécanique
	PARIAUD Jean-Charles	Chimie-Physique
	PAUTHENET René	Physique du solide
	PERRET René	Servomécanismes
	POLOUJADOFF Michel	Electrotechnique
	SILBER Robert	Mécanique des fluides

PROFESSEUR ASSOCIE

M.	ROUXEL Roland	Automatique

PROFESSEURS SANS CHAIRE

MM.	BLIMAN Samuel	Electronique
	BOUVARD Maurice	Génie mécanique
	COHEN Joseph	Electrotechnique
	LACOUME Jean-Louis	Géophysique
	LANCIA Roland	Electronique
	ROBERT François	Analyse Numérique
	VEILLON Gérard	Informatique fondamentale et appliquée
	ZADWORNY François	Electronique

MAITRES DE CONFERENCES

MM.	ANCEAU François	Mathématiques appliquées
	CHARTIER Germain	Electronique
	GUYOT Pierre	Chimie minérale
	IVANES Marcel	Electrotechnique
	JOUBERT Jean-Claude	Physique du solide
	MORET Roger	Electrotechnique nucléaire
	PIERRARD Jean-Marie	Mécanique
	SABONNADIERE Jean-Claude	Informatique fondamentale et appliquée
Mme.	SAUCIER Gabrièle	Informatique fondamentale et appliquée

MAITRE DE CONFERENCES ASSOCIE

M.	LANDAU Ioan	Automatique

CHERCHEURS DU C.N.R.S. (Directeur et Maîtres de Recherche)

MM.	FRUCHART Robert	Directeur de Recherche
	ANSARA Ibrahim	Maître de Recherche
	CARRE René	Maître de Recherche
	DRIOLE Jean	Maître de Recherche
	MATHIEU Jean-Claude	Maître de Recherche
	MUNIER Jacques	Maître de Recherche

à ma mère, à mon père

părinților mei

J'exprime ma profonde gratitude à Monsieur le Professeur Louis BOLLIET qui m'a fait l'honneur de présider le jury de cette thèse et qui a manifesté beaucoup d'attention à mon égard tout au long de mon activité à l'ENSIMAG ;

ma reconnaissance s'adresse à Claude BOKSENBAUM, Maître de Conférence à l'IUT de Montpellier, qui, au cours de ce travail, ne m'a ménagé ni son temps ni son soutien critique ;

je tiens à remercier Jean Pierre VERJUS, Professeur à l'Université de Rennes, pour l'intérêt qu'il a porté à mon travail, Michel SINTZOFF, Ingénieur à MBLE-Bruxelles, avec qui j'ai eu de nombreuses discussions lors de mon séjour à MBLE qui ont permis le raffinement de certaines idées, ainsi que Sacha KRAKOWIAK, Professeur à l'Université de Grenoble et Claude GIRAULT, Professeur à l'Institut de Programmation de Paris, qui ont accepté d'en être juges ;

je suis infiniment reconnaissant à Clotilde CHALAND pour la patience, la compétence et l'esprit d'initiative avec lesquels elle a bien voulu en assurer les frappes ;

que tous ceux qui m'ont aidé à poursuivre ce travail par leurs critiques, leurs conseils et leur amitié, trouvent ici l'expression de mes remerciements ;

enfin, je tiens à remercier les membres du Service de Tirage qui ont assuré la réalisation matérielle dans un temps record.

S O M M A I R E

INTRODUCTION ET OBJECTIFS

"though this be madness,
yet there is method in't"

Shakespeare, 'Hamlet'

Le travail de recherche exposé ici est né du désir de formalisation d'un problème complexe dans les systèmes informatiques modernes : la synchronisation des processus concurrents.

Pour ce problème, apparu comme conséquence des activités parallèles et des soucis d'efficacité, un formalisme de spécification ne peut que rendre les solutions moins "artisanales". C'est un objectif que nous poursuivons dans la suite.

Nous insisterons au chapitre suivant sur les notions de processus et de synchronisation, mais notons néanmoins dès maintenant que la plupart des dictionnaires classiques définissent un processus comme l'avance d'une série d'actions ou d'opérations conduisant à une fin ; de même, le verbe "synchroniser" signifie : se passer au même instant ou opérant dans la même période de temps (voir par exemple Larousse ou Oxford Dictionary).

Ainsi, une *définition informelle* de la synchronisation des processus, pourrait-elle être l'exécution correcte dans le temps (c'est-à-dire qui mène à un résultat) d'un ensemble de séries d'actions.

Les premiers systèmes informatiques ne permettaient pas de simultanéité entre les calculs effectués par l'unité centrale et d'autres opérations comme les entrées-sorties, par exemple. Généralement, un seul programme se déroulait sur l'unité centrale et les calculs étaient interrompus au moment où les entrées-sorties s'exécutaient.

Une analyse rapide des programmes s'exécutant dans un système montre que si un programme nécessite un certain ensemble de ressources (des périphériques en général), d'autres programmes utiliseront au même instant des ensembles différents.

La sous-utilisation des ressources (le terme de ressource apparaît ici au sens le plus large) en l'absence de simultanéité, a été résolue dans les systèmes au prix d'outils mis en oeuvre pour la synchronisation des activités parallèles. En effet, un système ayant des ressources très variées pour satisfaire à tous les utilisateurs, on essaie de faire travailler plusieurs processus parallèles sur des ensembles disjoints de ressources. Mais pour les ressources communes ou partagées par plusieurs processus, des mécanismes de synchronisation sont nécessaires.

La tendance actuelle des systèmes est de permettre un haut degré de parallélisme et par conséquent l'architecture sera orientée fortement pour faciliter les activités parallèles (systèmes multiprocesseurs, réseaux d'ordinateurs).

Pour le partage des ressources dans un système, les mécanismes mis en oeuvre doivent permettre l'exécution de deux ou plusieurs programmes dont les besoins ne dépassent pas la totalité des ressources disponibles dans les systèmes. De tels mécanismes sont donc constamment soumis à un ensemble de contraintes.

Pour des raisons de performances, il est apparu profitable de pouvoir interrompre et découper l'exécution de plusieurs programmes en unités de temps. Pour ce faire, il a été nécessaire d'introduire le partage de l'unité centrale entre plusieurs programmes (l'unité centrale apparaît donc comme une ressource particulière utilisée par tranches de temps). Cette possibilité de multiplexage dans le temps de l'unité centrale introduit une simultanéité "apparente" des programmes. Un autre argument qui est venu justifier le multiplexage d'un processeur, a été l'utilisation interactive des systèmes.

Notre travail est structuré de la façon suivante :

Le chapitre 1 contient une brève analyse du concept de processus et une présentation des outils de synchronisation existants ; cette présentation est surtout critique : elle fait apparaître d'une part l'évolution des outils, du plus simple (sémaphore) vers des structures plus évoluées (moniteurs, expressions de chemins, ..) et d'autre part la nécessité d'une telle évolution.

Le chapitre 2 présente le formalisme de synchronisation que nous introduisons, formalisme que nous voulons à la fois généralisateur, pour couvrir l'ensemble des outils de synchronisation, et simplificateur pour permettre de réduire au maximum l'artisanat et l'intuition dans la recherche des solutions aux problèmes de synchronisation.

C'est ce que nous nous efforcerons de montrer au chapitre 3 dans un cadre général.

Enfin, le chapitre 4 traite de l'apport de notre modèle à l'étude du deadlock.

Le lecteur intéressé trouvera également une bibliographie assez complète sur le contrôle du parallélisme et plus particulièrement sur la synchronisation des processus concurrents.

CHAPITRE I

PROCESSUS ET SYNCHRONISATION DE PROCESSUS

"quelques définitions
pour nous dérouiller l'entendement"

Boris Vian, 'En avant la zizique'

1. DÉFINITIONS ET GÉNÉRALITÉS

Les programmes se partageant les ressources et en particulier l'unité centrale, sont en général des parties indépendantes. L'exécution d'une telle partie sur un processeur est classiquement appelée *processus séquentiel*. Il faut remarquer tout de suite la nature "dynamique" de cette notion par opposition à la nature "statique" de la notion de programme ou de sous-programme. On peut donc considérer un système comme un ensemble de processus séquentiels coopérants et concurrents, se déroulant en parallèle.

Dijkstra (Di 68-a) a suggéré, il y a (déjà !) plus d'une dizaine d'années, de modeler des activités concurrentes sur un (ou plusieurs) processeur(s) par un ensemble de processus qui ont dans leur ensemble des périodes d'activité indépendantes et des périodes de communication entre eux.

Analysons plus en détail les causes des interactions ayant lieu dans un tel ensemble de processus.

Le nombre limité de ressources disponibles dans un système conduit inévitablement à une coopération et une compétition entre les processus qui se manifeste par un besoin de communication. Les programmes utilisateurs ne sont pas forcément conscients de cette coopération-compétition. Ce sont les fonctions du système d'exploitation qu'ils appellent, qui la gèrent. Cette interaction entre les processus a été modélisée par des *variables de communication*. Il est évident que des problèmes de cohérence apparaissent si deux processus qui coopèrent au moyen d'une variable de communication ont accès à cette variable en "même" temps. Cette exclusivité d'accès à une même variable de communication a engendré le problème de *l'exclusion mutuelle* entre processus concurrents.

Le mécanisme de coordination nécessaire dans un ensemble de processus concurrents pour assurer leur coopération, s'appelle *synchronisation des processus*. Notons dès maintenant que la synchronisation des processus apparaît comme le résultat des contraintes que les processus doivent respecter dans leur ensemble. Remarquons aussi qu'une communication correcte entre les processus n'est pas suffisante pour garantir leur coordination. L'exemple du système formé de deux processus occupant chacun une des deux ressources de ce système et demandant l'autre pour sa progression, est un problème "classique" de *deadlock* (ou *impasse*). La *famine* et la minimisation de l'attente active dans un ensemble de processus, constituent d'autres exemples.

Le modèle de processus comme programme en exécution communicant à travers des variables partagées avec d'autres processus, a servi à la définition d'outils de synchronisation pour les langages de programmation tels que sémaphores (Di 68-b), régions critiques conditionnelles (BH 72), moniteurs (Hoa 74). L'objectif de ces outils était d'aider le programmeur à utiliser les variables de communication correctement. Nous reparlerons plus en détail au paragraphe suivant de ces outils et d'autres structures de synchronisation, mais remarquons tout de suite qu'ils présentent des inconvénients importants.

i) Tout d'abord, l'expression de la synchronisation à l'aide des primitives usuelles n'est pas naturelle. En effet, une certaine expérience est nécessaire à la compréhension des solutions aux problèmes de synchronisation résolus à l'aide de ces primitives. Leurs actions sont "saupoudrées" dans les algorithmes et souvent très loin de l'énoncé initial du problème. Ce sont donc des outils rarement adaptés au programmeur d'application et un usage très "personnalisé" en est fait de la part du programmeur système.

Pour illustrer cet argument, prenons l'exemple des deux problèmes des "lecteurs - écrivains" résolus par Courtois, Heymans et Parnas (CHP 71). Les solutions ont été données en termes des primitives P et V introduites par Dijkstra (Di 68-b). Rappelons brièvement qu'il s'agit de deux classes de processus, l'une formée de "lecteurs" et l'autre "d'écrivains" consommant une ressource unique. Les processus "écrivains" doivent avoir un accès exclusif à la ressource, tandis que les "lecteurs" peuvent se partager la ressource entre eux. Poser le problème est chose simple et compréhensible ; pour le résoudre, je cite les auteurs : "... the solution presented is quite simple, but our experience has shown that it is not easily arrived at. Numerous solutions, which have quite unreasonable complexity, have been proposed" !

ii) De plus, il y a beaucoup d'encre et encore plus de réflexion dépensées pour prouver la correction de la solution ! Cela implique donc un autre inconvénient qui est celui de prouver la correction. Je ne veux pas dire ici que la correction des solutions soit douteuse dans tous les cas, mais seulement qu'elle est difficilement prouvable et souvent très coûteuse. On retient surtout des modèles de certains énoncé et de certaines solutions pour les cas courants (producteur-consommateur, lecteurs-écrivains,.

iii) Finalement, le critère de généralité est violé car ces outils de synchronisation sont souvent insuffisants ou inadaptés pour la résolution de certains problèmes de synchronisation. Nombreux sont les exemples de problèmes de synchronisation pour lesquels des adaptations aux outils de synchronisation ont été faites, ou pour lesquels d'autres outils ont été inventés De plus, il existe aussi les exemples de problèmes très difficiles et non résolus avec les outils existants. On peut surtout citer l'exemple des systèmes répartis qui posent actuellement de sérieuses difficultés (outils mal adaptés aux processus concurrents se déroulant sur les différents noeuds du réseau de commutation). Plusieurs problèmes sont à signaler : les actions lancées à travers un réseau le sont de façon asynchrone ; l'efficatité et l'architecture des systèmes répartis imposent de profiter du parallélisme ; en cas de panne, il faut pouvoir survivre ; pour le contrôle des applications on doit pouvoir discerner entre répartition ou quasi-centralisme. Des modèles nouveaux pour les programmes parallèles répartis apparaissent (Hoa 77, BH 77b) et des méthodes nouvelles seront inévitablement nécessaires dans ce contexte.

Ces arguments nous démontrent la nécessité d'un formalisme pour la synchronisation des processus qui soit assez général pour répondre à ces critères.

Il devra tout d'abord permettre d'aller plus vite et plus sûrement vers la solution. Cela signifie donc une certaine possibilité "d'automatisation" dans la recherche de la solution impliquant une élaboration plus systématique.

Une conséquence de cette automatisation sera de faciliter l'emploi des primitives de synchronisation choisies dans l'implémentation des solutions et de rendre les solutions plus compréhensibles.

La généralité du formalisme devrait permettre une classification des problèmes de synchronisation et surtout la détection du deadlock. On facilitera ainsi la preuve des solutions et dans certains cas elle devra même être superflue.

Nous développerons donc un formalisme pour *spécifier* la synchronisation des processus. La spécification constitue l'interface entre les outils de synchronisation qui l'implémentent et les modules de programmes qui l'utiliseront. Les méthodes de vérification de programmes (Fl 67a, Hoa 72b, O 77) tentent à prouver que l'implémentation satisfait les spécifications faites, ou à employer les spécifications pour vérifier les modules qui utilisent l'implémentation. C'est dans ce sens que nous affirmons que dans certains cas les preuves de non-deadlock seront superflues, car si les spécifications éliminent le deadlock et si le formalisme de spécification indique aussi un moyen systématique pour passer à une implémentation, alors aucune preuve n'est nécessaire (cf. chapitre 4) Remarquons que les méthodes de construction méthodique des programmes (Si 78) sont orientées dans le même sens.

Notre formalisme de synchronisation est naturel, dans ce sens qu'il est basé sur les contraintes introduites dans l'ensemble des processus par leur coopération et donc est défini comme un mécanisme capable de leur faire respecter ces contraintes dans leur avance et à chaque instant.

Avant de passer à la définition et à l'analyse du formalisme, nous allons énumérer les principaux mécanismes de synchronisation des processus concurrents définis ces dernières années.

2. OUTILS ET STRUCTURES DE SYNCHRONISATION "CLASSIQUES"

2.1. Les sémaphores

Dijkstra (Di 68-b) a introduit dans son article maintenant classique, un outil formel pour résoudre le problème d'exclusion mutuelle en définissant deux primitives P et V (du néerlandais Passeren et Vrijgeven) et un objet système de type particulier, *le sémaphore*.

Le sémaphore étant dans un espace accessible aux processus que l'on veut synchroniser, les deux primitives sont définies par :

```
P(s) ;                          V(s) ;
begin s := s - 1 ;              begin s := s + 1 ;
   if s < 0 then                   if s ≤ 0 then
   [mettre le processus            [réveiller un processus
    exécutant en attente]           en attente]
end                             end
```

Ces deux opérations sont considérées par le système comme *indivisibles*, en ce sens que si plusieurs processus tentent l'exécution simultanée d'une des deux (ou des deux) opérations sur le même sémaphore, les opérations ne sont exécutées que l'une après l'autre. En d'autres termes, l'accès au même sémaphore pour les opérations se fait en exclusion mutuelle.

La base du mécanisme d'exclusion mutuelle est, au niveau matériel, une instruction (TEST & SET par exemple) permettant l'implémentation des sémaphores, outils plus formels, pour résoudre les problèmes de synchronisation au niveau système. Il est important de souligner qu'à la base des mécanismes de synchronisation, il existe une *exclusion mutuelle minimale*. C'est en raffinant cette minimalité (Lam 77) que l'on peut obtenir des outils différents et souvent plus efficaces (augmentation du parallélisme, par exemple).

On peut réaliser un objet de type sémaphore à l'aide d'un entier S (dont on fixe la valeur initiale) et d'une file d'attente, initialement vide.

Pour les propriétés spécifiques des sémaphores, nous renvoyons le lecteur intéressé à la littérature maintenant très abondante. Nous allons cependant faire quelques remarques.

La définition de la primitive V n'indique pas comment est fait le choix du processus à réveiller, ce choix dépendant de la politique de gestion des files d'attente choisie dans une implémentation particulière. Cela signifie que l'outil de synchronisation est défini indépendamment de cette politique en assurant seulement qu'une requête quelconque sera prise en compte au bout d'un temps fini par le système. La séparation entre la politique de gestion et le réveil effectif a l'avantage de ne pas figer une fois pour toutes une politique donnée, mais il faut également souligner un inconvénient majeur : même pour une politique très simple (FIFO, par exemple), sa mise en oeuvre dans le système nécessite un travail considérable.

Les primitives de synchronisation favorisent la compétition entre les processus pour le partage des ressources. En effet, un processus réveillé par un V passe de l'état bloqué à l'état activable, mais en même temps le processus ayant exécuté la primitive V est à l'état activable ; les deux processus seront compétitifs pour les ressources. Dans l'état activable, un processus a franchi la fin de la primitive P qui l'avait bloqué.

Les primitives P et V sont d'ailleurs *compactes*, en ce sens qu'aucun processus ne pourra faire une action propre à l'intérieur de la primitive entre la modification de la valeur de S et le test qui la suit.

Pour les besoins de l'implémentation, on fait la distinction entre les sémaphores booléens (l'entier S ne prend que les valeurs 0 et 1) des autres (les premiers suffisent à implémenter les autres). C'est la base du mécanisme de synchronisation par *événements*. La coordination entre les processus est ainsi basée sur le fait que l'on considère les processus comme une succession d'événements synchrones avec les deux contraintes suivantes :

- un certain événement peut avoir lieu si *tous* les événements d'un ensemble donné ont eu lieu ;
- un certain événement peut avoir lieu si *un au moins* des événements d'un ensemble donné a eu lieu.

Ces deux contraintes génèrent les conditions booléennes linéaires apparaissant dans certaines extensions des primitives "wait" et "signa définies par Habermann (Hab 69).

Remarquons ici qu'un événement a un caractère "fugitif" dans ce sens qu'il est oublié dès qu'il est arrivé (voir la primitive "signal"). L'outil de synchronisation basé sur les événements n'a lui même aucune information relative au nombre de processus en attente sur un événement donné. Dans le cas des sémaphores non booléens, cette information est contenue dans la valeur du sémaphore.

Il n'est pas dans notre intention de détailler plus ici les travaux faits sur la synchronisation par événements et par sémaphores. Nous avons seulement rappelé certaines caractéristiques nous permettant de voir plus loin comment le formalisme de synchronisation que nous définissons couvrira ces cas particuliers. Quelques remarques sont toutefois nécessaires.

Les sémaphores (et les événements) sont des mécanismes de synchronisation "indirects" dans le sens suivant : un ensemble de processus concurrents ont accès à un ensemble d'objets du système destinés à les synchroniser, mais l'identité de ces processus n'intervient pas explicitement comme paramètre des primitives employées. Ainsi un processus particulier n'a-t-il pas la possibilité d'en bloquer un autre (Lam 76). Ces mécanismes sont donc utiles lorsqu'un processus ignore, en raison de la nature du problème, l'identité des processus avec lesquels il coopère ; les interactions dans l'ensemble des processus doivent avoir été prévues à l'écriture des programmes.

D'un autre côté, comme nous l'avons signalé au paragraphe précédent, des problème dont l'énoncé est relativement simple, ont avec les sémaphores des solutions assez complexes.

Toutes ces remarques suggèrent l'idée que les sémaphores et les primitives P et V telles qu'elles ont été définies par Dijkstra, ne sont pas adéquats pour la résolution de certains problèmes faisant intervenir des classes de processus ou des problèmes tels que "le dîner des cinq philosophes" posé aussi par Dijkstra (Di 71). Ainsi, plusieure extensions ont-elles été faites.

Remarquons que lors de la synchronisation par événements, un processus peut attendre plusieurs événements avant de continuer, alors qu'il ne peut agir que sur un seul sémaphore à la fois avec la primitive P. De même, alors qu'un événement devrait libérer tous les processus qui l'attendent, la primitive V n'en réveille qu'un seul à la fois.

Pour un sémaphore donné, deux nouvelles primitives ont été définies (VvL 72) : P(n,s) et V(n,s), qui sont aussi des opérations indivisibles et pour lesquelles n est un entier non négatif :

```
P(n,s) ;                          V(n,s) ;
if n ≤ s then s := s - n else     begin s := s + n ;
begin [mettre le processus           while [il existe un processus B en
       exécutant A en attente              attente et rang (B) ≤ s]
       et rang (A) := n]             do [réveiller un processus B tel que
end                                      rang (B) ≤ s et s := s - rang (B)]
                                  end
```

Ces deux primitives enlèvent la limitation décrite ci-dessus à l'aide de l'entier non négatif rang(B) associé à chaque processus B bloqué

Une réalisation possible de ces primitives consiste à maintenir une file d'attente de processus par sémaphore. Lors d'une opération V(n,s), soit m la valeur atteinte par le sémaphore. Si $\mathcal{P}$ est l'ensemble des processus en attente, on pourrait libérer un sous-ensemble $\mathcal{P}'$ de processus tel que :

$$\sum_{p \in \mathcal{P}'} n_p \leq m$$

$$\min_{p \in \mathcal{P}-\mathcal{P}'} n_p > m - \sum_{p \in \mathcal{P}'} n_p$$

où n_p est la variation du sémaphore induite par le $p^{\text{ième}}$ processus exécutant une primitive.

Cette réalisation peut aussi s'interpréter de la manière suivante : lors de la mise en attente on ne fait pas attendre un processus qui pourrait être réveillé comme faisant partie de l'ensemble $\mathcal{P}'$ défini ci-dessu

C'est la condition la plus faible sur la libération des processus en attente : elle permet un grand choix de stratégies de gestion de la file d'attente (réveil par ordre d'ancienneté dans la file, réveil en fonction des demandes les plus "importantes" en quantité de ressources par exemple, réveil en fonction des demandes les moins importantes). Evidemment, le choix de la stratégie est fait de manière à minimiser le coût du réveil. Dans ce cas, il restera toujours inférieur au coût d'au plus un balayage de l'ensemble des processus en attente. Si, par exemple, on ordonne les processus en attente dans l'ordre des demandes décroissantes, le coût des réveils sera moindre, mais la mise en file sera plus coûteuse.

Une autre généralisation est de définir des opérations indivisibles sur un nombre fini de sémaphores (Br 74) :

$P(s_1,n_1,s_2,n_2,\ldots,s_k,n_k)$;
[si $\forall i = 1,2,\ldots,K$,
$s_i \geq n_i$ alors $s_i := s_i - n_i$
sinon attente]

$V(s_1,n_1,s_2,n_2,\ldots,s_k,n_k)$;
[$\forall i = 1,2,\ldots,K$ faire $s_i := s_i + n_i$]

Une réalisation possible de ces primitives consiste à maintenir une *seule* file d'attente pour tous les processus en attente. Lors d'une opération V, soit m_i la valeur du sémaphore s_i. Si $\mathcal{P}$ est l'ensemble des processus en attente et $n_{i,p}$ la demande du processus p sur le sémaphore s_i, il faudra libérer un sous-ensemble P' de processus tel que :

$$\sum_{p \in \mathcal{P}'} n_{i,p} \leq m_i, \quad \forall i \in \{1,2,\ldots,K\}$$

$$p \in \mathcal{P}-\mathcal{P}', \ \exists\, i \in \{1,2,\ldots,K\} : n_{i,p} > m_i - \sum_{p' \in \mathcal{P}'} n_{i,p'}$$

Comme précédemment, les conditions ci-dessus laissent une grande latitude dans le choix des stratégies de réveil. Le coût du réveil est de l'ordre du coût d'un balayage complet de la file d'attente, avec en plus une itération à chaque pas sur chacun des sémaphores.

Il est important de remarquer l'existence de réveils simultanés possibles et nous verrons plus loin que cette solution est à rapprocher du mécanisme général de synchronisation. Le réveil simultané est ici conditionné par l'indépendance des actions des processus par rapport aux sémaphores concernés. Nous analyserons cet aspect dans un contexte plus général au chapitre 4, où nous montrerons comment l'outil décrit ci-dessus est un cas particulier d'un mécanisme plus général qui se déduira du formalisme introduit au chapitre 2.

Belpaire et Wilmotte (BW 73), en partant de la remarque qu'en général les conditions d'attente sont séparées des effets de la synchronisation, se proposent tout d'abord de définir des opérations plus élémentaires appartenant à un niveau inférieur d'abstraction que les primitives P et V, et ensuite de donner un moyen de composition de ces opérations élémentaires. Ce dernier point les amène à définir une généralisation des primitives de Dijkstra. Ils définissent d'abord les trois primitives suivantes opérant sur des sémaphores :

a) passage d'un sémaphore (s:) ; l'exécution de l'opération (équivalente à l'occurrence d'un événement) est possible si et seulement si $s \geq 0$. Dans ce cas, l'effet de la primitive est nul pour le processus exécutant. Si $s < 0$, l'exécution est bloquée. Remarquons ici que le passage d'un sémaphore apparaît uniquement comme une contrainte (une barrière) dans l'avance des processus.

b) retardement d'un sémaphore (down s) : l'exécution de cette primitive est toujours possible. L'effet en est la décrémentation de s de une unité.

c) avance d'un sépaphore (up s) : l'exécution de cette primitive est également toujours possible et elle a comme effet l'incrémentation de s de une unité.

Ces trois primitives élémentaires peuvent ensuite être composées pour former une autre opération indivisible plus générale. La notation est :

$$s_1 : s_2 : \ldots : s_n \text{ down } s'_1, \ldots, s'_m \text{ up } s''_1, \ldots, s''_k$$

avec $n, m, k \geq 0$.

L'exécution de cette opération est possible si et seulement si la contrainte suivante est respectée par les actions du processus :

$$s_1 + s_2 + \ldots + s_n \geq 0$$

et dans ce cas, son effet est obtenu par la conjonction des effets des primitives élémentaires qui la composent. Nous remarquons que la contrainte ci-dessus n'assure pas le passage de tous les sémaphores s_i, $i = 1,2,\ldots,n$ car $\sum_{i=1}^{n} s_i \geq 0$ n'implique pas que pour tout $i \in \{1,2,\ldots,n\}$ on ait $s_i \geq 0$.

Les primitives P et V de Dijkstra en sont un cas particulier car P(s) est équivalent à s : down s
et V(s) est équivalent à up s.

Pour d'autres détails relatifs à ces primitives et une "hiérarchie" des langages de synchronisation, nous invitons le lecteur à se référer à (BW 73).

De même, pour d'autres extensions des primitives P et V motivées par des exemples bien choisis, nous suggérons l'article de Presser (Notons toutefois que la voie de la généralisation des primitives n'est pas très fructueuse.

Nos objections sont les suivantes :

Premièrement, les extensions proposées tendent à élargir, ou plutôt à compliquer, la structure des unités ininterruptibles, ce qui naturellement diminue l'efficacité du système dans lequel l'implémentation est faite, car le temps passé en section critique est augmenté.

Les justifications de ces généralisations sont évidentes : il s'agit d'abord de rendre possible la résolution de certains problèmes de synchronisation (et encore, la totalité n'est pas assurée !) et ensuite de structurer les solutions. Cette structuration apparaît essentiellement dans une construction "standardisée" des programmes, mais rien n'est fait au niveau système.

Cela nous conduit à une seconde objection : puisque ces primitives feront partie des processus du système à un certain moment de leur existence, il en résulte des erreurs dues à leur emploi, exactement comme dans le cas des instructions. Ces erreurs d'utilisation (la "classique" inversion des P et V en est un exemple), rendent catastrophiques les interactions dans l'ensemble des processus (Br 74). C'est surtout le cas de l'emploi des primitives dans les processus utilisateurs, car pour les processus système, ces erreurs peuvent facilement être détectées.

Une troisième objection est donc l'impossibilité de détecter ces erreurs à un niveau tel que la compilation. Or, l'intérêt est de tenter la réalisation d'outils permettant une certaine "automatisation" du contrôle de l'interaction entre différentes unités des programmes. Cela suppose une grande modularité des systèmes.

Avant de passer à l'analyse d'autres outils pour la synchronisation des processus, donnons une autre définition des primitives P et V de Dijkstra (Ko 73) qui nous servira par la suite pour dégager une définition de notre formalisme de synchronisation :

P(s) : when s > 0 do s := s - 1 ;
V(s) : s := s + 1 ;

Dans cette définition, l'attente lors du P est suggérée par le test effectué avant le changement de la valeur du sémaphore (la signification de when au lieu de if est ici essentielle). La définition ci-dessus est à rapprocher de la définition donnée initialement par Dijkstra.

En effet, les processus bloqués lors d'un P dans la définition de Dijkstra, sont tous ceux qui attendent l'exécution du P dans la seconde définition, parce que la contrainte s > 0 est violée. La seule différence subsiste dans le moment où les processus sont mis en attente, à l'extérieur de la primitive l'effet étant le même. De même, un processus réveillé dans la première définition par un V, correspond à un processus et un seul autorisé à exécuter un P dans la seconde définition.

C'est donc la contrainte s ≥ 0 qui est voulue de façon *permanente* et de ce fait s > 0 est une pré-condition déduite de s ≥ 0 et associée à l'action s := s - 1.

A partir de cette simple remarque, le formalisme général présenté au chapitre 2 sera parfaitement compréhensible.

2.2. Régions critiques

Le concept de régions critiques a été introduit par Hoare et Brinch Hansen (Hoa 72a, BH 72) pour pouvoir exprimer l'exclusion mutuelle dans une construction unique de la forme :

region v do I_1 ; I_2 ; ... ; I_n od

où v est la ressource partagée et I_1 ; I_2 ; ... ; I_n constituent le corps de la région critique qui est exécuté comme une primitive.

Cet outil pour la synchronisation des processus accorde au compilateur la possibilité de contrôler à son niveau l'accès en exclusion mutuelle aux variables partagées.

Une extension naturelle des régions critiques a été l'introduction de l'attente pour un processus demandant l'entrée dans une région critique. Cette extension a été à l'origine proposée par Hoare (Hoa 72-a) et développée ensuite par Brinch Hansen (BH 73-a).
L'écriture devient ainsi :

région v when B do I_1 ; I_2 ; ... ; I_n od

où B est une condition exprimée sous la forme d'une expression booléenne qui doit être satisfaite avant l'entrée du processus exécutant dans la région critique. Quand un processus entre dans la *région critique conditionnelle*, la condition B est évaluée. Si le résultat de l'évaluation est vrai, l'entrée dans la région critique est autorisée au processus qui exécute les instructions I_1 ; I_2 ; ... ; I_n ayant potentiellement accès à la variable V. Si la condition est fausse, le processus est mis en attente dans une file associée à la variable v. L'attente est donc associée à la variable v et non à la condition. Si un processus termine une région critique sur la variable v, les conditions de synchronisation sont réévaluées. Si au moins une de ces conditions est satisfaite, un des processus en attente est autorisé à réentrer sa région critique. La politique d'attente n'est donc pas spécifiée comme dans le cas des primitives P et V. La politique d'attente doit assurer le réveil de tout processus en attente au bout d'un temps fini.

La simplicité de conception et la possibilité d'attente pour des conditions qui peuvent être très complexes, rendent cet outil facilement utilisable. Néanmoins, deux critiques importantes doivent être faites :

a) Il est nécessaire de réévaluer toutes les conditions pour lesquelles des processus sont en attente, chaque fois qu'un processus exécute une région critique. Ainsi, les régions critiques conditionnelles deviennent-elles inefficaces chaque fois que plusieurs processus sont en attente de ressources en même temps.

b) La répartition des ressources ne peut pas être contrôlée pour être faite dans un certain ordre.

Donc, une implémentation qui reflète exactement la forme linguistique des régions critiques conditionnelles n'est pas la meilleure, car elle considère la région uniquement comme une séquence d'instructions à exécuter dynamiquement. D'autres implémentations ont été proposées pour éliminer cet inconvénient (voir Schm 76).

2.3. Moniteurs

Les régions critiques conditionnelles ont introduit une "discipline" dans l'utilisation des ressources partagées en localisant l'accès aux variables représentant ces ressources.

L'introduction du concept de "classe" dans Simula 67 par DAHL & concordait avec cette idée en introduisant de plus une structuration : grouper ensemble la définition des objets partagés et des opérations les concernant. Cela a contribué à la définition d'autres concepts pour la synchronisation des processus que nous appellons "structures de synchronisation".

Le *moniteur* (initialement appelé "secrétaire" par Dijkstra (Di 72)) est la première structure ainsi présentée par Brinch Hansen (BH 72, BH 73-a) et développée par Hoare (Hoa 74).
La syntaxe même des moniteurs a été inspirée du concept de "classe" :

```
[class] nom du moniteur : monitor
   begin ... déclarations des données locales au moniteur ... ;
      procédure nom de procédure (... paramètres formels ...) ;
         begin ... corps de procedure ... end ;
      ... déclarations d'autres procédures du moniteur ... ;
      ... initialisation des données locales au moniteur ... ;
   end
```

Un moniteur consiste donc dans un ensemble de données et de procédures qui définissent ensemble un type abstrait. Les procédures du moniteur apparaissent ainsi comme des opérations de base associées au type abstrait.

La différence entre un moniteur et une classe de Simula 67 est que les procédures du moniteur sont destinées à être exécutées par des processus concurrents.

L'appel d'une procédure du moniteur par un processus est fait en précédant le nom de la procédure du nom du moniteur.

Remarquons le caractère plus général de cette structure de synchronisation, par le fait qu'en groupant toutes les régions critiques conditionnelles associées avec la même variable partagée dans un seul module, on obtient un moniteur dont les données locales seront formées par les composantes de la variable partagée.

A l'extérieur du moniteur, seuls les noms des procédures du moniteur sont connus et l'exclusion mutuelle à une donnée locale du moniteur est assurée par le fait *qu'une seule procédure du moniteur peut être exécutée à la fois*. C'est là un exemple de l'exclusion mutuelle minimale (§ 2.1). Donc les processus se partagent l'accès au moniteur d'une façon compétitive. Deux optiques différentes peuvent être envisagées : on peut soit considérer que l'exécution des procédures du moniteur appartient au processus appelant et est soumise aux contraintes d'exclusivité et de réveil, soit que le moniteur est un processus particulier et que l'appel d'une procédure du moniteur est l'exécution d'une fonction du processus appelant vers ce processus particulier. Le processus appelant attend la fin de la fonction pour continuer son avance. Le moniteur joue alors le rôle d'un processus "arbitre".

Mais pour assurer la synchronisation des processus il est nécessaire de suspendre l'exécution d'une procédure du moniteur jusqu'à ce que certaines conditions sur les données du moniteur soient satisfaites.

Comme dans le cas des régions critiques conditionnelles, l'attente a été introduite dans le cas des moniteurs, sous la forme du concept de *variables de type condition*. Une telle variable autorise un processus à relacher l'exclusion mutuelle sur une procédure du moniteur et à se mettre en attente (pour réaliser cette attente, Brinch Hansen et Hoare proposent l'utilisation des "queues d'événements") pour qu'une autre procédure du moniteur puisse lui signaler (lors d'une exécution future par un autre processus) que la condition pour laquelle il attend est satisfaite. Les primitives wait et signal sont introduites pour manipuler ces variables.

La différence avec les régions critiques conditionnelles est que, dans le cas des moniteurs, il n'existe pas de réévaluation implicite d'une expression booléenne, car l'utilisateur doit être sûr que l'expression booléenne implicite devient vraie avant l'opération signal.

La critique que nous faisons à cette implémentation de l'attente, est que toutes les interdépendances entre les différentes procédures du moniteur doivent être analysées, prévues et implémentées par le programmeur lui-même. Aucune analyse statique et globale des contraintes existant dans l'ensemble des processus se partageant (en se synchronisant) les procédures du moniteur n'est faite "automatiquement" au niveau de la compilation.

Une évaluation d'expressions booléennes est suggérée par Hoare dans sa définition des moniteurs (Hoa 74) : "the synchronizing primitive which is easiest to use is probably the conditional wait :

wait (B) ;

where B is a general boolean expression (it causes the given process to wait until B becomes true) ; but this may be too inefficient for general use in operating systems ...".

L'inefficacité suggérée ci-dessus est due au fait que B peut dépendre de données locales du processus exécutant (par exemple des données locales à la procédure du moniteur qui est exécutée). Cela implique qu'à chaque fois qu'un processus abandonne le contrôle du moniteur, l'expression B doive être réévaluée par chaque processus en attente sur cette condition, alors qu'une seule aura accès au moniteur.

On peut imaginer (Kes 77) une autre implémentation de l'attente conditionnelle pour pallier l'inconvénient ci-dessus. Des conditions peuvent être déclarées à l'entrée du moniteur, ce qui implique que ces conditions ne peuvent plus dépendre des données locales à une procédure quelconque du moniteur (donc données locales à un processus appelant). Ces conditions déclarées à l'entrée du moniteur, peuvent être utilisées à l'aide de la primitive wait dans les procédures du moniteur en indiquant, par un identificateur, la condition concernée. L'exécution d'une telle instruction wait invoquera l'évaluation de l'expression booléenne correspondante. Si elle est vraie, alors le processus continue, sinon il est bloqué sur la condition correspondante. Un processus abandonne le contrôle du moniteur quand il est mis en attente sur une certaine condition, ou quand il sort du moniteur.

Lorsqu'un processus abandonne le contrôle, les conditions ayant des processus en attente, et uniquement celles-ci, sont évaluées. Pour la première de ces conditions qui est vraie, un des processus en attente est réveillé ; si aucune condition n'est vérifiée, le moniteur sera accessible par un autre processus extérieur. Remarquons que les processus en attente ont une certaine priorité par rapport aux autres, mais que l'ordre d'évaluation des conditions n'est pas spécifié.

Plus exactement, on peut distinguer, dans le cas de moniteurs, trois niveaux de priorités possibles. Par ordre croissant on a :

. tout d'abord les processus en attente d'entrée dans le moniteur (processus extérieurs) ;

. le niveau des processus qui peuvent "potentiellement" prendre le contrôle du moniteur, mais

- ont réveillé un autre processus et attendent sa sortie du moniteur,

- attendent, à cause de l'exclusion mutuelle, d'entrer dans le moniteur.

. Le niveau du processus qui s'exécute.

Nous verrons (§ 1 Ch. 4) que l'inefficacité des implémentation faites pour les moniteurs est due à l'absence de relations entre ces trois niveaux.

Une autre critique que nous faisons à ce mécanisme, est l'abscence d'interactions entre les conditions. Plus explicitement, l'exécution d'un processus en attente sur une condition n'a aucune influence sur l'évaluation des autres conditions ; donc pas de relation explicite entre la condition de réveil et l'action du processus qui met en attente.

Des contraintes (exprimées, comme nous le verrons plus loin, à l'aide de variables de communication entre les processus) que devraient respecter les processus à l'extérieur du moniteur, permettent l'expression de relations dans l'ensemble des processus synchronisés par le moniteur. Le réveil systématique de tous les processus pour lesquels les contraintes deviendraient satisfaites à la suite de l'exécution d'une procédure du moniteur par un processus, ainsi que des mises en attente systématiques pourront être déduits

Après avoir défini notre formalisme de synchronisation, nous indiquerons la possibilité d'exprimer des contraintes globales pour l'ensemble des processus à synchroniser, ce qui permet une éventuelle implémentation "automatisée" des moniteurs et évite donc à l'utilisateur d'analyser des interactions entre les processus. Répétons encore que ce travail pourra ainsi être fait au niveau des langages ou du système.

Soulignons enfin que les moniteurs ne constituent pas des outils plus "puissants" que les primitives de Dijkstra, mais plus *pratiques* car plus *structurés*.

2.4. Les chemins de synchronisation
("path expressions")

Le concept de moniteur spécifie la synchronisation des processus par la définition d'un objet (ou classe), mais les opérations de synchronisation restent explicites, telles que wait et signal, et doivent être incluses dans le programme.

Une structure de synchronisation plus abstraite a été définie par Campbell et Habermann, où la synchronisation des processus est concentrée au niveau de la définition de types (CH 74).

On considère un processus opérant sur un ensemble donné d'objets par une séquence d'actions. Dans certains cas il est important qu'une seule action puisse être exécutée à un moment donné sur un objet particulier, l'ordre d'exécution des actions sur les différents objets étant indifférent ; dans d'autres cas, l'ordre d'exécution des actions est imposé (la production d'un bien doit précéder sa consommation, par exemple). Un ensemble d'objets peut être défini par son "type abstrait" qui décrit d'une part la structure interne de l'ensemble d'objets et, d'autre part, les procédures ayant accès à cet ensemble d'objets et qui peuvent éventuellement modifier sa structure interne.

Dans (Ca 74) Campbell introduit l'emploi des expressions de chemin définissant l'ordre d'exécution (ou des contraintes sur l'exécution) des actions sur un objet dans la définition de son type. La définition de type contient ainsi la description de la nature des données et leur utilisation. Pour un type donné, les "chemins" de synchronisation font partie de sa structure interne en définissant les contraintes d'ordonnancement auxquelles les actions invoquées par un processus ayant accès à ces objets doivent obéir.

On décrit ainsi les besoins de synchronisation, les définitions des procédures locales et les opérations possibles sur des objets du type spécifié.

Chaque fois qu'un objet d'un certain type est créé, on crée aussi une expression de chemin associée que les processus ayant accès à l'objet doivent respecter dans leurs actions.

L'écriture est de la forme :

type nom ;
... définition des données ...
path ... end
... opérations ...
end type

L'expression de chemin est définie par les mots path et end. L'expression elle-même est une expression régulière décrivant les séquences d'exécution possibles des actions sur un objet d'un type donné et est définie à l'aide des opérandes (nom de procédures) et des opérateurs suivants :

; est l'opérateur de séquencement
+ est l'opérateur de sélection exclusive
* est l'opérateur de répétition (de même les mots path et end représentent la répétition de l'exécution de l'expression qu'ils délimitent).

La syntaxe des expressions de chemin est complétée par les parenthèses ( ) exprimant la priorité.

Par exemple,
path a (b * + c) d end
représente l'exécution d'un nombre arbitraire d'actions b ou d'une seule exécution de l'action c entre l'exécution d'une action a et d'une action d. L'énumération implicite contenue dans l'expression de chemin implique une autre exécution possible de a après la fin de d et ainsi de suite.

Il est facile de remarquer que l'opérateur ; est distributif par rapport à l'opérateur +. Par contre, l'opérateur * n'est distributif ni par rapport à l'opérateur + ni par rapport à l'opérateur ;. Ainsi

$(b + c)^* \neq b^* + c^*$ car $b; c; b \in (b + c)^*$ mais $b; c; b \notin b^* + c^*$

et $(b ; c)^* \neq b^* ; c^*$ car $b; c; b \in (b ; c)^*$ mais $b; c; b \notin b^* ; c^*$

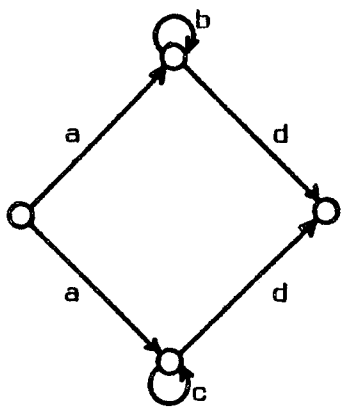
L'évaluation de l'état d'une expression de chemin et le test de l'éligibilité d'une action à exécuter sont faits entre les exécutions des opérations et sont indivisibles.

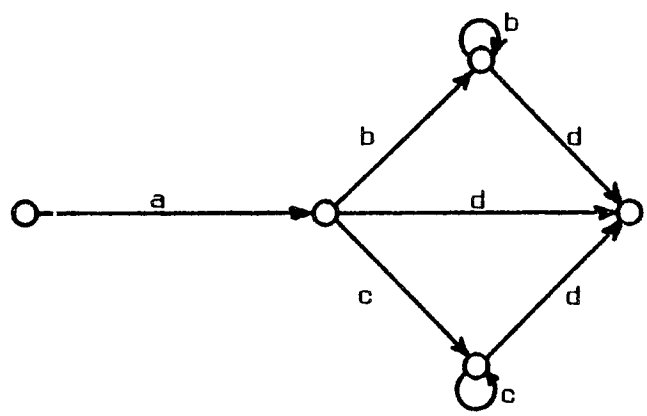
Etant une expression régulière, une expression de chemin peut être représentée par un graphe dont les arcs sont associés aux procédures et les noeuds représentent les états de la machine d'états finis associée. L'état initial et l'état final sont confondus.

L'expression de chemin est dite simple si dans le graphe représentatif il n'existe pas plusieurs arcs ayant le même nom ; elle est non simple dans le cas contraire. Pour une expression non simple, le graphe représentatif n'est pas unique.

Par exemple, path a(b* + c*)d end

est représenté par le graphe : ou par le graphe :





Campbell (Ca 74) a montré qu'une expression de chemin simple peut être implémentée à l'aide des primitives P et V sur des sémaphores booléens.

Ainsi les expression de chemins (et les moniteurs également) ne sont-elles pas des outils de synchronisation plus "puissants" que ceux énumérés auparavant, mais leur méthodologie de conception les rend plus pratiques que les primitives de synchronisation. L'équivalence entre ces structures et les primitives P et V de Dijkstra a été démontrée par l'implémentation des primitives P et V dans ces structures et inversement par leurs implémentations respectives à l'aide des primitives P et V

Comme dans le cas des régions critiques conditionnelles et des moniteurs, on doit avoir la possibilité de spécifier dans une expression de chemin le fait que l'exécution d'une action est soumise à la vérification de certaines conditions.

Pour cela on définit une *expression de chemin conditionnelle* à l'aide d'un élément conditionnel de la forme :

[<cond.1> : <elem.1>, ... <cond.n> : <elem.n>, {elem.(n+1)}]

qui est égal à l'élément le plus à gauche pour lequel la condition est vrai ou, si toutes les conditions sont fausses, au dernier élément qui est spécifié sans condition (Hab75-a).

Les conditions sont des expressions booléennes dans lesquelle les opérandes sont du même type que le type dans lequel l'expression de chemin est définie. De plus, toutes les opérations modifiant les opérandes doivent apparaître dans l'expression de chemin contenant l'élément conditionnel. Ces restrictions sont nécessaires pour assurer que l'évaluation d'une condition ne sera pas perturbée par d'autres actions opérant sur les opérandes de cette condition. C'est donc un moyen d'assurer l'exclusion mutuelle d'accès à ces opérandes.

Nous soulignons ici le besoin d'exprimer globalement pour un ensemble d'actions des conditions d'exécution. Le formalisme que nous développerons plus loin va dans ce sens en généralisant le caractère des conditions.

Par définition, les opérations dans une expression de chemin sont exécutées en exclusion mutuelle, mais dans une définition de type il est intéressant de pouvoir définir plus qu'une expression de chemin. Cela peut être justifié par la possibilité d'exprimer le parallélisme dans l'utilisation d'un type donné. Ces expressions peuvent être indépendantes, en ce sens qu'aucune opérande d'une expression n'apparaît dans les autres ou peut avoir les mêmes opérandes.

Par exemple, les deux expressions de chemin suivantes sur le même type :

path a; c end
path b; c end

expriment le fait que deux actions a successives sont séparées par une action c, de même pour deux actions b successives, et que deux actions c successives sont toujours séparées par une action a et une action b. Mais les deux expressions de chemin ne spécifient pas l'ordre entre a et b, donc ces deux actions peuvent s'exécuter en parallèle. Il s'ensuit que l'action c suivante ne peut commencer son exécution avant la fin des deux actions a et b.

Dans ces expressions de chemin multiples, l'exclusion mutuelle n'est donc pas assurée. Pour remédier à cela, on peut introduire une opération de concaténation notée "°", entre deux expressions de chemin sur le même type, qui définira une nouvelle expression de chemin qui assurera l'exclusion mutuelle. Ainsi :

path a; c ° b; c end

traduit l'exemple précédent.

Analysons les répercussions éventuelles dues aux restrictions imposées sur les éléments conditionnels dans les expressions de chemin conditionnelles. Dans une expression multiple, une variable appartenant à un élément conditionnel peut être modifiée uniquement par des actions appartenant à l'expression de chemin simple dans laquelle la condition apparaît. La concaténation de telles expressions sera licite car toutes les opérations sur les variables appartenant à une condition apparaissent dans l'expression résultante. Une nuance reste à signaler sur la concaténation et nous la signalons par l'exemple suivant.

Considérons l'expression de chemin multiple suivante :

path [x ≤ y : a ; c] end

path [x < y : b ; c] end

et l'expression de chemin résultant de leur concaténation :

path [x ≤ y : a ; c] ° [x < y : b ; c] end

Supposons que les actions a, b et c soient définies par :

a fait x := x - 1 ;

b fait y := y - 1 ;

c fait x := x + 1 ; y := y + 1 ;

L'expression de chemin résultant de la concaténation est correcte, mais pas ses deux expressions parallèles. En effet, a dans la première expression, modifie une variable appartenant à la condition du second et b dans la seconde, modifie une variable appartenant à la condition de la première.

Malgré cette possibilité de spécifier des expressions de chemin multiples pour les définitions de types, les objets ont un caractère figé après leur définition. Ainsi, un utilisateur ayant besoin d'objets pour lesquels d'autres règles de synchronisation soient valables que celles décrites initialement dans la définition du type de cet objet, doit-il redéfinir entièrement le type considéré.

Remarquons que dans une expression de chemin il existe des cas où l'exécution de deux procédures différentes est possible, mais l'exécution de l'une est plus prioritaire. On peut donc introduire dans ce cas, pour remplacer l'opérateur +, un opérateur > indiquant cette priorité. Notons toutefois que l'introduction des opérateurs de priorité augmente les risques de deadlock.

De même, on peut remarquer qu'à la différence d'autres outils de synchronisation, les expressions de chemin contiennent des *compteurs implicites* qui garantissent que la séquence d'exécution est légale par rapport à l'expression de chemin donnée. Comme exemple, rappelons l'extension proposée par Flon et Habermann (FH 76) en introduisant "l'élément de chemin numérique". Par exemple, path $(f - g - \ldots - k)^n$ end dénote une sélection d'une des actions $(f + g + \ldots + k)$ avec la contrainte supplémentaire que $\# f \geq \# g \geq \ldots \geq \# k \geq \# f - n$ où $\# f$ indique le nombre de fois que l'action f a été exécutée. (Par exemple, path $(f ; g)^*$ end s'exprime par path $((f - g)^1)^*$ end).

Soulignons le caractère global de cette contrainte sur l'ensemble des processus concernés par les actions définissant l'expression de chemin, caractéristique qui viendra appuyer notre formalisme.

En vue de besoins qui seront utilisés ultérieurement, rappelons l'implémentation des chemins de synchronisation à l'aide des primitives P et V données dans (CH 74).

On définit les deux primitives suivantes intervenant dans la traduction des parenthèses :

PP (x : integer ; S, Si : sémaphore) ;
begin P (S) ; x := x + 1 ; if x > 0 then P (Si) ; V (S) end
VV (x : integer ; S, Sj : sémaphore) ;
begin P (S) ; x := x - 1 ; if x ≤ 0 then V (Sj) ; V (S) end

Les règles de transformation des expressions de chemin sont les suivantes (on notera par ---> la transformation) :

path {expression} end ---> P (S1) {expression} V (S1)
avec S1 initialisé à 1,

; ---> V (S2) P (S2)
avec S2 initialisé à 0,

({expression 1} + {expression 2}) ---> Og {expression 1} Od
Og {expression 2} Od
où Og est une primitive P ou PP et Od est une primitive V ou VV

P (Si)({expression 3})*V (Sj) ---> PP (x, S, Si)
{expression} VV (x, S, Sj)
avec initialement x = 0 et S = 1.

L'exemple suivant donnera une idée plus précise de ces transformations :

```
path (a + b)* ; c end
          ⇣
P(S1) (a + b)* ; c V(S1)
          ⇣
P(S1) (a + b)* V(S2) P(S2) c V(S1)
          ⇣
PP(x, S, S1) (a + b) VV(x, S, S2) P(S2) c V(S1)
          ⇣
PP(x, S, S1) a VV(x, S, S2) PP(x, S, S1) b VV(x, S, S2)
                                P(S2) c V(S1)
```

avec initialement S1 = S = 1 et S2 = x = 0.

D'autres propositions d'opérateurs et éléments dans les expressions de chemin sont données par Habermann dans (Hab 75-a) et de nombreux exemples sont traités également.

Il est important de remarquer sur cette structure de synchronisation, "l'indépendance" voulue dans sa définition entre les règles que les processus doivent respecter dans leur avance dans l'exécution de certaines actions et l'expression proprement dite de ces actions qui n'est pas spécifiée dans la définition des expressions de chemin. Cette remarque souligne l'indépendance qui doit exister entre la spécification des processus eux-mêmes et les contraintes qu'ils doivent respecter dans leurs actions pour se synchroniser. Cette indépendance montre le caractère "haut niveau" de cette structure par rapport aux primitives de synchronisation, en laissant au programmeur le seul soin (non négligeable !) d'écriture correcte des programmes et d'introduire dans un langage spécifique la description des règles nécessaires à la synchronisation.

Mentionnons enfin le caractère d'inefficacité relative des propositions d'implémentation des expressions de chemin.

Une première ambiguité peut être illustrée par l'exemple suivant. Considérons une expression de chemin non simple :

path a (bc + bd) e end

Comme nous l'avons vu plus haut, cette expression a un graphe représentatif dont deux arcs portent le même nom : b. Une telle ambiguité ne peut pas être détectée quand l'expression sera compilée et donc le programmeur est obligé de la prévoir en transformant d'abord l'expression ci-dessus en :

path ab (c + d) e end

Mais le problème est plus compliqué si l'expression de chemin ci-dessus devient :

path a (bc + b*d) e end

Dans ce cas, l'inefficacité d'implémentation est due à la nécessité d'introduire un nombre non négligeable d'actions "vides" car b* doit être remplacé par Ø + b . b* où Ø est une action vide n'ayant aucune influence sur les variables de synchronisation, mais qui sera traitée au même titre qu'une procédure quelconque à synchroniser. Dans d'autres exemples, b* doit être remplacé par Ø + b (Ø + b (Ø ... bb*) ...).

L'emploi des expressions de chemin n'exclut pas la possibilité de saturation ou de deadlock.

Par exemple, dans l'expression multiple :

path a + c end

path b + c end

c pourra attendre indéfiniment si les exécutions de a et b sont demandées pendant l'exécution l'une de l'autre.

De même, dans l'expression multiple :

path a b c end

path a' c b end

le deadlock est inévitable après la première exécution de a et a'.

Remarquons toutefois l'avantage des expressions de chemin sur les primitives de synchronisation par le fait que le deadlock peut être détecté à la compilation des expressions. Mais comme nous le verrons au § le deadlock reste possible à l'exécution par les processus des actions synchronisées à l'aide des expressions de chemin.

Mentionnons enfin les travaux très récents effectués à l'Université de Newcastle Upon Tyne (LaTo 76) pour la définition formelle d'un langage pour la spécification des systèmes basée sur les notations et la définition des chemins de synchronisation. Le but de ce projet et les choix qui ont été faits sont justifiés de la façon suivante :

Si l'on veut spécifier un système dans un langage de programmation classique, les détails d'implémentation de ce langage influenceront parfois inutilement la spécification du système. Pour l'implémentation de concepts formulés d'une façon abstraite à l'aide d'un formalisme de spécifications, il suffit de donner des règles pour la portabilité des concepts du formalisme qui sert à leur spécification dans un langage évolué possédant des primitives appropriées pour la synchronisation ; c'est dans cette optique que nous développerons notre formalisme et également avec la possibilité (comme on le verra, cette possibilité existe pour la plupart des problèmes de synchronisation) que de telles règles de transition, puissent se déduire systématiquement de la spécification.

La syntaxe du langage développé à Newcastle Upon Tyne est brièvement :

program = begin {path / process @ ;} + end
process = process sequence end
sequence = sequence ; or element / or element
or element = or element, element / element
element = operation / (sequence)
path = path path sequence end
path sequence = start sequence
start sequence = start sequence : star or element / star or element
star or element = star or element, star element / star element
star element = path element * / path element
path element = element / (path sequence)
operation = name
name = letter {letter / digit} *
letter = a / ... / z
digit = 0 / ... / 9

où "+" et "*" indiquent respectivement des séquences d'au moins une ou plusieurs, ou zéro ou plusieurs, occurences de la production qui les précède : {element @ ;} + est équivalent à element {; element} * et "/" indique l'alternative.

Ainsi par exemple :

```
begin   path deposit 1 ; remove 1 end ;
        path deposit 2 ; remove 2 end ;
        process deposit 1 ; deposit 2 end ;
        process deposit 1 ; deposit 2 end ;
        process deposit 1 ; deposit 2 end ;
        process remove 1 ; remove 2 end ;
        process remove 1 ; remove 2 end ;
end
```

décrit l'existence dans ce système de deux ressources distinctes sur

lesquelles les opérations respectent les règles décrites par les deux expressions de chemin ; trois processus distincts qui peuvent déposer dans l'une ou l'autre des ressources et deux processus distincts qui peuvent enlever.

Le lecteur peut facilement constater que ce système n'est pas dépourvu de conflits ("inanition" du dernier processus dans certains cas).

2.5. Les réseaux de synchronisation

La théorie des graphes a été développée dans les trente dernières années pour la description, l'analyse et un même temps la modélisation des systèmes de processus du monde réel. En effet, des problèmes réels, tels les problèmes de transport, ont été à la base de cette théorie.

Le modèle de réseaux introduit par Petri a servi à la modélisation des systèmes de processus, car il permet l'expression de propriétés telles que le traitement parallèle, l'exclusion mutuelle, ... (HC 70). Les réseaux de Petri définissent plus qu'un outil de synchronisation ; ils constituent en même temps un formalisme pour la modélisation des processus et l'étude de leurs propriétés telles que l'absence du deadlock et les règles de synchronisation.

Nous insistons sur cette remarque, car elle concorde avec la description distincte de la structure interne du processus d'un côté et de leurs règles de synchronisation d'un autre côté, ou plus généralement de la définition du type d'un objet donné et de son comportement dans un environement concurrentiel formé d'un ensemble d'objets du même type ou de types distincts.

Remarquons toutefois que les réseaux de Petri ne représentent qu'un sous-ensemble des propriétés des processus, car ils n'incluent pas la structure de données des programmes, l'allocation des ressources physiques aux processus et le détail algorithmique quand il est purement séquentiel et local. Avec un réseau de Petri, on représente un "squelette" des algorithmes ou une "structure" vis à vis de la synchronisation.

Les premiers résultats théoriques sur les réseaux de Petri ont été obtenus par Petri lui-même en analysant la représentabilité des fonctio logiques élémentaires par flots dans les réseaux.

Plus tard, Holt (Holt 68), Genrich (GL 73), Holt et Commoner (HC 70) ont employé ce formalisme pour l'étude des modèles de systèmes informatiques.

Pour la présentation de ce formalisme, nous rappelons un certain nombre de notions classiques de la théorie des graphes (Ber 73).

Un *graphe* est un couple G = (X,U) formé d'un ensemble X (supposé fini) de "sommets" et d'un sous-ensemble U du produit cartésien X x X "d'arcs". Précisons que cette définition correspond à ce que l'on appelle couramment un *graphe orienté* car par cette définition tout graphe est orienté, mais pour des raisons liées à certaines applications (quand par exemple la relation U est symétrique), il est commode de considérer un graphe comme non orienté en ignorant le sens des arcs implicitement contenu dans le produit cartésien.

Si $U_k = (x_i, x_j) \in U$ alors le sommet x_i est appelé "extrêmité initiale" de l'arc U_k et x_j son "extrêmité finale".

Si i = j, alors U_k est une "boucle".

L'arc U_k est appelé "sortant" de x_i ou "incident" à x_j.

Un sommet n'ayant ni arcs incidents ni arcs sortants, est dit "isolé".

Un "chemin" (de longueur p) dans G est un ensemble $U_1,U_2,\ldots,U_p$ d'arcs et un ensemble $x_0,x_1,\ldots,x_p$ de sommets tels que $U_k = (x_{k-1},x_k)$ ou $U_k = (x_k,x_{k-1})$ avec $k = 1,2,\ldots,p$.

Si $x_0 = x_p$ alors le chemin est dit "fermé".

Si pour tout $k = 1,2,\ldots,p-1$ le sommet x_k n'apparaît qu'une seule fois dans la définition du chemin, alors ce chemin est "simple".

Un chemin simple et fermé s'appelle "cycle".

Si dans un chemin on a $U_k = (x_{k-1},x_k)$ pour tout k alors le chemin est appelé "chaîne".

Une chaîne simple et fermée est appelée "circuit".

Un graphe est "connexe" si pour tout couple de sommets il existe au moins un chemin qui les relie.

Nous nous limiterons ici aux graphes connexes et sans boucle. Un tel graphe est dit "biparti" si l'ensemble des sommets est la réunion de deux sous-ensembles distincts non vides ($X = P \cup T$, $P \neq \emptyset$, $T \neq \emptyset$, $P \cap T = \emptyset$) et dont les arcs n'ont jamais leurs extrêmités dans le même sous-ensemble.

On appelle couramment le couple (G,f) formé d'un graphe G et d'une application f sur G (sur X ou sur U ou sur les deux) à valeurs réelles un "réseau".

Un réseau de Petri (PN,μ) est défini par :

a) un graphe biparti $PN = (P, T, o\rightarrow|, |\rightarrow o)$ formé :
   - d'un ensemble non vide de "places", P
   - d'un ensemble non vide de "transitions", T avec $P \cap T = \emptyset$
   - de deux relations binaires $o\rightarrow| \in P \times T$ et $|\rightarrow o \in T \times P$ telles qu'il n'existe pas de places ou de transitions isolées.

b) une fonction $\mu : P \rightarrow N$ appelée marquage de places. Chaque place du réseau peut ainsi contenir un certain nombre de "marques".

Une place $p \in P$ est dite "place d'entrée" d'une transition $t \in T$ si la relation $p\ o\rightarrow|\ t$ est vraie ; si par contre $t\ |\rightarrow o\ p$ alors p est appelée "place de sortie".

Un réseau de Petri "pur" est un réseau de Petri tel que pour chaque transition l'ensemble des places d'entrée et de sortie est disjoint.

La définition ci-dessus des réseaux de Petri correspond à cell donnée initialement par Petri lui-même ; plusieurs généralisations en ont ét faites pour des besoins de représentation de systèmes complexes :

. réseaux de Petri "généralisés" pour lesquels une fonction de l'ensemble des arcs dans N - {0} a été rajoutée (c'est un réseau "capacitif" où chaque arc possède une capacité non nulle s'il existe) ;

. réseaux de Petri avec priorités

. réseaux de Petri temporisés

. réseaux de Petri synchronisés, .. (MPS 77).

Par la suite nous ne ferons pas de différence de langage entre réseaux de Petri et réseaux de Petri généralisés.

Dans les réseaux de Petri on peut définir une règle d'évolution permettant à un marquage d'en générer un autre. Cette règle d'évolution peut se définir ainsi :

si toutes les places d'entrée p d'une transition t possèdent un nombre de marques supérieur ou égal à la capacité de l'arc les reliant à cette transition, alors on enlève de chaque place d'entrée p un nombre de marques égal à la capacité de l'arc p o→| t et on ajoute dans chaque place de sortie p' de t un nombre de marques égal à la capacité de l'arc t |→o p'.

Ce changement d'état du réseau est une opération indivisible et on dit que la transition t a été *activée*.

Le flot des marques dans les chaînes du réseau sera ainsi synchronisé aux transitions.

Mais puiseque la possibilité d'activation d'une transition dépend uniquement de son environnement immédiat, il est possible que plusieurs de ces événements prennent place indépendamment l'un de l'autre "simultanément". Remarquons toutefois que dans ce cas, certains états du système peuvent ne pas apparaître explicitement dans son évolution.

Soient μ et μ' deux marquages de PN et soit τ un sous-ensemble des transitions activables indépendamment dans μ. Si le résultat de cet ensemble d'événements "simultanés" génère le marquage μ' on peut ainsi définir une relation $\overset{=>}{\tau}$ entre μ et μ' par :

$$\mu \overset{=>}{\tau} \mu' \text{ ssi } (\forall\, p \in P)\ (\forall\, t \in \tau)\ \mu(p) \geq \sum_{t \in \tau} \text{capa}(p,t)$$

$$\mu'(p) = \mu(p) + \sum_{t \in \tau} \text{capa}(t,p) - \sum_{t \in \tau} \text{capa}(p,t)$$

où capa est la fonction sur les arcs définissant un réseau de Petri généralisé.

A partir de cette relation on peut définir une relation d'équivalence entre les marquages par :

$$\mu \simeq \mu' \text{ ssi } (\exists \tau_1, \tau_2, \ldots, \tau_s)\ (\exists \mu_0, \mu_1, \ldots, \mu_s) :$$

$$\mu_0 = \mu,\ \mu_s = \mu' \text{ et } \forall\, i \in \{1,2,\ldots,s\} : \mu_{i-1} \overset{=>}{\tau_i} \mu_i$$

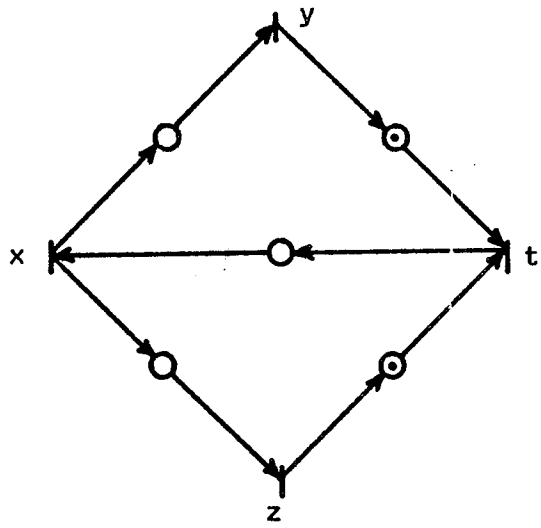
$$\text{ou } \mu_i \overset{=>}{\tau_i} \mu_{i-1}$$

Nous appelons *réseau de synchronisation* le couple formé de PN et de la classe d'équivalence (μ) correspondant à un marquage initial μ du réseau.

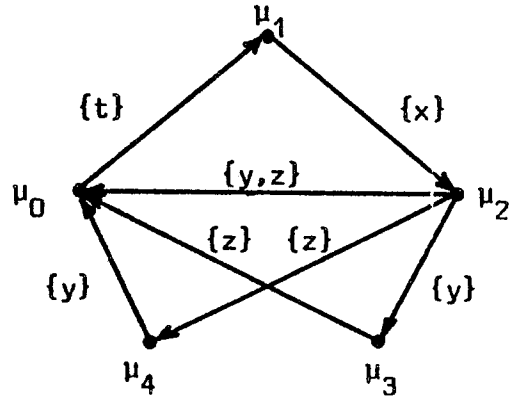
Une représentation d'un réseau de synchronisation peut être obtenue de la manière suivante :

- l'ensemble des marquages de (μ) définiront les sommets du graphe,
- les arcs du graphe correspondent aux relations $\overset{=>}{\tau}$ entre les différents marquages.

Exemple :



Le réseau de synchronisation correspondant au réseau de Petri ci-dessus peut être représenté par :



μ_0 étant le marquage initial indiqué ci-dessus.
Notons que l'ordre de marquage de {y, z} est indifférent.

Nous avons rappelé ci-dessus la définition des réseaux de Petri et des réseaux de synchronisation et nous insistons encore sur l'élégance de ce formalisme pour la modélisation des systèmes séquentiels et parallèles. De ce fait, il s'ensuit que la simulation des processus parallèles se base à profit sur ce modèle, étant donnée la facilité de compréhension et l'existence dans le modèle de certaines preuves sur le comportement du système de processus (absence du deadlock, vivacité, ...).

Il faut toutefois souligner que la représentation par un réseau de Petri des systèmes complexes devient rapidement laborieuse et elle ne peut se faire que dans le cas où ces systèmes restent déterministes (tous les cas d'exécution sont connus ou analysables à l'avance), mais ne comprend pas le cas des systèmes non-déterministes (programmes concurrents dans les applications réparties, par exemple).

L'apport des réseaux de Petri à la synchronisation des processus concerne la spécification sémantique de la synchronisation.

Nous soulignons encore la possibilité de séparation entre la représentabilité des contraintes de synchronisation d'une part, et de la structure des processus vis à vis de la synchronisation d'autre part.

Pour la modélisation des systèmes complexes à l'aide des réseaux de Petri, il existe des méthodes de décomposition permettant une analyse par "réduction", mais une question reste encore ouverte à notre connaissance : peut-on trouver, pour un problème donné, des méthodes *systématiques* de construction de réseaux de Petri représentatifs "corrects" ?

CHAPITRE II

FORMALISME DE SYNCHRONISATION PAR CONTRAINTES

"still glides the stream,
and shall for ever glide ;
the form remains,
the function never dies"

W. Wordsworth, 'After thoutht'

1. FORMALISATION DE LA NOTION DE PROCESSUS

1.1. Historique

Ce mot latin signifiant "progrès" est devenu, dans la théorie des systèmes, une notion abstraite à propos de laquelle Wilkes disait : "... in neither case it is very easy to pin the concept down in a definition".

Le 'fil rouge' dans la compréhension de cette notion peut être dégagé en se reportant aux nombreuses définitions données (Dijkstra, Denning, Dahl, Wirth, Knuth, Horning & Randell, Brinch Hansen, Crocus, etc ..), et en constatant que le facteur commun à toutes ces définitions est un ensemble "d'opérations" ou "actions" ou "instructions" temporelles qui, exécutées dans un certain ordre sur un ensemble fini de données, définissent une entité abstraite et dynamique qui créée à un instant donné disparaît en général au bout d'un temps fini en générant un autre ensemble fini de données.

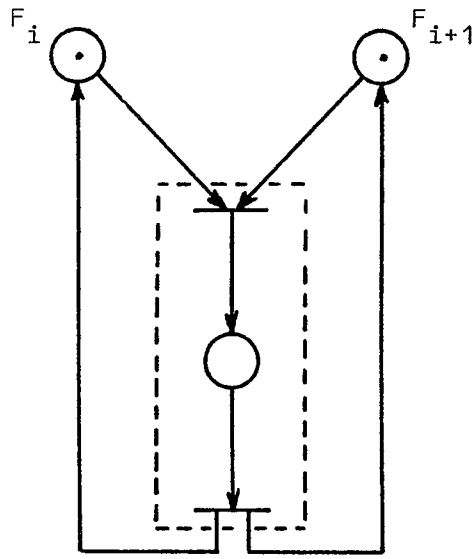
D'un autre côté, on peut remarquer le caractère très matériel de toutes ces définitions. En effet, elles tiennent compte des processeurs sur lesquels l'ensemble d'actions définissant le processus s'exécutent.

Seuls les réseaux de Petri peuvent être considérés comme un outil de modélisation abstrait permettant de représenter les caractéristiques essentielles de la spécification des processus.

Mais comme nous l'avons déjà remarqué au § 1.2.5. en rappelant la définition des réseaux de Petri, ces derniers ne peuvent représenter qu'un sous-ensemble des propriétés d'un processus. Tout en gardant la "puissance" des algorithmes, l'aspect "algorithmique" des réseaux de Petri reste aussi pénible que dans le cas d'une machine de Turing.

En effet, considérons le problème des "philosophes" (dans lequel un ensemble de processus P_i (i = 0,1,...,4) se partagent un ensemble de ressources F_i (i = 0,1,...,4). Pour s'exécuter, chaque processus doit pouvoir posséder les deux ressources F_i et F_{i+1} (modulo 5 dans l'exemple) qu'il relache au bout d'un temps fini.

Dans ce cas, le processus P_i peut être caractérisé par le réseau de Petri suivant :



Dans la suite de ce paragraphe, nous allons donner une formalisation de la notion de processus uniquement comme base pour la suite de ce travail, car la spécification des processus dans un système sort du cadre de nos objectifs.

1.2. Processus et algorithme

Il y a une trentaine d'années, Markov (Ma 51) définissait un algorithme comme une application obtenue par composition (dans un nombre fini de pas) d'applications "éléméntaires", ou plutôt, d'après sa terminologie, par "substitution de sous-chaînes dans une chaîne". En général, on n'a pas besoin d'énumérer la séquence d'application définissant l'algorithme, mais on décrit plutôt un sous-ensemble de ces applications qui composées, forment une "étape" de l'algorithme et la loi qui définira la prochaine étape. Cette démarche a marqué aussi la définition des machines de Turing et plus tard en pratique la définition des (sous)programmes. Ainsi, une équivalence apparaît-elle entre les classes de langages récursifs, les fonctions calculables ou récursives et les algorithmes.

Un algorithme peut être considéré formellement comme une fonction obtenue par composition d'applications "élémentaires".
Les langages de programmation permettant l'expression d'un algorithme, fournissent les applications "élémentaires" qui le composent.

On peut remarquer toutefois un point de rapprochement. La composition d'un ensemble (fini ou infini) de fonctions f_1, f_2, .., f_n .. (définies sur le même espace, produit cartésien de toutes les variables) est faite dans un certain ordre défini par la règle de composition.

A un certain moment, on applique la fonction f_{i+1} au résultat de la composition des i premières fonctions sur un argument pris dans l'ensemble de définition de ces fonctions. De même, dans un langage, les applications élémentaires sont définies à l'aide des opérations et de certaines instructions (affectations) ; le choix de la prochaine fonction à appliquer (instructions conditionnelles while, if, etc .. ou le séquencement) et finalement on peut rendre élémentaire une fonction déjà calculée (procédure).

Ce couple formé par l'ensemble des fonctions restant à appliquer (ensemble ordonné) et le résultat des fonctions précédentes (c'est l'ensemble des informations nécessaires et suffisantes pour poursuivre le calcul) est habituellement appelé "état des processus".

Un processus apparaît ainsi comme lié à l'exécution d'un algorithme suivant les règles définies par l'ordre de composition des fonctions sur un ensemble fini de données et à l'aide d'un ensemble fini de ressources.

Donc, le terme "processus" est une entité dynamique correspondant à des entités statiques (ensemble de procédures par exemple).

Du point de vue pratique, dans un système il est couramment accepté de dire qu'un processus est *créé* quand on associe un ensemble de ressources (physiques ou virtuelles) capables de supporter son exécution, à un (sous)programme.

En supposant l'existence d'un ensemble infini de ressources, un processus créé se comporte comme une machine logique ayant deux états d'exécution possibles. Il peut être dans l'état *actif* quand il exécute ses instructions, ou dans l'état *bloqué* quand il est en attente d'événements qui doivent avoir lieu dans d'autres processus avant qu'il puisse passer à l'exécution de ses instructions suivantes.

Mais en pratique, l'ensemble limité des ressources nécessite un nouvel état pour un processus créé. On dit qu'un processus créé est dans l'état *prêt* s'il est en attente de ressources pour son exécution.

Un processus coopère donc avec d'autres processus dans le partage des ressources physiques. Dans ce sens, un système apparaît comme un ensemble de processus séquentiels coopérants et compétitifs s'exécutant en parallèle. Ainsi des modèles de processus asynchrones ont-ils été développés en partant d'un ensemble de "variables partagées" suggérées par les programmes se déroulant dans un système multiprogrammé et qui communiquent à travers des espaces mémoire communs. Dans ce sens, nous rappelons les travaux de Horning et Randell (HR 73) qui sont les plus significatifs.

Pour un modèle mathématique caractérisant un ensemble de processus, il est essentiel de souligner que seules les fonctions définissant les processus et les relations résultant dans leur ensemble de leur existence et progression commune, sont à retenir.

Considérons un processus défini par une séquence de composition de fonctions définissant un certain algorithme et soit E l'ensemble de ses états possibles.

Un changement d'état des processus est couramment appelé un "pas" et consiste en affectation de nouvelles valeurs à un sous-ensemble des variables intervenant dans sa définition.

Nous formalisons cela sous la forme d'une *fonction atomique* f définie sur E et à valeurs dans $\mathcal{P}(A)$ où A est l'ensemble d'affectations possibles pour les valeurs des variables et $\mathcal{P}(A)$ est l'ensemble des parties de A. L'ensemble A peut être considéré comme le produit cartésien $A = V \times V'$ où V est l'ensemble des variables et V' l'espace des valeurs associé. La fonction f est supposée indivisible. Si $e_i \in E$ est un état des processus, alors l'état suivant e_{i+1} est obtenu en appliquant *une partie* de $f(e_i)$ à e_i. Nous faisons la restriction à une partie de $f(e_i)$ en éliminant la possibilité d'avoir deux affectations pour une même variable. Avec cette restriction, la preuve du déterminisme d'un processus est plus aisée. D'une façon plus générale, en supposant pour simplifier que les variables ont une existence non limitée, les états d'un processus sont définis par le couple $(\mathfrak{E}, F)$ où $\mathfrak{E}$ est le produit cartésien des espaces des variables et F est une séquence de fonctions $f_i : \mathfrak{E} \to \mathfrak{E}$ composées.

1.3. Relations entre les processus

Les problèmes posés par la formalisation des processus sont dûs essentiellement à la communication entre les processus comme résultat des relations existant dans leur ensemble.

Notons que les outils de synchronisation mis en oeuvre jusqu'à ce jour, sont le résultat de l'interprétation de la communication entre les processus d'un ensemble par des variables partagées.

Pour pouvoir dégager un formalisme pour la synchronisation, nous avons besoin d'une notion de processus comprenant leur existence plus générale ; par exemple, dans un système réparti (cas des réseaux) où des contraintes nouvelles (délais de transmission, absence de mémoire commune, etc ..), doivent intervenir.

Ainsi considérons-nous que pour un processus, la fonction que nous pouvons formellement lui associer, représente une entité interne au processus (son état n'a pas besoin d'être bien défini pendant l'exécution de cette fonction, en ce qui concerne les relations qu'il aura avec les autres processus. Chaque fonction atomique "lit" les variables au début de son exécution et les "écrit" à la fin de son action. Ces variables de "lecture" et "écriture" sont appelées les *entrées-sorties* des processus (Hoa 77).

Les actions des processus décrites par les fonctions atomiques et les *entrées-sorties* constituent les éléments essentiels pour l'expression des relations dans un ensemble de processus concurrents.

1.4. Programmes concurrents

Intuitivement, un programme est composé d'une collection d'opérations groupées en processus et de modules issus de la communication entre ces processus. Chaque opération peut être active ou non. Chaque période d'activation d'une opération (il est impossible pour une opération d'être active sans devenir passive après une durée finie mais indéfinie), est considérée comme une unité ininterruptible que nous avons appelée action atomique.

En général, toute action atomique d'un programme appartient à au moins un processus ou module exprimant la communication entre les processus. Une action atomique appartenant à un processus peut être active (quand le processus est actif) ou inactive (quand le processus est bloqué ou activable). Dans un processus, les actions atomiques peuvent être mutuellement exclusives, ce qui signifie que le processus demande l'activation d'une seule de ces actions pour progresser. Les actions peuvent être séquentielles, ce qui implique que leur activation doit être successive pour l'avance du processus. Efin, un processus peut demander l'activation répétée d'une même action atomique pour avancer.

Un ensemble de processus peuvent se "partager" une action atomique, ce qui signifie que cette action peut être activable, active ou inactive pour tous les processus de l'ensemble considéré, dans un ordre indifférent et sans avoir d'influence sur l'avance des autres processus. Une telle action atomique peut être active pour plusieurs processus à la fois ou dans un certain ordre (la concurrence).

Si une action atomique intervient dans la communication entre les processus, elle doit être active pour un seul processus à la fois. En cas de conflit, un choix reste donc à faire. Si plusieurs actions atomiques apparaissent dans la communication, leur activation devra être faite en fonction des contraintes de synchronisation entre les processus.

Une situation dans laquelle toutes les actions atomiques sont inactives est appelée *impasse*. Si un système n'est jamais en deadlock, il est dit *vivant*.

En cas de conflit entre les processus, c'est-à-dire si des actions atomiques sont activables en même temps par plusieurs processus, mais sont actives en exclusion mutuelle pour ces processus, un seul des processus est actif et les autres activables, mais rien ne peut garantir que dans le futur un processus ne pourra jamais être actif. Une telle situation est appelée *inanition* d'un processus.

Dans un programme concurrent, l'ensemble des variables peut être divisé en trois classes (FHdR 77) :

a. Variables d'entrée :

Ces variables reçoivent leurs valeurs de sources extérieures au processus qui les utilise. L'interprétation usuelle que l'on peut leur donner est sous forme de l'information nécessaire au déroulement du processus ou des messages auxquels le processus doit répondre. Ces variables sont complètement indépendantes de l'état courant du processus. Dans un programme, leur valeur peut être testée par des prédicats.

b. Variables locales :

Ce sont les variables propres au programme, mises à jour par les processus et testées par des tests dans le programme.

c. Variables de sortie :

Ce sont des variables par lesquelles un programme communique avec l'environnement. Elles sont mises à jour par les processus. On peut les interpréter comme des accusés de réception ou d'autres signaux pour l'extérieur du programme.

Un programme concurrent peut donc être considéré comme un ensemble de processus exécutant des actions dynamiques de la façon décrite ci-dessus et d'un interface entre ces processus formé d'actions qui assurent leurs communications avec l'environnement.

1.5. Processus distribués

On appelle ainsi un ensemble de processus tel que chacun d'eux ne peut accéder que des variables locales. Ils ne peuvent pas avoir de variables communes.

Un processus distribué peut seulement appeler des *procédures communes* définies dans d'autres processus. Ces procédures sont exécutées quand les autres processus sont en attente. Elles sont donc exécutées en exclusion mutuelle et c'est l'unique forme que peut prendre la communication entre les processus.

Nous considérons que l'interdiction d'accès à des variables communes est trop restrictive et est issue de considérations matérielles ou de langages procéduraux utilisés dans les systèmes distribués. Nous ne faisons aucune distinction entre processus et processus distribués.

Ces processus peuvent être utilisés dans les systèmes multiprocesseurs ou distribués, avec ou sans mémoire commune. Un autre domaine d'application est celui des systèmes temps réel.

Le non-déterminisme dans un ensemble de processus distribués a nécessité l'introduction d'outils langage nouveaux tels que les "guarded commands" (Di 75). On peut distinguer deux formes de non-déterminisme qui peuvent être caractérisées par les différentes implications sur la prévention du deadlock et de l'inanition.

Le non-déterminisme local (Hoa 76 , FS 77) concerne uniquement les variables locales d'un processus P_i. Par exemple, P_i peut communiquer avec P_{j1}, P_{j2}, ..., P_{jk} et la décision par laquelle cette communication aura lieu dépend des variables locales de P_i uniquement. Ce type de non-déterminisme a nécessité l'introduction des "guarded commands" de Dijkst pour pouvoir s'exprimer dans les langages.

Une "guarded command" permet donc à un processus de faire un choix entre différentes actions atomiques en analysant l'état de ses propres variables. Si aucune alternative n'est autorisée dans l'état courant des processus, alors la "guarded command" ne pourra pas être exécutée et le programme s'arrête.

La syntaxe est la suivante :

(*) if C1 : A1 || C2 : A2 || ... fi

(**) do C1 : A1 || C2 : A2 || ... od

avec la signification suivante :

(*) si des conditions Ci sont vraies, en choisir une et exécuter l'action qu'elle conditionne ; sinon stop du programme ;

(**) tant que des conditions sont vraies, en choisir une et exécuter l'actic qu'elle conditionne.

Le non-déterminisme global concerne l'ensemble des processus et les communications dans leur ensemble et nécessitera un mécanisme de synchronisation spécifié à l'aide du formalisme que nous introduirons plus loir Les caractéristiques peuvent être énoncées informellement dès maintenant. I s'agit de déterminer dans l'ensemble des processus des contraintes de synchr nisation qui définiront une région de synchronisation. Une région de synchrc nisation sera implémentée par des outils de synchronisation qui pourront blc quer une action à la différence des "guarded commands" qui ne le peuvent pas

2. EXPRESSION DE LA SYNCHRONISATION PAR CONTRAINTES

2.1. Contraintes de synchronisation

D'après la définition que nous avons donnée plus haut pour un processus, on peut remarquer que lors de l'exécution d'un processus nous pourrons distinguer :

- un sous-ensemble d'actions du processus exprimant la "communication" avec l'environnement,
- un sous-ensemble d'actions qui sont indépendantes de l'environnement et décrivent uniquement la structure interne du processus.

Le premier de ces sous-ensembles définit dans l'ensemble des processus des relations de communication concrétisées par la compétition dans l'acquisition des ressources et les contributions d'un processus à l'évolution des autres.

La synchronisation, dans l'ensemble des processus, apparaît comme une conséquence de ces relations.

Les ensembles de variables de communication entre les processus forment un ensemble de variables que nous appelons *variables de synchronisation*.

2.1.1. Variables appartenant aux actions des proces

Notre propos n'étant pas la spécification des processus d'un système eux-mêmes, mais plutôt leur synchronisation, nous rappelons simplement ici que les variables appartenant à la représentation concrète des processus sont couramment appelées "variables effectives". Il faut toutefois remarquer que ces variables peuvent être des variables de synchronisation dans l'ensemble des processus (représentant des variables partagées, par exemple). Ainsi, une variable x peut être manipulée uniquement par un processus P1 et une variable y par un processus P2, mais l'avance de ces deux processus doit respecter $x + y \geq 0$.

2.1.2. Variables appartenant aux contraintes de synchronisation

Du point de vue de la synchronisation, les relations dans un ensemble de processus peuvent être définies à l'aide d'un vecteur :

$$X = (x_j) \; ; \; j = 1, 2, \ldots, n$$

dont les composantes sont les variables de synchronisation.

Les relations des processus se traduisent par un ensemble de contraintes sur les variables x_j.

Ces variables peuvent être des variables effectives comme nous l'avons vu ci-dessus, ou des variables introduites pour l'expression des contraintes et souvent attachées aux processus eux-mêmes, ou des variables introduites pour faciliter l'expression des contraintes à l'aide d'un outil adéquat. Les exemples que nous donnerons dans les paragraphes suivants feront ressortir la nature de ces variables.

Mais nous considérons que la spécification d'un formalisme doit demeurer indépendante de l'expression des outils éventuels de mise en oeuvre. Cela ne résulte pas uniquement d'un souci de formalisation, mais l'expression mathématique des contraintes de synchronisation suffit souvent pour déterminer des relations dans l'ensemble de ces contraintes et aussi pour définir un moyen efficace d'utilisation de certains outils déjà existants. Tel est le cas des régions critiques conditionnelles et même des moniteurs (Schm 76).

Dans ce sens, l'expression formelle des contraintes de synchronisation pour un ensemble de processus, peut être utilisée pour faire la preuve de la solution de la synchronisation obtenue avec un outil de synchronisation donné, ou obtenir "automatiquement" cette solution avec des primitives de synchronisation telles que les sémaphores ou leurs variantes. (§II.3)

2.1.3. Définition des contraintes

Les variables de synchronisation d'un système de processus concurrents peuvent être considérées comme les coordonnées d'un point X dans l'espace $\mathcal{E}$ défini par le produit cartésien des espaces de valeurs de chacune des variables de synchronisation x_j, $j = 1, 2, \ldots, n$. Remarquons que souvent dans la pratique, les espaces peuvent être considérés comme identiques en prenant l'espace le plus général apparaissant dans l'expression des actions atomiques.

Il est évident que, sans restreindre la généralité du formalisme, pour que sa compréhension soit facilitée nous pouvons supposer que ces variables x_j prennent des valeurs réelles ou des valeurs entières.

L'ensemble des contraintes de synchronisation pour un système de processus concurrents sera défini par :

$$C = \begin{cases} f_i\ (x_1, x_2, \ldots, x_n) \geq 0 & \forall\ i \in I_1 \\ f_i\ (x_1, x_2, \ldots, x_n) > 0 & \forall\ i \in I_2 \\ f_i\ (x_1, x_2, \ldots, x_n) \neq 0 & \forall\ i \in I_3 \end{cases}$$

où $I = I_1 \cup I_2 \cup I_3$ est un ensemble d'indices que nous supposons ici fini, et $f_i : \mathcal{E} \rightarrow R \quad \forall\ i \quad I$ sont des fonctions réelles semi-calculables (pour assurer que le mécanisme pourra calculer ces fonctions à l'aide d'algorithmes).

On peut donc considérer que $C = \bigcap_{i \in I} C_i$ où chaque C_i est une contrainte de synchronisation.

Cette définition est très générale, mais nous verrons par la suite les restrictions à apporter pour l'expression des problèmes de synchronisation "classiques".

Précisons immédiatement le caractère non restrictif des inégalités. En effet, des contraintes éventuelles exprimées sous forme d'équations peuvent facilement être incluses dans la définition ci-dessus, car une contrainte de la forme

$$f_i\ (x_1, x_2, \ldots, x_n) = 0$$

est équivalente à l'ensemble des deux contraintes

$$f_i\ (x_1, x_2, \ldots, x_n) \geq 0$$

et

$$-\ f_i\ (x_1, x_2, \ldots, x_n) \geq 0$$

L'ensemble des contraintes pourra donc s'écrire :

$$C = F\ (X)\ \delta\ 0$$

où les f_i sont des composantes de F, $X = (x_1, x_2, \ldots, x_n)$ et δ peut être indifféremment $=, \geq, \leq, \neq, >, <$.

L'ensemble des contraintes de synchronisation définit dans $\mathfrak{X}$ un sous-ensemble que nous appelons *région de synchronisation*. Les contraintes de synchronisation peuvent donc être de formes très variées et nous n'imposons aucune autre restriction quant à leur expression. Réciproquement, toute région $C \subset \mathfrak{X}$ semi-calculable (i.e. telle qu'il existe un algorithme capable de déterminer si $X \in \mathfrak{X}$ appartient à C) est équivalente à un tel ensemble de contraintes qui définit la fonction caractéristique de cette région.

Par exemple :

$x_1 \geq x_2$ (forme de contrainte apparaissant dans les "guarded commands" de Dijkstra $(x_1, x_2) \in N^2$

$x_1 + x_2$ premier $(x_1, x_2) \in N^2$

$x^n + y^n = z^n$, $n > 2$, $(x, y, z, n) \in N^4$

La région de synchronisation, dans ce cas (si un problème de synchronisation devait répondre à une telle contrainte), ne peut être déterminée avec les résultats arithmétiques actuels ! (Théorème de Fermat).

$\sin(x_1) + \log(x_2) \geq 0$

$x_1 \cdot x_2 \cdot x_3 = 0$ ($\Leftrightarrow x_1 = 0$, ou $x_2 = 0$, ou $x_3 = 0$)

$x_1 + \frac{x_2}{2} > \frac{1}{2}$ et $2x_1 - x_2 < 0$ et $x_1 > 0$ et $x_2 > 0$

$x \bmod n = 0$

Nous laissons délibérément cette généralité dans l'expression des contraintes de synchronisation et c'est la "calculabilité" de ces contraintes dans différents contextes qui assurera la possibilité de synchroniser.

2.2. Actions des processus et contraintes de synchronisation

Un système de processus concurrents peut être considéré, à un instant donné, comme une entité définie par les valeurs des variables de synchronisation et par l'état d'avancement des processus de l'ensemble considéré. Parmi les variables de synchronisation, celles qui dépendent des actions des processus sont évidemment fonction de leur état d'avancement.

Voici quelques exemples de variables de synchronisation utilisées couramment :

a) variables exprimant le nombre de processus en attente pour une certaine ressource (un message ou un événement sont considérés comme des ressources) ;

b) variables exprimant le nombre d'unités disponibles pour chaque ressource ;

c) variables exprimant le nombre de demandes d'exécution d'un même procédure depuis un temps donné ;

d) variables exprimant le nombre d'actions autorisées pour un processus depuis un temps donné ;

e) variables exprimant le nombre d'actions terminées par un processus depuis un temps donné.

f) variables exprimant le nombre de processus actifs à un ins

Cette liste n'est pas limitative et il est souvent préférable de définir des variables nouvelles par combinaisons linéaires d'autres variables, pour exprimer plus facilement certains problèmes. Ainsi le langage des "compteurs" en est-il un exemple concluant (RV 77, Ger 77, V 78).

Les actions des processus peuvent aussi être considérées comm des relations dans $\mathbb{Z}$ définissant des transitions entre positions du point courant X dans la région de synchronisation.

2.3. Définition de la synchronisation

Il résulte des paragraphes précédents que la synchronisation d'un ensemble de processus est un mécanisme capable de contrôler la position du point courant X dont les composantes sont les variables de synchronisation dans la région de synchronisation définie par un ensemble de contraintes, de façon à faire respecter aux actions des processus, à chaque instant, la totalité de ces contraintes.

Dans le chapitre 4 , où nous analyserons le problème du deadlock à travers le formalisme de synchronisation par contraintes, une distinction sera faite entre les deux formes suivantes de deadlock : deadlock général et deadlock partiel ou local.

2.3.1. Deadlock général ("impasse")

On dit que les processus sont en état de *deadlock général* si la position du point X dans la région de synchronisation est telle qu'aucune action des processus n'est encore possible sans violer au moins une contrainte de synchronisation.

Dans ce cas, tous les processus de l'ensemble considéré sont bloqués pour toujours. Cette situation dépend du séquencement des actions atomiques dans les processus, c'est-à-dire de leur structure.

Remarque :

Nous pouvons distinguer deux cas essentiels qui apparaissent dans l'interprétation du deadlock général, déterminés par la structure de l'ensemble des processus à synchroniser :

. Paradoxalement *cette situation de deadlock est un objectif souhaitable* si l'ensemble des processus décrit la structure d'un algorithme qui est à la base d'un programme séquentiel, car une fois ce "but" atteint,c signifie la terminaison correcte de la construction répétitive décrivant l'algorithme à exécuter. Des preuves de la correction des programmes basées sur des "règles de dérivation" peuvent être faites dans ce contexte (vLs 76, vL 77). Comme exemple qui justifie ce cas, nous rappelons "l'algorithme d'Euclide" pour trouver le plus grand commun diviseur (pgcd) de deux nombre positifs exprimé par Dijkstra dans (Di 75). En effet, cet algorithme peu être considéré comme le résultat de la synchronisation de deux processus cycliques P1 : iter : a :=a - b ;

P2 : iter : b :=b - a ;

processus qui s'exécutent en respectant les contraintes de synchronisation

$$C \begin{cases} a > 0 \\ b > 0 \end{cases}$$

Quand ces deux processus seront en deadlock général, on aura max (a, b) = pgcd (a,b).

A souligner que, ni les corps de P1 et P2, ni la région de synchronisation exprimée par les deux contraintes, ne sont significatifs ; seule la conjonction des deux permet d'obtenir le pgcd des deux nombres.

. Si l'ensemble des processus est formé de programmes cycliques (FP 78) (un système par exemple),le deadlock général sera plutôt une "catastrophe" et des méthodes de prévention sont nécessaires

2.3.2. Deadlock local ("inanition")

Ce cas est caractérisé par le fait qu'un sous-ensemble seulement des processus à synchroniser est bloqué sur l'ensemble des contraintes de synchronisation, tandis que d'autres continuent leur progression.

Nous reviendrons plus tard sur ce cas en analysant l'exemple donné par Dijkstra dans son problème des "cinq philosophes" (Di 71). Cette situation de deadlock est appelée "inanition".

La distinction entre les deux cas de deadlock peut être caractérisée également par la remarque suivante : tandis que le deadlock général peut être constaté expérimentalement, l'inanition ne pourra qu'être prouvée et ces preuves doivent être faites *avant* exécution.

Ces deux formes peuvent également être caractérisées par les deux formes de non-déterminisme rappelées au § II.1.5.

3. VALIDATION DU FORMALISME. EXEMPLES

La définition des contraintes de synchronisation comme facteur essentiel de la synchronisation d'un ensemble de processus a, dans notre esprit, deux justifications essentielles.

D'abord, ces investigations font partie de notre souci principal d'apporter une appréciation à la question de déterminer quelles parties de la programmation peuvent être considérées comme "automatisables" et quell parties resteront à la charge du programmeur comme routines formelles liées aux langages ou aux systèmes utilisés. Evidemment, l'automatisation du fonctionnement des algorithmes à travers la coexistence d'un ensemble de processus concurrents amenés à respecter un ensemble de contraintes de synchronisation, ne peut se faire qu'avec un effort de formalisation et de structuration pour déterminer les contraintes de synchronisation.

Ensuite, des structures de synchronisation adéquates à la mise en oeuvre de ces contraintes de synchronisation dans différents contextes, rendront moins "artisanale" la résolution des problèmes de synchronisation. Dans ce travail, nous ne nous intéressons pas directement à de nouvelles structures de synchronisation, mais nous avons étudié leur nécessité dans (DM 78) et elle résulte de l'expression de la synchronisation à l'aide d'un ensemble de contraintes indépendamment des structures employées ultérieurement pour sa mise en oeuvre.

Cela implique une seconde justification qui consiste dans la nécessité de séparer le "traitement" du "contrôle" au niveau de la synchronisation des processus (BBV 77), l'expression des contraintes de synchronisation étant indépendantes de la structure interne des processus mis en cause. Le couplage des deux dans les applications, garde néanmoins toute son importance ; voir par exemple le calcul du pgcd avec les "guarded commands" de Dijkstra (§ II.2.3.1) où on ne peut séparer sans perdre toute signification

L'expression globale de l'ensemble des contraintes de synchronisation permet en outre de déterminer l'apport des actions effectuées par un processus à l'évolution des autres processus avec lesquels il est en relation.

En effet, chacune des actions d'un processus fait intervenir un ensemble de variables que cette action influence à travers les états du processus au cours de l'évolution déterminée par l'action considérée. Si aucune de ces variables n'est une variable de synchronisation, le point courant X ne sera pas affecté dans la région de synchronisation. Si au contraire, l'action change un sous-ensemble des variables de synchronisation tel que la nouvelle position du point courant n'appartient plus à la région de synchronisation, l'avance du processus est "bloquée".

La donnée explicite de l'ensemble des contraintes permet de déterminer pour chaque action l'ensemble des conditions à vérifier ; ce sont celles qui font intervenir des variables de synchronisation affectées par cette action.

Notons que chaque action nécessite la reconsidération des processus bloqués auparavant, mais que les relations que nous expliciterons au § IV.1. entre les actions des processus et l'ensemble des contraintes de synchronisation, nous permettront de déterminer une stratégie pour le choix des processus devenus réactivables après l'exécution d'une action d'un processus vérifiant l'invariant défini par l'ensemble des contraintes de synchronisation.

Remarque :

A partir de la définition de la synchronisation à l'aide d'un ensemble de contraintes que les processus doivent vérifier invariablement, il résulte deux structures de données utilisables successivement, capables de faciliter ce mécanisme :

a) pour chaque action des processus on peut déterminer un sous-ensemble des contraintes de synchronisation qui risquent de ne pas être satisfaites après son exécution ; seules les contraintes où apparaissent des variables modifiées par l'action sont à vérifier. (Dans certains cas, des considérations mathématiques peuvent éviter même ces vérifications, par exemple, la contrainte $s \geq 0$ et l'action $s := s + 1$ ne nécessitent qu'une vérification initiale et non après chaque exécution de l'action).

b) Chaque déplacement du point courant tenté ou réussi dans la région de synchronisation, peut déterminer un ensemble de processus qui risquent d'être débloqués à la suite de ce déplacement.

La première structure est la base de la plupart des implémentations faites pour les moniteurs et pour les régions critiques conditionnelles. La seconde peut essentiellement améliorer ces implémentations ; nous y reviendrons au § IV.1.

3.1. Exemple de contraintes exprimant les primitives de Dijkstra

Considérons la définition des primitives P et V de Dijkstra donnée dans (Di 68-a) (cf. § I.2.1).

Pour la synchronisation d'un ensemble de processus (dans les termes du formalisme introduit au §II.2) ceux-ci ont accès à une variable de synchronisation (s) et peuvent effectuer sur cette variable des actions soumises à certaines contraintes.

Plus précisément, ils ne peuvent qu'incrémenter ou décrémenter la variable de synchronisation en respectant toujours la contrainte :

$$s \geq 0$$

La forme de la contrainte et des actions correspondant aux primitives P et V est remarquablement simple.

Soulignons encore que la contrainte définit une région de synchronisation qui doit toujours être respectée et non une condition à vérifier avant une action particulière. Cela implique une parfaite "symétrie" entre les opérations P et V, vues sous cet angle.

Si maintenant on considère un ensemble fini $\{s_i\}$ $i \in I$ de variables de synchronisation, l'usage des primitives P et V sur cet ensemble est équivalent à un ensemble d'actions de la forme :

$$s_i := s_i \pm 1,\ i \in I$$

qu'effectuent les processus sur ces variables tout en respectant les contraintes

$$\{s_i \geq 0,\ \forall\ i \in I\}$$

3.2. Lecteurs et Ecrivains

Considérons le premier problème des "readers/writers" (c'est-à-dire sans priorités) proposé par Courtois & al. (CHP 71).

Rappelons qu'il s'agit de deux classes distinctes de processus indépendants, une classe de "lecteurs" qui peuvent se partager la ressource et une classe "d'écrivains" qui doivent avoir l'accès exclusif à la ressource.

En notant par l et e respectivement le nombre de lecteurs et d'écrivains actifs, les contraintes de synchronisation associées à ce problème s'expriment couremment par (e = 1 et l = 0) ou (e = 0 et l ≥ 0). Ces contraintes ne sont pas uniques et si nous voulons exprimer ce problème à l'aid d'un ensemble de contraintes, tel qu'il a été introduit au § II.2, nous devons écrire par exemple :

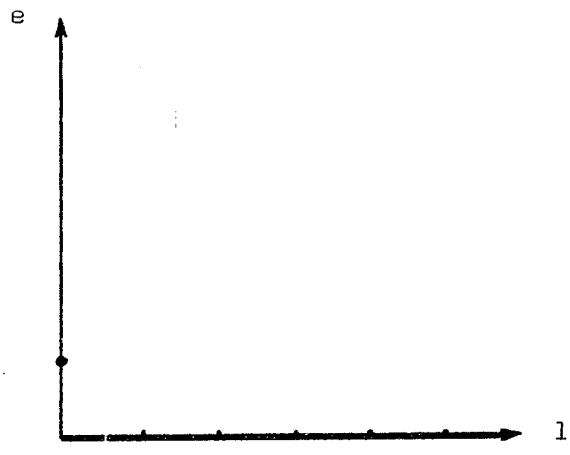
$$0 \leq \text{sign}(l) + e \leq 1$$

où la fonction "sign" est définie sur les réels par :

$$\text{sign} = \begin{cases} 1 \text{ si } x > 0 \\ 0 \text{ si } x = 0 \\ -1 \text{ si } x < 0 \end{cases}$$

Cette écriture fonctionnelle des contraintes, bien que moins évidente à trouver, contient en elle-même le fait qu'un seul sémaphore sera nécessaire pour l'implémentation à l'aide des primitives P et V, comme nous le verrons au chapitre suivant, en montrant que cette expression des contraintes permettra également une écriture "automatique" de la solution à l'aide de P et V.

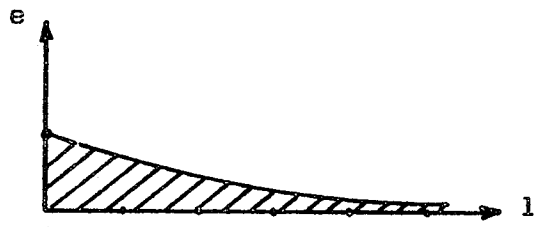
Les conditions initiales et l'ordre des actions des processus montrent que l'inégalité de gauche est redondante. La région de synchronisation peut donc être représentée par :



Cette forme algébrique de la contrainte (donc de la région de synchronisation) n'est pas unique et pour le problème donné on peut trouver que la contrainte

$$e \leq \frac{\lambda}{1 + \lambda}$$

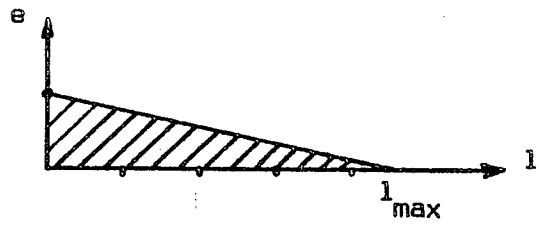
avec e et l entiers positifs et λ un paramètre quelconque positif, exprime aussi bien les besoins de synchronisation entre les deux classes de processus. Donc tout le faisceau d'hyperboles définira des régions de synchronisation valides pour le même problème.



C'est maintenant au programmeur de choisir en fonction de la structure de synchronisation choisie pour la spécification du problème, la forme algébrique la plus adéquate pour les contraintes. La nécessité de trouver des méthodes pour la construction "correcte" des contraintes de synchronisation en fonction du problème à traiter et des outils de synchronisation disponibles à cet effet, est d'autant plus impérative. C'est là une limite au travail exposé ici.

Si maintenant on suppose le nombre de lecteurs limité à l_{max}, le faisceau d'hyperboles défini plus haut sera transformé par exemple dans une contrainte linéaire :

$$l + l_{max} \cdot e \leq l_{max}$$



Si l'expression du problème devait se faire sous la forme d'un ensemble de contraintes linéaires, l'expression donnée à l'aide de la contrainte ci-dessus serait unique. Pour résoudre ce problème de synchronisation, nous reviendrons au chapitre suivant en donnant une solution d'une généralisation du problème.

3.3. Producteurs – consommateurs

Reprenons l'exemple des "pommes et des bananes" donné dans (BM 76). Il s'agit de maintenir un stock de deux produits, de telle manière que le nombre d'unités du premier produit x_1 et le nombre d'unités du deuxième produit x_2 (nombre d'unités disponibles, en cours de dépot ou de retrait), vérifient les contraintes suivantes :

$$\frac{2}{3} x_1 \leq x_2 \leq 2x_1$$

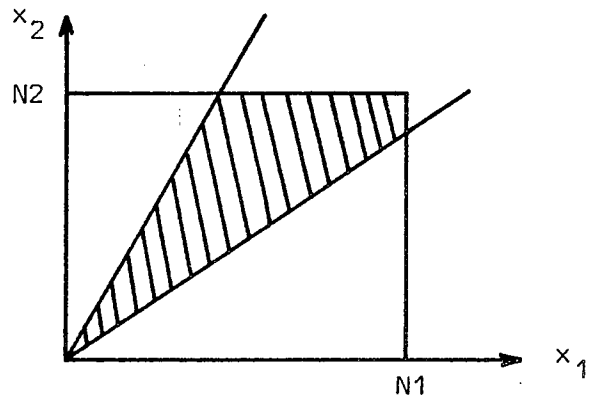
L'ensemble des processus à synchroniser est formé de deux couples "producteur-consommateur" qui sont soumis aux contraintes de synchronisation classiques à un problème du type "producteur-consommateur" et aux contraintes supplémentaires ci-dessus.

En supposant que chaque produit est livré par un processus "producteur" et consommé par un processus "consommateur", unité par unité, comment synchroniser ces processus sachant que la capacité de stockage est limitée à N1 unités pour le premier produit et N2 unités pour le second ?

L'ensemble des contraintes du problème s'écrira :

$$\begin{aligned} -2x_1 + 3x_2 &\geq 0 \\ 2x_1 - x_2 &\geq 0 \\ N1 - x_1 &\geq 0 \\ N2 - x_2 &\geq 0 \end{aligned}$$

et peut donc être représenté par la région de synchronisation* :



* Deux autres cas de figure correspondent à 2N1 > 3N2 et 2N1 < N2.

Ce problème très simple nous permet toutefois quelques remarques intéressantes. Par exemple, si le point courant initial est (0,0) qui appartient à la région de synchronisation et qui est un état initial naturel pour le système considéré, aucune action individuelle de production ou de consommation exécutée par les processus n'est licite. En effet, les déplacements du point courant conduiraient en des points extérieurs à la région de synchronisation, alors qu'une action de production d'une unité de x_1 et une de x_2 exécutées simultanément seraient licites.

De même, si le point courant initial est (1,1), l'état (0,0) devient inaccessible si l'on n'autorise pas une certaine "simultanéité" dans les actions des processus,simultanéité autorisée par les contraintes de synchronisation.

Nous soulignons donc que cet exemple nous permet en outre de donner un aperçu très simplifié de la possibilité qu'offre l'analyse des contraintes de synchronisation pour caractériser les différentes situations de deadlock. (*)

La formalisation des problèmes de synchronisation à l'aide d'ensemble de contraintes nous permet de souligner la nécessité d'analyser les points suivants pour la prévention des deadlocks :

a) la région de synchronisation,

b) la position du point courant dans la région de synchronisati

c) l'ordonnancement des actions futures des processus faisant varier la position du point courant (prendre comme point de départ dans l'exemple précédent, le point (1,1) par exemple, auquel cas en commençant par une production de y on reste dans la région de synchronisation),

d) une structuration des actions futures des processus (c'est la situation décrite plus haut où des actions groupées sont nécessaires).

(*) Une analyse détaillée de ce problème sera faite en indiquant une méthode de vérification du deadlock pour les programmes concurrents au § IV.2.

CHAPITRE III

UN CAS PARTICULIER :

LES CONTRAINTES LINÉAIRES DE SYNCHRONISATION

"example is always more efficacious than precept"

Dr. Samuel Johnson, 'Rasselas'

1. CONTRAINTES LINÉAIRES DE SYNCHRONISATION

Ce chapitre constitue un exemple d'application du modèle de synchronisation par contraintes introduit au chapitre précédent. Malgré sa simplicité, il couvre la plupart des problèmes "classiques" de synchronisation et l'objectif de ce chapitre est de développer des solutions pour quelques uns de ces problèmes.

Soit la région de synchronisation définie par l'ensemble :

$$C = \{x = (x_j)\ j = 1,2,\ldots,n : \sum_{j=1}^{n} a_{ij}\, x_j \geq b_i,\ i \in I\}$$

où I est un ensemble fini, a_{ij}, b_i sont des constantes entières et les x_j des variables de synchronisation.

Nous supposons donc pour la suite de ce chapitre que les contraintes de synchronisation d'un ensemble de processus définissent un invariant de cette nature.

1.1. Traduction automatique en primitives P et V

D'après un des objectifs que nous nous sommes fixés, montrons que, si pour un ensemble de processus l'invariant de synchronisation définit une région de synchronisation spécifiée par des contraintes linéaires, alors pour un tel problème de synchronisation, une solution peut être donnée "automatiquement" à l'aide des primitives P et V proposées par Dijkstra. Ainsi, la formalisation d'un problème de synchronisation à l'aide seulement d'un ensemble des contraintes que les processus concernés doivent vérifier, peut-elle éliminer la nécessité d'un certain "doigté" de la part du programmeur pour l'écriture ou la compréhension des solutions. De plus, comme nous allons le constater, cela facilitera la preuve de la correction des solutions apportées à un tel problème de synchronisation.

Notons

$$s_i = \sum_{j=1}^{n} a_{ij} x_j - b_j \text{ pour } i \in I$$

et supposons que les actions des processus sur les variables de synchronisation sont de la forme (*) :

$$x_k := x_k + 1 \text{ ou } x_k := x_k - 1 \qquad k \in \{1, 2, \ldots, n\}$$

Ces actions impliquent sur le système $\{s_i\}$ $i \in I$ des variations du type :

$$s_i := s_i + a_{ik} \text{ ou } s_i := s_i - a_{ik} \qquad i \in I, k \in \{1, 2, \ldots, n$$

dans la région de synchronisation définie par les contraintes linéaires

$$C = \{s_i \geq 0, i \in I\}$$

Ainsi, si par exemple une variation $s_i := s_i + a_{ik}$ est autorisée, elle est équivalente à

$$a_{ik} \text{ fois } s_i := s_i + 1, \text{ si } a_{ik} \geq 0$$

$$\text{ou } |a_{ik}| \text{fois } s_i := s_i - 1, \text{ si } a_{ik} < 0$$

Si le problème de synchronisation est résolu à l'aide des primitives de Dijkstra, l'invariant $\{s_i \geq 0, i \in I\}$ peut alors définir un nouvel ensemble de variables de synchronisation (telles que les sémaphores par exemple) et donc le problème de synchronisation des processus affectant les variables x_j peut être résolu facilement à l'aide de primitives P et V sur ces sémaphores.

Nous remarquons ici le caractère hiérarchique des variables de synchronisation en fonction des outils de synchronisation employés pour la mise en oeuvre de la solution.

(*) La forme des actions n'est pas restrictive car des actions plus complexes alourdissent l'exposé sans modifier la méthode.

De plus, le sous-ensemble des contraintes affecté par une action du type décrit ci-dessus sur une variable x_k peut être explicitement déterminé et indiquera l'ensemble des sémaphores affectés par cette action. Les coefficients des variables de synchronisation x_j dans les contraintes indiqueront en outre automatiquement le nombre d'opérations P ou V à effectuer sur chacun des sémaphores affectés. Le choix particulier des primitives de Dijkstra pour cette traduction, introduit des problèmes de 'scheduling' particuliers que nous supposerons implicitement résolus dans tous les exemples du chapitre. Par exemple, si plusieurs opérations P sont exécutées sur le même sémaphore par un processus, d'autres processus ne doivent pas modifier ce sémaphore avant que le premier processus n'ait terminé ses actions ou ne soit parvenu au blocage (cette remarque justifie en outre l'introduction des primitives généralisées (VvL 72)).

1.2. Exemple

Reprenons le problème de stocks que nous avons formulé au chapitre précédent. Les contraintes de synchronisation étaient :

$$s_1 = -2x_1 + 3x_2 \geq 0$$

$$s_2 = 2x_1 - x_2 \geq 0$$

$$s_3 = N_1 - x_1 \geq 0$$

$$s_4 = N_2 - x_2 \geq 0$$

Les diverses actions de production et de consommation se traduisent ainsi :

$x_1 := x_1 + 1$ se traduit par :

[P(s_3) ; P(s_1) ; P(s_1) ; {production d'une unité du premier produit} V(s_2) ; V(s_2) ;]

$x_1 := x_1 - 1$ se traduit par :

[P(s_2) ; P(s_2) ; {consommation d'une unité du premier produit} V(s_1) ; V(s_1) ; V(s_3) ;]

$x_2 := x_2 + 1$ se traduit par :

[P(s_4) ; P(s_2) ; {production d'une unité du second produit} V(s_1) ; V(s_1) ; V(s_1) ;]

$x_2 := x_2 - 1$ se traduit par :

[P(s_1) ; P(s_1) ; P(s_1) ; {consommation d'une unité du second produit} V(s_2) ; V(s_4) ;]

Remarque :

Il est intéressant, à notre avis, de remarquer dès maintenant et sur cet exemple, que dans tout problème de synchronisation, une action effectuée par un des processus concurrents sera *parenthésée* par une "pré-opération" et une "post-opération" mettant en cause les variables de synchronisation et donc la région de synchronisation.

La synchronisation des processus ne met pas en cause le déroulement même des actions des processus (sauf en ce qui concerne l'exclusion), mais surtout l'autorisation pour effectuer une certaine action (pré-opération) et l'influence qu'une action aura sur les actions futures (post-opération).

C'est pour cette raison que, dans l'exemple ci-dessus, les primitives P et V sont groupées respectivement avant l'action effective et après le déroulement de l'action.

Dans le cas des moniteurs, cette remarque est encore plus évidente car toute action sera parenthésée par deux procédures du moniteur qui mettent en oeuvre les contraintes de synchronisation.

2. CONTRAINTES DE SYNCHRONISATION POUR QUELQUES PROBLEMES "CLASSIQUES" DE SYNCHRONISATION

Le formalisme de synchronisation par contraintes linéaires couvre, comme nous l'avons souligné plus haut, la plupart des problèmes connus de synchronisation. Analysons quelques exemples pour souligner l'obtention systématique des solutions à l'aide de P et V à partir de contraintes exprimées convenablement.

2.1. Exclusion mutuelle

Un ensemble de processus exécutent des actions qui doivent être mutuellement exclusives, c'est-à-dire qu'il y a au plus une action en cours.

En notant par :

x_1 = nombre d'actions commencées

x_2 = nombre d'actions terminées,

les contraintes de synchronisation entre les processus s'écrivent :

$$0 \leq x_1 - x_2 \leq 1$$

En supposant qu'une action est d'abord commencée et ensuite terminée (ce qui rend implicite la contrainte $x_1 - x_2 \geq 0$) et si l'on note par

$$s = x_2 - x_1 + 1$$

alors l'exécution d'une action par un processus se traduira par :

$$[x_1 := x_1 + 1 \; ; \; \{action\} \; ; \; x_2 := x_2 + 1 \; ;]$$

A l'aide des primitives de Dijkstra, la contrainte d'exclusion mutuelle, $s \geq 0$, s'exprimera sur le sémaphore s par (la valeur initiale du sémaphore peut se déduire aussi de la forme de la contrainte pour $x_1 = x_2 = 0$)

P(s) ; car $x_1 := x_1 + 1$ implique $s := s - 1$

{action}

V(s) ; car $x_2 := x_2 + 1$ implique $s := s + 1$

qui est l'écriture bien connue donnée par Dijkstra (Di 68-a).

2.2. Producteurs - Consommateurs

Il s'agit de synchroniser les actions de deux classes de processus, l'une formée de "producteurs" et l'autre de "consommateurs", suivant certaines contraintes.

Notons par :

x_1 = nombre d'actions de production commencées

x_2 = nombre d'actions de production terminées

x_3 = nombre d'actions de consommation commencées

x_4 = nombre d'actions de consommation terminées

(on produit et on consomme une unité à la fois).

2.2.1. Buffer illimité

Les processus concurrents des deux classes échangent de l'information à travers un buffer dont la capacité est supposée ici infinie Les "producteurs" insèrent une unité d'information dans la prochaine place libre du buffer et les "consommateurs" enlèvent une unité de la première place occupée, par exemple.

Les contraintes de synchronisation sont dues uniquement au fait qu'on ne peut consommer plus que ce qu'on produit, donc :

$$x_3 \leq x_2$$

Si on introduit le sémaphore :

$$s_1 = x_2 - x_3 \geq 0$$

et si l'on tient compte de l'exclusion mutuelle entre les actions, exclusio exprimée au § 2.1 par :

$$s_2 = 1 - x_1 + x_2 - x_3 + x_4 \geq 0$$

la synchronisation de ces processus s'écrit, à l'aide des primitives de Dijkstra, comme suit :

$$\begin{cases} x_1 := x_1 + 1 \ ; \\ [\text{action de production}] \\ x_2 := x_2 + 1 \ ; \end{cases} \Rightarrow \begin{cases} P(s_2) \ ; \\ [\text{action de production}] \\ V(s_2) \ ; \\ V(s_1) \end{cases}$$

$$\begin{cases} x_3 := x_3 + 1 \ ; \\ [\text{action de consommation}] \\ x_4 := x_4 + 1 \ ; \end{cases} \Rightarrow \begin{cases} P(s_1) \ ; \\ P(s_2) \ ; \\ [\text{action de consommation}] \\ V(s_2) \ ; \end{cases}$$

qui est l'écriture bien connue donnée par Habermann (HAB 72).

Remarquons que le fait que le buffer soit illimité implique que la fin de la consommation et le début de la production, n'aient une influence sur les contraintes de synchronisation qu'en ce qui concerne l'exclusion mutuelle.

Nous analyserons les relations dans la région de synchronisation entre les actions des processus et l'ensemble des contraintes au chapitre suivant.

Les valeurs initiales des sémaphores se déduisent systématiquement de l'expression des contraintes, car initialement toutes les variables de synchronisation x_i (i = 1,2,3,4) sont nulles ; donc $s_1 = 0$ et $s_2 = 1$.

2.2.2. Buffer N-limité (N ≥ 1)

Dans ce cas, les "producteurs" et le "consommateur" doivent être synchronisés de façon à ce qu'un producteur n'essaie jamais d'insérer une unité produite dans un buffer plein et qu'un consommateur n'essaie jamais de consommer dans un buffer vide. Donc, la contrainte suivante est à considérer : on ne peut produire plus de N unités (N est la capacité de stockage du buffer) de plus que le nombre d'unités consommées. De plus, l'accès au buffer reste en exclusion mutuelle.

Il en résulte qu'aux contraintes exprimées plus haut dans le cas du buffer illimité, on ajoute la contrainte suivante :

$$s_3 = x_4 - x_1 + N \geq 0$$

et donc une action de production (parenthésée par $x_1 := x_1 + 1$ et $x_2 := x_2 +$ sera traduite par :

$P(s_3)$;
$P(s_2)$;
{production} ;
$V(s_2)$;
$V(s_1)$;

et une action de consommation (parenthésée par $x_3 := x_3 + 1$ et $x_4 := x_4 + 1$) sera traduite par :

$P(s_1)$;
$P(s_2)$;
{consommation} ;
$V(s_2)$;
$V(s_3)$;

Comme initialement $x_i = 0$ $(i = 1,2,3,4)$, le sémaphore s_3 est initialisé à N.

Remarque :

Nous avons donné ici les expressions des solutions à l'aide de primitives de Dijkstra, car dans ce cas il ressort clairement que l'influenc de chaque action sur les contraintes et l'application des primitives aux dif férents sémaphores, est indiquée explicitement par la forme des contraintes. De plus, les processus dont les actions ne vérifieraient pas les contraintes seront en attente sur ces contraintes, ce qui ne ressort pas d'une solution obtenue à l'aide des expressions de chemin, par exemple.

Remarque :

Nous observons également que si des contraintes supplémentaires venaient s'ajouter à un ensemble de contraintes donné, alors des sémaphores correspondants exprimeront ces contraintes en fonction des variables de synchronisation et la traduction, à l'aide des primitives choisies, en tiendra compte en fonction des variables de synchronisation qui interviennent dans ces nouvelles contraintes.

2.3. Un modèle de synchronisation

Une nature très particulière possible pour les contraintes ressort de l'exemple suivant où des fonctions non nécessairement linéaires peuvent apparaître dans l'expression des contraintes. Nous donnons cet exemple pour montrer que l'expression des solutions à l'aide de P et V reste systématique. Considérons m classes de processus qui doivent s'exécuter en respectant les conditions suivantes :

- exclusion mutuelle entre les m classes de processus,
- le nombre d'actions parallèles autorisées pour les processus d'une même classe, est limité à $n_i > 0$.

Notons par x_i le nombre d'actions (sections critiques) en exécution dans la classe i. Alors, les deux conditions ci-dessus s'expriment par les contraintes globales :

$$0 \leq \sum_{i=1}^{m} \frac{x_i + n_i \text{ sign } (x_i)}{n_i} \leq 2$$

où la fonction sign est définie comme au § II.3.2. En effet, on vérifie qu'un terme de la somme est non nul au plus (puisque > 1) et que lorsque $x_i > n_i$ ce terme est > 2. Remarquons que, en supposant les actions des processus d'abord commencées puis terminées, alors la contrainte de positivité est toujours vérifiée. Cette expression en une seule contrainte algébrique est "acrobatique" mais nous l'exprimons ainsi car elle indique qu'un seul sémaphore sera nécessaire pour sa traduction et elle indique également la valeur initiale à prendre pour ce sémaphore.

En appelant

$$N = n_1, n_2 \ldots n_m$$

$$N_i = \frac{N}{n_i} \qquad i = 1, 2, \ldots, m$$

la contrainte de synchronisation devient :

$$s = 2N - \sum_{i=1}^{m} [N \text{ sign } (x_i) + N_i x_i] \geq 0$$

qui est l'invariant de ce problème de synchronisation.

D'après cette représentation, il s'ensuit qu'une action faite par un processus sur une des variables de synchronisation x_i est répercutée via la fonction sign à une variation correspondante du sémaphore choisi pour l'expression de ce problème à l'aide des primitives P et V de Dijkstra. De plus, cette variation est explicitement indiquée par la contrainte.

Par exemple, considérons comme plus haut (cf. § III.1.1) que les processus doivent exécuter des actions du type suivant sur les variables de synchronisation :

$x_i := x_i + 1$;
{action} $\qquad i = 1, 2, \ldots, m$
$x_i := x_i - 1$;

Mais si x_i devient $x_i + 1$ dans l'expression de s, nous constatons que s devient $s - N - N_i$ ou $s - N_i$ suivant la variation de sign (x_i). Une remarque analogue pour la décrémentation de x_i nous conduit à la solution suivante :

pour tout i = 1, 2, ..., m

P (mutex) ;
$x_i := x_i + 1$;
if sign $(x_i - 1)$ < sign (x_i) then $M_i := N + N_i$
else $M_i := N_i$;
V (mutex) ;
{M_i fois P (s)} ;
{action}
P (mutex) ;
$x_i := x_i - 1$;
if sign $(x_i + 1)$ > sign (x_i) then $M_i := N + N_i$
else $M_i := N_i$;
V (mutex) ;
{M_i fois V (s)} ;

Ici, le sémaphore mutex est un sémaphore supplémentaire d'exclusion mutueelle destiné à assurer la cohérence entre la variation de x_i affectant la fonction sign et éventuellement affectant le sémaphore s associé à la contrainte de synchronisation. La valeur initiale du sémaphore s est également indiquée par la contrainte globale pour $x_i = 0$, $i = 1,2,\ldots,m$, c'est-à-dire initialement $s = 2N$. Les M_i sont de variables locales aux processus et ne servent qu'à répercuter en dehors de l section critique le nombre d'actions P à faire. Remarquons que dans ce cas, la fonction sign ne peut prendre que les valeurs 0 ou 1 et donc :

$$\text{sign } (x_i - 1) < \text{sign } (x_i) \text{ est équivalent à } x_i = 1$$

$$\text{et sign } (x_i + 1) > \text{sign } (x_i) \text{ est équivalent à } x_i = 0$$

2.3.1. Cas particulier

Supposons que parmi les m classes de processus il en existe k (avec $k < m$) numérotées de m-k+1 à m pour lesquelles le nombre d'actions parallèles autorisées est non limité. La contrainte de synchronisation devi dans ce cas :

$$s = 2N - \sum_{i=1}^{m} [N \text{ sign } (x_i) + N_i x_i] - 2N \sum_{i=m-k+1}^{m} \text{sign } (x_i) \geq 0$$

où N et N_i sont définis comme plus haut, mais le produit intervenant dans leurs définitions est limité aux premières m-k classes.

Pour ces m-k classes, l'interprétation de

```
x_i := x_i + 1 ;
{action}                    i = 1, 2, ..., m-k
x_i := x_i - 1 ;
```

restera la même.

Pour les autres k classes on aura :

pour tout i = m-k+1, m-k+2, ..., m

P (mutex) ;
$x_i := x_i + 1$;
if sign (x_i - 1) < sign (x_i) then $M_i := 2N$;
V (mutex) ;
{M_i fois P (s)} ;
{action}
P (mutex) ;
$x_i := x_i - 1$;
if sign (x_i + 1) > sign (x_i) then $M_i := 2N$;
V (mutex) ;
{M_i fois V (s)} ;

Il est intéressant d'analyser le cas où k = m pour lequel la contrainte de synchronisation est :

$$s = 1 - \sum_{i=1}^{m} \text{sign } (x_i) \geq 0$$

Les actions décrites ci-dessus se traduisent donc (avec s initialisé convenablement) par :

pour i = 1, 2, ..., m

P (mutex) ;
$x_i := x_i + 1$;
if sign (x_i - 1) < sign (x_i) then $M_i := 1$;
V (mutex) ;
{M_i fois P (s)} ;
{action}
P (mutex) ;
$x_i := x_i - 1$;
if sign (x_i + 1) > sign (x_i) then $M_i := 1$;
V (mutex) ;
{M_i fois V (s)} ;

Mais dans l'écriture ci-dessus, tout ce qui précède {action} et tout ce qui suit {action}, correspond respectivement aux primitives PP et VV de Campbell et Habermann (CH 74) introduites pour la traduction des expressions de cheminà l'aide des primitives P et V. Cela est tout à fait naturel en remarquant que PP et VV interviennent dans la traduction de l'opérateur * de répétition dans les expressions de chemin.

2.3.2. Lecteurs et écrivains

Revenons au premier problème des Readers & Writers (CHP 71) c'est-à-dire sans priorités. Ce problème est un cas particulier du modèle ci-dessus. En effet, si on considère deux classes en exclusion mutuelle, l'une avec exclusion mutuelle entre les écrivains et l'autre avec des lecteurs opérant simultanément, la solution à ce problème de synchronisation est évidente avec $N = N_1 = n_1 = 1$, d'après le § 2.3.1. Dans ce cas, la contrainte s'écrit (en prenant e et 1 pour x_1 et x_2 respectivement) :

$$1 - \text{sign}(e) - e - 2\,\text{sign}(1) \geq 0$$

ce qui est équivalent à la contrainte $\text{sign}(1) + e \leq 1$ donnée au § II.3.2.

Remarque :

Deux points sont à souligner dans les solutions donnés ci-dessus :

a) des P - V conditionnels sont nécessaires pour leur traducti

b) le nombre de processus dans chaque classe est indifférent.

3. EXPRESSION DES RÉSEAUX DE PETRI PAR CONTRAINTES

Nous avons donné au chapitre I (§ 3.5) un rappel des définitions des réseaux de Petri et des réseaux de synchronisation, et nous avons vu qu'un réseau de Petri modélise aussi bien un ensemble de processus que les règles de synchronisation les concernant. Dans ce paragraphe, nous insistons sur l'expression des réseaux de Petri représentant les règles de synchronisation dans un ensemble de processus concurrents, par un ensemble de contraintes linéaires.

A partir d'un graphe $G = (X, U)$, on peut définir sa matrice d'incidence sommets - arcs, $E = (e_{ij})$, de la manière suivante :

$$e_{ij} = \begin{cases} -1 \text{ si l'arc } U_i \text{ est sortant de } x_j \\ +1 \text{ si l'arc } U_i \text{ est incident à } x_j \\ 0 \text{ sinon} \end{cases}$$

Cette matrice a exactement deux éléments non nuls par ligne, un +1 et un -1, car chaque arc a exactement un sommet initial et un sommet final. Il est facile de démontrer également (par récurrence, par exemple), que cette matrice est *totalement uni-modulaire*, c'est-à-dire que le déterminant de toute sous-matrice carrée extraite de E vaut +1, -1 ou 0.

Cette propriété implique que tout ensemble de la forme

$\{Y \geq 0 \mid EY \leq b \; ; \; b \text{ vecteur entier}\}$

est un polyèdre qui possède uniquement des "extrêmités" de coordonnées entières.

Pour un réseau de Petri généralisé, la matrice d'incidence sera définie par :

$$e_{pt} = \begin{cases} -\ \text{capa}\ (t, p) \text{ si } (t, p) \in |\!\rightarrow o \\ +\ \text{capa}\ (p, t) \text{ si } (p, t) \in o\!\rightarrow| \\ 0 \text{ sinon} \end{cases} \qquad \forall\ p \in P,\ \forall\ t \in T$$

A chaque place d'un réseau de Petri, nous associons également un entier $b(p)$ représentant un *potentiel initial* pour le nombre $\mu(p)$ de marques initialement dans cette place.

Proposition :

Un réseau de synchronisation (cf. § I.3.5) associé à un réseau de Petri PB = $(P, T, o\!\rightarrow|\ ,\ |\!\rightarrow o)$ est représenté par le système de contraintes linéaires :

$$EY \leq b$$

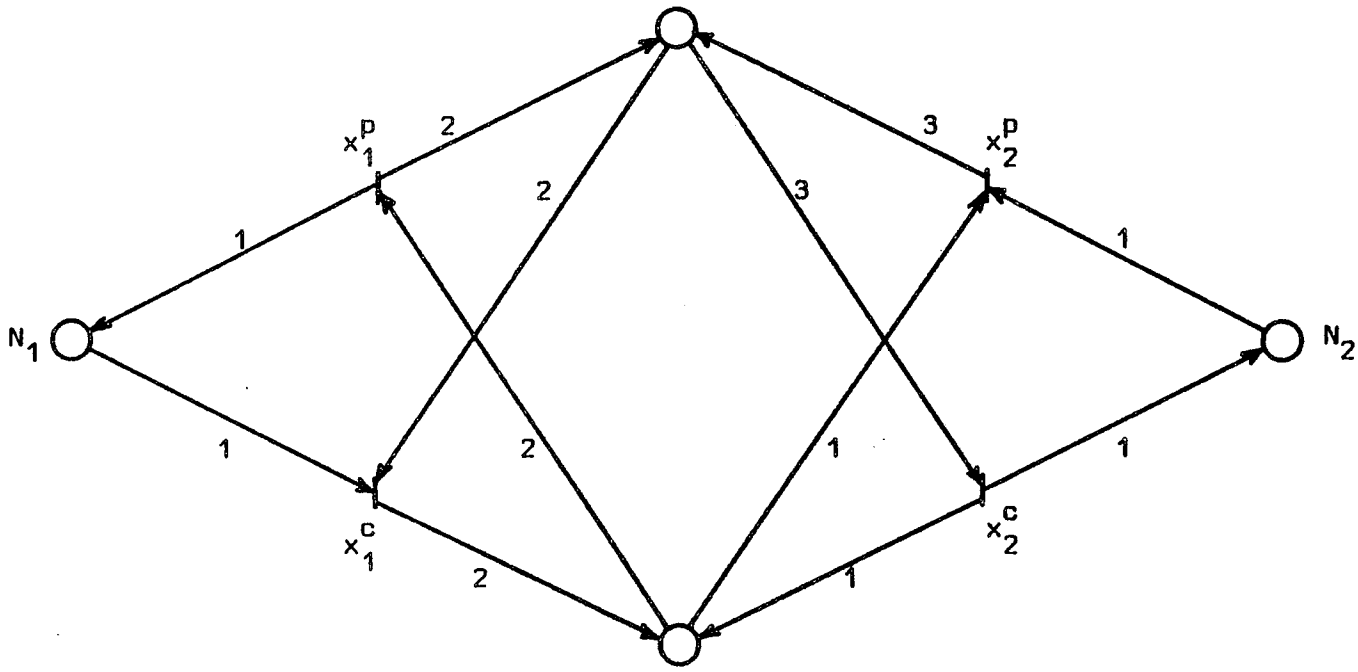
où E est sa matrice d'incidence définie comme ci-dessus, Y est un vecteur associé aux transitions et b est le vecteur potentiel initial associé à l'ensemble P des places.

La démonstration est immédiate d'après les définitions données Le système de contraintes représentant un réseau de Petri définit une région de synchronisation associée au réseau.

Remarque :

La région ainsi définie *ne possède pas* uniquement des sommets de coordonnées entières, car la matrice d'incidence d'un réseau de Petri généralisé *n'est pas totalement uni-modulaire*. Cette matrice continue néanmoins d'avoir seulement deux éléments non nuls sur chaque ligne.

Exemple : considérons le réseau suivant :



dont la matrice d'incidence est :

$$E = \begin{bmatrix} 2 & -2 & -3 & 3 \\ -2 & 2 & 1 & -1 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix}$$

Si le vecteur Y associé aux transitions est $Y = (x_1^p, x_1^c, x_2^p, x_2^c)^t$ et $b = (0,0,N_1,N_2)$, alors la région de synchronisation associée à ce réseau de Petri n'est autre que celle du problème de stock pour deux producteurs - deux consommateurs défini aux § II.3.2 et III.2.2.3.

Remarque :

Un réseau de Petri "pur" (cf. § II.3.5) ne peut représenter la région de synchronisation à l'aide des variables x_1 et x_2 sans les "dédoubler" en x_1^p, x_1^c, x_2^p et x_2^c.

Pour chaque marquage μ des places d'un réseau de Petri, l'équation

$$E\ Y = \mu$$

a comme solution un vecteur associé aux transitions dont les coordonnées représentent les coordonnées du point courant dans la région de synchronisation associée au réseau.

D'après les définitions ci-dessus, la relation $\underset{\tau}{=>}$ définie au § I.3.5, exprimant la transition entre deux marquages μ et μ' d'un réseau donné par la "mise à feu simultanée" d'un sous-ensemble τ de transitions, nous pouvons énoncer la proposition suivante :

Proposition :

Deux marquages μ et μ' vérifient la relation $\underset{\tau}{=>}$ si et seulement si :

$$\mu \geq \frac{1}{2} (|E| + E) \cdot Y^{\tau}$$

et alors :

$$\mu' = \mu - E \cdot Y^{\tau}$$

où $|E|$ est la matrice des valeurs absolues des éléments de E et Y^{τ} est le vecteur caractéristique de l'ensemble τ des transitions.

Cette proposition exprime donc la relation de transition entre un point courant et son successeur dans la région de synchronisation correspondant au réseau de Petri.

Un réseau de synchronisation (§ I.3.5) représente donc tous les points obtenus dans la région de synchronisation à partir d'un point initial correspondant à un marquage initial du réseau de Petri associé.

Note :

Nous venons de démontrer qu'un réseau de Petri peut toujours s'exprimer par un ensemble de contraintes linéaires de synchronisation. Mais comme nous l'avons indiqué au début de ce chapitre, notre formalisme de synchronisation n'impose aucune restriction quant à la linéarité des contraintes, c'est-à-dire que les fonctions intervenant dans l'expression des contraintes sont considérées quelconques. La question reste posée de savoir si, inversement, tout ensemble de contraintes de synchronisation peut toujours être exprimé par un réseau de Petri ?

CHAPITRE IV

IMPACT SUR L'ÉTUDE DU DEADLOCK

"pour aller où tu ne sais pas,
va par où tu ne sais pas"

Saint Jean de la Croix

1. RELATIONS DANS LA RÉGION DE SYNCHRONISATION

Nous avons vu au § 2 du chapitre II que les variables de synchronisation d'un système de processus concurrents peuvent être considérées comme les coordonnées d'un point courant X dans l'espace défini par le produit cartésien des espaces des valeurs de chacune des variables de synchronisation x_j, $j = 1,2,\ldots,n$.

L'état d'un système de processus est ainsi déterminé par la position du point X dans la région de synchronisation et l'état d'avancement des processus du système.

Rappelons que les variables de synchronisation doivent aussi bien appartenir aux contraintes de synchronisation qu'aux actions des processus. Par exemple, ces variables peuvent être pratiquement :

a) associées aux *ressources* : nombre de processus en attente pour une certaine ressource, nombre d'unités disponibles pour une ressource, etc ..

b) associées aux *opérations* ("compteurs" associés aux procédures, (RV 77) : nombre de processus exécutant une action atomique donnée à l'instant considéré, nombre de processus en attente d'exécution d'une certaine action atomique, nombre de processus ayant exécuté une certaine action atomique depuis l'instant initial. Dans notre formulation, ces compteurs seront donc attachés aux actions atomiques des processus.

c) associées aux *contraintes* de synchronisation : nombre de processus en attente sur une contrainte de synchronisation pour une action atomique donnée ou pour des actions différentes, nombre d'actions atomiques exécutées après la vérification d'une certaine contrainte de synchronisation, nombre de processus actifs après vérification d'une certaine contrainte de synchronisation, etc ..

d) associées à la *région* de synchronisation : nombre de processus en attente sur des contraintes définissant la région de synchronisation, nombre de processus exécutant une certaine action atomique dans la région de synchronisation, nombre de processus exécutant des actions atomiques vérifiant la région de synchronisation, nombre de processys ayant terminé des actions atomiques vérifiant la région de synchronisation, etc ..

Pour résoudre un problème spécifié par un ensemble de contraintes de synchronisation, il faudra traduire les contraintes et les variables de synchronisation en fonction des outils disponibles dans le langage d'implémentation des actions des processus.

Considérons une région de synchronisation définie par un ensemble de contraintes C_i, $i \in I$, et soit A_k une action atomique sur les variables de synchronisation. L'effet que peut avoir l'exécution d'une action A_k sur l'ensemble des contraintes de synchronisation nous intéresse de deux manières :

. en supposant que le point courant X a été déplacé à l'extérieur de la région de synchronisation, l'exécution de l'action ne pourrait le ramener à l'intérieur de la région ;

. l'exécution d'une action A_k ne peut jamais déplacer le point courant à l'extérieur de la région de synchronisation.

Ainsi définit-on deux relations dans la région de synchronisation :

. Une action atomique A_k et une contrainte de synchronisation C_i vérifient la *relation de déblocage* $\mathcal{D}$ si l'exécution de l'action A_k pourra rendre la contrainte C_i vérifiée alors qu'elle ne l'était pas avant cette exécution ;

. de même, on définit une *relation de blocage* $\mathcal{B}$ entre une action A_k et une contrainte C_i si l'exécution de l'action A_k rend la contrainte C_i non vérifiée alors qu'elle était vérifiée avant l'exécution de A_k

Pour donner un exemple de ces relations, prenons l'exemple du chapitre précédent où les contraintes étaient linéaires et les actions des incrémentations ou décrémentations de variables.

Ce type d'actions sur des variables appartenant à des contraintes linéaires, couvrent les actions sur des variables de synchronisation faites par les primitives de synchronisation classiques.

Supposons donc que les processus peuvent agir sur les variables de synchronisation par des actions du type :

$$x_k := x_k + y_k \; ; \; k \in \{1, 2, \ldots, n\}$$

On suppose également y_k entier.

Remarquons que ces actions peuvent être assimilées à une séquence d'actions du type $s_k := s_k \pm 1$ que nous avons analysé auparavant, mais peuvent aussi être indivisibles dans le cas de certaines contraintes sur les variables de synchronisation. Ainsi, par exemple, l'action :

$$x_k := x_k + 2$$

avec la contrainte x_k impaire, ne peut-elle pas être assimilée à une séquence d'actions du type $x_k := x_k + 1$. Les primitives P et V définissent des déplacements élémentaires du point courant dans la région de synchronisation (suivant les "axes" s_k). Ils imposent donc des déplacements souvent différents de ceux réellement voulus, ce qui explique que dans certains cas le déplacement voulu satisfait les contraintes de synchronisation, mais son expression à l'aide de P et V ne le satisfait pas.

Notons comme auparavant par

$$s_i = \sum_{j=1}^{n} a_{ij} x_j = b_j \qquad i \in I$$

la valeur numérique d'une contrainte de synchronisation C_i : $\{s_i \geq 0, \; i \in I\}$.

L'exécution d'une action

$$A_k := \{x_k := x_k + y_k\}, \; k \in \{1, 2, \ldots, n\}$$

modifie les valeurs numériques des contraintes de la façon suivante :

$$s_i' = s_i + a_{ik} y_k \qquad i \in I, \; k \in \{1, 2, \ldots, n\}.$$

Nous allons prendre en considération l'effet que peut avoir l'exécution d'une action du type décrit ci-dessus sur l'ensemble des contraintes de synchronisation.

Remarquons qu'à un moment donné, un ensemble de processus est en attente d'exécution, car les actions qu'ils tentent d'exécuter déplacent le point X en dehors de la région de synchronisation en violant un sous-ensemble de contraintes. Supposons qu'il existe au moins une action A_k qui puisse être exécutée par un processus.

Proposition :

Un processus viole un sous-ensemble $I_1 \subset I$ de contraintes de synchronisation en exécutant une action atomique A_k si

$$a_{ik}\, y_k < 0, \quad \forall\, i \in I_1 \qquad (*)$$

La démonstration est évidente et basée sur la relation

$$s_i' = s_i + a_{ik}\, y_k$$

donnant la valeur numérique des contraintes. En effet, la condition nécessaire (*) découle du fait que $s_i \geq 0$, $i \in I$, avant l'exécution de l'action A_k par le processus. Un tel processus sera donc bloqué après A_k sur les contraintes de l'ensemble I_1.

Remarque :

De la même façon, pour qu'un processus puisse rendre vérifiées les contraintes d'un sous-ensemble $I_2 \subset I$ en exécutant une action A_k, il faut que :

$$a_{ik}\, y_k > 0, \quad \forall\, i \in I_2 \qquad (**)$$

Les conditions (*) et (**) sont nécessaires (elles ne sont pas suffisantes !) pour que la valeur d'une contrainte soit changée ; elles déterminent les deux relations $\mathcal{B}$ et $\mathcal{D}$ définies ci-dessus. La relation $\mathcal{B}$ (respectivement $\mathcal{D}$) contiendra tous les couples (A_k, C_i) pour lesquels la condition (*) (respectivement (**)) est vérifiée.

Remarque :

Si $(A_k, C_i) \in$ alors on peut avoir deux situations :

i) $a_{ik}\ y_k = 0$

cela correspond au cas où l'action du processus ne fait pas intervenir de variables appartenant à la contrainte C_i (cf. § la distinction faite entre les variables appartenant aux contraintes et les variables appartenant aux actions).

ii) $a_{ik}\ y_k > 0$

dans ce cas, l'action des processus ne fait qu'accroître la valeur numérique de la contrainte en question.

Les actions définies ci-dessus définissent dans l'ensemble des contraintes des dépendances dues aux actions des processus sur les variables de synchronisation. Nous remarquons que la forme particulière des contraintes linéaires et des actions ne restreint pas la généralité, car des relations analogues peuvent être définies dans un cas tout à fait général où les contraintes sont quelconques et les actions atomiques définissent des transitions quelconques entre les différentes positions du point courant X dans la région de synchronisation.

A l'aide des relations $\mathfrak{B}$ et $\mathfrak{D}$ définies ci-dessus, nous nous proposons maintenant de décrire une méthode pour le contrôle efficace de la synchronisation dans un ensemble de processus. Comme notre propos est uniquement la synchronisation de ces processus concurrents, on suppose connues

. les variables de synchronisation $x_1, x_2, \ldots, x_n$;

. les contraintes de synchronisation C_i, $i \in I$, faisant intervenir ces variables et définissant la région de synchronisation. (Nous ne nous intéressons pas ici à la façon dont ces contraintes de synchronisation ont été déduites à partir de la spécification du problème. Il est toutefois intéressant de noter que ces contraintes peuvent être déduites par des règle de construction des programmes concurrents à partir des invariants attachés à ces programmes) ;

. les actions atomiques des processus définies comme des modules séquentiels décrivant l'avance des processus. Une action atomique sera donc exécutée comme une région critique dans le sens donné par Hoare.

Un processus doit attendre avant l'exécution d'une action atomique qu'une contrainte C_i, $i \in I$, soit satisfaite. De même, après une telle action atomique, un processus doit pouvoir transmettre sa "contribution" à l'avance des autres processus.

Ainsi, *chaque action sera-t-elle parenthésée par une préface et une postface* ; la préface consiste essentiellement dans la vérification des contraintes de synchronisation que l'action doit vérifier avant son exécution ; la postface résume la nouvelle position du point courant dans la région de synchronisation et donc l'influence de l'action exécutée sur les actions futures des processus.

Remarquons que tous les outils qui demandent de morceler la préface, introduisent des risques de deadlock supplémentaires.

Ainsi, chaque action atomique aura-t-elle la forme générale :

$x_1, x_2, x_3, \ldots, x_n$: ? begin act ;

A_k ;

end act ! ;

où ? begin act et end act ! sont respectivement la préface et la postface.

Tout accès aux variables de synchronisation x_j, $j = 1, 2, \ldots, n$ doit se faire en exclusion mutuelle pendant l'exécution d'une action atomique, c'est-à-dire entre ? et !.

L'action A_k sera exécutée uniquement si la vérification des contraintes de synchronisation dans la préface s'avère vraie. Pour plus de clarté, détaillons la préface par :

? begin act ::= if $\neg C_{ik}$

then (mettre en attente le processus exécutant dans une file F_{ik} attachée à la condition C_{ik} (une ou plusieurs contraintes sur lesquelles porte l'action A_k) et lacher l'exclusion mutuelle sur les variables de synchronisation)

Remarquons que l'exclusion mutuelle peut donc être relachée à la fin de la préface et donc l'exécution de l'action atomique en cours est abandonnée momentanément par le processus qui sera mis en attente sur la préface de cette action. Notons aussi que les risques de deadlock sont ainsi diminués.

Initialement, les processus tentant l'exécution d'actions sur les variables de synchronisation seront mis en attente dans une file d'attente F d'où ils seront transférés dans les files F_{ik} quand la préface de leurs actions respectives le demandera.

Quand une action A_k a été exécutée par un processus, l'exclusi mutuelle est gardée jusqu'après la postface de l'action.

Dans une liste L_k seront placés tous les processus en attente sur des contraintes C_i telles que $(A_k, C_i) \in \mathcal{D}$. Ce sont des processus qui pourront entrer dans les préfaces de leurs actions respectives après la fin la postface de l'action en cours et de toutes les postfaces des actions commencées avant (élimination en cascade des postfaces).

Si la liste L_k n'est pas vide, alors les processus lui appartenant seront activés suivant la procédure suivante :

si sur les contraintes C_i telles que $(A_k, C_i) \in \mathcal{D}$ il existe des processus en attente d'exécution d'actions A_i telles que $(A_i, C_{ik}) \in \mathcal{B}$ alors un seul de ces processus de la liste L_k sera activé, sinon ils seront tous activés.

Il est évident que cette construction peut remédier à l'implémentation des moniteurs faisant intervenir le réveil "potentiel" de plusieur processus qui vérifient une certaine contrainte, mais l'entrée du moniteur n'est accessible qu'à un processus à la fois (cf. § I.2.3.). Pour une implémentation dans le cas des moniteurs, le lecteur se reportera à (SCHM 7E

Nous soulignons enfin le caractère performant d'un mécanisme de contrôle de la synchronisation basé sur les relations ci-dessus, en ce qui concerne l'augmentation du parallélisme

2. MODÈLE REPRÉSENTATIF DU DEADLOCK

A partir des relations $\mathcal{S}$ et $\mathcal{D}$ définies au paragraphe précédent, nous allons développer un modèle pour la détection et la caractérisation du deadlock dans un système formé d'un ensemble *fini* de processus concurrents. A chaque état observable du système, nous allons associer un *graphe représentatif* $G_{\mathcal{E}}$ défini de la façon suivante :

(i) $G_{\mathcal{E}}$ est un graphe biparti (l'ensemble des sommets étant l'union des deux ensembles disjoints Γ et Λ) ;

(ii) chaque sommet de Γ correspond à un processus du système (l'ensemble des processus étant supposé fini à un instant donné) et chaque sommet de Λ correspond à un ensemble d'actions atomiques des processus ;

(iii) les arcs orientés de $G_{\mathcal{E}}$ sont définis par : si à l'état considéré l'action A_k d'un processus P_k vérifie la relation $\mathcal{S}$ avec une contrainte de synchronisation G_i et si le déblocage de ce processus peut être assuré par l'exécution d'un ensemble λ d'actions appartenant aux processus $P_{j1}, P_{j2}, \ldots, P_{jk}$ (ensemble que l'on peut obtenir à partir de la fermeture transitive de la relation de transition du point courant et de la relation $\mathcal{D}$), alors on définit un arc orienté de chacun des sommets ($\in \Gamma$) qui correspondent aux processus $P_{j1}, \ldots, P_{jk}$ vers le sommet ($\in \Lambda$) qui correspond à l'ensemble d'actions λ et on définit également un arc orienté de λ vers P_k.

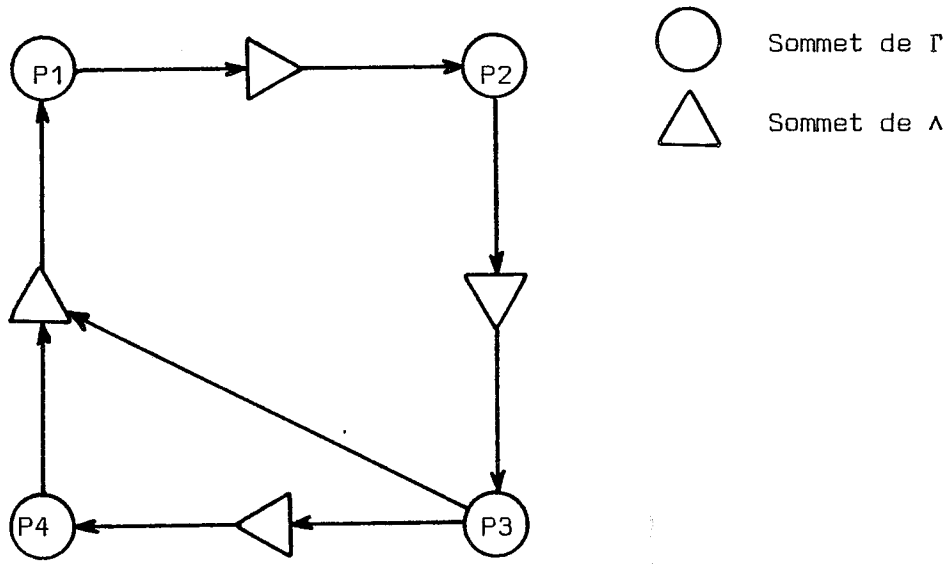
Remarque :

$k \notin \{j_1, \ldots, j_k\}$ car $(A_k, C_i) \in \mathcal{S} \Rightarrow (A_k, C_i) \notin \mathcal{D}$.

Remarque :

Un ensemble λ d'actions n'est évidemment pas unique pour le déblocage d'un processus P_k et donc le sommet P_k aura un "degré intérieur" (nombre d'arcs orientés vers P_k) égal au nombre de tels sous-ensembles (non nécessairement disjoints) d'actions qui pourraient débloquer le processus P_k à cet état d'avancement du système.

Exemple :



Comme le lecteur pourra facilement le constater, le graphe ci-dessus représente un système de quatre processus en deadlock (général).

Remarque :

Un sommet de Γ de degré intérieur nul correspond à un processus actif (ou activable) du système.

2.1. Algorithme pour la détection du deadlock dans un système

Nous décrivons un algorithme de marquage du graphe G_Σ permettant la caractérisation du deadlock dans le système de processus représenté :

```
begin
   {marquer tous les sommets de Γ de degré intérieur nul} ;
    co c'est le marquage initial du graphe oc ;
   while {il existe λ ∈ Λ avec tous les prédécesseurs marqués}
         do {choisir un tel λ};
            {marquer les successeurs de λ};
         od
end
```

Remarque :

Cet algorithme est basé sur la résolution itérative d'une équation à point fixe (remarque due à M. SINTZOFF).

Proposition

Après l'application de l'algorithme de marquage à G_Σ, chaque sommet de Γ non marqué représente un processus en deadlock (local) et chaque sommet marqué représente un processus actif ou un processus déblocable dans cet état du système.

Preuve

Initialement, tous les processus actifs (ou activables) sont marqués. Un sommet P_k sera marqué par l'algorithme s'il existe un ensemble d'actions atomiques qui correspondent au sommet λ prédécesseur de P_k, vérifiant avec la contrainte de synchronisation sur laquelle P_k est bloqué, la relation $\mathcal{D}$. Donc le processus P_k sera activé au bout d'un temps fini (la durée des actions atomiques est finie).

Montrons maintenant qu'un sommet non marqué par l'algorithme représente un processus en deadlock. En effet, si un sommet P_1 n'est pas marqué, son degré intérieur n'est pas nul. Il existe donc un ensemble de processus correspondant à l'ensemble des actions représenté par un prédécesseur de P_1 et parmi ces processus il existe au moins un, soit P_2, non marqué. Le degré intérieur de P_2 étant non nul (sinon il aurait été marqué par le marquage initial), on peut déterminer par le même raisonnement un processus P_3 non marqué, et ainsi de suite P_4, P_5, ..., P_n, P_{n+1}, ... Mais le nombre de processus du système étant fini, nous obtiendrons forcément un circuit dans G_Σ formé des processus P_1, P_2, ... P_1. Chacun des processus du circuit a besoin, pour être débloqué, de débloquer lui-même un processus au moins pour lequel il est en attente. Chaque processus non marqué est donc en deadlock.

2.2. Conditions nécessaires et suffisantes pour l'absence du deadlock

Théorème :

Une condition nécessaire et suffisante pour qu'un système de processus concurrents ne soit pas en deadlock à un certain état d'avancement des processus est que tous les sommets de Γ soient marqués par l'algorithme (§ 2.1).

La démonstration est une conséquence immédiate de la proposition du paragraphe précédent.

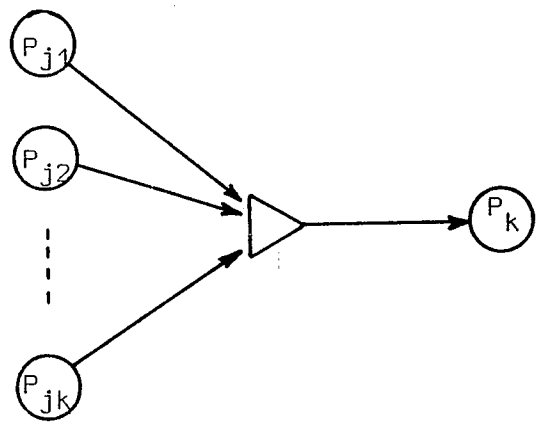
On en déduit qu'un processus P_i n'est pas en deadlock (local) à un certain état d'avancement du système, si et seulement si dans le graphe G_ξ correspondant à cet état, le sommet P_i est de degré intérieur nul ou s'il existe un ensemble de sommets de Γ ayant tous un degré intérieur nul, tel que l'algorithme marquera P_i à partir de cet ensemble.

Donc, à partir d'un ensemble de sommets de Γ de degré intérieur nul, on peut déterminer un sous-graphe de G_ξ contenant tous les processus qui ne sont pas en deadlock à cet instant.

Le théorème ci-dessus assure l'absence du deadlock dans un ensemble de processus à un certain état d'avancement. Nous allons établir des conditions nécessaires et suffisantes pour que le deadlock ne soit pas introduit dans le système par l'avancement des processus.

Plusieurs cas peuvent se présenter :

1) Considérons d'abord un système représenté par un graphe G_ξ et supposons que le système ne soit pas en deadlock à cet instant. Soit P_k un processus déblocable à cet instant par des actions des processus P_{j1}, ..., P_{jk}, c'est-à-dire :



Proposition :

Si un ou plusieurs processus P_{j1}, ..., P_{jk} est autorisé à exécuter une action atomique, le système ne sera pas en deadlock dans l'état obtenu après cette exécution.

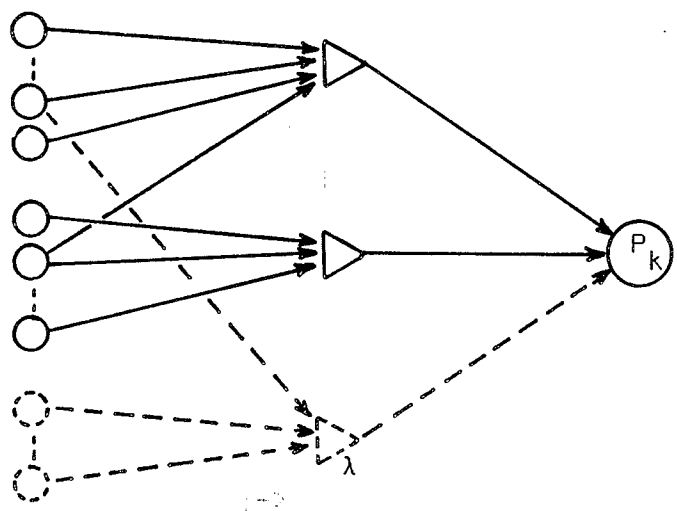
Preuve :

Si tous les processus P_{j1}, ..., P_{jk} sont autorisés à exécuter une action atomique, alors dans le graphe représentatif du nouvel état, P_k aura un degré intérieur nul et donc il sera marqué.

Si seulement une partie des processus P_{j1}, ..., P_{jk} sont autorisés à exécuter une action, alors si P_k était marqué dans le graphe représentatif avant cette exécution (le système n'était pas en deadlock), il pourra être marqué dans le graphe représentatif après l'exécution.

2) Supposons toujours un système sans deadlock représenté par un graphe G_ξ à un état de son avancement. Le passage du système à l'état suivant implique les transformations suivantes pour le graphe G_ξ :

a) Pour un sommet $P_k \in \Gamma$ un nouveau prédécesseur $\lambda \in \Lambda$ est rajouté, représentant des actions atomiques supplémentaires capables de débloquer P_k (en pointillés, la transformation de G_ξ) :



Proposition :

(i) Si le degré intérieur de P_k était nul avant la transformation du graphe, alors une condition nécessaire et suffisante pour que le système reste sans deadlock est que les sommets de Γ qui définissent la transformation de G_{Σ} aient tous un degré intérieur nul ou qu'il existe un ensemble de sommets de Γ ne contenant pas P_k et de degrés intérieurs nuls à partir desquels un sous-graphe de marquage existe et contient tous les sommets rajoutés ;

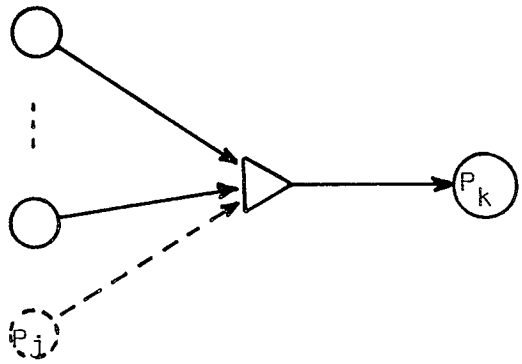
(ii) si le degré intérieur de P_k n'était pas nul avant la transformation du graphe, alors le système restera toujours sans deadlock dans le nouvel état.

Preuve :

(i) La condition est nécessaire car s'il existe un sommet rajouté à Γ, soit P_j, de degré intérieur strictement positif et s'il n'existe pas un ensemble de sommets de Γ ne contenant pas P_k et de degrés intérieurs nuls à partir desquels on puisse marquer tous les sommets rajoutés à Γ alors P_j ne pourra pas être marqué et par conséquent P_k ne pourra pas être marqué non plus, ce qui implique, d'après le théorème, que P_k est en deadlock. La condition est suffisante car, après la transformation du graphe, aucun processus rajouté ne peut être en deadlock s'il a un degré intérieur nul ou s'il existe un sous-graphe de marquage auquel il appartient.

(ii) S'il existait un sous-graphe de marquage contenant P_k avant la transformation, alors un tel sous-graphe existera après la transformation (le système était supposé sans deadlock).

b) Pour un sommet $P_k \in \Gamma$ au moins un des prédécesseurs représentant un ensemble d'actions débloquantes pour P_k doit être complété par une action atomique supplémentaire du processus P_j $(j \neq k)$.



Proposition :

Le système des processus restera sans deadlock après cette transformation si et seulement si le degré intérieur de P_j est nul ou s'il existe un ensemble de sommets de Γ de degré intérieur nul ne contenant pas P_k et de degré intérieur nul à partir duquel un sous-graphe de marquage contient P_j.

La démonstration est analogue à celle de la précédente proposition.

3. CARACTÉRISATION ET VÉRIFICATION DU DEADLOCK

Dans ce paragraphe, nous allons suggérer une méthode pour la vérification des solutions des problèmes de synchronisation basée sur le formalisme de synchronisation par contraintes.

Nous avons vu que les contraintes de synchronisation ont un caractère *universel* (ou invariant) pour l'ensemble des processus considérés. Des *assertions globales* pour les programmes considérés peuvent être déterminées automatiquement pour prouver deux sortes de propriétés pour les programmes concurrents. La première propriété est la prévention du deadlock et la seconde concerne la preuve de correction des programmes séquentiels.

Les programmes calculent formellement une fonction partielle (ou composition de fonctions partielles) sur un domaine de variables "d'entrée" en générant un ensemble de valeurs de "sortie" (cf. chapitre II). C'est donc la réalisation algorithmique d'une relation donnée "d'entrées-sorties" (Fl 67, Ho 77, Ma 74).

Nous faisons la distinction entre deux sortes de programmes concurrents pour lesquels les propriétés ci-dessus ont des interprétations et des résultats différents.

Pour les programmes acycliques (ou avec terminaisons), la première propriété signifie simplement une coopération correcte, c'est-à-dire que le deadlock peut, dans ce cas, être une situation souhaitée signifiant la terminaison du programme. Donc, une propriété fondamentale des programmes acycliques est la propriété de terminaison. Un tel programme est correct si, lors de sa terminaison, il délivre "en sortie" la fonction souhaitée en fonction de ses "entrées".

Pour les programmes cycliques (ou sans terminaisons), tels que les Operating Systems ou certains programmes de simulation par exemple, ces propriétés signifient la nécessité de vérifier constamment les assertions globales pour ne pas atteindre l'état de deadlock (le système reste "vivant") et assurer qu'une requête lancée par un processus à l'adresse d'ur autre processus, sera honorée au bout d'un temps fini (prévention de la "starvation") dans l'ensemble des processus considérés.

Nous ne faisons aucune restriction quant à la structure de l'ensemble des processus considérés (CG 77, DL 77).

3.1. Modèle de programmes concurrents

Un programme concurrent (cf. § II.1.4.) est défini par un ensemble

$$P_1 \,||\, P_2 \,||\, \ldots \,||\, P_n$$

de processus opérant sur un ensemble de variables (parmi lesquelles il y a des variables de synchronisation).

Comme nous avons vu au Chap. II , chaque processus peut être considéré comme un ensemble d'actions atomiques et des "entrées-sorties" (Hoa 77) constituant des actions de communication avec l'environnement.

Nous considérons (voir également Kes 76)) que chaque processus peut être décrit par un graphe orienté fini sans boucles (Σ, τ) où Σ est l'ensemble des états du processus et τ une relation binaire sur Σ définissant les transitions entre les différents états du processus (il existe un seul état initial $S_0 \in \Sigma$).

Une transition entre deux états S et S' correspond à une action (atomique ou de communication) qui fait progresser le processus considéré de l'état S à l'état S'.

A cause du parallélisme, la relation τ n'est pas une fonction, mais dans la plupart des cas (correspondant à un programme *déterministe*), on peut considérer qu'à un état S correspond au plus un état S' tel qu'il existe une transition entre S et S'.

Les transitions dans le graphe peuvent être marquées par deux sortes d'étiquettes :

a) opérationnels

qui correspondent, par exemple, à des instructions d'affectation, primitives de synchronisation, branchements, ..

b) logiques

qui correspondent, par exemple, aux "guarded commands" de Dijkstra.

Pour un programme déterministe, chaque état S a soit :
. un seul arc sortant marqué par une action (label de type a),
. plusieurs arcs sortant marqués par des conditions exclusives
. aucun arc sortant (état *final*).

Remarque :

Les actions atomiques et la communication entre les processus nous conduisent à considérer l'ensemble Σ des états comme le produit cartésien de deux ensembles d'états appelés respectivement *états de contrôle* et *états de traitement*. Le lecteur trouvera dans (Kes 76) un exemple où l'ensemble des états de contrôle est infini et également un modèle pour la représentation des programmes concurrents basé sur une généralisation des réseaux de Petri .

Nous allons maintenant revenir sur l'affirmation faite (chap. I concernant la caractérisation du deadlock dans un ensemble de processus concurrents par la position du point courant dans la région de synchronisation, en donnant des arguments plus formels.

Pour un programme $\{||P_i, i = 1, 2, \ldots, n\}$, soit Σ_i l'ensemble des états de chaque P_i. La position du point courant dans la région de synchronisation est caractérisée par le couple :

$$M = (\bar{S}, \bar{X})$$

où $\bar{S} \in \prod_{i=1}^{n} \Sigma_i$ représente l'état d'avancement des processus

et $\bar{X}$ est la valeur actuelle de l'ensemble des variables de synchronisation.

Dans la région de synchronisation, nous pouvons définir une relation de succession comme suit :

$M' = (\bar{S}', \bar{X}')$ est un successeur de $M = (\bar{S}, \bar{X})$ si et seulement si :

. il existe un processus et un seul P_i qui, par une action, change un état $S \in \bar{S}$ en un état $S' \in \bar{S}'$, les autres états intervenant dans S et $\bar{S}'$ restant inchangés,

. l'action marquant cette transition entre S et S' respecte les contraintes de synchronisation et $\bar{X}'$ est obtenu en appliquant cette action à $\bar{X}$.

Par exemple, l'action peut être une affectation pour une variable locale au processus P_i (c'est un cas limite qui pourrait ne pas être considéré dans la relation de succession) ; c'est la même chose pour une instruction de branchement. Une primitive de synchronisation peut être une telle action si le sémaphore auquel elle est appliquée est dans $\bar{X}$ une variable de synchronisation compatible avec cette primitive (si on effectue un P (s) sur une variable $s \in \bar{X}$, il faut que s soit positif (voir les contraintes de synchronisation exprimant les primitives de synchronisation, § II.3.1.)). Si cette action est une action conditionnelle, il faut que la condition soit vraie avec $\bar{X}$ (c'est le cas des sections critiques conditionnelles ou des moniteurs).

Remarquons que la distinction que nous avons faite entre variables appartenant aux contraintes de synchronisation et variables appartenant aux actions des processus, facilite la transition de $\bar{X}$ à $\bar{X}'$. De même, dans une action conditionnelle pour laquelle $\bar{X}$ n'intervient pas dans la condition, on a $\bar{X}' = \bar{X}$ et c'est encore un cas limite qui pourrait ne pas intervenir dans la succession.

Remargue :

La relation de succession peut être considérée comme une relation quotient de la relation τ restreinte aux variables de synchronisation et aux actions qui les concernent (nous continuons à noter la relation de succession par τ).

Avec ces définitions, nous sommes en mesure de caractériser formellement le deadlock :

Si, pour un point courant, il n'existe pas de successeur, alors ce point est appelé "point de deadlock" pour l'ensemble des processus considérés.

Un point de deadlock pour lequel chaque $S \in \bar{S}$ est un état fina est appelé "point final" du programme concurrent considéré.

Remargue :

Etant donné un vecteur d'états initiaux

$$\bar{S}_0 = (S_0^1, S_0^2, \ldots, S_0^n)$$

pour les processus d'un programme concurrent, alors son déroulement peut êtr caractérisé du point de vue de la synchronisation par une séquence :

$$(\bar{S}_0, \bar{X}_0), \ldots, (\bar{S}_i, \bar{X}_i)$$

où chaque $(\bar{S}_{i+1}, \bar{X}_{i+1})$ est un successeur de $(\bar{S}_i, \bar{X}_i)$ $(i = 0, 1, \ldots)$.

3. 2. Methode inductive de vérification

Nous rappelons brièvement que les méthodes de vérification des programmes font la distinction entre programmes corrects "pas à pas" et programmes "globalement" corrects (Fl 67a, OG 76, Hoa 72b). Ce paragraphe contient une traduction de ces méthodes dans le cadre du formalisme de synchronisation introduit.

Si $(M, M') \in \tau$, c'est-à-dire M' est le successeur immédiat de M (on notera $M \rightarrow M'$), les deux types de preuves s'énoncent :

a) Pour toute séquence de calcul correspondant au déroulement d'un programme (soit infinie, soit finissant avec un point de deadlock), avec un point initial $M_0 = (\bar{S}_0, \bar{X}_0)$ vérifiant un prédicat π, un prédicat invariant Π sera vrai pour tout point atteint par transition à partir de M_0.

b) Pour toute séquence de calcul vérifiant au point initial M_0 un prédicat π, il doit exister un point successeur M_1 vérifiant l'invariant Π.

Mais d'après notre définition des contraintes de synchronisation, la région de synchronisation C est un invariant universel pour l'ensemble des processus et donc les méthodes de vérification s'appliquent.

La méthode de vérification "pas à pas" s'écrit :

$$\begin{cases} \pi\ (\bar{X}_0) \Rightarrow C\ (M_0) \\ C\ (M) \wedge (M \rightarrow M') \Rightarrow C\ (M') \\ C\ (M) \Rightarrow \Pi\ (M) \end{cases}$$

L'invariance de C peut être établie pratiquement en considérant chaque processus comme un programme dont les états sont déterminés par le compteur ordinal des instructions, et vérifiant la région de synchronisation par un essai systématique de l'effet de chacune des instructions des processus considérés.

Remargue :

On peut supposer, sans beaucoup de restrictions, que les processus ne possèdent pas de boucles strictement intérieures (i.e. le processus peut être cyclique lui-même). Cela implique que toute action d'un processus une fois commencée, sera terminée au bout d'un temps fini, exception faite pour certaines actions de communication avec d'autres processus (par exemple, une primitive P sur un sémaphore).

Bien qu'il ne soit pas difficile de donner une description formelle de la correction globale dans un ensemble de processus vérifiant un ensemble de contraintes de synchronisation, nous préférons donner une description plus intuitive à l'aide d'un exemple.

Considérons un programme concurrent formé de deux processus cycliques (ou sans terminaison) P1 et P2 définis comme suit :

```
P1 : iter
         A1
         x := x + 1
    S1 →
         B1
     end

P2 : iter
         A2
         y := y + 1
    S2 →
         B2
     end
```

avec x et y initialisés à 0.

Nous imposons aux processus P1 et P2 les contraintes de synchronisation suivantes :

$$C : \begin{cases} x \geq 0 \\ y \geq 0 \\ x = y \end{cases}$$

Nous supposons encore que dans les sections A_i et B_i ($i = 1, 2$) les variables de synchronisation x et y peuvent intervenir mais ne seront pas changées.

Il est évident qu'après l'initialisation à 0 des deux variables de syncrhonisation, les deux premières contraintes seront invariablement respectées.

Il nous reste à prouver que le programme est globalement correct pour $x = y$. (*)

Cette contrainte n'est pas invariante dans tous les états des processus. Si P1 est dans l'état S_1 et P2 n'est pas encore dans l'état S_2, la contrainte est non vérifiée, tandis qu'au moment où P2 arrive en S_2, la contrainte redevient vraie.

Pour la preuve formelle, nous définissons une distance entre le point courant et la région de synchronisation. Plus exactement, si une action tend à déplacer le point courant à l'extérieur de la région de synchronisation, nous définissons cette distance comme étant égale à l'effet de cette action sur les variables de synchronisation. Cette distance sera une fonction

$$\delta : \bar{S} \rightarrow N$$

Plusieurs valeurs de cette fonction peuvent donc intervenir au cours des actions d'un même processus.

Remarque :

Souvent cette distance sera définie par une fonction caractéristique de la région de synchronisation. Ainsi, pour l'exemple considéré, définit-on (comme $\bar{S}$ est un produit cartésien, des distances δ_i peuvent être définies suivant chacun des facteurs dans le produit) :

pour i = 1, 2 : δ_i (S) = { 1 si S = S_i correspondant aux états de P_i
0 sinon

Alors la relation

$$x + \delta_2 (S) = y + \delta_1 (S) \qquad (\ast\ast)$$

est toujours vraie dans l'évolution des deux processus.

En effet, initialement x = y = 0 et δ_1 (S) = δ_2 (S) = 0 et don la relation est vérifiée.

Si P_i (i = 1, 2) n'a pas encore atteint l'état S_i, la relation (**) reste vraie car les actions des deux processus n'affectent pas les variables de synchronisation et la distance est nulle.

Si P1 atteint l'état S_1 sans que P2 ait atteint S_2 (ou l'inver alors la distance δ_1 (S) compense l'écart provoqué par l'action x := x + 1 sur la région de synchronisation (*) et l'invariant (**) demeure toujours vr

Remarque :

L'exemple ci-dessus nous montre également que la région de syn chronisation et la position du point courant à l'intérieur de cette région peuvent indiquer une stratégie dans l'avance des processus. Si les deux processus parallèles exécutent "en même temps" les actions respectives x := x + et y := y + 1, leur avance est autorisée par la région de synchronisation, tandis que l'avance indépendante serait interdite.

Remarque :

Intuitivement, la distance δ indique la stratégie souhaitée po les actions futures des processus pour que le point courant puisse regagner une position réalisable si des actions des processus avaient tendance à le dépla à l'extérieur de la région de synchronisation.

3.3. Exemples de vérifications

La spécification des contraintes de synchronisation (et donc de la région de synchronisation), forme l'interface entre la structure de synchronisation qui implémente ces contraintes et les processus qui l'utiliseront pour leur synchronisation. Mais nous ne possédons pas actuellement de méthodes formelles (et cela constitue une limite à notre travail) pour la construction "correcte" des contraintes de synchronisation, auquel cas aucune vérification ne serait nécessaire quant aux solutions basées sur ces contraintes. Pour la construction de programmes "corrects" des études sont en cours dans cette direction (Si 78).

Nous avons donné au chapitre III des exemples de régions de synchronisation pour quelques problèmes de synchronisation "classiques". Nous allons montrer ici que ces régions sont correctes.

3.3.1. Sections critiques

Nous avons vu (§ III.2.1.) que la région de synchronisation s'écrit dans ce cas :

$$0 \leq x_1 - x_2 \leq 1$$

où x_1 est le nombre d'actions (pour lesquelles l'exclusion mutuelle est exigée) commencées par les processus,

et x_2 est le nombre d'actions terminées par les processus.

A partir de cette région de synchronisation, nous démontrons que :

a) jamais plusieurs processus P_i ne peuvent entrer simultanément leurs sections critiques ;

b) tout processus P_i pourra entrer dans sa section critiqueavec la même probabilité que les autres processus.

Pour simplifier la démonstration, nous allons considérer un ensemble de deux processus P1 et P2 (le raisonnement est le même pour un nombre quelconque).

Rappelons que les contraintes de synchronisation doivent être vérifiées avant que chacun des processus puisse entrer dans une section critique.

Démontrons d'abord a). Supposons par l'absurde, que les deux processus P1 et P2 tentent l'entrée dans leurs sections critiques. Dans ce cas, $x_1 = 2$ et $x_2 = 0$, ce qui ne vérifie pas la région de synchronisation.

Pour démontrer b), supposons que le processus P1 souhaite entrer sa section critique. Si $x_1 - x_2 < 1$, cela signifie que toutes les sections critiques commencées ont été terminées et donc P1 pourrait entrer la sienne. Si par contre $x_1 - x_2 = 1$, comme P1 n'a pas commencé sa section critique, il résulte que P2 a commencé la sienne. Mais comme chaque action d'un processus a une durée finie, P1 pourra entrer sa section une fois celle de P2 terminée.

Remarque :

Nous avons également vu au § III.2.1. que la contrainte de synchronisation ci-dessus suggérait le choix systématique du sémaphore

$$s = x_2 - x_1 + 1$$

et sa valeur initiale s = 1 pour l'implémentation des sections critiques à l'aide des primitives de Dijkstra.

La démonstration ci-dessus prouve également que l'implémentation des sections critiques par

```
s := 1
P_i : iter
        P (s)
        A_i
        V (s)
      end
```

est correcte, car cette implémentation se déduit automatiquement de l'influence des variables x_1 et x_2 sur le sémaphore s au cours de l'avance des processus considérés.

3.3.2. Producteurs - consommateurs (buffer N-limité)

Rappelons que la région de synchronisation est dans ce cas :

$$C : \begin{cases} s_1 = x_2 - x_3 \geq 0 \\ s_2 = 1 - x_1 + x_2 - x_3 + x_4 \geq 0 \\ s_3 = x_4 - x_1 + N \geq 0 \end{cases} \qquad \text{(cf. § III.2.2.2)}$$

A l'aide des sémaphores s_1, s_2 et s_3 que les trois contraintes déterminent et initialisent respectivement à 0, 1 et N, la solution s'écrit simplement (en introduisant une variable y représentant le nombre d'unités dans le buffer) :

```
Prod : iter
           P (s3)
           P (s2)
           y := y + 1
           V (s2)
           V (s1)
       end

Cons : iter
           P (s1)
           P (s2)
           y := y - 1
           V (s2)
           V (s3)
       end
```

Le fait que $0 \leq y \leq N$ résulte de C et nous ne détaillons pas ici la preuve (qui est évidente d'ailleurs).

Montrons que la région de synchronisation élimine le deadlock. Nous avons vu que le passage de la région de synchronisation au programme concurrent exprimé par les deux processus "Prod" et "Cons" ci-dessus, est immédiatement obtenue en introduisant s_1, s_2 et s_3.

Supposons que le deadlock soit possible. Donc "prod" et "cons" ne peuvent exécuter quelques primitives P. D'après le § IV.3.3.1 les deux processus ne peuvent pas rester bloqués dans l'état situé avant P (s_2). Donc chacun est bloqué à son début sur les sémaphores s_3 et s_1 respectivement. Cela implique donc que $s_1 = s_3 = 0$ et donc le point courant se situe "sur la frontière" de la région de synchronisation et les actions des processus tentent de le déplacer à l'extérieur de C. Mais $s_1 = s_3 = 0$ implique $N = 0$ ce qui contredit l'hypothèse.

Remarque :

Nous pouvons aussi introduire dans ce cas la distance δ par :

$$\delta(\bar{S}) = \begin{cases} 1 \text{ si l'état courant de Prod est après P } (s_3) \text{ et celui de Cons avant V } (s_3), \text{ ou si l'état de cons est après P } (s_1) \text{ et celui de prod avant V } (s_1) \\ 0 \text{ sinon} \end{cases}$$

et montrer que le programme est globalement correct à l'aide de l'invariant :

$$s_1 + s_3 + \delta(\bar{S}) = N$$

pour tout état S des deux processus parallèles. Intuitivement, cela revient à dire que le point courant sera réintroduit à l'intérieur de la région de synchronisation, malgré une tendance des processus à le déplacer à l'extérieur pour certains états. (Ce sont les actions V (s_1) et V (s_3) qui réaliseront cela).

Remarque :

Le formalisme de synchronisation par contraintes indique ici également l'avantage et la possibilité de permettre des actions groupées de processus, plutôt qu'indépendantes. Ici, une intention de consommation peut être groupée avec une fin de production, ou une fin de consommation avec un début de production.

3.3.3. Lecteurs - Ecrivains

Rappelons qu'au § III.2.3 nous avons traité de la spécification de la solution pour un problème plus général que celui proposé dans (CHP 71). Ici, nous nous limitons au cas du premier problème des lecteurs et écrivains, c'est-à-dire :

- plusieurs "lecteurs" peuvent entrer leurs sections critiques simultanément,
- au plus un "écrivain" peut exécuter sa section critique,
- exclusion mutuelle entre "lecteurs" et "écrivains".

Soulignons la particularité de ce problème où le nombre des processus dans chaque classe n'est pas limité.

La solution classique fait intervenir le programme concurrent suivant :

```
L_i : iter
          P (mutex)
          if x_i = 0 then P (s)
          x_i := x_i + 1
          V (mutex)
          {action}
          P (mutex)
          x_i := x_i - 1
          if x_i = 0 then V (s)
          V (mutex)
      end
E_j : iter
          P (mutex)
          {action}
          V (mutex)
      end
```

Initialement, mutex = s = x_i = 0 et x_i représente le nombre de processus dans les sections critiques respectives du programme ci-dessus.

En tenant compte de l'exclusion mutuelle que nous avons traitée au § IV.3.3.1et de la contrainte exprimée au § II.3.2. , l'invariant peut s'écrire :

$$C = \begin{cases} (1) & 0 \leq \text{mutex} \leq 1 \\ (2) & s + e + \text{sign}(1) = 1 \end{cases}$$

Supposons que les processus soient en deadlock. Si mutex = 1, aucun lecteur n'est en attente sur P (s) et donc il n'y a pas de deadlock. On doit donc avoir mutex = 0. Alors un lecteur est dans L_i et, puisqu'on a supposé le deadlock, il en résulte qu'il est bloqué sur P (s), c'est-à-dire que l = 0. Le deadlock implique également que tous les écrivains sont bloqués sur P (s), c'est-à-dire que e = 0, ce qui signifie donc que s = 0. Cela est en contradiction avec (2).

3.3.4. "Le dîner des philosophes"

Dans cet exemple, nous allons souligner une interprétation particulière de la distance définie au § IV.3.2.Il s'agit d'une décomposition de cette distance suivant les contraintes définissant la région de synchronisation. Cela a une justification pratique évidente, étant donné que le point courant peut quitter la région de synchronisation en violant au moins une contrainte apparaissant dans la définition de la région.

Rappelons le problème posé par Dijkstra. Cinq "philosophes" (nous nous limitons au problème initial uniquement pour des raisons d'écriture ; le cas de N philosophes ne pose aucune difficulté supplémentaire), peu s'asseoir autour d'une table avec un bol de riz (pourquoi pas !) devant eux et une baguette à la droite de chacun des cinq philosophes. Comme ce sont de vrais philosophes, leur vie se limite aux actions consécutives : "penser" et "manger". Pour penser, la matière grise que chacun possède (étant donnée sa qualité de philosophe !) lui suffit largement. Pour manger, par contre, chaque philosophe a besoin de deux baguettes, ce qui exclut la possibilité pour deux voisins de se nourrir en même temps (on n'autorise donc pas la galanterie d'un convive de nourrir ses voisins !). On suppose que les deux actions ont une durée limitée. Le problème est de synchroniser les actions des philosophes pour que :

- le deadlock soit impossible,
- chaque philosophe ait accès au bol de riz au bout d'un temps fini (éviter l'inanition)

La solution traduisant ces contraintes de synchronisation a été donnée par Dijkstra (Di 71). Elle nécessite N + 1 sémaphores, un pour chaque ressource (baguettes et table). Les sémaphores associés aux baguettes sont initialisés à 1 et celui correspondant à la table est initialisé à N - 1. La vie de chaque philosophe peut donc être décrite par :

P_i : iter
{penser}
P (table)
P ($mutex_i$)
P ($mutex_{i+1}$)
{manger}
V ($mutex_{i+1}$)
V ($mutex_i$)
V (table)
end

$i = 0, 1, 2, \ldots, N - 1$

Remarque :

Courtois et Georges (CG 77) ont récemment montré, indépendamment de Dijkstra, que la contrainte de synchronisation qui introduit le sémaphore "table" est redondante et qu'il suffit d'ordonner les opérations P (mutex$_i$) avant l'action de "manger" pour éliminer les cycles d'Euler de "contrôleurs effectifs" dans le graphe de contrôle qu'ils définissent.

L'inanition et l'impasse restent impossibles si on suppose qu'au plus N - 1 personnes obéissent au programme :

P_i : iter
{penser}
P (mutex$_i$)
P (mutex$_{i+1}$)
{manger}
V (mutex$_{i+1}$)
V (mutex$_i$)
end

et les autres philosophes obéissent à :

P_i : iter
{penser}
P (mutex$_{i+1}$)
P (mutex$_i$)
{manger}
V (mutex$_i$)
V (mutex$_{i+1}$)
end

On définit les distances de la manière suivante (il est facile de vérifier que ce sont des distances entre le point courant et les contraintes de synchronisation) :

$$\delta_i = \begin{cases} 1 & \text{si l'état de } P_i \text{ est situé après P(table) et avant V(tabl} \\ 0 & \text{sinon} \end{cases}$$

$$\delta'_i = \begin{cases} 1 & \text{si l'état de } P_i \text{ est situé après P(mutex}_i\text{), avant V(mutex} \\ 0 & \text{sinon} \end{cases}$$

$$\delta''_i = \begin{cases} 1 & \text{si l'état de } P_i \text{ est situé après P(mutex}_{i+1}\text{)} \\ & \qquad \text{et avant V(mutex}_{i+1}\text{)} \\ 0 & \text{sinon} \end{cases}$$

Alors, les relations suivantes sont invariantes pour l'ensemble des processus (les opérations sont modulo N) :

$$\text{(a) } mutex_i + \delta'_i + \delta''_{i-1} = 1 \; ; \qquad i = 0, 1, \ldots, N-1$$

$$\text{(b) } table + \sum_{i=0}^{N-1} \delta_i = 4$$

Supposons que les processus soient en deadlock. Cela signifie qu'ils sont tous en attente sur des sémaphores. Mais cela est possible uniquement si leurs états respectifs se situent avant P (table) ou avant P ($mutex_i$), ou encore avant P ($mutex_{i+1}$). Donc, pour tout i on a $\delta'' = 0$.

Supposons maintenant que $\delta_i = 1$ et $\delta'_i = 0$ pour tout i, ce qui signifie que l'état de P_i est avant P ($mutex_i$), donc que P_i est en attente sur $mutex_i$. Mais $\delta'_i = 0$ et $\delta''_i = 0$ pour tout i, implique $mutex_i = 1$ dans (a), ce qui contredit le fait que P_i est en attente sur $mutex_i$. On peut donc remplacer (a) par :

$$\text{(a1) } mutex_i + \delta_i = 1, \qquad i = 0, 1, \ldots, N-1$$

Supposons maintenant que P_i est en attente sur $mutex_{i+1}$. Comme dans ce cas $\delta_i = 1$, on a forcément :

(a2) $mutex_{i+1} + \delta_i \leq 1$

Si on soustrait (a2) de (a1) on obtient :

$$mutex_i - mutex_{i+1} \geq 0 \quad \forall\, i$$

ce qui est possible uniquement si $mutex_i = mutex_{i+1}$, $\forall$ i. Supposons donc que $mutex_i = 1$, $\forall$ i. Alors, d'après (a1) $\delta_i = 0$, $\forall$ i et d'après (b) il résulte que table = 4. Cela signifie donc que tous les processus attendent sur P (table) alors que table > 0 ; absurde.

Si maintenant on suppose $mutex_i = 0$, $\forall$ i, alors d'après (a1) $\delta_i = 1$, $\forall$ i et (b) implique table = -1 ; absurde.

On en conclut donc que les contraintes de synchronisation qui impliquent les invariants (a) et (b) pour ce problème, excluent le deadlock.

3.4. Prévention du deadlock

Comme exemple d'application de notre formalisme à la prévention du deadlock, nous allons analyser et justifier une méthode très connue : l'algorithme du banquier défini par Dijkstra.

Rappelons qu'il s'agit d'un ensemble de N processus définis par des programmes acycliques, qui se partagent un ensemble fini d'unités d'une même ressource ; soit $\bar{M}$ le nombre total d'unités dans le système (le capital du banquier). Pour terminer leurs actions, chaque processus P_i ($i = 0, 1, \ldots, N-1$), a besoin d'un nombre d'unités fixé à l'avance, soit $\overline{M_i}$. Un processus P_i peut bénéficier d'une ou plusieurs unités qui seront ajoutées à son crédit courant M_i ; il peut également retourner une ou plusieurs unités dans les limites de son crédit courant, et à la fin de son activité il retournera toutes les unités en sa possession. On aura donc pour chaque processus P_i :

$$0 \leq M_i \leq \overline{M_i} \leq \bar{M}$$

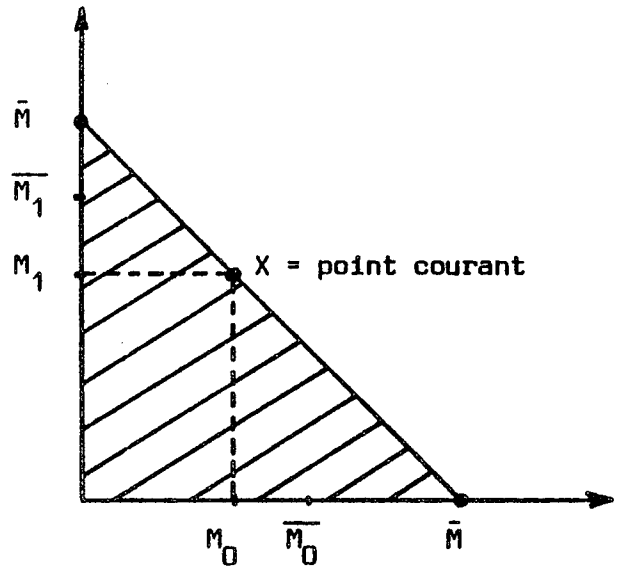
A tout moment, le banquier dispose d'un capital courant donné par :

$$M = \bar{M} - \sum_{i=0}^{N-1} M_i$$

et son objectif est, connaissant les besoins $\overline{M_i}$ de chacun des processus, d'assurer que toutes les demandes présentes et futures seront satisfaites au bout d'un temps fini. Il est évident que son objectif n'est pas assuré d'avance et des situations de deadlock sont facilement observables.

Un rapprochement est à faire entre l'algorithme défini par Dijkstra pour le banquier et le "dîner des philosophes" où l'introduction d'un état supplémentaire "affamé" était nécessaire pour chacun des processus. Ici, chaque "emprunt" doit être divisé en deux actions : demande au banquier et attente pour que le compte courant soit crédité.

La région de synchronisation correspondant à l'ensemble des processus est (pour deux processus P_0 et P_1) :



Si le banquier avait le contrôle absolu du système (distribuer et relancer des unités suivant sa volonté), son seul souci serait de maintenir à chaque instant le point courant X dans la région de synchronisation. Mais, comme chaque processus P_i a un besoin $\overline{M_i}$ annoncé d'avance, soit D_i le nombre maximum d'unités dont il a besoin avant de retourner les unités en sa possession. Donc :

$$D_i = \overline{M_i} - M_i$$

Avec cette nouvelle contrainte, le banquier doit trouver un cheminement du point courant X dans la région de synchronisation pour satisfaire tous les besoins des processus.

S'il existe une permutation des processus telle que pour tout $0 \leq i \leq N-1$ on ait :

$$D_i \leq M + \sum_{j=0}^{i} M_j$$

alors une stratégie existe pour garantir l'objectif du banquier.

En effet, une stratégie évidente est de servir le processus P_i uniquement si tous les processus P_j ($0 \leq j < i$) ont terminé leurs transactions. Alors, la relation ci-dessus implique simplement que le capital est suffisant pour garantir la demande du processus P_i pour $i = 0, 1, .., N-1$ successivement. Comme il est supposé que P_i termine au bout d'un temps fini (il est acyclique), i peut être incrémenté par 1 chaque fois.

L'algorithme de Dijkstra cherche une telle permutation des processus pour que la relation

$$D_i \leq M + \sum_{j=0}^{i} M_j \quad , \forall\ 0 \leq i < k$$

reste invariante avec $k = 1, 2, .., N$.
Comme pour $k = 1$ la relation est évidente, l'algorithme incrémente k de 1 jusqu'à N sans changer l'ordre des processus $p_0, .., p_{k-1}$ et détermine un l (s'il existe) tel que $k \leq l < N$ et

$$D_l \leq M + \sum_{j=0}^{k} M_j$$

Pour prévenir le deadlock, l'algorithme du banquier cherche donc un cheminement particulier dans la région de synchronisation.

CONCLUSIONS ET LIMITES

"maintenant nous savons
que l'infamie des méthodes
se multiplie
dans l'infamie des résultats"

A. Soljenitsyne, 'Appel à la résistance'

Dans cette étude, nous définissons essentiellement une méthodologie pour la spécification de la synchronisation dans un ensemble de processus concurrents. L'avance des processus est conditionnée par des relations de coopération, compétition et communication. A partir de ces relations, nous définissons les *régions de synchronisation* par des contraintes auxquelles les actions des processus sont soumises.

Le formalisme de spécification par contraintes nous a permis de rendre plus systématique la recherche des solutions aux problèmes de synchronisation. Son impact sur l'étude du deadlock a été ensuite analysé.

Bien que la méthode soit exprimée dans un cadre général, nous n'avons pas approfondi dans ce travail l'étude de contraintes quelconques de synchronisation. Cette limite est due au souci de présenter des exemples de problèmes classiques de synchronisation, problèmes qui peuvent s'exprimer généralement par des contraintes linéaires.

Pour un même problème, les contraintes qui le représentent ne sont pas uniques ; trouver des méthodes pour leur élaboration systématique reste un problème ouvert.

AD 78 ANDRE E. & DECITRE P.
On providing distributed application programmers with control over synchronization
Computer Networks Protocols, Université de Liège, A. Danthine Ed.

Ag 77 AGERWALA T.
Some extended semaphore primitives
Acta Informatica, 8, 201-220

As 75 ASHCROFT E.A.
Proving assertions about parallel programs
Journal of Computer and System Sciences, 10, 110-135

At 76 ATWOOD J.W.
Concurrency in operating systems
Computer, Oct., 18-26

BBV 77 BEKKERS Y., BRIAT J., VERJUS J.P.
Construction of a synchronization scheme by independent definition of parallelism
IFIP Conference on Constructing Quality Software, North Holland Publishing Co. (à paraître)

BDM 78 BEKKERS Y., DARONDEAU Ph., MUNTEAN T.
Types de synchronisation et types abstraits concurrentiels
Rapport de Recherche n° 108, ENSIMAG, Grenoble 1978

Bek 74 BEKKERS Y.
A comparison of two high-level synchronizing concepts
T.R., Department of Computer Science, The Queen's University, Belfast

Ber 73 BERGE C.
Graphes et Hypergraphes
Dunod - Bordas, Paris.

BH 70 BRINCH HANSEN P.
The nucleus of a multiprogramming system
Communications ACM, 13, 4, 238-250

BH 72 BRINCH HANSEN P.
Structured multiprogramming
Communications ACM, 15,7, 574-578

BH 73 a BRINCH HANSEN P.
Operating system principles
Prentice Hall, Englewood Cliffs, New Jersey

BH 73 b BRINCH HANSEN P.
Concurrent programming concepts
Computing Surveys, 5, 4

BH 75 BRINCH HANSEN P.
The programming language concurrent Pascal
IEEE Transactions on Software Engineering, SE-1, 2, 199-207

BH 77 a BRINCH HANSEN P.
The architecture of concurrent programming concept
Prentice Hall, Englewood Cliffs, New Jersey

BH 77 b BRINCH HANSEN P.
Distributed process, a concurrent programming concept
Computer Science Dept., University of Southern California,
Los Angeles

BHS 77 BRINCH HANSEN P. & STAUNSTRUP J.
Specification and implementation of mutual exclusion
Computer Science Dept., University of Southern California, Los Angeles

BM 76 BOKSENBAUM C. & MUNTEAN T.
Synchronization by means of constraints, formulation and applications
Colloque International de Programmation, Paris 1976

Br 74 BRON C.
Trends in the synchronization of parallel processes
Symposium of the Dutch Computer Society, February 1974

BW 73 BELPAIRE G. & WILMOTTE J.P.
A semantic approach to the theory of parallel processes
I.C.S. North Holland Publishing Co., 1974, 159-164.

Ca 74 CAMPBELL R.H.
The specification of process synchronization by path expressions
Colloque sur les aspects théoriques et pratiques des systèmes d'exploitation, IRIA, Paris, 93-106.

Ce 72 CERF V.G.
Multiprocessors, semaphores and a graph model of computation
UCLA Computer Science Dept., Report ENG-7223, University of California, Los Angeles

CES 71 COFFMAN E.G., ELPHICK M.J., SHOSHANI A.
System deadlocks
Computing Surveys 3, 2, 67-78

CG 77 COURTOIS P.J. & GEORGES J.
On starvation prevention
RAIRO Informatique, 11, 2, 127-141

CH 74 CAMPBELL R.H. & HABERMANN A.N.
The specification of process synchronization by path expressions
T.R. 55, Computing Laboratory, University of Newcastle Upon Tyne

CHCP 74 COOPRIDER L.W., HEYMANS F., COURTOIS P.J., PARNAS D.L.
Information streams sharing a finite buffer : other solutions
Information Processing Letters, 3,1, 16-21

CHP 71 COURTOIS P.J., HEYMANS F., PARNAS D.L.
Concurrent control with readers and writers
Communications ACM, 14, 16, 667-668

CL 77 COTRONIS J.Y. & LAUER P.E.
Verification of concurrent systems of process
International Computing Symposium, 197-207

DDH 72 DAHL O.J., DIJKSTRA E.W., HOARE C.A.R.
Structured programming
Academic Press, New York

De 77 DEVILLERS R.
Game interpretation of the deadlock avoidance problem
Communications ACM, 20, 10, 741-745.

Di a DIJKSTRA E.W.
A strong P/V implementation of conditional critical regions
Communication privée, EDW/651 et 652

Di b DIJKSTRA E.W.
On two beautiful solutions designed by Martin Rem
Communication privée, EDW/607

Di 65 DIJKSTRA E.W.
Solution of a problem in concurrent programming control
Communications ACM, 8, 9, 569-

Di 68 a DIJKSTRA E.W.
The structure of THE multiprogramming system
Communications ACM, 11,5,

Di 68 b DIJKSTRA E.W.
Cooperating sequential processes
Programming Languages, Academic Press, 1968

Di 71 DIJKSTRA E.W.
Hierarchical ordering of sequential processes
Acta Informatica, 1, 2, 115-138

Di 72 DIJKSTRA E.W.
A class of allocation strategies inducing bounded delays only
Proceedings SJCC, 933-936

Di 74 DIJKSTRA E.W.
Self stabilizing systems in spite of distributed control
Communications ACM, 17, 11, 643-644

Di 75 DIJKSTRA E.W.
Guarded commands, non determinacy and formal derivation of programs
Communications ACM, 18, 8, 453-457

Di 76 DIJKSTRA E.W.
A discipline of programming
Prentice Hall, Engelwood Cliffs, New Jersey

DL 77 DEVILLERS R.E. & LAUER P.E.
A general mechanism for avoiding starvation with distributed control
Computing Laboratory, University of Newcastle upon Tyne

DLMSS 75 DIJKSTRA E.W., LAMPORT L., MARTIN A.J., SCHOLTEN C.S., STEFFENS E.
On the fly garbage collection : an exercise in cooperation
NATO Summer School on Language hierarchies and interfaces, Marktoberforf 1975, Lecture Notes in Computer Science, 46, Springer Verlag

DM 78 DARONDEAU Ph. & MUNTEAN T.
Linking synchronization types with abstract data types
ACM Computer Science Conference, Detroit

DS 77 DANG Ng.X. & SERGEANT G.
Expression of parallelism and communication in distributed network processing
International Conference on Parallel Processing, Bellaire.

FH 76 FLON L. & HABERMANN A.N.
Towards the construction of verifiable software systems
ACM Conference on Data : abstraction, definition and structure, SIGPLAN Notices 8, 2, 141-148

FHdR 77 FRANCEZ N., HOARE C.A.R., de ROEVER W.P.
Concurrency and guarded commands : an exercise in formulation
Communication privée, Wolfson College, Université d'Oxford

Fl 67a FLOYD R.W.
Assigning meanins to programs
Symposia in Applied Mathematics, vol. 19, Amer. Math. Soc., Providence, Rhode Island, 19-32

Fl 67b FLOYD R.W.
Non-deterministic algorithms
Journal ACM, 14, 4, 636-644

FP 78 FRANCEZ N. & PNUELI A.
A proof method for cyclic programs
Acta Informatica 9, 2, 133-157

FS 77 FLON L. & SUZUKI N.
Non-determinism and the correctness of parallel programs
IFIP Congress, Toronto

Ger 77 GERBER A.J.
Process synchronization by counter variables
O.S.R., 11, 4, 6-17

Gi 78 GIRAULT C.
Réseaux de Petri et synchronisation de processus
Institut de Programmation, Paris

GL 73 GENRICH H.J. & LAUTENBACH K.
Synchronizationsgraphen
Acta Informatica, 2, 143-161

Gr 77 GRIES D.
An exercise in proving parallel programs correct
ACM, 20, 12, 921-930

Hab 67 HABERMANN A.N.
On the harmonious cooperation of abstract machines
Technische Hogeschool, Eindhoven

Hab 69 HABERMANN A.N.
Prevention of systems deadlock
Communications ACM, 12, 7, 373-377, 385.

Hab 72 HABERMANN A.N.
Synchronization of communicating processes
Communications ACM, 15, 3, 177-184

Hab 74 HABERMANN A.N.
A solution and a generalization of the cigarette smokers problem
Computing Science Dept., Carnegie Mellon University, Pittsburgh

Hab 75a HABERMANN A.N.
Path expressions
Computing Science Dept., Carnegie Mellon University, Pittsbrugh,
Projet ARPA

Hab 75b HABERMANN A.N.
A new approach to avoidance of system deadlocks
RAIRO Informatique, Septembre, 19-28

Had 77 HADDON B.K.
Nested monitor calls
O.S.R., 11, 4, 18-23

Hav 68 HAVENDER J.W.
Avoiding deadlock in multitasking systems
IBM System Journal, 7, 2, 74-84

HC 70 HOLT A.W. & COMMONER F.
Events and conditions
ACM Project MAC, Conference on Concurrent Systems and Parallel Computation, New York

He 77 HERZOG O.
Automatic deadlock analysis of parallel programs
I.C.S., North Holland Publishing Co., 209-216

Hoa 69 HOARE C.A.R.
An axiomatic basis for computer programming
Communications ACM, 12, 10, 576-583

Hoa 72a HOARE C.A.R.
Towards a theory of parallel programming
Operating Systems Techniques, Academic Press, New York

Hoa 72b HOARE C.A.R.
Proof of correctness of data representations
Acta Informatica 1972, 1, 271-281

Hoa 73 HOARE C.A.R.
A structured paging system
Computer Journal, 16, 3, 209-215

Hoa 74 HOARE C.A.R.
Monitors : an operating system structuring concept
Communications ACM, 17, 10, 549-557

Hoa 76 HOARE C.A.R.
Some properties of non-deterministic computations
Dept. of Computer Science, the Queen's University, Belfast

Hoa 77 HOARE C.A.R.
Communicating sequential processes
Dept. of Computer Science, The Queen's University, Belfast

Hoa 78 HOARE C.A.R.
Notes on communicating automata
IFIP W.G. 2.3, Warwick

Hola 72 HOLAGER
A consolidation of synchronization primitives
SIGPLAN Notices, 7, 42-52

Holt 68 HOLT R.C.
Theory of occurence systems
Princeton N.J., AD 676972

Holt 71 HOLT R.C.
Comments on prevention of system deadlocks
Communications ACM, 14, 1, 36-38

Holt 72 HOLT R.C.
Some deadlock properties of computer systems
ACM Computing Surveys, 14, 3, 179-196

How 73 HOWARD J.H.
Mixed solutions for the deadlock problem
Communications ACM, 16, 7, 427-430

How 76 HOWARD J.H.
Proving monitors
Communications ACM, 19, 5, 273-278

HR 73 HORNING J.J. & RANDELL B.
Process structuring
Computing Surveys, 5, 1, 5-30

Jo 77 JONES C.B.
Program development using data abstraction
IBM, International Education Centre, Bruxelles

Ka 71 KAISER C.
Quelques problèmes de parallélisme et leurs solutions par sémaphores
Projet ESOPE, IRIA, Rocquencourt.

Kel 76 KELLER R.M.
Formal verification of parallel programs
Communications ACM, 19, 7, 371-384

Kel 77 KELLER R.M.
Semantics of parallel program graphs
Computer Science Dept., University of Utah, UUCS, 77-110

Kes 77 KESSELS J.L.W.
An alternative to event queues for synchronization in monitors
Communications ACM, 20, 7, 500-503

Ko 73 KOSARAJU S.R.
Limitations of Dijkstra's semaphore primitives and Petri nets
Symposium on Operating System Principles, New York

KPH 76 KAUBISCH W.H., PERROTT R.H., HOARE C.A.R.
Quasiparallel programming
Software, Practice and Experience, 6, 341-356

Lam 74 LAMPORT L.
A new solution of Dijkstra's concurrent programming problem
Communications ACM, 17, 8, 453-455

Lam 76 LAMPORT L.
The synchronization of independent processes
Acta Informatica, 7, 15-34

Lam 77 LAMPORT L.
Concurrent reading and writing
Communications ACM, 20, 11, 806-811

LaTo 76 LAUER P.E. & TORRIGIANI P.R.
Toward a system specification language based on paths and processes
The University of Newcastle Upon Tyne, ASM 19

Lau 75 LAUESEN S.
A large semaphore based operating system
Communications ACM, 18, 7, 377-389

LC 74 LAUER P.E. & CAMPBELL R.M.
A description of path expressions by Petri nets
Proceedings of the 2nd ACM Symposium on Principles of Programming Languages, Palo Alto, Cal., 95-105

LC 75 LAUER P.E. & CAMPBELL R.M.
Formal semantics of a class of high-level primitives for coordinat concurrent processes
Acta Informatica, 5, 237-332.

Led 75 LEDGARD H.F.
A genealogy of control structures
Communications ACM, 18, 11, 629-631

Lev 72 LEVITT K.N.
The application of program-proving techniques to the verification of synchronization processes
Fall Joint Conference, 33-47

Lip 73 LIPTON R.J.
On synchronization primitive systems
Carnegie Mellon University, Computing Science Dept.

Lip 75 LIPTON R.J.
Reduction : a method of proving properties of parallel programs
Communications ACM, 18, 12, 717-721

Lis 77 LISTER A.M.
The problem of nested monitor calls
O.S.R., 11, 3, 5-7

LiSa 77 LISTER A.M. & SAYER P.J.
Hierarchical monitors
Software, Practice and Experience, 7, 613-623

LiTu 75 LIPTON R.J. & TUTTLE R.W.
A synchronization anomaly
Information Processing Letters, 3, 3, 65-66

LM 76 LISTER A.M. & MAYNARD K.J.
An implementation of monitors
Software, Practice and Experience, 6, 377-385

Lo 77 LOMET D.B.
Process structuring, synchronization and recovery using atomic actions
SIGPLAN Notices, Conference on Language Design and Reliable Software, 128-137

LS 77 LAUER P.E. & SHIELDS M.W.
Abstract specification of resource accessing disciplines : adequacy, starvation, priority and interrupts
University of Newcastle Upon Tyne, ASM/41.

Ma 51 MARKOV A.A.
Theory of algorithms
Works of the Mathematical Institute Imeni Steklov, vol 38, 176-189

MPS 77 MOALLA M., PULOU J., SIFAKIS J.
Réseaux de Petri synchronisés
Rapport de Recherche n° 80, ENSIMAG, Grenoble

Mu 77 MUNTEAN T.
Formalism for the specification of unities controlling process cooperation
Conférence on Information Science and Systems, Baltimore

Ne 75 NEWTON G.
Proving properties of interacting processes
Acta Informatica, 4, 117-126

OG 76 OWICKI S. & GRIES D.
Verifying properties of parallel programs : an axiomatic approach
Communications ACM, 19, 5, 279-285

Ow 77 OWICKI S.
Verifying concurrent programs with shared data classes
Digital systems Laboratory, Stanford University, Cal 94305

Par 75 PARNAS D.L.
On a solution to the cigarette smokers' problem (without conditional statements)
Communications ACM, 18, 3, 181-183

Par 77 PARNAS D.L.
The non-problem of nested monitor calls
O.S.R.

Pat 71 PATIL S.S.
Limitations and capabilities of Dijkstra's semaphore primitives for coordination among processes
Project MAC, MIT, Cambridge, Mass.

Pe 75 PETRI C.A.
Interpretations of net theory
Institut für Informationssystemforschung, GMD, Bonn

Pr 75 PRESSER L.
Multiprogramming coordination
Computing Surveys, 7, 1, 21-44

Ra 78 RAYNAL M.
Une expression de la synchronisation pour les types abstraits
IRISA, Rennes (RAIRO Informatique, à paraître)

RB 77 RUSSELL D.L. & BREDT T.H.
Error resynchronization in producer-consummer systems
Digital Systems Laboratory, Stanford University, California

RM 75 RABUE J.E. & MULLINS J.M.
Solving synchronization problems using semaphores
Software, Practice and Experience, 5, 51-64

Ro 78 ROUCORAIL G.P.
Mots de synchronisation
Institut de Programmation, Paris (RAIRO Informatique, à paraître)

RV 77 ROBERT P. & VERJUS J.P.
Towards autonomous description of synchronization modules
IFIP Congress 77, Toronto

Schm 76 SCHMID H.A.
On the efficient implementation of conditional critical regions and the construction of monitors
Acta Informatica, 6, 227-249

Scho 75 SCHOLTEN C.S.
Self-stabilization in general event graphs
Communication privée, CD 75/200/1

Sh 75 SHRIVASTAVA S.K.
A view of concurrent process synchronization
Computer Journal, 18, 4, 375-379

Sh 76 SHRIVASTAVA S.K.
Systematic programming of scheduling algorithms
Software, Practice and Experience, 6, 357-370

Si 78 SINTZOFF M.
On language design for program construction
Rapport de Recherche 78.1.001, Nancy

So 77 SOKOLOWSKI S.
Axioms for total correctness
Acta Informatica, 9, 61-71

SR 71 SAAL J.H. & RIDDLE W.E.
Communicating semaphores
IFIP Congress

SvL 75 SINTZOFF M. & Van LAMSWEERDE A.
Constructing correct and efficient concurrent programs
Conference on Reliable Software, Los Angeles

V 78 VERJUS J.P.
Expression du contrôle du parallélisme à l'aide d'invariants
Groupe Programmation et Langages, AFCET, La Bresse

vL 77 Van LAMSWEERDE A.
From verifying termination to guaranteeing it : a case study
IFIP Congress 77, Toronto

vLS 76 Van LAMSWEERDE A. & SINTZOFF M.
Formal derivation of strongly correct parallel programs
MBLE Research Laboratory, RR 338, Bruxelles

VMT 77 VERJUS J.P. MOSSIERE J. TCHUENTE M.,
Sur l'exclusion mutuelle dans les réseaux informatiques
IRISA, Rapport de Recherche 75, Rennes

VV 77 VAUDENE D. & VIGNAT J.C.
Enoncés de synchronisation et moniteurs : deux entités regroupées
Journées AFCET Applications en Temps Réel, Institut de Programmation, Paris

VvL 72 VANTILBORGH H. & Van LAMSWEERDE A.
On an extension of Dijkstra's primitives
Information Processing Letters, 1, 181-186

We 77 WETTSTEIN H.
The implementation of synchronizing operations in various environments
Software, Practice and Experience, 7, 115-126

Wi 77 WIRTH N.
Toward a discipline of real-time programming
Communications ACM, 20, 8, 577-583

ERRATA :

Page 74, ligne 2 :

$$s_i = \sum_{j=1}^{n} a_{ij} x_j - b_i \quad \text{pour } i \in I$$

Page 84, ligne 17 :

$$s = 2N - \sum_{i=1}^{m-k} [N \operatorname{sign}(x_i) + N_i x_i] - 2N \sum_{i=m-k+1}^{m} \operatorname{sign}(x_i) \geq 0$$

Page 94, ligne 24 :

$$s_i = \sum_{j=1}^{n} a_{ij} x_j - b_i \quad i \in I$$

dernière page de la thèse

AUTORISATION DE SOUTENANCE

VU les dispositions de l'article 3 de l'arrêté du 16 Avril 1974,

VU le rapport de présentation de Messieurs :

L. BOLLIET, Professeur U.S.M.G.

C. BOKSENBAUM, Maître de Conférences à l'I.U.T. de MONTPELLIER

Monsieur Traian M U N T E A N

est autorisé à présenter une thèse en soutenance pour l'obtention du titre de DOCTEUR de TROISIEME CYCLE, spécialité "Génie Informatique".

Grenoble, le 7 Juin 1978

Ph. TRAYNARD
Président
de l'Institut National Polytechnique