Validation et mise en oeuvre de la synchronisation dans un système multiprocesseur à mémoire dupliquée

Guy Latapie

N° d'ordre : 10

Année 1980

THÈSE

présentée

DEVANT L'INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE TOULOUSE

en vue de l'obtention

du Titre de DOCTEUR de 3e CYCLE

Spécialité : E.E.A. - Mention : Automatique

par

Guy LATAPIE

Maître ès Sciences

VALIDATION ET MISE EN ŒUVRE DE LA SYNCHRONISATION DANS UN SYSTEME MULTIPROCESSEUR A MEMOIRE DUPLIQUEE

Soutenue le 14 Novembre 1980, devant la Commission d'Examen :

MM.	A. TITLI	Président
	P. AZEMA	Examinateurs
	M. COURVOISIER	
	H. HECKENROTH	
	J. MOSSIERE	
	R. VALETTE	

A mes parents,
A Martine et à Jackie,
Avec toute mon affection

AVANT-PROPOS

- - - -

Ce document représente la synthèse des travaux que nous avons effectués au Laboratoire d'Automatique et d'Analyse des Systèmes du C.N.R.S. au sein de l'équipe Parallèlisme Analyse Spécification et Test En Ligne des Structures.

Nous tenons à remercier :

- *Monsieur A. TITLI, Professeur à l'Institut National des Sciences Appliquées de Toulouse,*
- *Monsieur P. AZEMA, Maître de Recherche au C.N.R.S., Responsable de la division II du L.A.A.S,*
- *Monsieur M. COURVOISIER, Professeur à l'Université Paul Sabatier de Toulouse, Responsable de l'équipe P.A.S.T.E.L.S.,*
- *Monsieur H. HECKENROTH, Ingénieur de Recherche au Service Base de Données du C.E.T.E. d'Aix en Provence,*
- *Monsieur J. MOSSIERE, Maître-Assistant à l'Institut National Polytechnique de Grenoble,*
- *Monsieur R. VALETTE, Chargé de Recherche au C.N.R.S.,*

pour leurs remarques concernant le manuscrit et leur participation au jury,

- *Brigitte PRADIN, Jocelyne NOUBEL, Bernard BERTHOMIEU, pour leurs conseils et leur amitié à notre égard, ainsi que les membres de l'équipe P.A.S.T.E.L.S.,*
- *Brigitte DAROLLES, pour la dactylographie,*
- *Emile LAPEYRE-MESTRE, Jean CATALA, Bernard MEUNIER, Guy GALINIER, pour les témoignages d'amitié qu'ils nous ont donnés au cours de ces dernières années.*

INTRODUCTION

L'augmentation de la puissance d'intégration des composants électroniques qui a permis la naissance des microprocesseurs est à l'origine d'une pénétration de l'informatique dans des domaines d'application très diversifiés. Un domaine, particulièrement favorisé est celui de la commande de processus industriels qui requièrent une puissance de calcul et une rapidité suffisante pour analyser et commander des phénomènes en temps réel.

Les réalisations obtenues par l'utilisation des microprocesseurs sont alors beaucoup plus élégantes au point de vue encombrement, prix, souplesse .... que les solutions traditionnelles. Toutefois, pour compenser une relative lenteur de ces microprocesseurs, beaucoup de systèmes sont construits à partir d'une structure multimicroprocesseur travaillant en parallèle (quand cela est possible). Deux facteurs sont alors à prendre en compte ; l'indépendance dans le déroulement des processus, c'est-à-dire, l'autonomie du déroulement de chaque processus par rapport aux autres ; et le partage des ressources nécessaires à un processus, c'est-à-dire, le fait qu'une ressource puisse être affectée à un processus ou non. C'est à partir de ces facteurs que se définit généralement la synchronisation réalisée dans le système.

Un autre intérêt de ces systèmes, est qu'il peut leur être adaptées des stratégies telles que la présence de pannes est un événement prévu et tolérable. Ces systèmes tolérants aux pannes, sont de plus en plus utilisés, surtout depuis que la sûreté de fonctionnement est devenue une contrainte première pour la conception de systèmes. En conséquence, généralement les ressources sont au moins dupliquées.

Quand une ressource est constituée par un ensemble de données, cette "duplication" introduit de nouveaux problèmes de synchronisation qui sont liés au maintien de l'intégrité des données. En effet, pour un bon fonctionnement du système, il faut que les données contenues dans les deux mémoires soient identiques. Ainsi, chaque modification apportée sur un exemplaire des données doit être répercutée sur les autres (on dit encore que l'on assure "l'unicité" des données).

Cependant, cette duplication peut être également utilisée pour augmenter le parallèlisme et ainsi diminuer le temps de réponse. Pour cela, il suffit de répartir la charge sur les différentes ressources. Ainsi, l'utilisation plus rationnelle des ressources permet de diminuer fortement les accès à cette ressource, et d'éviter des phénomènes de saturation ou d'engorgement.

Dans la réalisation de ces systèmes, deux types de mise en oeuvre de la synchronisation existent; centralisé ou distribué. Une solution centralisée, quand elle est possible, offre fonctionnellement plus d'avantages au point de vue temps de réponse et sûreté de fonctionnement. En effet, il est plus facile de surveiller un processeur responsable plutôt que la communication entre plusieurs processeurs. En particulier, une solution centralisée offre la possibilité de se protéger contre un mauvais fonctionnement d'un microprocesseur en introduisant des possibilités de surveillance plus sévères de la synchronisation. Evidemment, en dehors de ce problème de synchronisation, ces solutions centralisées sont toutefois structurellement plus fragiles que les autres, et il est nécessaire de prévoir pour ces systèmes une reprise en cas de panne des organes centraux.

Le travail que nous présentons dans ce mémoire, correspond à l'étude de la faisabilité d'un système de commande de processus industriels, localement distribué et sûr de fonctionnement. Ce système est conçu de la manière suivante :
Un ensemble de micro-calculateurs, banalisés du point de vue matériel, effectue un certain nombre de tâches qui ne sont pas totalement indépendantes. Par exemple, certains calculateurs sont chargés de régulations locales pendant que d'autres ont à assurer des tâches de coordination et de gestion globale. Le système étant géographiquement centralisé, les échanges d'information se font par l'intermédiaire d'une mémoire partagée et dupliquée.

Plus précisément, notre travail a consisté à montrer qu'il était possible de spécifier, puis de valider formellement et d'implémenter la synchronisation dans un tel système en utilisant les réseaux de Petri. Valider, consiste principalement à vérifier de façon rigoureuse, l'absence de blocage dans le comportement du système étudié ainsi que la bonne spécification des contraintes qui le définissent en utilisant les réseaux de Petri.

En fait, nous ne nous sommes attachés qu'au problème du maintien de l'intégrité des données de la mémoire partagée puisque les autres aspects de la synchronisation dépendent de l'application spécifique considérée.

L'organisation de ce mémoire est la suivante :

Dans le premier chapitre, nous présenterons rapidement les principaux outils qui permettent de spécifier et d'implémenter les mécanismes de synchronisation dans un système monoprocesseur puis multiprocesseur. Nous définirons un critère qui nous permettra de classer quelques exemples de systèmes déjà étudiés, et nous nous appuyerons sur ces systèmes pour présenter la structure et le fonctionnement de la solution retenue. En particulier, nous décrirons les points importants liés à la sûreté de fonctionnement de ce système.

Dans le deuxième chapitre, nous présenterons l'outil que nous utiliserons pour spécifier les mécanismes de synchronisation ; les réseaux de Petri. Après les avoir définis, et montré leurs règles d'évolution, nous détaillerons les diverses méthodes d'analyse qu'ils autorisent. Toutefois, cet outil général est mal adapté pour représenter des phénomènes particuliers tels que des contraintes de sécurité. Aussi, nous proposerons un modèle dérivé que nous appellerons réseaux de Petri à jetons individualisés. Pour ces réseaux, et pour l'utilisation que nous en ferons, nous proposerons une méthode d'analyse s'appuyant sur celle des réseaux de Petri à jetons banalisés.

Dans le troisième chapitre, nous décrirons l'algorithme de la synchronisation du système présenté au chapitre premier. Pour le présenter, nous suivrons la méthodologie suivante :
Tout d'abord, nous présentons les contraintes à remplir, ensuite nous spécifions ces contraintes par réseaux de Petri (à jetons individualisés), et enfin nous vérifions, à l'aide des résultats procurés par un système d'aide à la preuve automatique, d'une part que ces réseaux sont sans blocage, et d'autre part que les contraintes sont bien respectées.

Pour faciliter la compréhension de cet algorithme, nous considérerons dans un premier temps que le milieu environnant est non défaillant.

Puis nous envisagerons les pannes des éléments composant ce système, et nous analyserons les conséquences qu'elles ont pour l'algorithme.

Dans le quatrième chapitre, nous proposerons une mise en oeuvre de cet algorithme directement à partir des réseaux de Petri à jetons individualisés.

La solution retenue, simule le réseau en fonction des messages reçus par le processeur de synchronisation. Comme le temps de réponse est un facteur important, nous avons essayé d'optimiser cette simulation en fonction des évolutions possibles du réseau. En fait, nous utilisons les messages reçus par le processeur de synchronisation pour déterminer les tirs successifs des transitions à accomplir sur le réseau.

C H A P I T R E I

- - - -

OUTILS DE SYNCHRONISATION ET STRUCTURES

- -

I. GENERALITES

L'informatique a été marquée ces dernières années par l'apparition des microprocesseurs sur le marché international. Ces micro-calculateurs ont permis d'agrandir les secteurs d'application de l'informatique, en particulier celui de la conception de systèmes de commande et de surveillance en temps réel de processus industriels, car ils permettent des architectures plus complexes tout en restant bon marché.

Ces sytèmes de commande sont généralement décomposables fonctionnellement en tâches, souvent indépendantes les unes des autres, et donc pouvant évoluer de façons silmutanées.

Ces systèmes possèdent dans la plupart des cas des structures modulaires composées de processeurs se partageant des ressources communes. Ainsi dans leurs utilisations et par leur conception, le degré de parallélisme est beaucoup plus grand pour de tels systèmes que pour ceux réalisés par des monoprocesseurs multiprogrammés. Ils ont donc de meilleures performances surtout en ce qui concerne le temps de réponse et la disponibilité.

De plus, il est à noter, que la modularité qui caractérise certains systèmes, autorise leur extensibilité et leur fonctionnement en mode dégradé. Ces propriétés sont importantes pour classer le grand nombre de systèmes multiprocesseurs ou multimicroprocesseurs ayant fait l'objet d'une étude. Cependant, d'autres critères de classement doivent être pris en compte tels que spécificité ou universalité du système, distribution locale ou géographique, sécurité et fiabilité ....
En fait, ces caractéristiques déterminent les possibilités et les domaines d'applications [MAZ 78] des systèmes. Pour ces raisons, nous allons définir les grandes lignes du système que nous voulons étudier.

Cette étude sera celle d'un système temps réel de commande de processus industriels, à évolutions parallèles sûr de fonctionnement et localement distribué. Nous considérons que ce système est formé d'un ensemble de calculateurs (ou processeurs) banalisés du point de vue matériel.

Par contre chaque tâche à effectuer est affectée a priori a un processeur donné. La mémoire partagée n'est utilisée que pour stocker les données communes à plusieurs tâches, chaque processeur ayant sa mémoire privée.

Notre travail est centré sur le problème de la spécification de l'analyse et de l'implémentation de la synchronisation d'un tel système. Aussi dans le prochain paragraphe, nous récapitulons les principaux outils qui permettent d'implémenter la synchronisation. Nous considérons d'abord le cas des systèmes monoprocesseurs multiprogrammés car le problème de la synchronisation y a déjà été abondamment traité, puis nous abordons le cas des systèmes multiprocesseurs, et les nouveaux problèmes qu'ils impliquent. Un critère de classement peut alors être défini pour ces systèmes qui est fonction de leurs structures et de leurs modes de fonctionnement.

Suivant ce critère, nous donnons quelques exemples de systèmes dans le quatrième paragraphe, avant de présenter dans le dernier, la structure et le principe de fonctionnement du système que nous avons retenu.

2. *MODELISATION ET IMPLEMENTATION DE LA SYNCHRONISATION*

2.1. *Systèmes monoprocesseurs multiprogrammés :*

Dans les systèmes monoprocesseurs multiprogrammés, plusieurs programmes se partagent des ressources (unité centrale, mémoire, périphériques....). L'indépendance de ces programmes, c'est-à-dire des processus séquentiels qu'ils engendrent, permet qu'ils s'exécutent virtuellement en parallèle dans le système.

Mais , la mission à remplir par le système (commande d'un procédé par exemple), impose une certaine coopérative entre ces processus.

Ainsi un système monoprocesseur multiprogrammé est à considérer comme un ensemble de processus séquentiels coopérants et concurrents qui s'exécutent en parallèle [MUN 78].

Cette coopération et cette concurrence, généralement appelée synchronisation inter-tâches, est réalisée à l'aide de variables manipulées par les processus séquentiels. Toutefois, cette manipulation doit se faire en prenant quelques précautions car un processus peut toujours être suspendu entre deux accès pour être remplacé par un autre plus prioritaire. C'est pourquoi, l'emploi de primitives spécialisées manipulant les variables de synchronisation est nécessaire.

Plusieurs outils ont été créés pour assurer la protection des variables de synchronisation. Les plus connus et utilisés sont décrits dans ce qui suit.

2.1.1. *"TEST AND SET"*

C'est la primitive de base. Elle est composée de deux accès successifs à la variable de synchronisation considérée, l'un pour tester et l'autre pour la modifier éventuellement. Evidemment, l'ensemble formé par ces deux accès est rigoureusement indivisible.

Son utilisation est très simple ; quand un processus veut une ressource, il teste la variable spécifiant l'état de la ressource. Si la ressource est libre, c'est-à-dire si le test est positif, il change la variable pour spécifier que la ressource est prise; si le test est négatif, la demande faite par le processus est rejetée.

Une grosse lacune de cette primitive est son incapacité à gérer une file d'attente. Ceci introduit des demandes d'accès répétitives quand un processus est en attente (attente active) ce qui risque d'engorger le système. Aussi par son manque de performance, cette primitive reste un outil élémentaire, seulement utilisé dans le cas de problèmes de synchronisation très simples.

2.1.2. *Les sémaphores* [DIJ 68] .

Digkstra est l'un des premiers à avoir pensé à regrouper dans une primitive le test de la variable de synchronisation et la gestion de la file d'attente correspondante.

Pour cela, le concept de sémaphore a été créé. Un sémaphore est formé d'un compteur S dont la valeur initiale est S_o, d'une file d'attente et des deux primitives suivantes:

```
P(S) ;
début  S : = S - 1 ;
       si S<O  alors "mise en file d'attente" ;
fin ;

V(S) ;
début S : = S + 1 ;
       si S⩽O  alors "réveiller un processus en file d'attente".
fin ;
```

Plusieurs exemples [CRO 73] ont mis en valeur l'utilisation de ces primitives. L'un d'eux a plus particulièrement retenu notre attention c'est le problème des "lecteurs écrivains" [COU 71], devenu depuis très célèbre, et dont nous nous servons tout au long de ce travail.

Cependant même avec l'évolution que présente ces primitives par rapport à la primitive "Test and Set", elles sont difficiles à mettre en oeuvre pour des problèmes de grande complexité. Ceci est dû à leur manque de structuration lorsque de nombreuses variables de synchronisation interviennent simultanément.

Ces difficultés sont alors résolues par l'introduction du concept de moniteur.

2.1.3. Moniteurs spécialisés :

Le premier de ces moniteurs [HOA 74] est une extension des régions critiques conditionnelles [HOA 72 - BRI 72] .

Un moniteur est à considérer comme un ensemble de données et de procédures qui gère un problème de synchronisation. Ainsi, un processus voulant effectuer une opération sur la mémoire partagée, devra auparavant exécuter la procédure du moniteur y correspondant.

Seuls les noms des procédures du moniteur sont connus du monde extérieur (processus), et elles se déroulent en exclusion mutuelle. L'organisation générale d'un moniteur est la suivante : l'ensemble des conditions, pour l'acquisition d'une ressource, est déclaré à l'entrée du moniteur. Ces conditions sont utilisées dans le déroulement de certaines procédures par l'intermédiaire d'une primitive propre au moniteur, "WAIT". L'exécution de cette primitive implique alors l'évaluation de la condition associée ; si elle est vraie la procédure continue, si elle est fausse le processus est mis dans la file d'attente correspondant à cette procédure. Ces processus seront alors réveillés par l'exécution de certaines procédures qui rendront vraie la condition associée à leur passage.

Une critique de ce moniteur peut cependant être faite, car le concepteur doit prévoir, analyser et implanter la gestion des réveils des processus en attente. Ce délicat travail peut être la source de nombreuses erreurs. Aussi il a été imaginé un type de moniteur [KES 77] dont le principe de fonctionnement diffère en ce qui concerne la gestion de ces réveils. La stratégie retenue est d'examiner une à une toutes les files d'attente . Quand l'une d'elles est non vide, la condition associée à cette file d'attente est reévaluée pour voir si le processus peut être réveillé.

Les implémentations suggérées par les auteurs de ces moniteurs sont réalisées à l'aide des sémaphores. On peut donc considérer le concept de moniteur comme un moyen de structurer une synchronisation implantée par sémaphores.

Par l'emploi de tous ces outils, dans l'élaboration de la synchronisation, le concepteur n'a aucun moyen pour valider facilement, et montrer formellement l'absence de blocage mortel. Si pour certains systèmes, la validation de la synchronisation s'avère nécessaire, l'utilisation des réseaux de Petri est séduisante. En effet, ces derniers sont des outils formels de spécification dont le support mathématique permet une analyse fine.

2.1.4. Les réseaux de Petri :

Les réseaux de Petri permettent une méthodologie rigoureuse de conception de mécanismes de synchronisation.

La spécification basée sur un graphe peut être validée en grande partie d'une manière automatique.

Les réseaux de Petri seront présentés de façon formelle au chapitre II et seront utilisés dans la suite de ce travail pour la spécification et la validation du mécanisme de synchronisation du système étudié.

Toutefois, cet outil n'est pas à opposer au concept de moniteur puisqu'un réseau modélisant un mécanisme d'allocation de ressource peut très aisément être implémenté sous cette forme [VAL 79]. Il faut plutôt y voir une complémentarité; l'un étant plus orienté vers l'analyse et la preuve, l'autre vers la facilité d'implantation.

2.2. *Systèmes multiprocesseurs* :

A l'inverse des systèmes monoprocesseurs, où l'on est sûr qu'à un instant donné un seul processus est actif (une instruction est en cours d'exécution), dans les systèmes multiprocesseurs cette hypothèse ne peut plus être faite. En effet pour ces systèmes le parallèlisme existe à des niveaux beaucoup plus fin ; il n'est plus apparent mais bien réel.

Aussi, toutes les primitives de synchronisation définies dans le cadre des systèmes monoprocesseurs sont à remettre en cause. Toutefois, l'utilisation des réseaux de Petri pour la spécification et la validation n'est, elle, par contre pas affectée [AZE 80] les difficultés n'apparaissant qu'au moment de l'implémentation.

Deux classes de systèmes multiprocesseurs peuvent être rencontrées ; les systèmes qui utilisent une mémoire commune pour stocker les variables de synchronisation, et ceux où l'ensemble de ces variables est réparti sur tous les "processeurs".

2.2.1. *Variables centralisées* :

En fait, pour cette classe de systèmes, utilisant les services d'une mémoire commune pour stocker les variables de synchronisation, l'unicité du processeur touchant ces variables est très facile à réaliser

par le contrôle des accès à cette mémoire. Bien sûr, le besoin de protéger ces variables est tout aussi pressant (sinon plus), et l'emploi des primitives précédemment présentées est obligatoire pour les manipuler.

Plusieurs solutions sont possibles pour réaliser le partage de l'accès mémoire entre plusieurs processeurs, deux sont détaillées dans le paragraphe 4.1.1. de ce chapitre.

2.2.2. *Variables distribuées* :

Pour les systèmes utilisant ce mode d'implémentation des variables de synchronisation, une redéfinition de leur composition s'impose. Généralement, ces systèmes sont constitués de plusieurs sites communiquant entre eux, chaque site étant à considérer comme un mini (ou micro) calculateur, c'est-à-dire qu'il est formé schématiquement d'un "processeur", d'un "moniteur", et de la "mémoire". Les informations, permettant de réaliser la synchronisation du système, transitent par le support de communication.

Donc pour cette classe de système il faut définir un nouvel ensemble d'outils qui permet que cette communication se déroule suivant une certaine norme. Aussi plusieurs outils ou concept ont été développés pour aider à cette implémentation dans un système complétement réparti.

Parmi ces outils, citons les principaux, en commençant par les compteurs d'événements [REE 79] : variables entières non décroissantes, où une seule opération (incrémentation) peut être faite par l'intermédiaire d'une primitive "augmenter la variable E de 1". Deux autres primitives sont utilisées pour réaliser la synchronisation ; la primitive "lire la variable E" et la primitive "suspendre le processus jusqu'à ce que la variable atteigne la valeur n".

Rien n'empêche pour ce genre d'outil que plusieurs primitives soient exécutées simultanément, cela permet donc un plus grand parallélisme, mais par contre, tous les problèmes de synchronisation ne sont pas résolubles ainsi.

En particulier, c'est le cas de la résolution des conflits dans un mécanisme d'allocation de ressources. L'emploi d'autres stratégies est alors obligatoire et diverses solutions sont possibles, telle la méthode utilisant des horloges logiques [LAM 78] incrémentées après chaque événement. Les problèmes d'exclusion, pour l'allocation de ressources, sont alors résolus par l'envoi de messages entre sites qui recalent sur chacun d'eux l'horloge logique. Par la lecture de son horloge, le site sait s'il peut ou non utiliser la ressource. D'autres méthodes ont été définies par exemple, celle de l'ordonnancement des allocations par séquenceur circulant [DIJ 74 - VER 77] ou non [BAN 79].

3. *PROBLEMES LIES A L'IMPLEMENTATION DES DONNEES*

3.1. *Aspect structurel :*

En fait, dans chaque système on peut différencier deux types de données ; d'une part celles se rapportant à la synchronisation et qui sont manipulées par les processeurs pour contrôler s'ils peuvent utiliser une ressource (nous les appelons variables de synchronisation), et d'autre part celles qui constituent une ressource propre du système et qui sont utilisées par les processeurs pour l'exécution d'un processus (nous les appelons "données d'intérêt général").

Pour ces deux ensembles de données, deux manières de les implémenter sont possibles : centralisée ou répartie. Bien que nous ayons vu les problèmes liés à ces deux types d'implémentation pour les variables de synchronisation, nous allons toutefois résumer les précautions qu'elles demandent avant d'aborder le cas des données d'intérêt général.

3.1.1. *Variables de synchronisation :*

Dans le cas d'une implémentation centralisée, le système correspondant diffère très peu dans son principe de fonctionnement des systèmes monoprocesseur, et l'exclusivité de l'accès aux variables de synchronisation est facilement résoluble.

En ce qui concerne une implémentation répartie, le principe de fonctionnement du système repose sur un échange de message entre "processeurs". Ces échanges doivent alors obéir à certains protocoles définis par les contraintes de synchronisation du problème traité.

3.1.2. Données d'intérêt général :

Implanter ces données de manière centralisée sur une mémoire commune, n'implique rien de très particulier sinon la résolution des conflits d'accès des processeurs à la mémoire.

Par contre, distribuer les données sur les différents processeurs amène à plusieurs autres problèmes. En effet, si la distribution des données présente le grand avantage de rendre les processeurs autonomes, (on parle alors de sites), par contre, quand l'un d'eux ne possède pas les données nécessaires, il doit les rechercher sur les autres sites.
Ceci implique que le concepteur doit définir des facteurs tels que :

I) le nombre d'accès aux données d'une mémoire à partir des autres sites

II) le trafic entraîné sur le support de communications par ces accès,

III) la taille de la mémoire sur chaque site,

IV) le moyen rapide de localiser une donnée non contenue dans sa mémoire

pour optimiser la place mémoire maximale sur chaque site, et le temps de réponse du système.

3.1.3. Unicité ou duplication des données :

Pour les systèmes multiprocesseurs, les données communes sont soit uniques pour tous le système, soit "n-pliquées".

L'avantage de "n-pliquer" les données est d'augmenter la disponibilité du système en lui permettant de continuer à fonctionner en cas de panne d'un des exemplaires.

L'introduction de cette "duplication" (au sens large du terme) amène de nouveaux problèmes dans l'élaboration de la synchronisation car il est nécessaire que toutes les copies possèdent exactement les mêmes informations. Ces problèmes de cohérence ont fait l'objet de nombreuses recherches qui à notre connaissance, se rapportent toutes au maintien de la cohérence de données dupliquées dans des bases de données réparties [ELL 77 - MIR 79 - ...] .

Avant d'examiner les problèmes de base nous allons rappeler quelques notions élémentaires :

Un accès à une mémoire est une opération indivisible. Il peut être de deux types : lecture ou écriture. Le premier consiste en une prise d'informations, alors que le second modifie le contenu de la mémoire.

Une transaction est une suite d'accès vérifiant certaines propriétés dont principalement l'atomicité. Deux types sont différenciés : les transactions lecture , formées par un ensemble d'accès lecture et uniquement lecture, et les transactions écriture , formées d'une suite d'accès dont au moins l'un d'eux est un accès écriture.

Une transaction lecture ne modifie pas l'état de la mémoire, mais pour qu'elle s'effectue correctement, cet état ne doit pas être modifié pendant son exécution.

Une transaction écriture est telle que l'ensemble des mémoires ne peut être uniforme à un instant donné, (c'est-à-dire que l'ensemble des mémoires peut passer par des états incohérents), cependant à sa fin, l'état de cohérence doit être retrouvé. Aussi pour qu'elle soit exécutée correctement aucun autre accès de type écriture en dehors de cette transaction et se rapportant aux données modifiées par elle, ne doit s'effectuer dans le système.

Nous voyons donc qu'il doit y avoir exclusion mutuelle entre les transactions de type écriture et de type lecture, ainsi qu'entre les transactions de type écriture elles-mêmes.

Par contre plusieurs transactions lecture peuvent s'exécuter en parallèle.

Ces quelques remarques introduisent les problèmes de synchronisation auxquels nous sommes confrontés quand on oborde les systèmes à données dupliquées.

3.2. *Aspect fonctionnel* :

L'algorithme de synchronisation peut être centralisé ou distribué. A priori un algorithme distribué semble plus fiable pour le système qu'un algorithme centralisé, cependant nous verrons que dans certains cas, cela peut être le contraire.

4. *QUELQUES SOLUTIONS POUR UN SYSTEME MULTIPROCESSEUR*

4.1. *Données centralisées (mémoire commune partagée)*

4.1.1. *Algorithme de synchronisation réparti.*

Un exemple de système basé sur ce principe a été décrit dans [ADA 78] . Il s'agit d'une structure composée de plusieurs microprocesseurs se partageant une mémoire commune.

Les processeurs sont liés à la mémoire commune par l'intermédiaire d'un bus. Pour résoudre les problèmes d'accès au bus, deux solutions sont possibles. L'une (figure 1a) consiste à relier les processeurs entre eux par une connexion en guirlande "Daisy chain" avec tous les inconvénients qu'elle entraîne (rigidité, fragilité), l'autre (figure 1b) de les connecter parallèlement à un arbitre qui détermine en cas de conflits le plus prioritaire.

La protection des variables de synchronisation est assurée par l'intermédiaire de primitives du type "Test and Set", et aucune gestion des files d'attente n'est prévue. L'attente des processeurs est donc active (répétition de la demande).

Ce type de système, s'il possède le grand avantage d'être commercialisé, présente toutefois de graves lacunes.

La première est la possibilité "d'engorgement du Bus" ("bottle neck") dans le cas de nombreuses demandes d'accès aux données. Ce type de système n'est donc intéressant que pour des applications où la mémoire sert de "boîte aux lettres" entre les processeurs, et où les échanges sont limités.

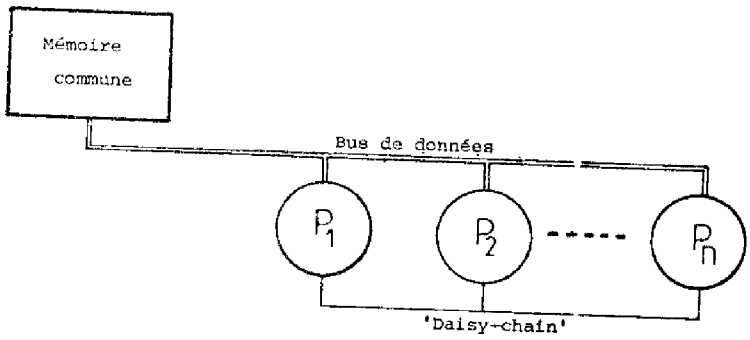


Figure 1a.

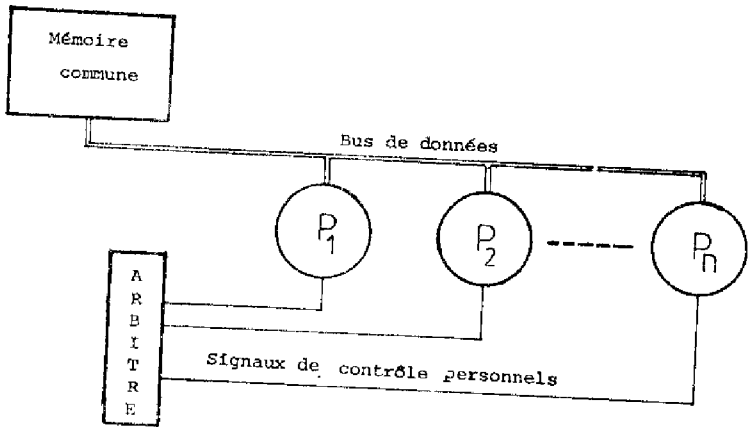


Figure 1b.

Une autre lacune apparaît quand on considère la sûreté de fonctionnement et la fiabilité de ce système. En ce qui concerne la sûreté, rien n'empêche à un processeur dans un fonctionnement anormal de détruire la synchronisation par des changements de variables incorrects. De plus la disponibilité du système repose sur le bon fonctionnement de la mémoire commune, or nous savons que ces éléments sont parmi les plus fragiles.

4.1.2. *Algorithme de synchronisation centralisée* :

Bien qu'à priori une telle configuration paraisse très mauvaise pour la fiabilité du système, nous allons montrer sur une structure particulière, entièrement centralisée, et avec un mode de fonctionnement adapté, qu'elle ne présente pas de plus graves inconvénients qu'une synchronisation répartie.

L'exemple [LAG 76] dont nous nous servons est le système représenté figure I_2. Cette structure est composée de plusieurs processeurs de constitution strictement identique, reliés à une mémoire commune. Dans leur fonctionnement, deux des processeurs sont spécialisés, ils sont appelés régisseur (R) et secrétaire (S).

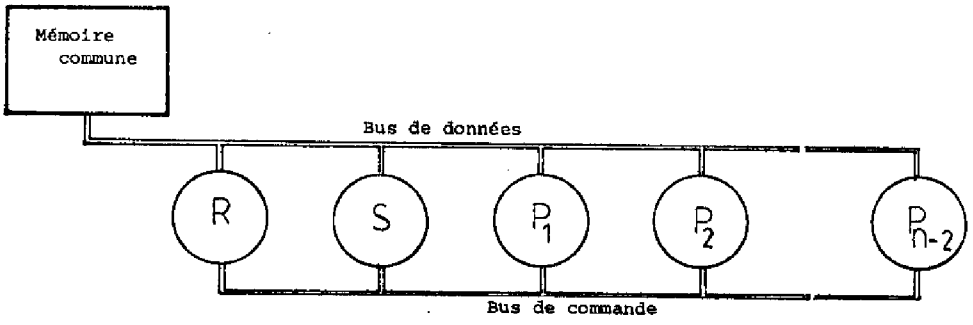


Figure I_2.

Le régisseur synchronise les travaux faits par les processeurs banalisés, et assure le partage de la mémoire entre eux.

Pour cela un dialogue s'établit entre le régisseur et les autres processeurs par l'intermédaire du bus de commande.

Le secrétaire lui, s'occupe de gérer principalement l'allocation du bus de données.
L'originalité de ce système est qu'en cas de panne d'un des processeurs spécialisés, l'autre élit automatiquement un processeur banalisé pour qu'il tienne le rôle du processeur en panne. Les tâches, que remplissait ce processeur banalisé, étant elles distribuées sur les autres processeurs.

L'avantage que possède ce système par rapport au précédent est qu'en cas de panne sur l'élément contenant les variables de synchronisation, une reconfiguration est possible par l'élection d'un nouveau régisseur qui reconstitue les valeurs de ces variables. Donc, fait très important, la vie du système n'est pas mise en danger par la centralisation de l'algorithme de synchronisation, pour peu que l'on tienne compte de la défaillance de l'organe qui le supporte.

En fait le plus gros défaut de ces systèmes, si les contraintes de fiabilité et de sûreté sont primordiales, provient de l'unicité de la mémoire de données d'intérêt général. Si cette mémoire est défaillante tout le système est en panne.

4.2. *Systèmes à données dupliquées et réparties* :

Une solution pour améliorer la fiabilité des systèmes est alors de "dupliquer" les données sur différentes mémoires, et de permettre au système de continuer à fonctionner en cas de panne d'une mémoire.

Nous avons déjà signalé dans un précédent paragraphe les nouveaux problèmes introduits par la duplication des données. Un système de gestion de base de données réparties est un exemple typique de système distribué avec données dupliquées (n-pliquées) et réparties. A cause de l'importance du sujet, de nombreuses solutions ont été proposées suivant les caractéristiques propres des applications considérées [THO 80 - SER 79 - MUL 75 ...]

Une étude comparative [WIL 79 - WIL 80] en a été faite pour dégager leurs principaux domaines d'application à la vue de leurs caractéristiques.

Comme hypothèse de base, ils sont tous bâtis sur la même architecture de communication sans plus de précision sur leur structure matérielle.

Dans leurs principes, ces algorithmes fonctionnent généralement sur deux pas de synchronisation ; un premier pour s'assurer qu'un site peut exécuter une transaction, et un second où la transaction est lancée.

Deux genres de fonctionnement, cependant, sont usuellement rencontrés : un considérant que les messages entre sites transitent pas à pas sur un anneau virtuel, l'autre au contraire, que tous les messages sont diffusés en parallèle d'un site vers tous les autres.

Un exemple classique de fonctionnement de la transmission pas à pas sur un anneau virtuel est l'algorithme d'ELLIS [ELL 77], où l'on considère que chaque processeur est relié à deux voisins sur l'anneau : un prédécesseur et un successeur.

Quand un site veut effectuer une transaction (autre qu'une transaction lecture sur ses propres données) il lance une requête sur l'anneau en direction de son successeur. Celui-ci l'examine, et dans le cas d'une réponse favorable, il la transmet à son successeur après s'être mis dans l'état passif. Cette requête transite ainsi de site en site jusqu'à ce qu'elle revienne au site émetteur (donc le seul actif). Ce site envoie , à ce moment là, la transaction à son site successeur qui aprés son exécution, la transmet à son successeur et redevient libre.

Dans la première étape de synchronisation, c'est-à-dire au cours de la recherche par un site, de son droit de passage, il peut y avoir plusieurs sites actifs sur l'anneau. Cela se traduit pour chacun par la circulation d'une requête propre.

Or chacune va rencontrer un site actif qui ne sera pas celui qui l'aura émise, et créer un conflit sur ce site. Il est résolu alors par priorité fixe ; si la requête est plus prioritaire, elle est transmise à son successeur, et le site devient passif ; si elle est moins prioritaire, elle est stoppée et mise en file d'attente.

Cet algorithme a été conçu pour un environnement fiable. Cependant une solution pour reconfigurer l'anneau en cas de défaillance d'un site peut être réalisé à partir de la solution proposée par [LEL 78] où il est décrit un mécanisme de suspicion pour détecter les sites en panne. Quand un est détecté, il est isolé de l'anneau et celui-ci est reconfiguré en établissant une liaison entre les deux sites encadrant le site en panne. A l'inverse un site peut réintégrer le système quand il le désire en établissant des liaisons avec ses voisins immédiats.

En fait, si tous ces algorithmes sont très intéressants pour s'affranchir de divers problèmes liés à la cohérence de données partagées dupliquées, ils paraissent lourds à mettre en oeuvre pour une application distribuée localement. Aussi il nous a paru bon de nous intéresser à une solution centralisée.

4.3. Conclusion :

Avant de tirer quelques enseignements des exemples de systèmes précédents, nous allons rappeler le cadre de notre étude. Nous voulons élaborer un système multiprocesseur pour la commande de processus industriels. Pour ce système, les contraintes de sûreté de fonctionnement, disponibilité, et temps réel sont les plus importantes. De plus nous considérons des installations occupant une surface au sol relativement faible (type unité pilote par exemple) de telle sorte qu'une distribution géographique de la commande n'est pas nécessaire.

Donc pour respecter ces contraintes, il est judicieux de concevoir un système possédant un maximum de redondance matérielle, et ayant un temps de réponse le plus court possible. Aussi, les solutions présentées dans les exemples ne nous satisfont pas pleinement pour les raisons suivantes.

En ce qui concerne les algorithmes de maintien de la cohérence dans les bases de données réparties et dupliquées, le défaut est leur temps de réponse. Pour les autres systèmes, c'est la sûreté de fonctionnement qui est en cause avec, pour le système où les mécanismes de synchronisation sont répartis sur l'ensemble des processeurs [ADA 77] , un défaut supplémentaire celui de l'attente active en cas d'insuccès pour l'acquisition de la ressource. La solution centralisée [LAG 76] , permet de s'affranchir de ce problème tout en restant aussi fiable.

De plus, bien que cela ne soit pas fait pour ce système, nous pensons qu'une solution centralisée facilite la protection, et le filtrage des erreurs provenant des processeurs (vis à vis de la synchronisation) par rapport à une solution distribuée. Cependant, il est nécessaire dans tous les cas de prévoir la panne des éléments centraux afin que le système continue à fonctionner.

5. *SYSTEMES PASTELS*

5.1. *Présentation générale*

5.1.1. *Introduction :*

La structure générale de ce système [COU 80] est celle de la figure I_3 . Pour présenter son principe de fonctionnement, nous supposerons dans un premier temps qu'elle est composée de n processeurs, de deux mémoires dont les accès élémentaires sont contrôlés par des processeurs d'arbitrage, et d'un processeur de synchronisation. Pour des raisons de sûreté de fonctionnement, ce processeur sera doublé ultérieurement.

L'allocation de la mémoire est traitée à deux niveaux :

- demande d'autorisation d'accès gérée par le processeur de synchronisation au début d'une transaction,

- demande d'accès élémentaire gérée par les processeurs d'arbitrage.

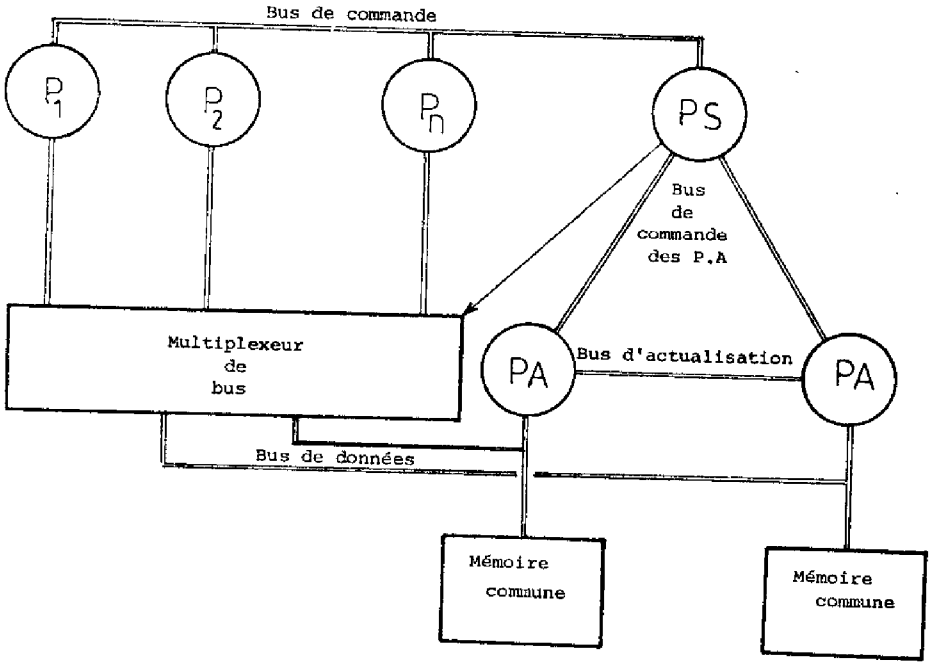


Figure I3.

5.1.2. Fonctionnement normal :

Chaque fois qu'un processeur veut utiliser les données contenues dans une des deux mémoires, (n'importe laquelle puisque leur contenu est identique), il fait une demande de transaction au processeur de synchronisation. Cet échange se fait au moyen d'un bus parallèle comprenant une partie données commune à tous les processeurs et une partie formée de signaux de contrôle (demande, acquittement) personnelle à chaque processeur.

Le processeur de synchronisation dans un fonctionnement normal (sans conflit de demande), permet au processeur questionneur de formuler sa demande. Ce processeur adresse alors sa requête sous la forme d'un message codé.

Le nombre de messages différents étant très restreint, une procédure de codage particulier est applicable afin de filtrer certaines erreurs de communication .

A la réception du message codé, le processeur de synchronisation le décode. La gestion des requêtes est faite selon l'algorithme de synchronisation (chapitre III) qui tiend compte des charges respectives des deux mémoires.

Si la transaction demandée ne peut être accordée, à cause par exemple, d'un manque de ressources, il met le processeur en attente sur la file correspondant à cette transaction, sachant que ce processeur en attente sera réveillé dès que la transaction pourra se dérouler.

Si la transaction peut se dérouler, le processeur de synchronisation informe les divers éléments concernés, c'est-à-dire : le processeur arbitre de la mémoire allouée en lui signalant le nom du processeur utilisateur, et la nature de la transaction qu'il veut effectuer; le multiplexeur de bus pour établir la connexion entre le processeur utilisateur et la mémoire concernée; et enfin, le processeur utilisateur.

Celui-ci, chaque fois qu'il veut accéder à la mémoire, envoie une requête au moyen d'un signal individuel en direction des mémoires. Ces accès à une mémoire sont transparents aux utilisateurs, c'est-à-dire qu'ils ne savent pas sur quelle mémoire ils travaillent. L'aiguillage se fait d'une part au niveau du multiplexeur lors de la connexion du processeur utilisateur à la mémoire, et d'autre part au niveau des processeurs arbitres par leur connaissance des processeurs pouvant accéder à la mémoire.

Donc à la réception d'un requête d'accès, le processeur arbitre autorise l'accès au moyen d'un signal d'acquittement individuel. Le processeur arbitre examine la nature des accès faits par l'utilisateur et contrôle qu'elle soit bien identique à celle signalée par le processeur de synchronisation.

Quand un processeur a fini une transaction, il le signale au processeur de synchronisation au moyen d'un message codé et de façon identique à la procédure qu'il avait suivi pour demander la transaction.

Quand une transaction écriture est accomplie, les transformations des données faites sur une mémoire sont répercutées sur l'autre à partir d'une mémoire tampon (ou "buffer"), interne au processeur arbitre, et au moyen d'un bus d'actualisation reliant les deux ensembles mémoire .

Bien sûr, ces mises à jour sont pilotées automatiquement par le processeur de synchronisation, quand toutes les contraintes,pour qu'elles puissent s'exécuter, sont remplies.

5.1.3. *Fonctionnement en présence de conflits*

Deux sortes de conflits peuvent être rencontrés dans ce système. L'un au niveau des accès au processeur de synchronisation quand un processeur signale un début ou une fin de transaction, et l'autre au niveau des accès élémentaires sur une mémoire quand plusieurs processeurs s'en partagent l'utilisation.

Dans les deux cas, une même procédure classique pour résoudre ces conflits est appliquée ; chaque processeur se voit fixer un degré de priorité, en cas de conflit, le processeur ayant le plus fort degré est sélectionné, les autres restent en attente pour une sélection ultérieure.

5.2. *Rappel du rôle de chaque élément*

5.2.1. *Processeurs utilisateurs : (figure I_4)*

Ce sont eux qui sont chargés d'exécuter les processus. Pour un accès mémoire le séquencement des actions qu'ils ont à accomplir est le suivant.

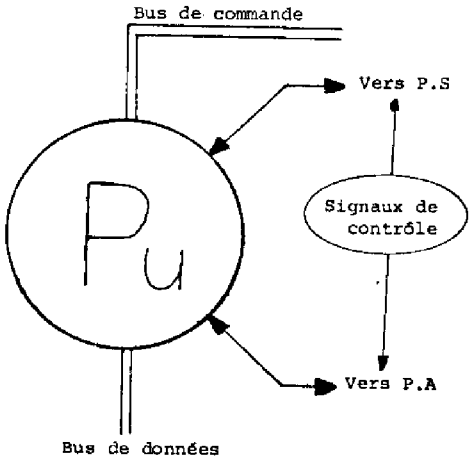


Figure I_4.

Envoyer une requête au processeur de synchronisation, attendre son acquittement, formuler la nature de l'utilisation qu'il veut faire de la mémoire, attendre l'accord du processeur de synchronisation. Chaque fois que ce processeur voudra faire un accès mémoire élémentaire, il devra envoyer un signal en direction des mémoires, et attendre l'accord d'un processeur d'arbitrage. A la fin d'une session d'utilisation, il faudra qu'il le signale au processeur de synchronisation.

Bien sûr, pendant toutes les attentes de réponse, chacun d'eux peut effectuer d'autres tâches.

5.2.2. Processeurs arbitres :

Ils ont trois fonctions principales :

I) dialoguer avec le PS, soit quand celui-ci lui envoie des informations sur l'identité des occupants de sa mémoire, soit pour faire une actualisation

II) gérer les accès des processeurs utilisateurs que PS lui a signalé, et surveiller leurs communications

III) effectuer les mises à jour sur l'autre mémoire par l'intermédiaire de sa mémoire tampon.

Ces fonctions définissent sa structure (figure I_5).

Elles sont assez indépendantes pour que leur déroulement se fasse en parallèle. Comme elles doivent être accomplies rapidement, le processeur d'arbitrage sera réalisé de façon microprogrammée.

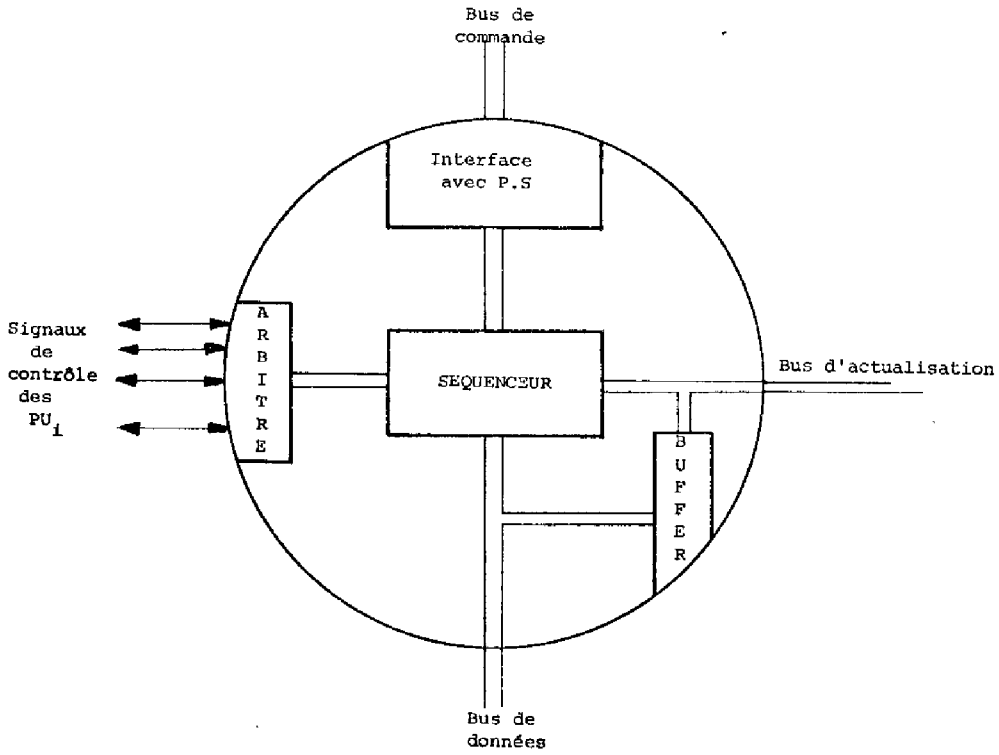


Figure I_5.

5.2.3. *Le processeur de synchronisation : (figure I_6)*

Il sera présenté plus en détail au chapitre IV. Sachons toutefois qu'il est l'organe de commande hiérarchiquement le plus haut. Il gére les échanges entre processeurs utilisateurs et mémoires. Pour cela, il communique avec tous les éléments du système. Lui seul connait l'état complet du système.

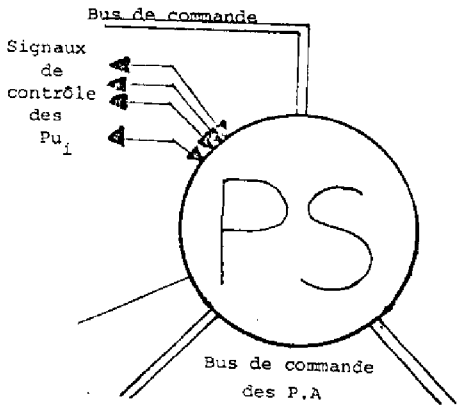


Figure I_6.

5.3. *Aspect sécurité et tolérance aux pannes*

5.3.1. *Sécurité* :

L'objectif est d'effectuer une détection en ligne des anomalies de fonctionnement et une localisation des pannes au niveau des modules, afin d'en réduire les conséquences.

Pour cela, chaque processeur analyse le dialogue qu'il a avec les processeurs qui lui sont liés, et peut ainsi détecter des incohérences de comportement dont les principales sont les suivantes :

- entre Pu et PS.
  - . défaut de réponse (dans les deux sens)
  - . demandes interdites ou erronées (dans les deux sens), réponses intempestives.

- entre Pu et PA
  - . pas d'utilisation mémoire après une demande
  - . utilisation prohibée (tentative d'écriture par un lecteur)

. accès intempestif
. accès abusif de la mémoire (en temps et quantité de données écrites)
. pas de liaison de données
. pas de réponse

- entre PS et PA

. information incorrecte
. manque de réponse
. réponse erronée.

Ces fautes sont détectées par les fonctions normales programmées dans les processeurs (redondance fonctionnelle logicielle) et quelques modules de temporisation pour le contrôle des défauts de réponse ou la détection d'une occupation abusive.

Pour chaque processeur, on implantera de plus des programmes assurant des fonctions locales de test qui seront déroulées pendant les temps morts.

Certaines pannes seront détectées au moyen de tests continus en ligne. Ceci nécessite des redondances aussi bien matérielles que logicielles telles que arbitres autotestables, code redondant...

La détection de panne se traduit par des signaux d'alarme qui sont transmis à l'organe de décision le plus élevé, le PS.

5.3.2. Tolérance aux pannes :

La fragilité de la structure provient du processeur de synchronisation. En effet, une panne de celui-ci entraîne un arrêt total du système.

Pour y remédier, une solution consiste à observer le comportement de PS en lui adjoignant un processeur qui a accès à toutes ses entrées-sorties. La structure devient alors celle de la figure I_7 .

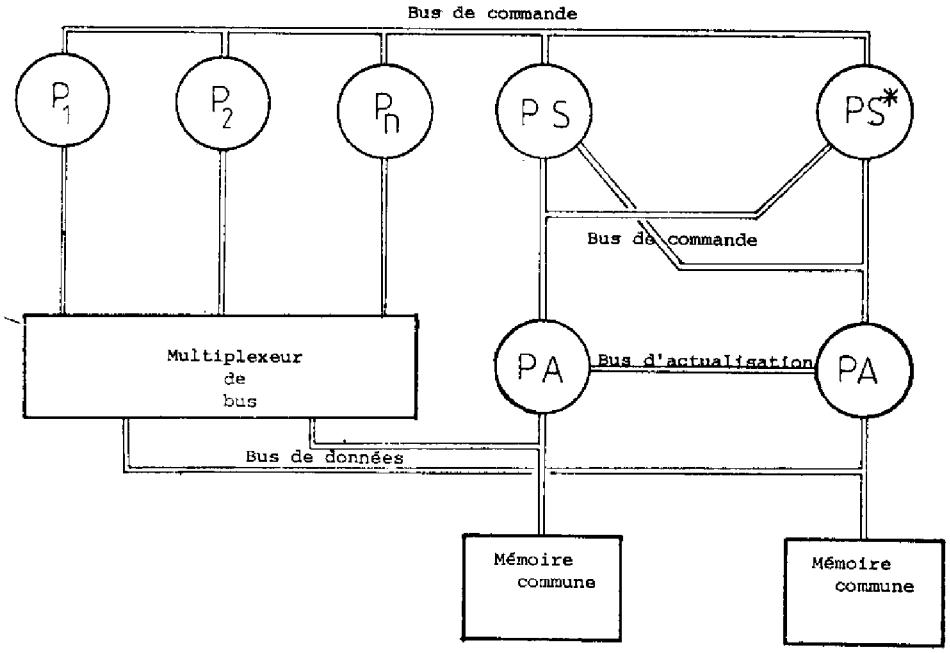


Figure I_7 .

Le processeur PS* ne participe pas au fonctionnement normal du système, et peut être testé hors ligne. Par contre, il peut être utilisé pour reconfigurer le système en cas de panne de PS, la tâche consistant alors à isoler le PS et à reprendre la gestion du système, éventuellement en mode dégradé sur une seule mémoire, après en avoir averti le monde extérieur.

Tous les problèmes de restauration de l'état du sytème lors d'une panne de PS par PS* doivent encore être appréhendés.

6. CONCLUSION

Nous venons de décrire un système multiprocesseur multimémoire pour la commande de processus industriels à haute sécurité.

Pour le bâtir, nous avons profité de l'expérience de quelques systèmes afin d'en tirer les avantages et les inconvénients.

La solution proposée, bien qu'encore incomplète dans son étude pour rendre le système tolérant aux pannes du "PS", met en oeuvre des procédures de redondance (surtout au niveau logiciel) pour garantir une bonne sécurité de fonctionnement.

A la vue de son principe de fonctionnement, il est facile de se rendre compte de la complexité de la synchronisation à remplir. Il paraît donc intéressant, sinon vital pour le système, de la prouver correcte. Pour cela, nous pensons que les réseaux de Petri sont très bien adaptés. Aussi, avant de rentrer plus dans les détails de cette synchronisation, nous allons définir cet outil de spécification.

CHAPITRE II

- - - -

RÉSEAUX DE PETRI

- - -

1. INTRODUCTION

Nous venons de montrer dans le chapitre précédent, que l'évolution technologique permet de développer des systèmes à fort degré de parallèlisme. Leur complexité de fonctionnement, en conséquence augmente aussi.

Au-delà d'un certain niveau de complexité, il est nécessaire de modéliser et d'analyser les mécanismes de synchronisation avant de les implémenter afin d'éviter toute erreur de conception. En fait la modélisation, riche en enseignements lors de la conception du système est également très importante comme outil de documentation pendant les phases de maintenance.

Beaucoup d'outils [PET 77-GSA 77-HOA 78-ROU 78...] ont été créés et utilisés pour spécifier les systèmes parallèles. Entre tous, nous avons retenu les réseaux de Petri dont plusieurs travaux [KOT 78-GIR 78-AGE 79-VAL 79] ont montré leur aptitude à représenter de façon formelle les spécifications des systèmes.

Le formalisme des réseaux de Petri a permis la mise en oeuvre de nombreuses propriétés et l'élaboration de méthodes d'analyse basées soit sur l'énumération des marquages [KAR 69 - BER 77], soit sur la structure elle-même du réseau [LAU 74 - LAU 75 - MEN 77 - BER 79]. De plus, plusieurs systèmes d'analyse automatique ont été créés [BOU 78 - MOA 79 - CHE 79], afin de valider ces réseaux.

Enfin il faut souligner l'existence d'outils dérivés des réseaux [CRI 77 - SCH 78 - GEN 79 - JEN 80] plus adaptés à représenter des phénomènes tels que priorité, surveillance...

Nous allons présenter dans les prochains paragraphes les réseaux de Petri, leurs règles d'évolutions et leurs propriétés. Une description rapide de leur analyse sera donnée, puis nous introduirons un modèle dérivé des réseaux de Petri, nous l'appellerons réseaux à jetons individualisés. Comme pour les réseaux de Petri, nous indiquerons leurs règles d'évolutions leurs propriétés, et nous traiterons de leur analyse.

2. *RESEAUX DE PETRI*

2.1. *Définition et représentation*

2.1.1. *Définition :*

Un réseau de Petri est un quintuplet $[P,T,E,S, M_o]$ où :

- $P = \{P_1, P_2 \ldots . P_i \ldots . P_n\}$ est un ensemble fini de place, il sert à caractériser l'état du système.

- $T = \{t_1, t_2 \ldots . t_m\}$ est un ensemble fini de transition, il sert à caractériser les changements d'état possibles du système.

- E est la fonction " places précédentes ". C'est une application de T X P dans N

$$E : T \times P \longrightarrow N$$

- S est la fonction " places suivantes ". C'est une application de T X P dans N

$$S : T \times P \longrightarrow N$$

- M_o est la fonction marquage initial. C'est une application de P dans N

$$M_o : p \longrightarrow N$$

où N est l'ensemble des entiers positifs ou nul.

M_o donne la distribution initiale des jetons dans les places et caractérise ainsi l'état initial du système.

2.1.2. Représentation :

La représentation classique des réseaux de Petri est donnée à la figure II_1.

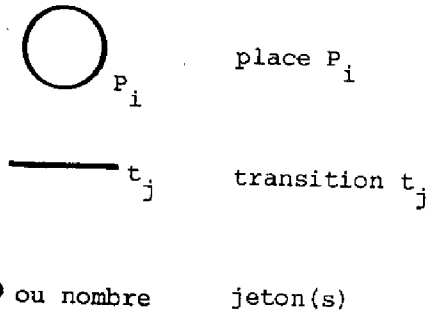


Figure II_1.

Les réseaux de Petri sont en fait des graphes orientés où les sommets sont les ensembles P et T (places, transitions) et les arcs qui lient ces sommets sont définis par les matrices E et S.

Il est important de noter qu'aucun arc ne peut relier deux sommets de même type.

$$\forall\, p_i,\ p_j \in P \Longrightarrow p_i \times p_j = \emptyset$$

$$\forall\, t_i,\ t_j \in T \Longrightarrow t_i \times t_j = \emptyset$$

Pour des sommets de types différents nous aurons :

Si : $E(t_j, p_i) = n$

signifie qu'un arc de poids n relie la place p_i à la transition t_j (figure II_{2a}) . $E(t_j, .)$ représentera le poids de tous les arcs entrant de la transition t_j.

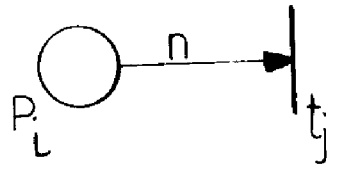


Figure II_{2a}.

SI : $S(t_i, p_j) = m$

signifie qu'un arc de poids m relie la transition t_i à la place p_j (figure II_{2b}) . $S(t_i, .)$ représentera le poids de tous les arcs sortant de la transition t_i.

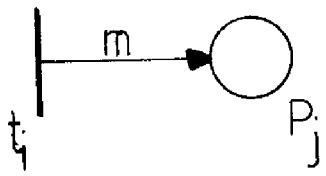


Figure II_{2b}

En fait cette représentation permet de modéliser de façon condensée les arcs multiples reliant deux sommets. Par convention le poids "zéro" représente l'absence d'arc entre sommets.

Un exemple de modélisation par réseaux de Petri est donné à la figure II_3. Ce réseau spécifie le mécanisme classique de synchronisation entre lecteurs et écrivains travaillant sur une mémoire partagée tout en sauvegardant l'intégrité des données [COU 71].

Pour ce problème, les contraintes sont que des lectures et écritures ne puissent s'exécuter simultanément, et qu'une seule écriture se fasse à la fois. En outre, nous supposerons que le nombre d'utilisateurs de la mémoire est connu.

Le contenu en jetons de la place P_3 donne le nombre d'utilisateurs oisifs à un instant donné. La place P_4 sert à exprimer l'exclusion.

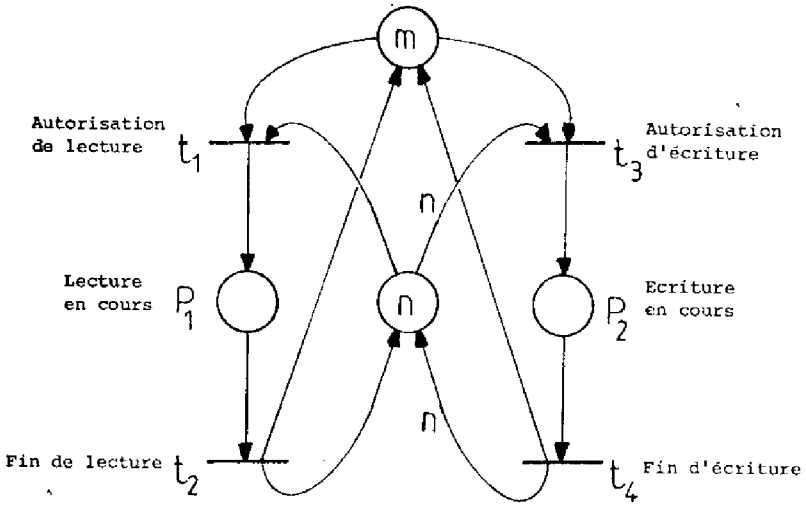


Figure II_3.

$P = \{P_1, P_2, P_3, P_4\}$

$T = \{t_1, t_2, t_3, t_4\}$

$$E = \begin{array}{c} \begin{matrix} P_1 & P_2 & P_3 & P_4 \end{matrix} \\ \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & n \\ 0 & 1 & 0 & 0 \end{bmatrix} \end{array} \begin{matrix} t_1 \\ t_2 \\ t_3 \\ t_4 \end{matrix}$$

$$S = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & n \end{bmatrix}$$

$$M_o = \begin{bmatrix} 0 , & 0 , & m & , n \end{bmatrix}.$$

2.2. *Règles d'évolution*

2.2.1. *Marquage* :

Le marquage peut être considéré comme étant une application de l'ensemble des places P dans l'ensemble des entiers positifs ou nuls, N.

$$M : P \longrightarrow N$$

La notation généralement employée est : $M(p) = n$ où $n \in N$ représente le nombre de jetons de la place p.

2.2.2. *Transition sensibilisée* :

Une transition t_i est dite sensibilisée par un marquage M si et seulement si, le nombre de jetons contenus dans chaque place qui la précède est supérieur ou égal au poids de l'arc la reliant à la transition.

$$\forall p \in P, \quad M(p) > E(t_i, p)$$

La figure II_4 donne un exemple de transition sensibilisée (b) et non sensibilisée (a).

Figure II_4.

2.2.3. *Tir d'une transition* :

Une transition t_i sensibilisée est dite " tirable ". Son tir, opération indivisible, correspond à une évolution du système donc à un changement d'état. Ce changement se traduit sur le réseau de Petri par un nouveau marquage, obtenu par les règles suivantes :

- à chaque place précédant la transition t_i, est enlevé un nombre de jetons égal au poids de l'arc qui la relie à la transition,

- à chaque place suivant la transition t_i, est ajouté un nombre de jetons égal au poids de l'arc qui la relie à la transition.

$$M_{i+1} = M_i + S(t_i,.) - E(t_i,.)$$

où M_{i+1} est le marquage obtenu à partir du marquage M_i par le tir de la transition t_i

$$M_i \xrightarrow{t_i} M_{i+1}$$

La figure II_5 donne un exemple de tir de la transition.

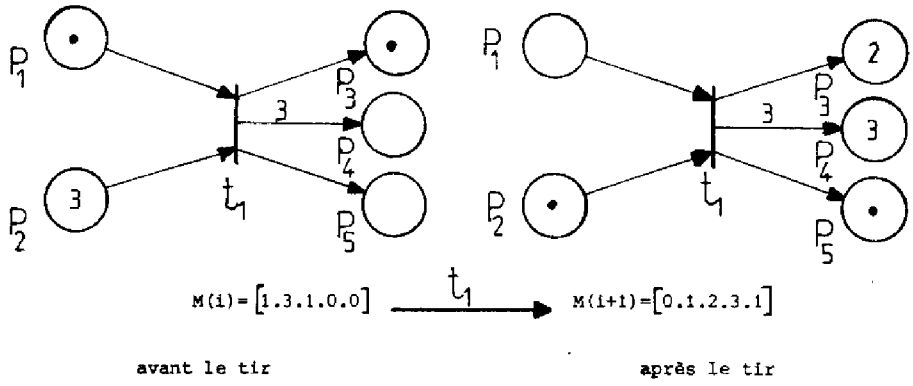


Figure II_5.

2.2.4. *Séquence de tir* :

On appelle séquence de tir, tirable à partir d'un marquage M_i une séquence de transitions $\sigma = (t_i . t_{i+1} . \ldots . t_{i+k})$, telle que le tir d'une transition conduit à un marquage qui sensibilise la suivante. La première transition est bien sûr sensibilisée par le marquage M_i.

$$M_i \xrightarrow{t_i} M_{i+1} \xrightarrow{t_{i+1}} \ldots M_{i+p} \ldots \xrightarrow{t_{i+k}} M_{i+k+1}$$

ou encore $M_i \xrightarrow{\sigma} M_{i+k+1}$.

Un exemple de séquence de tir dans le réseau de la figure II_3 est $\sigma = (t_1 . t_1 . t_2)$ qui conduit à partir du marquage initial M_o au marquage M_1.

$$M_o \xrightarrow{t_1} M_1 \xrightarrow{t_1} M_2 \xrightarrow{t_2} M_1$$

avec $M_o = [o, o, n, m]$ $M_1 = [1, o, n-1, m-1]$

$M_2 = [2, o, n-2, m-2]$.

2.2.5. Classe des marquages conséquents :

On appelle **classe** des marquages conséquents du marquage initial M_o, l'ensemble des marquages M_i accessibles à partir de M_o;
Cet ensemble est noté $\overrightarrow{M_o}$.

$$M_i \in \overrightarrow{M_o} \Longleftrightarrow \exists \sigma \text{ telle que } M_o \longrightarrow M_i$$

Pour le réseau de Petri de la figure II_1, nous aurons comme ensemble des marquages conséquents si m = n = 2

$M_o = [0,0,2,2]$ $M_3 = [0,1,1,0]$

$M_1 = [1,0,1,1]$

$M_2 = [2,0,0,0]$

2.2.6. Graphe des marquages :

On appelle graphe des marquages conséquents de M_o, le graphe orienté dont l'ensemble des sommets est l'ensemble des marquages conséquents, $\overrightarrow{M_o}$. Les arcs sont définis par ; (M_i, M_j) est un arc si et seulement si :

$$M_i \in \overrightarrow{M_o},\ M_j \in \overrightarrow{M_o} \text{ et } \exists\, t_k \text{ telle que } M_i \xrightarrow{t_k} M_j$$

Pour le réseau de la figure II_3, le graphe des marquages conséquents est représenté sur la figure II_6.

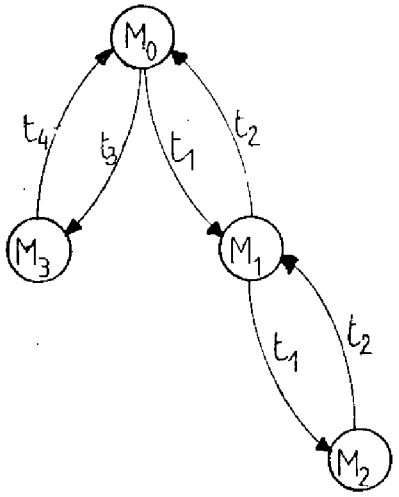


Figure II_6.

2.3. *Représentation matricielle*

2.3.1. *Matrice d'incidence* :

La matrice d'incidence C d'un réseau de Petri est égale à la différence entre la matrice de sortie et la matrice d'entrée.

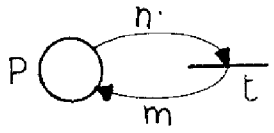
$$C = S - E$$

Pour notre exemple, nous aurons :

$$C = \begin{bmatrix} 1 & 0 & -1 & -1 \\ -1 & 0 & 1 & 1 \\ 0 & 1 & -1 & -n \\ 0 & -1 & 1 & n \end{bmatrix}$$

2.3.2. Remarque :

Cette représentation peut supprimer des informations pour une classe de réseaux, appelés "non purs". En effet, un réseau "non pur", est un réseau comportant des boucles élémentaires (figure II_7), et pour ces réseaux la matrice d'incidence omet tout ou partie de la boucle.



$$C(t,p) = S(t,p) - E(t,p) = n - m$$

figure II_7.

2.4. Propriétés et analyse des réseaux de Petri

2.4.1. Propriétés classiques

Nous nous limiterons à présenter trois de ces propriétés dites classiques, cependant d'autres pourront être rencontrées [VAL 76].

Un réseau de Petri est dit borné pour un marquage initial M_o, si quel que soit le marquage accessible, et quelle que soit la place considérée, le nombre de jetons contenus par chaque place est inférieur à une borne donnée.

Dans le cas où la borne est unitaire, on dit que le réseau est sauf.

Le réseau de Petri de la figure II_3 est borné pour m=n=2 car toutes les places de réseau ont un marquage inférieur ou égal à 2.

On dit qu'un réseau de Petri est vivant pour un marquage initial M_o si et seulement si pour tout marquage M appartenant à la classe des marquages conséquents $\overrightarrow{M_o}$ et pour toute transition t, il existe une séquence de tir finie tirable à partir de M et qui comprend t.

$$\forall M \in \overrightarrow{M_o}, \forall t \in T, \exists \sigma \text{ telle que } t \in \sigma, \quad M \xrightarrow{\sigma} M_{+1}$$

A partir du graphe des marquages (figure II_6) il est facile de vérifier que toutes les transitions de notre exemple de base répondent à cette définition. Ainsi le réseau de Petri est vivant.

Enfin un réseau de Petri est réinitialisable pour un marquage initial M_o si et seulement si ce marquage M_o est accessible à partir de n'importe quel marquage appartenant à l'ensemble des marquages conséquents $\overrightarrow{M_o}$.

$$\forall M \in \overrightarrow{M_o}, \exists \sigma \text{ telle que } M \xrightarrow{\sigma} M_o$$

Pour cette propriété aussi, il est facile de vérifier sur le graphe des marquages (figure II_6) que le réseau de Petri respecte bien cette définition. En effet, il existe toujours une séquence de tir de transitions qui ramène de n'importe quel marquage M_1, M_2 ou M_3 au marquage initial M_o.

2.4.2. *Analyse directe*

Le but de cette analyse est de vérifier les propriétés classiques telles qu'elles ont été définies. Sa base est la construction du graphe des marquages conséquents. Cependant pour les réseaux dont la borne n'est pas à priori connue, il faut préciser que l'algorithme d'étude est dérivé de celui de KARP et MILLER [KAR 69].

Ainsi, un réseau de Petri sera dit borné pour un marquage initial M_o si et seulement si, son ensemble des marquages conséquents $\overrightarrow{M_o}$ est fini.

De la connaissance de l'ensemble des marquages conséquents, la construction du graphe est très facile. Ensuite les autres propriétés sont aisément démontrables.

En effet, pour un réseau borné, une condition nécessaire et suffisante pour que ce réseau soit réinitialisable, est que son graphe des marquages accessibles soit fortement connexe. Il est vivant, si et seulement si, toutes les transitions apparaissent au moins une fois dans chacune des composantes fortement connexes pendantes.

Certaines conclusions peuvent être tirées de ces propriétés. Par nature, un système physique est borné : un réseau non borné traduit en général une erreur de conception ou de description et son implémentation est impossible. Le caractère vivant assure l'absence de blocage du système spécifié par réseau de Petri ; de plus il assure que toute partie du système est accessible à partir de l'état initial. Enfin un réseau réinitialisable traduit la possibilité de retour à l'état initial du système représenté.

Aussi il est très utile qu'un réseau de Petri possède ces propriétés. Si l'on considère notre exemple, nous sommes confrontés au problème de fixer une valeur à m et n pour pouvoir analyser le réseau. Même si l'on prend des valeurs très grandes (toutefois limitées par le nombre des marquages accessibles qui peut être déduit), rien dans les résultats obtenus ne permet de conclure sur la validité des propriétés classiques pour un n quelconque. Cependant dans bien des cas ces résultats partiels sont suffisants pour conclure sur la présence de blocage.

Donc le principal défaut de cette analyse, est qu'elle repose sur l'énumération des marquages, car elle tend à exploser dès que le réseau devient un peu important.

2.4.3. *Analyse par réduction*

Une solution pour combattre cette explosion combinatoire en l'absence des règles de construction des réseaux de Petri, consiste à les réduire avant de les analyser.

Bien entendu, les règles de réductions employées n'influent pas sur les propriétés classiques des réseaux de Petri.

Les règles les plus communément employées [BER 78] sont succintement les trois suivantes (seuls des cas particuliers évidents de ces règles sont exposés (ci-dessous) :

I) Subtitution d'une place

Une place servant de relais (figure II $_{8a}$) entre deux transitions peut être supprimée (figure II $_{8b}$). En effet, si la première transition est tirée, la seconde est obligatoirement sensibilisée.

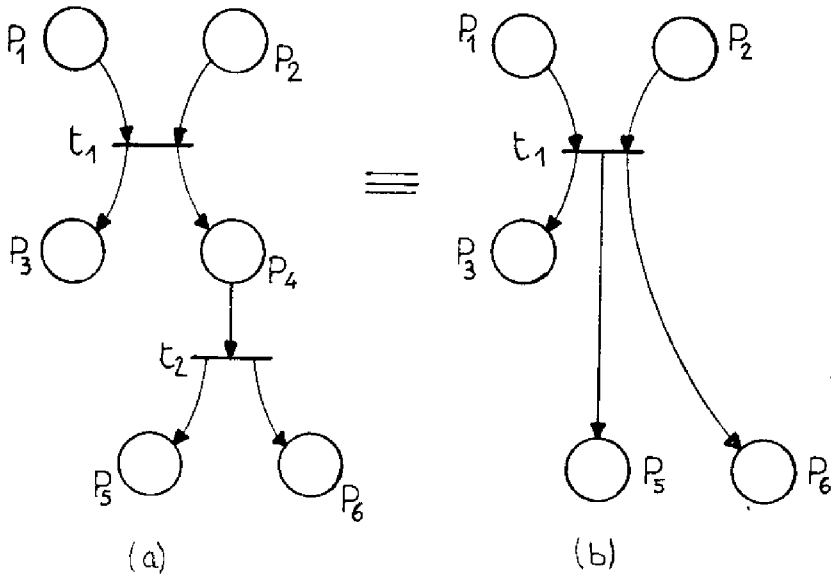


Figure II$_8$.

II) Place implicite

C'est une place qui peut être supprimée (figure II$_{9b,d}$) sans pour autant modifier les séquences de tir (figure II$_{9d,c}$)

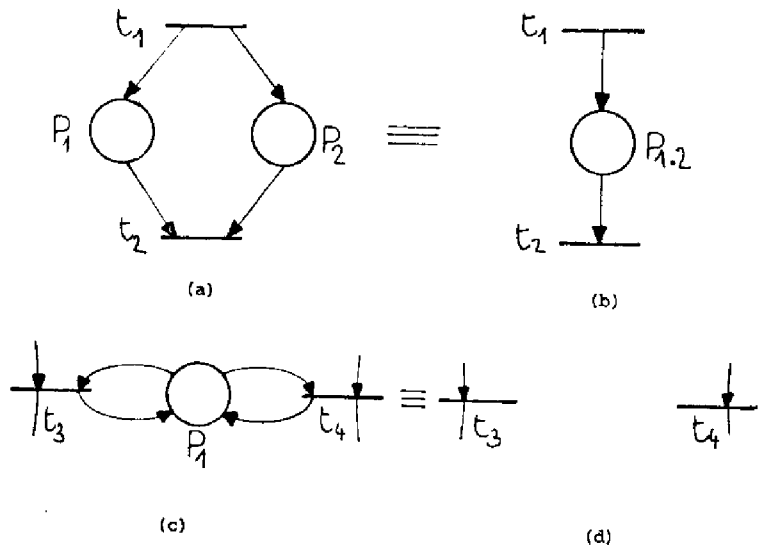


Figure II_9.

III) <u>transition identité</u>

C'est une transition dont le tir ne modifie pas le marquage, et dont la sensibilisation dépend du tir d'une autre (figure II_{10}).

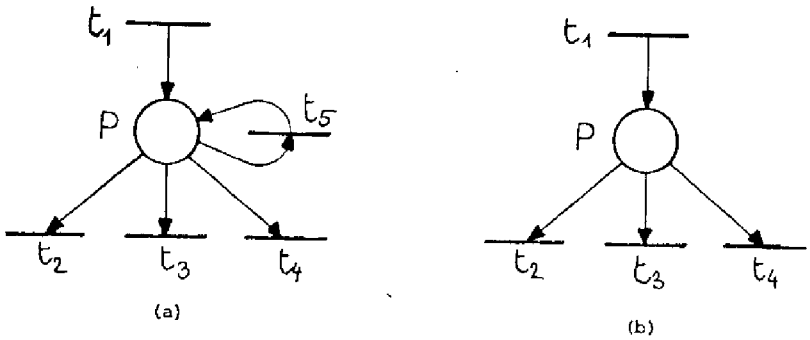


Figure II_{10}.

Par ce mode d'analyse, le sens donné au réseau initial est perdu après réduction. Cependant cette analyse permet de limiter très fortement le nombre des sommets, pour arriver dans de nombreux cas à la réduction totale des réseaux. Cette réduction totale est très intéressante car alors on peut conclure que ces réseaux possèdent les propriétés borné et vivant. De plus, les règles de réduction sont indépendantes du poids sur les arcs et donc pour un réseau totalement réduit la vérification des propriétés classiques est rapidement faite pour n'importe quels poids sur les arcs.

Le réseau représentant le problème des lecteurs et écrivains répond à cette classe de réseaux totalement réductibles, et ainsi par réduction nous montrons pour tous m et n qu'il est un réseau borné et vivant.

Pour les autres types de réseaux (non réductibles) le problème de l'énumération des marquages reste posé ainsi que ses limitations.

2.4.4. *Invariance des réseaux*

Pour les deux modes d'analyse précédents, la vérification des propriétés repose sur l'énumération des marquages, et donc du marquage initial. Dans le cas où le nombre de marquages est trop grand, il devient fastidieux de vérifier un à un tous ces marquages pour prouver les diverses contraintes composant le cahier des charges du système. Soulignons ici que pour cette preuve, l'analyse par réduction est impossible vu la perte de sens du réseau réduit (et donc des marquages accessibles).

Une autre technique a été élaborée ces dernières années [LAU 74- MEM 77 - SIF 79 - BER 79] pour vérifier très facilement des propriétés spécifiques des réseaux. Cette technique est basée sur l'analyse de la structure du réseau et elle utilise la notion d'invariance définie par LAUTENBACH et SCHMID [LAU 74] et mise en application par BERTHOMIEU [BER 79].

Ainsi nous dirons dans un réseau de Petri défini par $[P,T,E,S,M_o]$ que :

- le vecteur $\underline{X}$ ($x_i \in N$) est un invariant linéaire de places si et seulement si, il est solution du système d'équations linéaires.

$$C\underline{X} = \underline{O}$$

- le vecteur $\underline{X}$ ($x_i \in N$) est un invariant linéaire de transitions du réseau si et seulement si il est solution du système d'équations linéaires

$$C^T \underline{X} = 0$$

C & C^T sont respectivement la matrice d'incidence du réseau et sa transposée.

Nous appellerons couverture d'invariants linéaires de places (respectivement de transitions) d'un réseau de Petri, un ensemble d'invariants linéaires de places (transitions) qui regroupe toutes les places (transitions) du réseau. Elle est obtenue par la résolution en nombres entiers des systèmes d'équations linéaires : $C \underline{X} = 0$ et $C^T \underline{X} = 0$ [BER 79].

L'existence d'une couverture d'invariants linéaires de places pour un réseau de Petri permet, en plus de l'analyse structurelle, de conclure sur la propriété de réseau borné [SIF 79].

Pour le réseau de la figure II_3 une couverture d'invariants de places est donnée par les deux invariants suivants :

$$II_1 \begin{cases} M(P_1) + M(P_2) + M(P_3) = m \\ M(P_1) + nM(P_2) + M(P_4) = n \end{cases}$$

2.4.5. *Analyse structurelle*

A l'aide des invariants, la preuve de certaines contraintes est rapidement faite sans avoir à énumérer tous les marquages du réseau (en considérant seulement le marquage initial comme paramètre). De plus un parallèle peut être fait entre invariance d'un réseau et propriétés classiques [SIF 79].

Pour notre exemple de base, à l'aide des invariants du système II_1 nous pouvons conclure :

de $M(P_1) + n.M(P_2) + M(P_4) = n$

- que s'il y a au moins un lecteur ($M(P_1) \geqslant 1$), aucun écrivain ne peut plus être autorisé ($M(P_2) = 0$),
- qu'il ne peut y avoir qu'un seul écrivain à la fois ($0 \leqslant M(P_2) \leqslant 1$),
- que s'il y a un écrivain ($M(P_2) = 1$), aucun lecteur ne peut plus être autorisé ($M(P_1) = M(P_4) = 0$).

de $M(P_1) + M(P_2) + M(P_3) = m$

- que le nombre d'utilisateurs de la ressource reste constant.

2.4.6. *OGIVE, outil graphique interactif de vérification*

Divers systèmes de vérification de réseaux de Petri ont été créés. Certains sont basés sur la simulation [MOA 79 - MIC 79] du système décrit par réseaux, d'autres sur l'énumération des marquages [BOU 78 - PRA 79]. Le dernier né, OGIVE, présente en plus, l'avantage de rechercher les invariants linéaires de places et de transitions [BER 79]. De plus les règles de réduction définies par BERTHELOT [BER 78] sont implantées pour autoriser l'étude de réseau de complexité moyenne.

Il est important de souligner l'intérêt d'un tel outil au moment de l'élaboration du réseau de Petri car il permet de détecter le maximum d'erreurs de conception ou de modélisation en évitant les procédures de simulation coûteuses, et aux résultats moins rigoureux.

2.5. *Interprétation des réseaux de Petri*

Nous venons de voir que les réseaux de Petri sont un très bon outil pour spécifier et valider les contraintes de synchronisation qu'ils représentent.

Cependant, les évolutions de la synchronisation se traduisent sur le réseau par le tir de transitions. Ce tir est piloté par le monde extérieur, et il apparaît ainsi sur le réseau de Petri par un prédicat associé à la transition qui modélise l'évolution.

Ainsi le tir d'une transition peut être soumis à deux conditions :

- la transition doit être sensibilisée
- le prédicat associé éventuellement à la transition doit être vrai.

Modélisons sur notre exemple les demandes de lecture ou d'écriture . Le réseau qui en résulte est celui de la figure II_{11}. Ainsi, dans ce réseau, quand la synchronisation évolue, c'est-à-dire par exemple, lorsqu' une demande de lecture est faite, le prédicat associé étant vrai et la transition t_1 sensibilisée, son tir a lieu. La transition t_2 n'a pas de prédicat et est sensibilisée, son tir a donc lieu. Si une demande d'écriture intervient, le prédicat associé étant vrai et la transition t_4 étant sensibilisée, elle est donc tirée. La transition t_5 qui n'a pas de prédicat ne peut pas être tirée car elle n'est plus sensibilisée. Par contre si une fin de lecture est faite, le prédicat associé est vrai, la transition t_3 étant sensibilisée, son tir se fait. La transition t_4 est alors sensibilisée et est tirée. Plusieurs autres configurations peuvent apparaître.

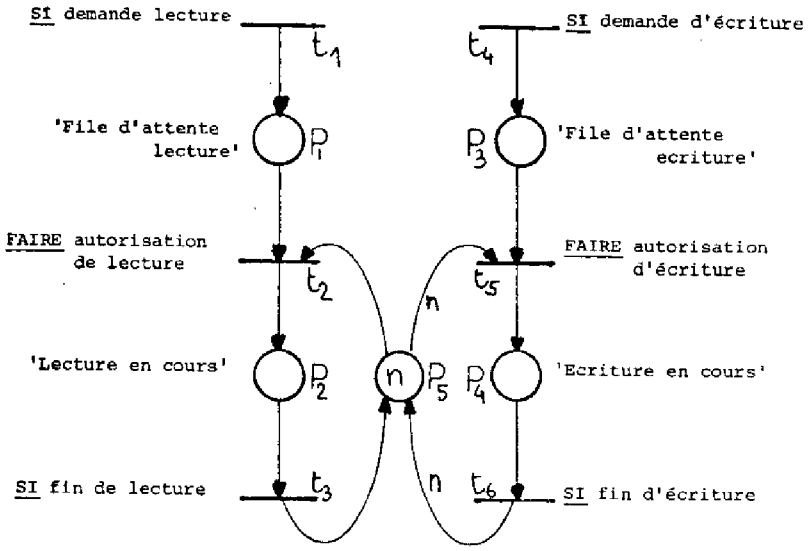


Figure II_{11}

3. *RESEAUX DE PETRI A JETONS INDIVIDUALISES*

3.1. *Limites des réseaux de Petri*

En plus de celles se rapportant à l'analyse par réduction et invariants, diverses méthodes peuvent éviter l'explosion du nombre des marquages accessibles [VAL 76]. Mais dans de nombreux cas les limites des réseaux de Petri se font sentir.

Par exemple, il n'y a pas de solution satisfaisante pour exprimer directement qu'une évolution est soumise à la non existence d'un état (condition d'inhibition).

En effet, les diverses solutions sont obtenues soit par l'introduction d'arcs inhibiteurs (figure II_{12a}) soit par l'adjonction de prédicats sur les transitions (figure II_{12b}). Cependant dans ces cas la preuve automatique par OGIVE (CHE 79) du bon fonctionnement est impossible. La dernière solution restante consiste en l'introduction de boucles élémentaires qui ne sont pas prises en compte dans l'analyse structurelle (§2.3.2). De plus, les réseaux possédant des boucles élémentaires sont difficilement réductibles et l'analyse reposant sur l'énumération des marquages peut être très vite saturée (figure II_{12c}).

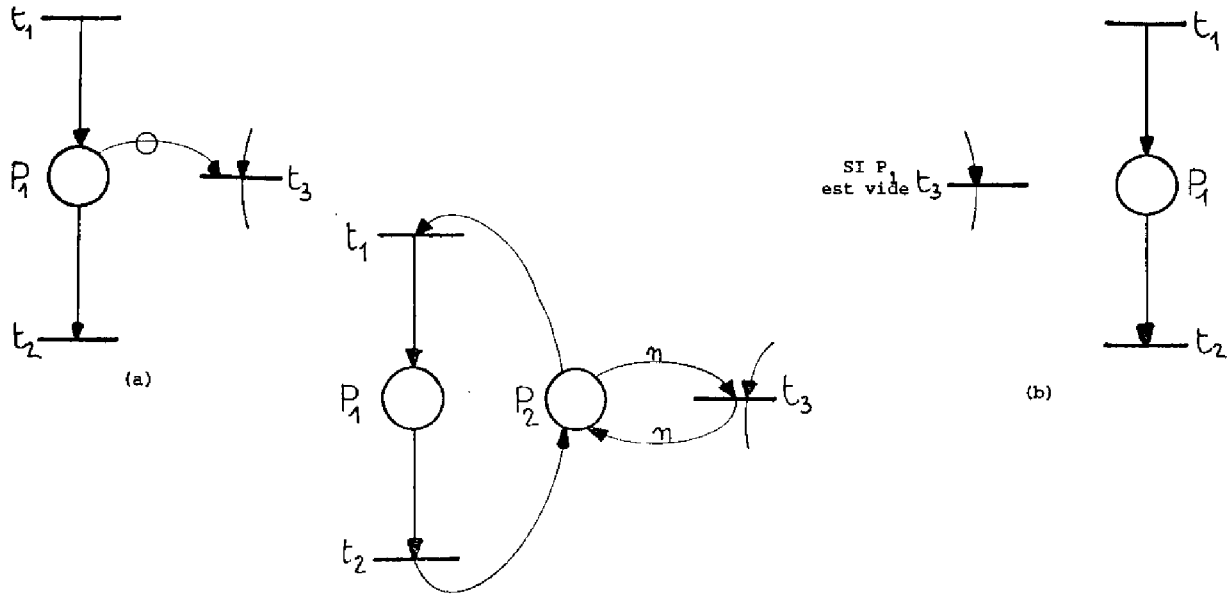


Figure II_{12}

Une autre limitation des réseaux de Petri est que ces derniers sont basés sur une utilisation exclusive de variables booléennes et de compteurs. L'introduction de stratégies faisant intervenir des noms (mots de synchronisation par exemple [ROU 78]) implique des codages qui compliquent fortement le réseau et introduisent encore de nombreuses boucles élémentaires.

Le plus ennuyeux est que ces limitations n'apparaissent pas seulement dans des cas exceptionnels peu réalistes. En effet, dès que l'on cherche à rendre un mécanisme non vulnérable à son environnement [VAL 80], il faut introduire des tâches de signalement d'erreur et de reconfiguration qui ne sont exécutées que lorsque certaines conditions ne sont pas remplies.

Les réseaux de Petri seuls sont donc mal adaptés à la spécification de systèmes de commande sûrs de fonctionnement. Mais à notre connaissance, il n'existe aucun modèle basé sur une approche différente qui soit mieux adapté à la fois pour la spécification et pour la validation. La seule solution est semble-t-il de surmonter les difficultés éventuelles en modifiant le modèle.

Plusieurs modèles dérivés des réseaux de Petri ont été proposés [CRI 77 - SCH 78 - GEN 79] .

Nous introduisons dans les paragraphes suivants le formalisme des réseaux de Petri à jetons individualisés qui sont à considérer comme un cas particulier du modèle défini par GENRICH et al .

3.2. *Réseaux de Petri à jetons individualisés*

3.2.1. *Présentation informelle* :

Un réseau de Petri à jetons individualisés n'est autre qu'un réseau de Petri dont les jetons sont personnalisés par des identificateurs. Ainsi des noms sont associés aux jetons, et des règles de transfert de noms sont associées aux tirs des transitions.

Pour introduire ce genre de réseau, prenons un exemple tel que celui de la gestion d'une ressource. Le problème est alors le suivant ; des processus utilisateurs demandent une ressource, attendent qu'elle soit disponible et l'utilisent. Si l'on se contente de cet énoncé, le réseau de Petri (très simple) est celui de la figure II_{13}, où m est le nombre total d'utilisateurs.

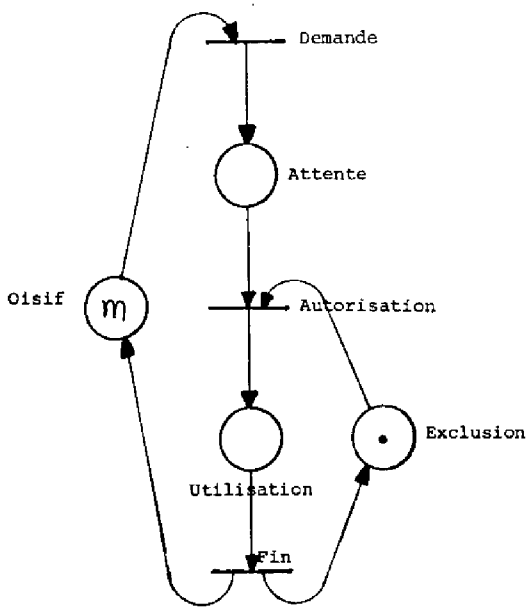


Figure II_{13}.

En fait pour ce réseau rien n'empêche par exemple que par un message erroné, un utilisateur signale sa fin d'occupation de la ressource alors qu'elle est encore utilisée par un autre.

Si pour des raisons de sûreté, on veut interdire un tel phénomène, il faut m-pliquer la transition "fin" et en conséquence les places "utilisation"et"attente" ainsi que les transitions "autorisation" et "demande".

Ceci afin de différencier l'utilisation de la ressource par un processus i de celle faite par un processus j.

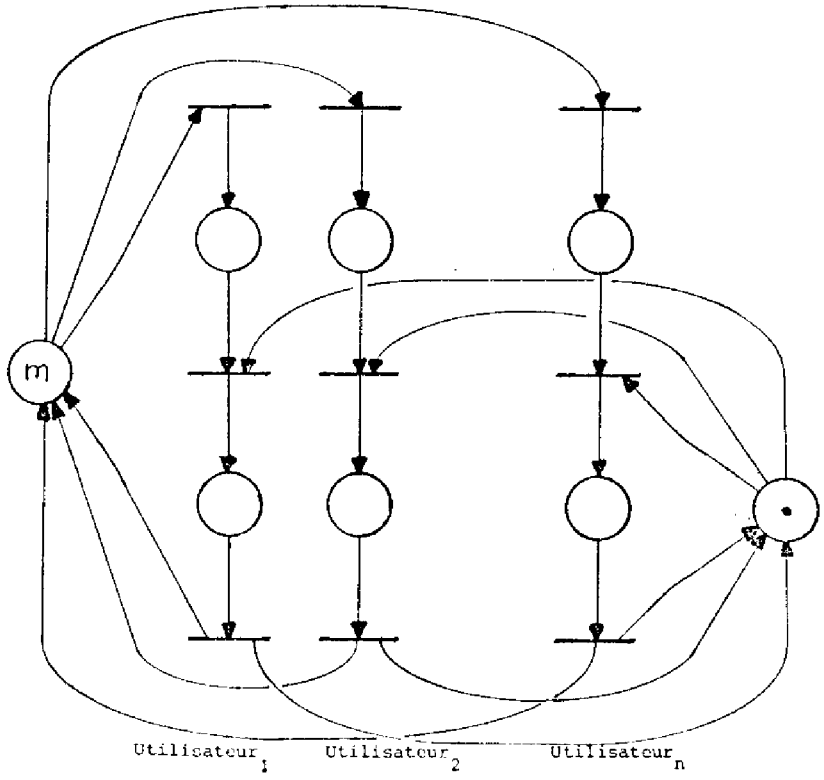


Figure II_{14}.

Ainsi pour tenir compte de ces nouvelles contraintes le réseau initial s'alourdit, et peut devenir rapidement inexploitable quand on spécifie des problèmes plus complexes.

Aussi notre but avec les réseaux de Petri à jetons individualisés est-il de spécifier, avec un même nombre de sommets que le réseau de la figure II_{13}, les contraintes représentées sur celui de la figure II_{14}.

Pour cela nous allons affecter un identificateur à chaque jeton contenu dans la place "oisif", qui correspondra au nom d'un utilisateur.

Remarquons ici que si $U = \{u_1, u_2, \ldots, u_m\}$ est l'ensemble des identificateurs des utilisateurs, les marquages de la place "oisif" peuvent s'exprimer par des sommes formelles d'élément de U. Par exemple nous pouvons avoir $M(oisif) = u_1 + u_7 + u_8$ qui indique que les utilisateurs u_1, u_7 et u_8 sont oisifs tandis que les autres $(U - \{u_1, u_7, u_8\})$ sont dans un autre état.

Reste alors à compléter les règles d'évolution (tir de transition) pour prendre en compte ces identificateurs. Cela se fait par l'intermédiaire de variables associées aux arcs, variables qui seront remplacées par des identificateurs au moment du tir. Ainsi sur le réseau de la figure II_{15} le tir de la transition t_1 par l'utilisateur i impliquera qu'un jeton u_i contenu dans la place "oisif" est transféré dans la place "attente".

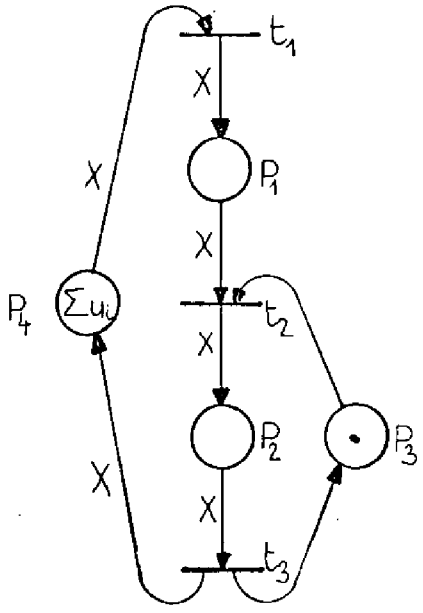


Figure II_{15}.

Après cette rapide présentation, nous allons définir plus formellement cet outil que nous appelons réseaux de Petri à jetons individualisés.

3.2.2. Définition :

On appelle réseau de Petri à jetons individualisés, le 7 uplet $[P,T,L,I,E,S,M_o]$ où ,

- $P = \{P_1, P_2, \ldots, P_n\}$ est l'ensemble des places du réseau,

- $T = \{t_1, t_2, \ldots, t_p\}$ est l'ensemble des transitions,

- $L = \{\varepsilon, l_1, \ldots, l_k\}$ est l'ensemble des variables utilisées comme étiquettes sur les arcs,

- $I = \{\varepsilon, n_1, \ldots, n_q\}$ est l'ensemble des identificateurs de jeton ,

- E : est la fonction des places précédentes, elle est définie par

$$E : P \times T \longrightarrow N \times L$$

où N est l'ensemble des entiers positifs ou nuls,

- S : est la fonction des places suivantes, elle est définie par

$$S : P \times T \longrightarrow N \times L$$

- M_o est le marquage initial, chaque place aura comme marquage une somme formelle composée d'éléments banalisés et/ou d'identificareurs. Le marquage initial sera donc un vecteur de sommes formelles.

3.2.3. Représentation :

Les principales différences par rapport aux réseaux de Petri sont :

- ⓐ jeton individualisé
- les arcs reliant places et transitions pourront avoir comme poids des étiquettes (figure II_{16})

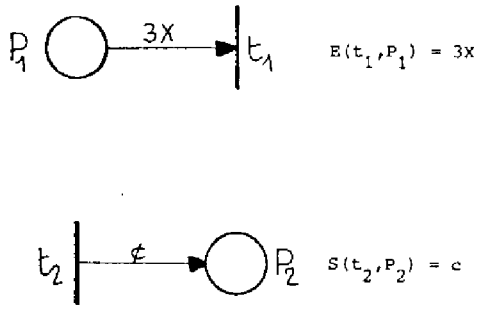


Figure II_{16}.

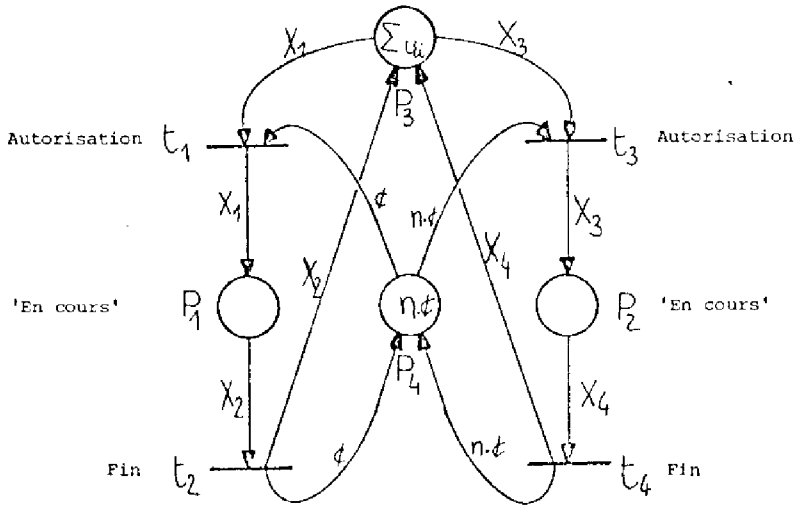
Dans la figure II_{16} la transition t_1 ne sera sensibilisée que si la place P_1 comprend trois jetons ayant le même identificateur. Lorsque la transition t_2 est tirée un jeton banalisé est mis dans la place P_2.

Remarquons que ¢ est l'élément banalisé des ensembles L et I, et que si ces ensembles ne contiennent que cet élément, on retrouve le cas des réseaux de Petri.

Ainsi on pourra facilement se ramener au cas des réseaux de Petri en supprimant les identificateurs et en les remplaçant par l'élément banalisé.

Revenons à notre exemple de base des lecteurs écrivains, où nous ajoutons des contraintes de surveillances sur les utilisateurs. Le réseau résultant est celui de la figure II_{17}.

Remarquons dans cet exemple, qu'il n'est pas nécessaire de distinguer les variables x_1, x_2 ... associées aux diverses transitions du réseau, et l'on peut se contenter d'utiliser un seul nom de variable.



$$P = \{P_1, P_2, P_3, P_4\} \qquad T = \{t_1, t_2, t_3, t_4\}$$

$$L = \not{\epsilon}, x_1, x_2, x_3, x_4 \qquad I = \{\not{\epsilon}\} \cup \{ui\}$$

$$E = \begin{bmatrix} 0 & 0 & x_1 & \not{\epsilon} \\ x_2 & 0 & 0 & 0 \\ 0 & 0 & x_3 & n \cdot \not{\epsilon} \\ 0 & x_4 & 0 & 0 \end{bmatrix} \qquad S = \begin{bmatrix} x_1 & 0 & 0 & 0 \\ 0 & 0 & x_2 & \not{\epsilon} \\ 0 & x_3 & 0 & 0 \\ 0 & 0 & x_4 & n \cdot \not{\epsilon} \end{bmatrix}$$

$$M_o = \left[0, 0, \sum_i ui, n\not{\epsilon}\right]$$

figure II_{17}

3.3. *Règles d'évolution*

3.3.1. *Marquage* :

Le marquage d'un réseau de Petri à jetons individualisés est un vecteur M dont le nombre de composantes est égal au nombre d'éléments de P.

Chaque composante $M(P_i)$ indique pour chaque identificateur le nombre de jetons contenus dans la place P_i et ayant cet identificateur. Ainsi ces composantes sont des sommes formelles d'identificateur .

Par exemple $M(P_i) = 3a + 2\phi$ exprime le marquage M de la place P_i comprend 3 jetons de nom "a" et 2 jetons banalisés.

3.3.2. *Transition sensibilisée* :

Une transition t est dite sensibilsée si on peut affecter des identificateurs aux variables des étiquettes des arcs entrant de la transition de telle manière que le vecteur marquage soit supérieur ou égal au poids de ces arcs.

$$\forall p \implies M(p) \geqslant E(t,p)$$

Des exemples de transitions sensibilisées (d,e,f) et non sensibilisées (a,b,c) sont présentés à la figure II_{18}.

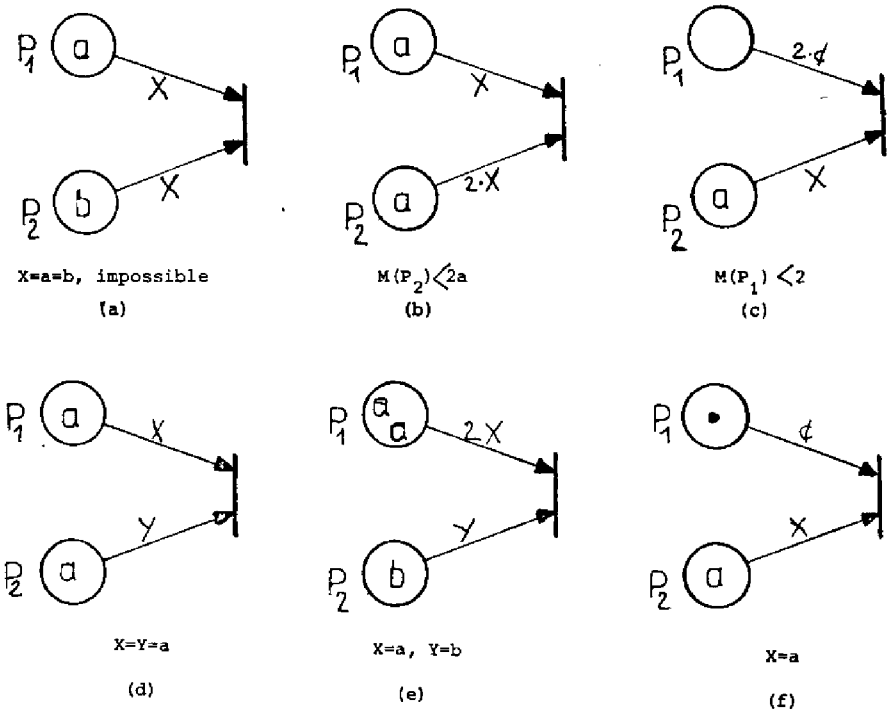


Figure II_{18}.

3.3.3. Tir d'une transition :

Une transition sensibilisée peut être tirée. A son tir, opération indivisible, correspond :

- affectation d'identificateurs aux variables des étiquettes des arcs de la transition,

- à chaque place précédente est enlevé un nombre de jetons individualisés égal au poids de l'arc étiqueté qui la relie à la transition,

- à chaque place suivante est ajouté un nombre de jetons individualisés égal au poids de l'arc étiqueté qui la relie à la transition.

$$M_{i+1} = M_i - E(t,.) + S(t,.)$$

où M_{i+1} est le marquage obtenu à partir du marquage M_i par le tir de la transition t. Les opérations "+" et "-" correspondent à la somme formelle définie pour les marquages.

La figure II_{19} donne un exemple de tir de transition.

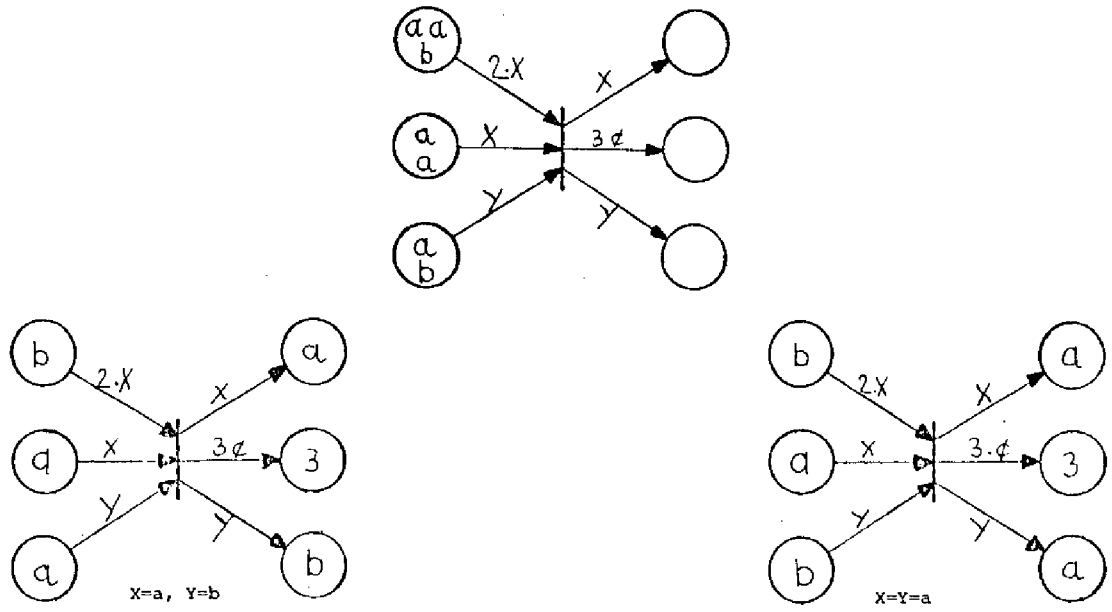


Figure II_{19}.

3.3.4. *Séquence de tir* :

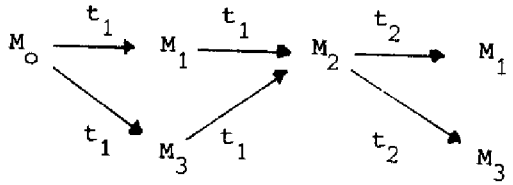
Comme pour les réseaux de Petri, on appellera "séquence de tir, tirable à partir d'un marquage M_i" une séquence $\sigma = (t_i.t_{i+1}...t_{i+k})$ telle que le tir d'une transition sensibilise la suivante, Ie marquage M_i sensibilisant la première transition.

$$M_i \xrightarrow{t_i} M_{i+1} \cdots\cdots \xrightarrow{t_{i+k}} M_{i+k+1}$$

$$M_i \xrightarrow{\sigma} M_{i+k+1}$$

Ainsi pour notre exemple de base, en considérant seulement deux utilisateurs u_1 et u_2, une séquence de tir possible est :

$\sigma = (t_1, t_1, t_2)$ qui se traduit par



$$M_o = [o,\ o,\ u_1 + u_2,\ 2]$$

$$M_1 = [u_1,\ o,\ u_2,\ 1]$$

$$M_3 = [u_2,\ o,\ u_1,\ 1]$$

$$M_2 = [u_1 + u_2,\ o,\ o,\ o]$$

3.3.5. *Classe des marquages conséquents* :

On appelle classe des marquages conséquents, issus du marquage initial M_o, l'ensemble des marquages M_i accessibles à partir du marquage initial.

$$M_i \in \overrightarrow{M_o} \iff \exists\ \sigma \text{ telle que } M_o \xrightarrow{\sigma} M_i$$

Ainsi avec toujours deux utilisateurs nous aurons pour le réseau de la figure II_{14} l'ensemble suivant :

$$\overrightarrow{M_o} = [M_o, M_1, M_2, M_3, M_4, M_5]$$

avec $M_o = [0, 0, u_1+u_2, 2]$ $\quad M_3 = [u_2, 0, u_1, 1]$

$M_1 = [u_1, 0, u_2, 1]$ $\quad M_4 = [0, u_2, u_1, 0]$

$M_2 = [u_1+u_2, 0, 0, 0]$ $\quad M_5 = [0, u_1, u_2, 0]$

Remarquons que si l'on ne tient pas compte des identificateurs, nous aurons $M_1 = M_3 = [1, 0, 1, 1]$ et $M_4 = M_5 = [0, 1, 1, 0]$ ce qui réduit la taille de cet ensemble. En fait l'utilisation des identificateurs, modifie aussi l'ensemble des marquages.

3.3.6. Graphe des marquages :

On appelle graphe des marquages conséquents du marquage initial M_o, le graphe orienté, dont les sommets sont l'ensemble des marquages conséquents, et les arcs qui les relient sont définis par :

$$\forall\ M_i \text{ et } M_j,\ \exists\ t \in T \text{ telle que } M_i \xrightarrow{t} M_j$$

Pour notre exemple, le graphe des marquages est représenté sur la figure II_{20}.

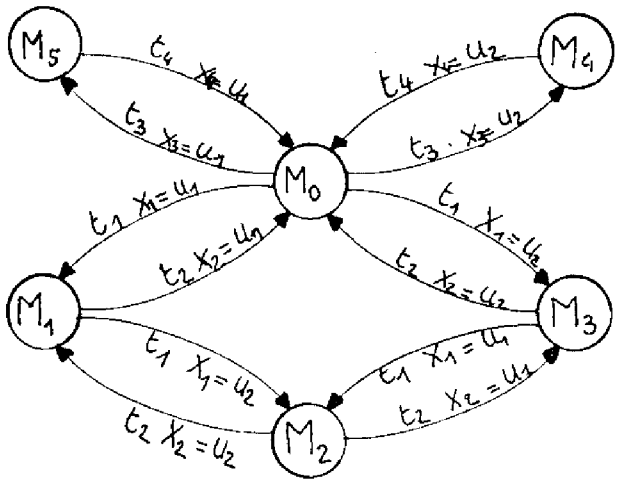


Figure II_{20}.

3.4. Propriétés et analyse

Comme pour les réseaux de Petri, nous pouvons définir de manière similaire les propriétés de réseau borné, vivant et réinitialisable pour ce type de modèle. L'analyse basée sur l'énumération des marquages est identique à celle des réseaux de Petri (§ 2.4.2.). Cependant, les défauts présentés par les réseaux de Petri sont ici aggravés par l'explosion combinatoire, encore bien supérieure, de l'ensemble des marquages accessibles. C'est pourquoi ce type d'analyse n'est pas applicable. Une issue envisageable est alors de décomposer cette analyse en deux phases :

I) la première consiste à analyser le réseau de Petri déduit du réseau à jetons individualisés, en supprimant les identificateurs,

II) l'autre à prouver que l'introduction des identificateurs n'apporte pas de blocage.

Ainsi la première phase de cette analyse est identique à celle des réseaux de Petri, et les mêmes conclusions pourront être tirées.

La preuve des propriétés spécifiques se fait, elle, généralement par invariants. Deux types d'invariants linéaires sont à différencier ; ceux qui traduisent la conservation du nombre de jetons et ceux qui traduisent la conservation du nom des jetons.

Le premier type d'invariants est déduit des systèmes d'équations linéaires $|C|\,\underline{\rho} = \underline{0}$ et $|C^T|\,\underline{\rho} = \underline{0}$ où la matrice $|C|$ est obtenue à partir de la matrice d'incidence du réseau à jetons individualisés en remplaçant toutes les variables par le symbole $\not\in$. C'est en fait la matrice d'incidence du réseau de Petri à jetons non individualisés correspondant. Le vecteur $\underline{\rho}$ est un vecteur d'entiers relatifs.

Ainsi, on obtient des invariants tels que :

$$\sum_i \rho_i \; |M(P_i)| \; = \; k$$

où $|M(P_i)|$ est le nombre de jetons contenus dans la place P_i sans tenir compte de leur nom.

Le second type d'invariants, est déduit des équations $C\,\underline{\rho} = \underline{0}$ et $C^T\,\underline{\rho} = \underline{0}$ où C est la matrice d'incidence du réseau de Petri à jetons individualisés et C^T sa transposée.
Les invariants sont alors du genre

$$\sum_q \rho_q \; M(P_q) \;=\; \sum_i k_i \; n_i$$

où $M(P_q)$ est le nombre de jetons individualisés contenus dans la place P_q et, $k_i\ n_i$ est le nombre de jetons individualisés par l'identificateur n_i.

A titre d'exemple, analysons notre réseau de base (figure II_{14})

Le réseau de Petri à jetons banalisés est celui de la figure II_3 : c'est un réseau borné réinitialisable et vivant. La même couverture d'invariants linéaires montre l'exclusion entre lecteurs et écrivains, et écrivains eux-mêmes, ainsi que la conservation du nombre des utilisateurs.

Ainsi la première phase de l'analyse reste inchangée. Pour la seconde, détaillons-la un peu plus et écrivons le système d'équations linéaires $C\,\underline{\rho} = 0$

$$\begin{bmatrix} \rho_1 \\ \rho_2 \\ \rho_3 \\ \rho_4 \end{bmatrix}$$

$$\begin{bmatrix} X_1 & 0 & -X_1 & -\not{\epsilon} \\ -X_2 & 0 & X_2 & \not{\epsilon} \\ 0 & X_3 & -X_3 & -n\not{\epsilon} \\ 0 & -X_4 & X_4 & n\not{\epsilon} \end{bmatrix} = \underline{0} \quad \text{qui donne} \quad \begin{aligned} (\rho_1-\rho_3)X_1 - \not{\epsilon}\,\rho_4 &= 0 \\ -(\rho_1-\rho_3)X_2 + \not{\epsilon}\,\rho_4 &= 0 \\ (\rho_2-\rho_3)X_3 - n\not{\epsilon}\,\rho_4 &= 0 \\ -(\rho_2-\rho_3)X_4 + n\not{\epsilon}\,\rho_4 &= 0 \end{aligned}$$

Ce système d'équations doit être résolu quels que soient X_1, X_2, X_3 et X_4 . Il se réduit donc à

$$\rho_1 - \rho_3 = 0$$

$$\rho_2 - \rho_3 = 0$$

$$\rho_4 = 0$$

Une solution à ce système d'équation est alors

$$\rho_4 = 0 \qquad \rho_1 = \rho_2 = \rho_3 = 1$$

On peut en déduire l'invariant suivant :

$$M(P_1) + M(P_2) + M(P_3) = \sum_i ui$$

qui montre la conservation des noms entre les places P_1, P_2 et P_3.

De plus si tous les utilisateurs ui sont différenciés, c'est-à-dire s'il n'existe pas deux utilisateurs de même nom, on est sûr qu'un utilisateur donné ne peut être, à un instant précis, que dans un seul des trois états suivants : lecteur (place P_1), écrivain (place P_2) ou oisif (place P_3).

3.5. *Interprétation des réseaux de Petri à jetons individualisés*

Le principe est le même que pour les réseaux de Petri (§2.5.). Cependant dans ce cas, l'information est plus précise puisque certains jetons sont individualisés. Les noms des jetons peuvent donc éventuellement être utilisés dans les prédicats et les actions associés aux transitions. Ceci se fera en utilisant les mêmes variables que celles qui apparaissent dans les poids assignés aux arcs (ensemble L).

Ainsi en reprenant l'exemple de la figure II_{11}, et en transformant ce réseau en réseau de Petri à jetons individualisés pour tenir compte des contraintes de sécurité, nous obtenons le réseau de la figure II_{21}.

Comme précédemment, considérons une demande de lecture faite par un utilisateur "a". Le prédicat demande de lecture est vrai, x prend la valeur "a", et la transition t_1 est tirée. Ceci implique que la place P_1 est marquée par un jeton identifié par la variable "a" ($M(P_1) = a$). La transition t_2 , ne devant pas vérifier un prédicat et étant sensibilisée, est alors elle aussi tirée. Ceci implique d'une part qu'un message autorisation de lecture est émis vers l'utilisateur "a", et d'autre part que $M(P_2) = a$, et $M(P_5) = 0$.

Si maintenant, un utilisateur "b" fait une demande d'écriture, la transition t_3 est tirée ($M(P_3) = b$). La transition t_4 elle par contre n'est pas sensibilisée et donc l'utilisateur "b" reste en attente.

Jusque là, pas de grands changements par rapport à ce qui avait été dit au paragraphe 2.5. Aussi examinons maintenant ce qui se passe à l'annonce d'un fin de lecture. Le prédicat de la transition t_3 est vrai, et la variable x prend la valeur du nom de l'utilisateur. Comme le marquage de la place P_2 est égal à "a", cette transition t_3 sera tirée si x = a, c'est-à-dire si l'utilisateur est "a".

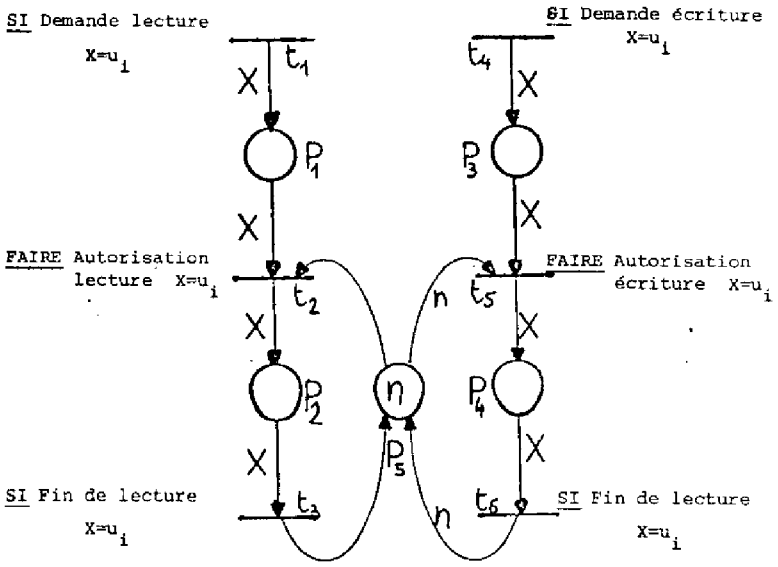


Figure II_{21}.

Regardons ce qui se passe si la transition t_3 est tirée($M(P_5) = n$). La transition t_5 ne devant satisfaire à aucun prédicat, est sensibilisée et donc tirée. Ceci implique qu'un message d'autorisation d'écriture est envoyé vers l'utilisateur b. Ainsi, autre avantage de cette représentation pour cette interprétation, c'est qu'elle permet de connaître directement vers qui est envoyé le message.

4. CONCLUSION

Dans la première partie de ce chapitre, une description désormais classique des réseaux de Petri a été rappelée, ainsi que les diverses techniques existant pour les analyser. Nous tenons à souligner l'aide que peut procurer un outil tel qu'OGIVE lors de la conception d'un réseau, en particulier pour localiser ses imperfections.

Si les réseaux de Petri sont un très bon outil pour spécifier des problèmes de synchronisation complexes, ils sont toutefois très mal adaptés pour représenter certains phénomènes. En particulier, représenter formellement des contraintes de sécurité amènent à construire des réseaux comportant un grand nombre de sommets. Pour ces réseaux, pas nécessairement difficiles à comprendre, une preuve formelle d'absence de blocage peut dans certains cas être très complexe, voire même impossible, principalement à cause de ce nombre de sommets.

Pour pallier cette lacune, nous avons proposé un outil : les réseaux de Petri à jetons individualisés. Ce modèle est proche des réseaux de Petri colorés [JEN 80]; cependant, son utilisation en diffère en ce sens que l'identité n'est pas employée pour spécifier des problèmes de synchronisation. mais uniquement pour modéliser des contraintes de sécurité.

Ainsi, la validation des systèmes spécifiés par cet outil est bien plus simple, et repose en grande partie sur celle des réseaux de Petri généralisés. Donc, la validation de ces réseaux est réalisée en employant les mêmes outils que ceux utilisés pour les réseaux généralisés.

CHAPITRE III

ALGORITHME DE MAINTIEN DE LA COHÉRENCE DES DONNÉES DUPLIQUÉES

1. INTRODUCTION

Dans ce chapitre, nous présentons l'algorithme de synchronisation qui sera implanté dans le processeur de synchronisation de la structure PASTELS (§ I.4).

Nous avons vu dans le chapitre I que la duplication des données impliquait l'utilisation d'algorithmes complexes pour maintenir leur intégrité et leur cohérence. Aussi l'algorithme retenu sera-t-il décrit à l'aide de réseaux de Petri à jetons individualisés, outil autorisant une bonne validation des contraintes et permettant de prouver l'absence de blocage du mécanisme qu'ils spécifient.

Ayant conscience de la difficulté, et de la complexité du problème de synchronisation à résoudre, nous présenterons dans le deuxième paragraphe l'algorithme dans le cas d'un milieu non défaillant. Ensuite dans le troisième paragraphe, nous analyserons les conséquences d'une panne extérieure au processeur de synchronisation. L'algorithme proposé alors protège les données et, mettant à profit leur duplication, permet des procédures de restauration et de fonctionnement en mode dégradé. En particulier, il autorise la mise hors service d'une mémoire, et sa réinsertion aprés réparation pendant le fonctionnement normal du sytème. Enfin, dans le dernier paragraphe nous établirons les caractéristiques que présente cet algorithme d'après certains critères. Nous dresserons un tableau pour le comparer avec les algorithmes de maintien de la cohérence des données réparties et dupliquées dans des bases de données, bien que ces deux catégories de système n'aient pas les mêmes applications.

La méthodologie que nous suivons pour décrire cet algorithme est composée de trois phases :

I) élaboration des contraintes du problème
II) spécification par réseaux de Petri de ces contraintes
III) validation : d'une part de l'absence de blocage du réseau, et d'autre part de la "bonne" représentation des contraintes.

2. *ALGORITHME DE BASE*

Nous allons chercher à adapter le problème des lecteurs-écrivains [COU 71] au cas de deux mémoires.

Dans le but de généraliser cet algorithme au cas de m mémoires, nous essayerons de modéliser la synchronisation pour chaque mémoire de façon à ce qu'elle présente un aspect modulaire.

Nous avons vu que le réseau de la figure III_1 modélise le problème des lecteurs-écrivains, ou plus exactement les contraintes suivantes:

a) exclusion entre lecteurs et écrivains
b) exclusion entre les écrivains

Ces contraintes assurent l'intégrité de la mémoire,et nous servirons de base par la suite.

Quand on aborde la cohérence mutuelle, deux solutions s'offrent , soit l'écriture est faite simultanément sur toutes les mémoires, soit elle est réalisée sur une mémoire et les mises à jour sur les autres, sont différées

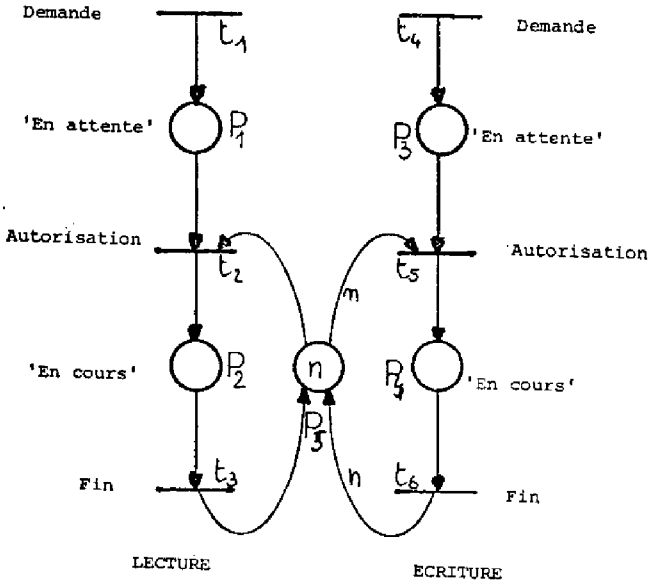


Figure III_1.

2.1. *Mises à jour simultanées*

C'est le cas qui correspond aux algorithmes des bases de données réparties. On attend que toutes les mémoires possédant les données à modifier soient disponibles pour lancer la modification (écriture) sur toutes.

Cette écriture étant considérée comme indivisible, elle interdit toute autre transaction tant qu'elle n'est pas finie sur toutes les mémoires.

Ceci se traduit par la contrainte suivante :

- il doit y avoir exclusion entre lecteurs et écrivains sur toutes les mémoires.

Le réseau de la figure III_1 décrit bien encore ce type de contrainte et nous avons vu au chapitre II l'analyse qui peut être faite sur ce réseau.

Ce mode de fonctionnement ne nous satisfait pas à cause des pertes de temps qu'il impose. En effet, il faut que toutes les mémoires remplissent les conditions d'exclusion avant d'autoriser la modification.

En fait, pendant une transaction écriture, tout parallélisme est interdit et cet algorithme ne présente un intérêt que si l'occurrence des transactions écriture est très faible.

Aussi nous avons défini un autre mode de fonctionnement pour diminuer les pertes de temps et augmenter le parallèlisme du système.

2.2. *Mises à jour différées*

2.2.1. *Contraintes* :

Afin d'augmenter le parallèlisme, on peut se contenter de modifier une seule mémoire pendant l'écriture et lorsque cette écriture est terminée, la répercuter sur les autres (par des mises à jour ou actualisation) dès que possible, c'est-à-dire dès que les transactions "lecture" en cours dans la mémoire sont toutes terminées.

Le parallèlisme peut être alors augmenté en autorisant de nouvelles transactions de lecture dès la fin de l'écriture et donc simultanément avec la mise à jour des autres mémoires

Ceci conduit à décomposer l'écriture. Nous définissons un cycle d'écriture comme l'écriture elle-même et les diverses mises à jour (ou actualisations).

Pour que les mises à jour se fassent correctement, il est nécessaire d'avoir au plus une écriture à la fois dans le système.

Ainsi en plus des contraintes (a et b) définies par [COU 71] ce mode de fonctionnement en impose de nouvelles :

c) Dés qu'une requête d'écriture est faite, et pendant toute l'écriture plus aucune autorisation de lecture ne sera donnée.

d) Un seul cycle d'écriture à la fois.

e) Exclusion entre écriture et mises à jour, et entre mises à jour. (Pour être sûr que les transactions de type écriture se font en séquence sur des versions mises à jour et donc que la cohérence est sauvegardée).

f) Enfin nous voulons qu'aucune transaction (soit de lecture, soit d'écriture) ne puisse être retardée indéfiniment (phénomène de famine).

2.2.2. *Spécification par réseau de Petri* :

La modélisation de ces contraintes par réseau de Petri est faite à la figure III_2, et la signification des places et des transitions est donnée par le tableau III_1.

Plus précisément pour ce réseau, les sommets ont le sens suivant :

- Les places TL_i sont utilisées pour empêcher toute autorisation de lecture tant qu'une mémoire n'a pas été mise à jour (c'est-à-dire tant qu'elle n'a pas la version la plus récente).

Elles servent d'autre part à assurer l'unicité du cycle d'écriture.

- La place TE pour réaliser l'exclusion entre écritures et mises à jour.

- La transition PAE est tirée lorsque commence un nouveau cycle d'écriture.

Les transitions DL et DE et les places FAE et FAL n'ont d'autre intérêt que de modéliser respectivement les demandes et les files d'attente des transactions de lecture et d'écriture.

	ECRITURE	LECTURE
Demande	DE	DL
File d'attente	FAE	FAL
pré-autorisation	PAE	
attente pour mémoire i	EA_i	
autorisation sur mémoire i	AE_i	AL_i
en cours sur mémoire i	EC_i	LC_i
fin sur mémoire i	FE_i	FL_i
Test commun	TLE_i	
Test d'exclusion	TE	TL_i

Tableau III_1.

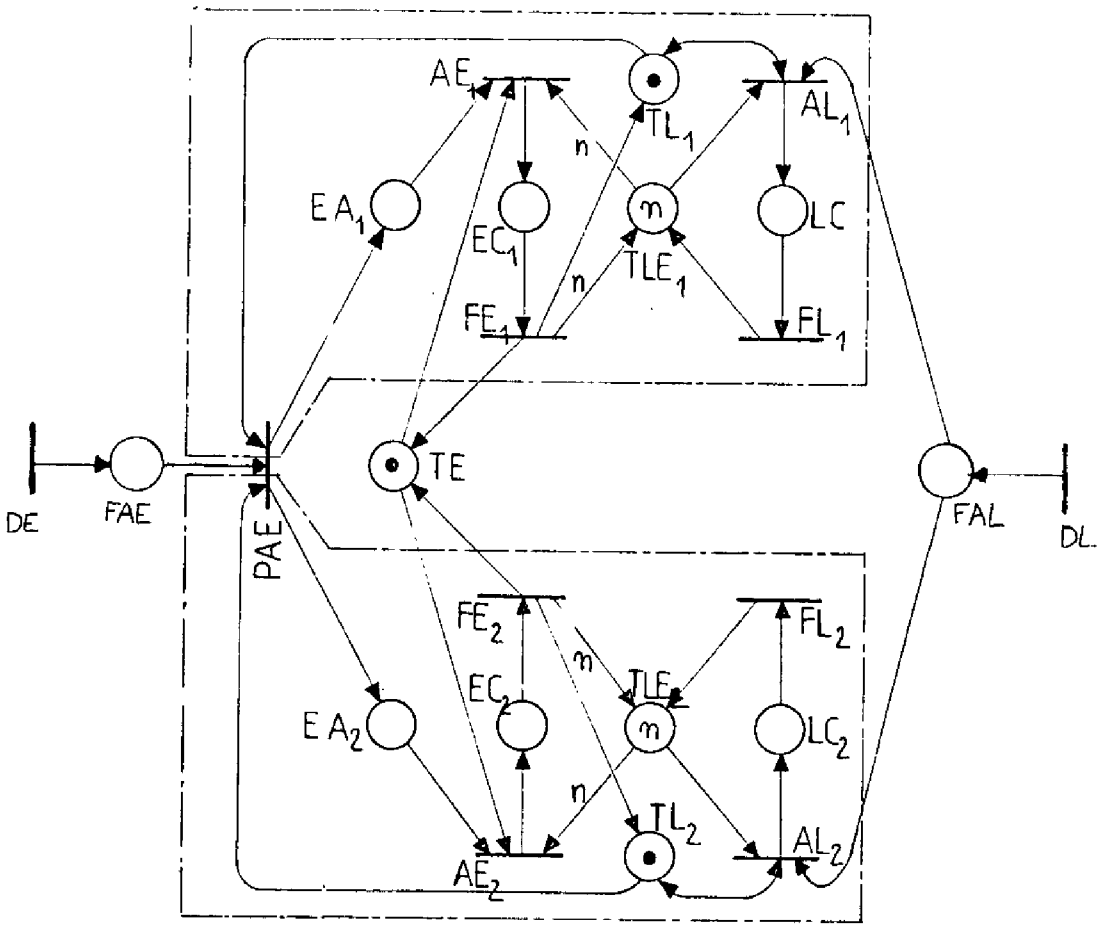


Figure III_2.

La spécification de l'algorithme est faite pour deux mémoires, cependant du fait de la modularité présentée par le réseau, il est très facile de le généraliser au cas de m mémoires par le fusionnement des transitions de même nom des réseaux modulaires et du réseau de la figure III_3.

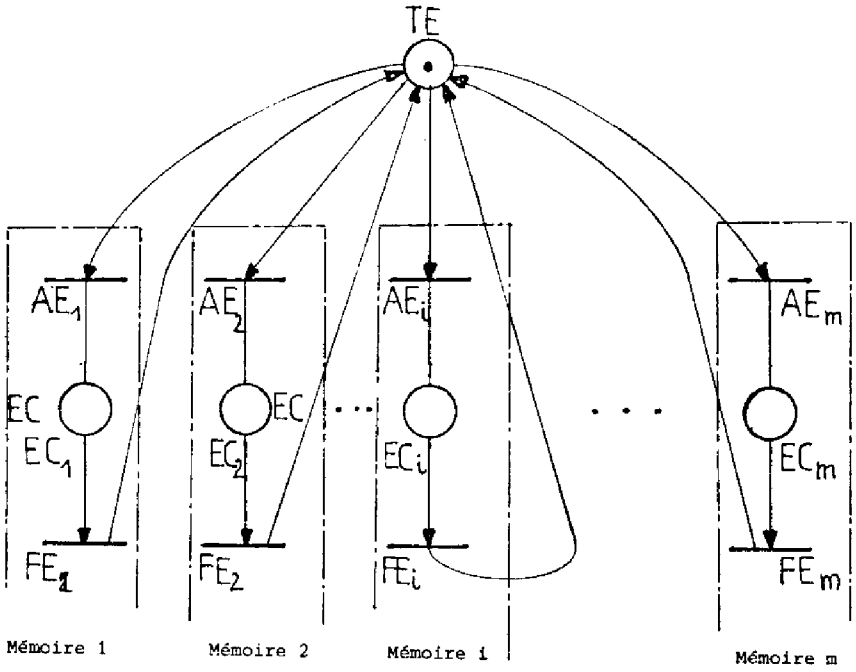


Figure III_3.

Quand on impose des contraintes de surveillance sur les utilisateurs des mémoires, le réseau se transforme en réseau à jetons individualisés de la figure III_4.

2.2.3. Analyse :

Nous avons analysé le réseau à l'aide d'OGIVE. Ce réseau étant complétement réductible, il est vivant et donc sans blocage , ceci quelle que soit la valeur de n (il est possible de vérifier manuellement que la réduction ne dépend pas de n).

La seconde phase de l'analyse consiste à prouver d'une part la "bonne" modélisation des contraintes, et d'autre part que l'introduction des identificateurs n'amène pas de blocage.

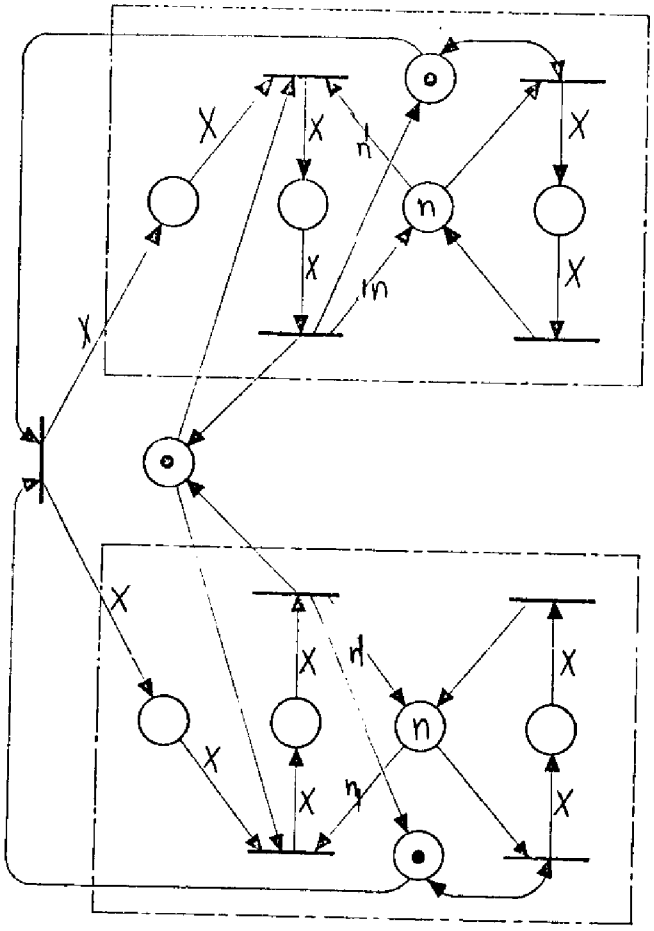


Figure III_4.

Pour ce réseau les invariants intéressants sont donnés dans ce qui suit

En premier lieu examinons ceux obtenus (système III_1) à l'aide d'OGIVE par la résolution automatique du système d'équations linéaires $|C|\underline{P} = \underline{0}$ avec n = 3

$$III_1 \begin{cases} 3\,|M(EC_1)| + |M(TLE_1)| + |M(LC_1)| = 3 & (1) \\ 3\,|M(EC_2)| + |M(TLE_2)| + |M(LC_2)| = 3 & (2) \\ |M(EC_1)| + |M(EC_2)| + |M(TE)| = 1 & (3) \\ |M(EA_1)| + |M(EC_1)| + |M(TL_1)| = 1 & (4) \\ |M(EA_2)| + |M(EC_2)| + |M(TL_2)| = 1 & (5) \end{cases}$$

Pour les invariants (1) et (2), il est possible de les généraliser pour n'importe valeur de n par une résolution partielle et manuelle du système $|C|\underline{\rho} = \underline{0}$.

En effet, ils sont déductibles du système d'équations linéaires $|C_1|\,\underline{\rho}_1 = \underline{0}$ où $|C_1|$ est la matrice d'incidence du réseau formé exclusivement par les places appartenant aux invariants et leurs transitions d'entrée et de sortie. ρ_1 est lui un vecteur entier se rapportant aux places de ces invariants.

Donc le système III_1 est généralisable pour n'importe quel n. On obtient alors le système III_2.

$$III_2 \begin{cases} n\,|M(EC_1)| + |M(TLE_1)| + |M(LC_1)| = n & (6) \\ n\,|M(EC_2)| + |M(TLE_2)| + |M(LC_2)| = n & (7) \\ |M(EC_1)| + |M(EC_2)| + |M(TE)| = 1 & (8) \\ |M(EA_1)| + |M(EC_1)| + |M(TL_1)| = 1 & (9) \\ |M(EA_2)| + |M(EC_2)| + |M(TL_2)| = 1 & (10) \end{cases}$$

Considérons maintenant le cas de m réseaux modulaires fusionnés avec le réseau de la figure III_3.

En ce qui concerne les invariants (6) et (7), ils restent valables quelque soit le nombre de modules puisque l'on ne touche pas aux places internes des modules.

$$\forall\, i,\quad n\,|M(EC_i)| + |M(TLE_i)| + |M(LC_i)| = n \qquad (11)$$

Rappelons que ces invariants assurent le respect des contraintes (a) et (b) pour le réseau. En effet, nous pouvons en déduire

$$M(EC_i) \;.\; M(LC_i) = 0$$

et

$$0 \leqslant M(EC_i) \leqslant 1$$

qui montrent les exclusions à remplir sur chaque mémoire entre écritures et lectures, et entre écritures.

La généralisation au cas de m mémoires de l'invariant (8) se fait à partir de la figure III_3.

Cette figure est la représentation d'une machine à états et pour ce réseau, l'existence d'un jeton unique, lors du marquage initial prouve que l'invariant suivant est vérifié.

$$\sum_{i=1}^{m} |M(EC_i)| + |M(TE)| = 1 \qquad (12)$$

ceci montre le respect de la contrainte (e).

En effet, si une mémoire i est dans une phase du cycle d'écriture, c'est-à-dire s'il existe une place EC_i dont le marquage est égal à 1,

$$\exists i \quad \text{tel que} \quad |M(EC_i)| = 1$$

Alors nous aurons obligatoirement pour toutes les autres places EC_j le marquage nul.

$$\forall j \neq i \qquad |M(EC_j)| = 0$$

ceci prouve bien l'exclusion entre écriture et mise à jour ou entre les mises à jour elles-mêmes.

De la même manière que pour les invariants (6) & (7), les invariants (9) & (1o) sont généralisables pour un nombre quelconque de modules, puisqu'ils sont spécifiques à chaque module. Ils deviennent :

$$|M(EA_i)| + |M(EC_i)| + |M(TL_i)| = 1 \qquad (13)$$

Ces invariants aident à vérifier la contrainte (c). En effet, si une mémoire i est dans une phase du cycle d'écriture ($|M(EC_i)| = 1$) ou en attente ($|M(EA_i)| = 1$), la place TL_i a obligatoirement un marquage nul ($|M(TL_i)| = 0$) et donc la transition AL_i n'est pas sensibilisée, et réciproquement.

En faisant la somme membre à membre de tous les invariants (13) se rapportant à une mémoire, on obtient :

$$\sum_{i=1}^{m} \left[|M(EA_i)| + |M(EC_i)| + |M(TL_i)| \right] = m \qquad (14)$$

qui aide à montrer l'unicité de l'écriture. En effet, c'est seulement quand toutes les mémoires sont à jour ($\sum_{i=1}^{m} |M(TL_i)| = m$) et donc qu'il n'y a pas d'écriture ou de mise à jour en attente ou en exécution ($\sum_{i=1}^{m} \left[|M(EA_i)| + |M(EC_i)| \right] = 0$) que la transition PAE est sensibilisée.

Ainsi une transaction écriture quand elle commence, empêche le passage d'une autre de même type tant que toutes les mémoires ne possèdent pas les mêmes données.

Abordons maintenant le cas de la contrainte (f), et examinons tour à tour les deux types de privations.

I) Famine des écrivains :

Plaçons-nous dans le cas favorable, où toutes les mémoires sont affectées à des transactions de lecture.

Dès qu'un écrivain apparaît, la transition PAE est tirée. Ce tir implique que le marquage des places TL_i est nul :

$$\forall i \qquad |M(TL_i)| = 0$$

ainsi plus aucune transaction lecture n'est autorisée (aucune transition AL_i n'est sensibilisée).

Notons toutefois que la transaction d'écriture ne pourra commencer que lorsque toutes celles de lecture seront finies sur une mémoire

II) Famine des lecteurs

Pour ce cas aussi plaçons-nous dans la configuration la plus défavorable ; lorsque la transition PAE vient d'être tirée et que plus aucune lecture n'est possible.

Comme le cycle d'écriture se décompose en l'écriture elle-même et ses diverses mises à jour et du fait de leur exclusion mutuelle, il existera, avant ou pendant une actualisation, une mémoire i qui vérifiera:

$$\exists i \text{ tel que } |M(EA_i)| = |M(EC_I)| = 0$$

impliquant $|M(TL_i)| = 1$ et donc que la transition AL_i est sensibilisée en cas de lecture en attente.

Enfin il nous reste à prouver que l'introduction d'étiquettes sur les arcs ne peut provoquer de blocage. En fait cette preuve est immédiate, puisque chaque transition du réseau de la figure III_4 possède au plus un arc d'entrée étiqueté, et donc ces arcs ne peuvent pas introduire de blocage tel que celui de la figure II_{18a}. De plus la préservation du nom est assurée lors du tir d'une transition par l'identité des étiquettes sur les arcs entrant et sortant de chaque transition. Ainsi par exemple, ne peuvent déclarer une fin de lecture que ceux qui ont demandé une lecture et ont été autorisés à lire.

3. *FONCTIONNEMENT EN MODE DEGRADE*

Avant d'étudier plus en détail les possibles dégradations du système, nous rappelons ici les éléments qui composent un ensemble (ou bloc) mémoire. A savoir :

- la mémoire,
- le processeur arbitre,
- le bus qui les relie aux processeurs utilisateurs.

Dans la structure globale deux types de dégradations peuvent se produire :

I) Le premier, par la panne d'un élément entraînant la mise hors service d'un ensemble mémoire ; ce qui obligera à l'isoler des autres jusqu'à sa réintégration après réparation.

II) Le second, par la perte, localement sur une mémoire, de la cohérence des données due par exemple à un mauvais fonctionnement d'un processeur en écriture. Dans ce cas aussi la mémoire est isolée avant d'être restaurée à partir des autres.

Les paragraphes qui suivent montrent les incidences qu'ont ces mauvais fonctionnements sur la synchronisation et donc sur l'algorithme.

3.1. Panne d'un élément d'un bloc mémoire

3.1.1. Contraintes :

C'est le cas rencontré quand au moins un des éléments d'un bloc mémoire est en panne. Les nouvelles contraintes sont :

(g) aucune transaction, (lecture et écriture), ne sera autorisée sur une mémoire défaillante,

(h) le système doit continuer à fonctionner en mode dégradé tant qu'il reste au moins un ensemble mémoire valide.

3.1.2. Spécification par réseau de Petri :

L'incidence de ces nouvelles contraintes se traduit sur le réseau de la figure III_3 par de nouveaux sommets. Les liaisons entre ces sommets et le réseau initial sont données dans le réseau de la figure III_5. En fait comme ces contraintes sont applicables individuellement nous n'avons représenté leurs impacts que sur un seul module.

La signification des nouveaux sommets de ce réseau est donnée par le tableau III_3.

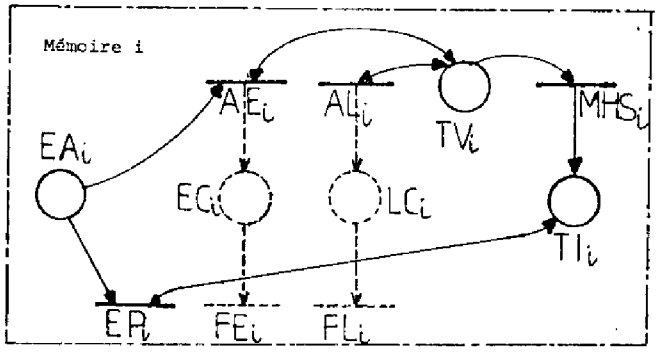


Figure III$_5$.

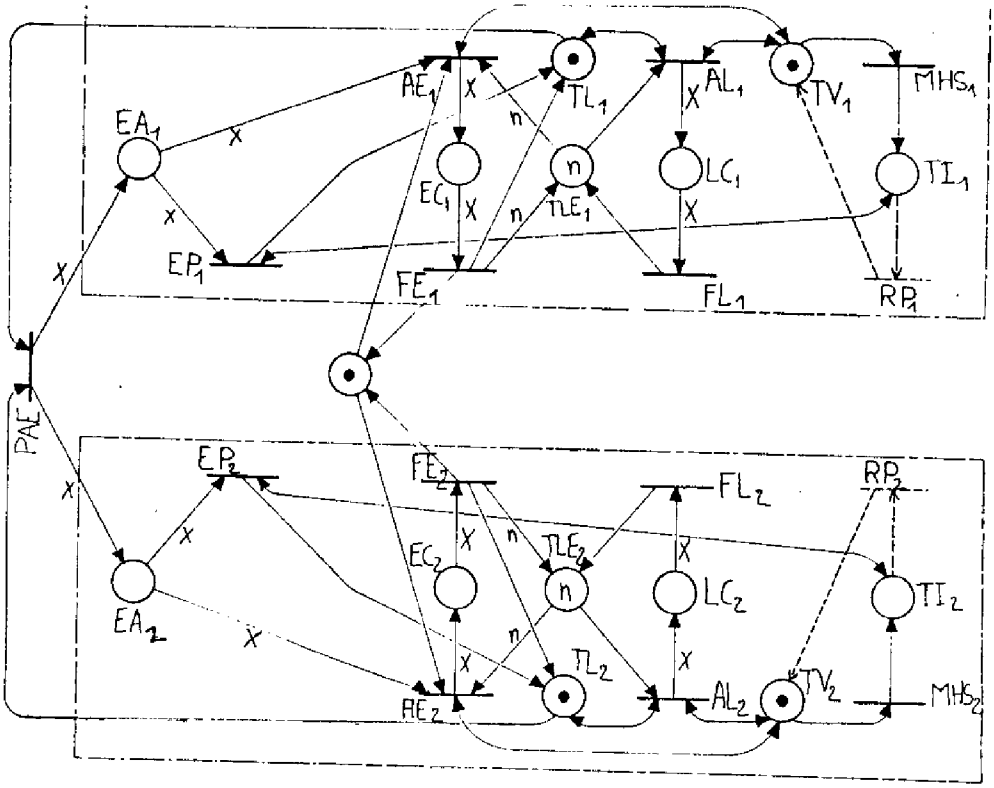


Figure III$_6$.

Ecriture prohibée	EP_i
Mise hors service	MHS_i
Test de validité	TV_i
Test d'invalidité	TI_i

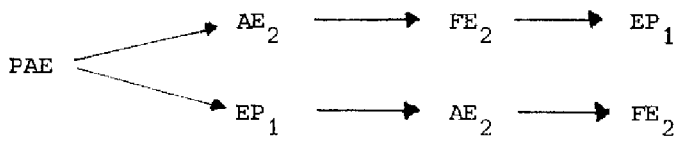
Tableau III_3.

3.1.3. Analyse :

Nous avons fusionné les sommets de même nom des deux réseaux de Petri des figures III_4 et III_5, et nous avons étiqueté les arcs pour tenir compte des contraintes de surveillance. Le réseau de Petri à jetons individualisés est alors celui de la figure III_6.

Pour que ce réseau soit vivant, il nous a fallu rajouter à chaque module la transition RP_i dont le sens est le suivant : si une mémoire est déclarée en panne, il arrive un moment où étant réparée, elle est remise en service.

Ainsi ce réseau possède les propriétés d'être vivant, borné et réinitialisable. En fait la propriété de réseau vivant ne suffit pas à elle seule pour vérifier la contrainte (h) (par la présence des transitions RP_i). Cependant elle est vérifiable simplement à partir de séquences de tir. En effet, si une mémoire i est déclarée en panne, (tir de la transition MHS_i) ce qui implique que $|M(TV_i)| = 0$ et donc que les transitions AL_i et AE_i ne sont plus sensibilisées (contrainte g) ; rien n'empêche, si l'autre mémoire est bonne, de reporter toutes les transactions sur elle. Si le réseau se suffit à lui-même pour le vérifier dans le cas des lectures, nous tenons à préciser le déroulement du cycle d'écriture, en présence par exemple de la mémoire 1 en panne $|M(TV_1)| = 0 \Longleftrightarrow |M(TI_1)| = 1$. L'exécution du cycle se fera alors par l'une des séquences suivantes :



Le marquage résultant autorise, le cas échéant, le passage d'une autre écriture.

3.2. *Reconfiguration mémoire*

Dans le dernier réseau (figure III_6), rien n'indique la façon dont est réalisée la réintégration de la mémoire dans le système (restauration) ni quelles sont les contraintes que doit respecter le système pendant une réinsertion d'une mémoire.

3.2.1. *Contraintes* :

Cette nouvelle transaction (restauration) doit s'effectuer en ligne à partir d'une mémoire valide. Dans le cas de la structure PASTELS (figure III_7) cette restauration s'effectuera à partir de l'autre mémoire.

D'autre part pour conserver l'unicité des données il faut interdire toute écriture pendant la restauration.

Ainsi le système devra répondre aux contraintes suivantes :

(i) la restauration doit être faite en exclusion mutuelle par rapport aux cycles d'écriture.

(j) la restauration doit être faite à partir de l'autre mémoire.

3.2.2. *Spécification par réseau de Petri*

Ces deux contraintes sont représentées par les réseaux de Petri des figures III_7 & III_8 (respectivement les contraintes (i) et (j)).

Les nouveaux sommets correspondent à la transaction restauration sur chaque mémoire, sauf la place CM qui sert à l'exclusion entre la restauration et les cycles d'écriture. Le tableau III_4 résume leur signification.

Autorisation de restaurer	AR_i
Restauration en cours	RC_i
Fin de restauration	FR_i
Compteur mémoire	CM

Tableau III_4.

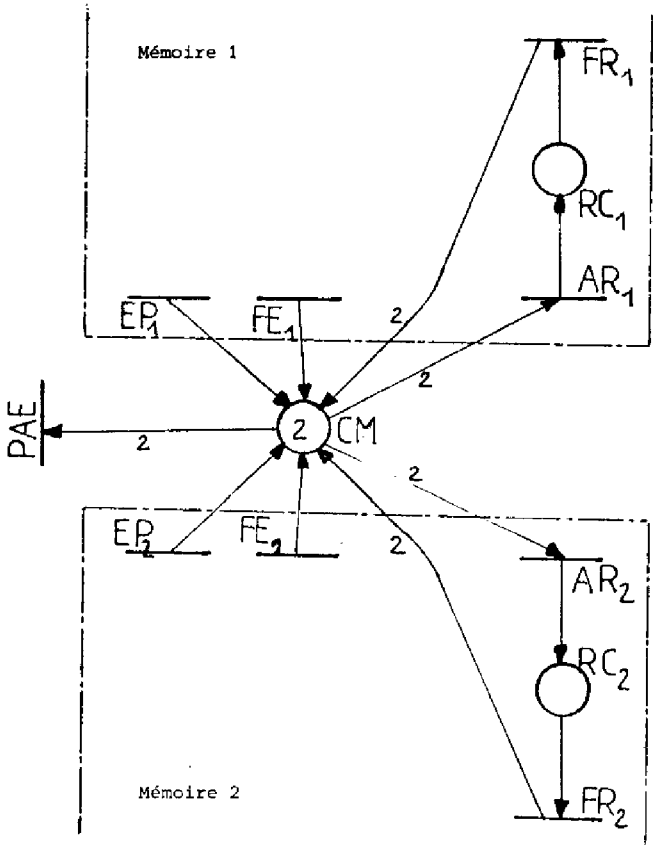


Figure III_7.

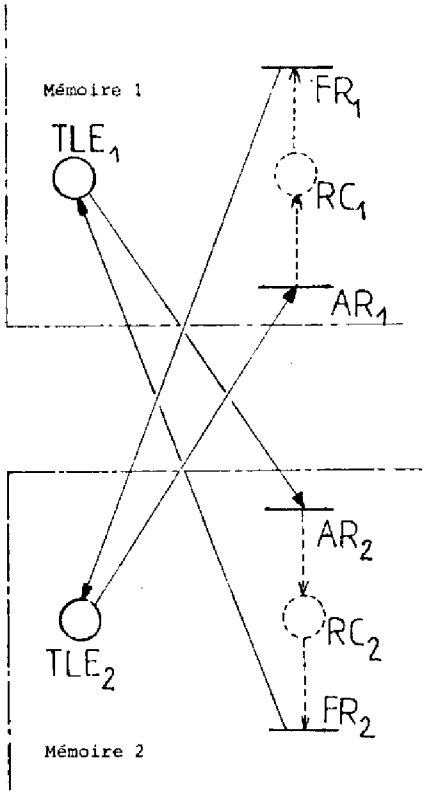


Figure III_8

Dans notre cas (système PASTELS actuel), nous considèrerons qu'il a au plus une mémoire en panne à tout instant. Aussi n'y a-t-il pas d'autres contraintes à introduire dans les spécifications.

Dans le cas d'un plus grand nombre de modules, il faut s'assurer, par contre, que la mémoire servant de source est bien valide. Ceci se traduit par une nouvelle contrainte dont nous verrons les effets à l'annexe II_1.

La conséquence finale de la restauration d'une mémoire est de la rendre valide pour le système. Ceci se traduit par le réseau de la figure III_9.

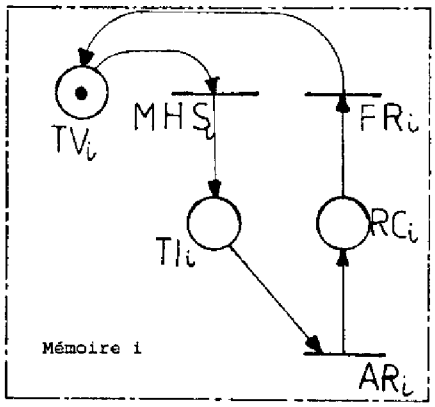


Figure III_9.

3.2.3. Analyse :

Si on fusionne les sommets de même nom des réseaux des figures III_6, III_7, III_8, III_9 on obtient le réseau de Petri à jetons individualisés qui spécifie l'algorithme tolérant les pannes mémoires.

Le réseau de Petri simple a été analysé à l'aide d'OGIVE, et il possède les propriétés "borné, vivant et réinitialisable". Ainsi, il est sans blocage.

Les invariants linéaires intéressants pour l'analyse structurelle sont en plus des invariants (8,9,10), les invariants suivants :

$$III_3 \begin{cases} |M(TV_1)| + |M(TI_1)| + |M(RC_1)| = 1 & (15) \\ |M(TV_2)| + |M(TI_2)| + |M(RC_2)| = 1 & (16) \\ |M(CM)| + |M(EA_1)| + |M(EA_2)| + |M(EC_1)| + |M(EC_2)| + 2|M(RC_1)| \\ \quad + 2|M(RC_2)| = 2 & (17) \\ n|M(EC_1)| + |M(TLE_1)| + |M(LC_1)| + |M(RC_2)| = n & (18) \\ n|M(EC_2)| + |M(TLE_2)| + |M(LC_2)| + |M(RC_1)| = n & (19) \end{cases}$$

Les conclusions faites par l'analyse des invariants (8,9,10) restent vraies, et donc les contraintes (c d e) sont toujours vérifiées par ce réseau.

Les invariants (15) & (16) montrent respectivement sur les deux modules les états que peuvent prendre une mémoire.

$\lvert M(TV_i)\rvert = 1$	mémoire valide
$\lvert M(TI_i)\rvert = 1$	mémoire invalide
$\lvert M(RC_i)\rvert = 1$	mémoire en cours de restauration

En fait ils montrent qu'une mémoire ne peut être que dans un seul de ces états à la fois.

L'invariant (17) lui permet de vérifier la contrainte (i). De (17) nous pouvons déduire :

$$\forall\, i \in \{1,2\} \quad , \quad M(RC_i) \;.\; \left[\sum_{j=1}^{2} \left[\lvert M(EA_j)\rvert + \lvert M(EC_j)\rvert\right]\right] = 0$$

ceci montre bien l'exclusion entre restauration et écriture.

Les invariants (18) & (19) eux sont respectivement la suite logique des invariants (6) & (7). Ils montrent en plus des contraintes (a) et (b), qu'une restauration d'une mémoire se fait à partir de l'autre (contrainte j). Sur cette mémoire source, la restauration est donc considérée comme une transaction de lecture (particulière toutefois).

Remarquons cependant, que la contrainte (f) n'est plus vérifiée, car si une mémoire est invalide, il y a possibilité de famine des lecteurs par l'arrivée continue d'écrivains (cf. § 3.1.3.).

3.3. Panne processeur

Une panne d'un processeur peut produire une perte d'intégrité des données sur une mémoire. Par exemple, cette perte peut être provoquée par la panne d'un processeur en train d'écrire (d'autres cas sont examinés au paragraphe IV_4). Une des particularités de la structure PASTELS est qu'elle permet de surmonter cette perte d'intégrité par une procédure de reprise très proche de ce qui est couramment connu sous le nom de "Roll-back".

Ainsi pour tenir compte de ce type de panne entraînant une perte d'intégrité locale des données nous avons créé une nouvelle transaction, appelée "restauration partielle". Elle consiste à ne restaurer que les données erronées à partir de l'autre mémoire.

3.3.1. *Contraintes* :

Pour l'intérêt de la structure, cette restauration doit se faire en ligne et est déclenchée automatiquement par PS.

Ainsi les contraintes qu'elle impose sont :

(k) la restauration partielle se fait à partir de l'autre mémoire et en ligne

(l) la restauration partielle et les lectures sur la mémoire source sont en exclusion mutuelle.

Cette dernière contrainte est d'ordre technologique et imposée par le processeur d'arbitrage qui ne peut gérer en même temps ces deux types de transactions.

Ici aussi, il faudrait tester la validité de la mémoire source pour être sûr du fonctionnement, cependant nous nous plaçons dans un premier temps dans le cas d'une seule mémoire en panne. Nous examinerons la généralisation pour m mémoires dans l'annexe II_2.

3.3.2. *Spécification par réseau de Petri* :

La figure III_{10} représente le réseau de Petri à jetons individualisés modélisant la mise hors service partielle des mémoires, alors que celui de la figure III_{11} représente la restauration partielle. Il s'interprète de la manière suivante. Pour autoriser une restauration partielle, il suffit que les lectures soient finies sur l'autre mémoire (contrainte l). A la fin de cette restauration, il faut remettre le système dans l'état correspondant à la fin d'un cycle d'écriture.

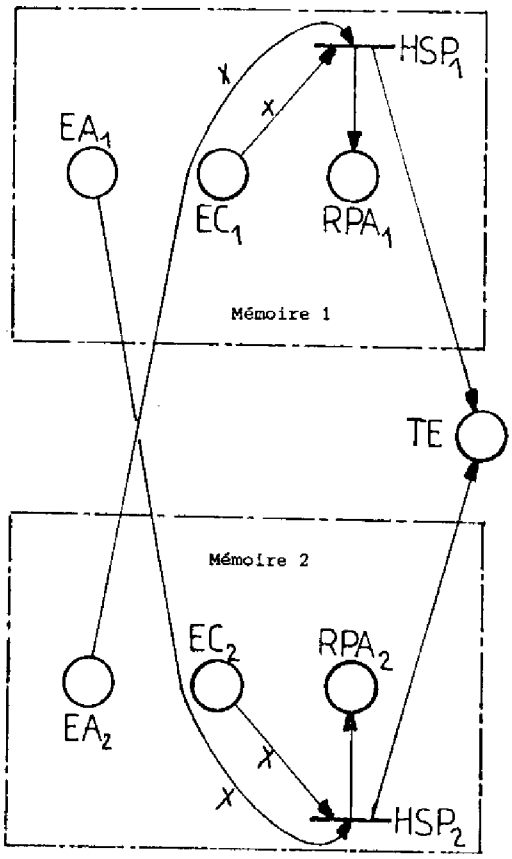


Figure III$_{10}$

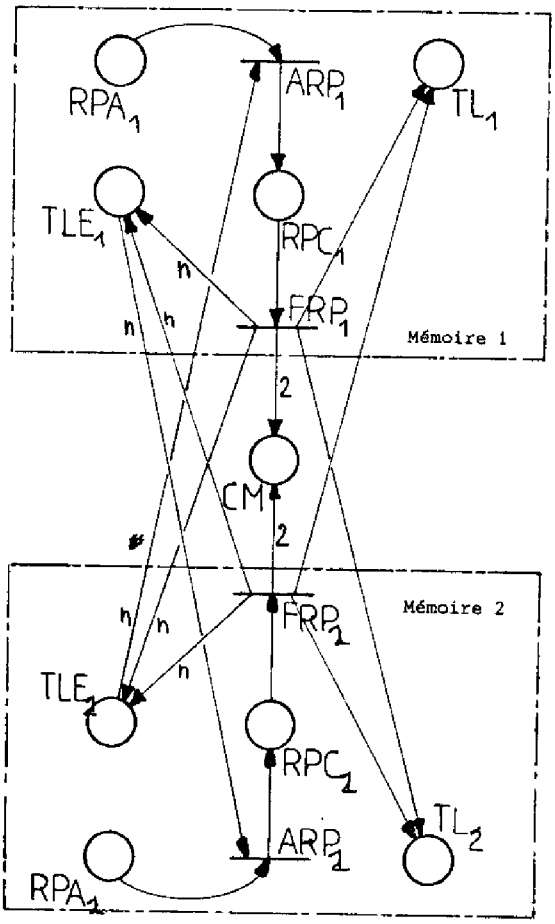


Figure III$_{11}$

Les nouveaux sommets spécifiant cette transaction de restauration ont la signification donnée par le tableau III$_5$.

Mise hors service temporaire	HSP_i
Restauration partielle en attente	RPA_i
Autorisation de restauration partielle	ARP_i
Restauration partielle en cours	RPC_i
Fin de restauration partielle	FRP_i

Tableau III$_5$

3.3.3. Analyse :

En fusionnant les sommets de même nom des réseaux des figures III_{10} et III_{11} avec le résultat du dernier fusionnement, nous avons spécifié par réseau de Petri l'algorithme global du maintien de la cohérence de données dupliquées incluant la restauration partielle.

Le réseau sans contrainte de surveillance sur la validité de la mémoire source a été analysé à l'aide d'OGIVE. Le résultat est que ce réseau est absent de blocage, et suffit dans le cas de deux mémoires si l'on considère qu'il y en a toujours au moins une de bonne. Cependant ce réseau n'est pas complétement réductible, et l'analyse faite à l'aide de l'énumération des marquages n'est valable que pour n = 3.

Les invariants linéaires de places qui aident à la vérification des contraintes sont en plus des invariants (8), (15) et (16), les suivants (système III_4) :

$$III_4 \begin{cases} |M(EA_1)| + |M(EC_1)| + |M(RPA_1)| + |M(RPC_1)| + |M(RPA_2)| + |M(RPC_2)| + |M(TL_1)| = 1 & (20) \\ |M(EA_2)| + |M(EC_2)| + |M(RPA_1)| + |M(RPC_1)| + |M(RPA_2)| + |M(RPC_2)| + |M(TL_2)| = 1 & (21) \\ |M(EA_1)| + |M(EA_2)| + |M(EC_1)| + |M(EC_2)| + |M(CM)| + 2\left[|M(RPA_1)| + |M(RPA_2)| + |M(RPC_1)| + |M(RPC_2)| + |M(RC_1)| + |M(RC_2)|\right] = 2 & (22) \\ n\left[|M(EC_1)| + |M(RPA_1)| + |M(RPC_1)| + |M(RPC_2)|\right] + |M(TLE_1)| + |M(LC_1)| + |M(RC_2)| = n & (23) \\ n\left[|M(EC_2)| + |M(RPA_2)| + |M(RPC_2)| + |M(RPC_1)|\right] + |M(TLE_2)| + |M(LC_2)| + |M(RC_1)| = n & (24) \end{cases}$$

(20) & (21) sont la suite logique des invariants (9) & (10) quand on introduit la restauration partielle. Les invariants (20) & (21) interviennent donc pour la preuve de la contrainte (c).

De plus, ils montrent l'exclusion entre cycle d'écriture et restauration partielle. En effet pour chacun d'eux nous aurons :

$$\forall\, i = \{1,2\} \quad , \quad \left[|M(EA_i)| + |M(EC_i)| \right] . \left[\sum_j |M(RPA_j)| + |M(RPC_j)| \right] = 0$$

Quelle est la raison de cette exclusion entre cycle d'écriture et restauration partielle ?

Si un cycle d'écriture se déroule normalement, il ne peut y avoir de restauration partielle et réciproquement si une mémoire est invalide temporairement, le cycle d'écriture est interrompu.

Nous pouvons remarquer de plus, qu'une mémoire ne peut être déclarée hors service temporairement que si elle est bien dans sa phase d'écriture ; tir de $HSP_i \Longrightarrow |M(EC_i)| = 1$ et $\forall j \neq i \quad |M(EA_j)| = 1$.

(22) est l'invariant dérivé de (17) quand on considère le réseau global. Donc les conclusions faites avec (17) pour montrer l'exclusion entre cycle d'écriture et restauration sont toujours valables. (22) permet en plus de montrer l'exclusion entre les types de restauration car de cet invariant il peut être déduit :

$$\forall\, i = \{1,2\} \quad , \quad \left[|M(RPA_i)| + |M(RPC_i)| \right] . |M(RC_i)| = 0.$$

(23) & (24) sont le prolongement logique des invariants (18) & (19). La vérification des contraintes (a), (b) et (j) est la même avec ces invariants (23) & (24) qu'avec (18), (19).

Ils permettent en plus de montrer les exclusions entre restauration partielle et transaction de lecture, en effet nous pouvons en déduire :

$$\forall\, i = \{1,2\} \quad , \quad |M(TL_i)| . \left[\sum_{j=1} |M(RPC_j)| \right] = 0$$

Enfin il nous reste à montrer que l'introduction des identificateurs et variables étiquetées sur le réseau de Petri ne crée pas de blocage. En fait le seul point délicat est celui qui correspond au tir des transitions HSP_i (cf. figure III_{10}).

En effet ce sont les seules qui possèdent plusieurs arcs entrant étiquetés, et donc qui introduisent une possibilité de blocage du type de celui de la figure II_{18a}.

Pour lever cette incertitude, rappelons qu'un cycle d'écriture est unique (quand la transition PAE est tirée elle ne peut l'être à nouveau que lorsque toutes les mémoires sont à jour). Aussi l'identificateur des jetons des places EA_i sera le même puisqu'il est produit par le tir de la transition PAE. Au tir d'une transition AE_j (correspondant au début de la transaction écriture) la conservation du nom est assurée, et le même identificateur se retrouve pour le jeton de la place EC_j. Si cette écriture se déroule mal au point qu'il faille déclarer la mémoire temporairement hors service, rien n'empêche le tir de la transition HSP_j.

4. *CARACTERISTIQUES*

4.1. *Introduction*

Dans ce paragraphe nous allons mettre en évidence les principales caractéristiques de notre algorithme. Pour cela nous allons le comparer suivant certains critères [WIL 79] à plusieurs algorithmes [ELL 77 - LEL 78 - MIR 79 - MUL 75 - SER 79] de maintien de la cohérence de données dupliquées, bien que les deux types d'application (base de donnée, système de commande de processus industriels) soient très différents.

4.2. *Critères*

4.2.1. *Granularité* :

On appelle granule, la plus petite entité à verrouiller durant l'exécution d'une transaction. Ce facteur est très important en ce qui concerne le parallèlisme autorisé par les algorithmes.

Dans le cas de l'algorithme présenté dans ce chapitre, le granule est le plus défavorable possible puisque l'exécution d'une transaction requiert un verrouillage de l'ensemble de la mémoire.

Beaucoup d'algorithmes présentent ce défaut, et seuls quelques uns [THO 78 - BER 79 ...] possèdent l'avantage d'une granularité fine. Cependant cet avantage est contrebalancé par l'accroissement de la taille mémoire nécessaire au stockage des variables de synchronisation servant de verrous.

[ROS 78] a montré qu'il existe un seuil à ne pas dépasser dans la finesse du granule sans nuire aux performances du système.

4.2.2. Parallélisme :

I) intra-transactionnel; il correspond au cas où plusieurs processeurs coopèrent pour effectuer une transaction. Ce type de parallèlisme est impossible à cause des procédures de surveillances mutuelles que nous imposons.

II) inter-transactionnel; il correspond à l'exécution simultanée de plusieurs transactions dans le système. C'est ici que nous voyons l'importance de la finesse du verrouillage puisque le parallélisme y est directement lié. Cependant rappelons que dans notre cas des transactions de lecture peuvent se dérouler sans pour autant que toutes les mémoires aient été mises à jour. Aussi entre transactions différentes, il existe une possibilité de parallèlisme.

Soulignons cependant que dans le cas de transactions de type lecture, leur déroulement simultané sur une mémoire est contrôlé par l'intermédiaire des places TLE_i. La valeur limite n est telle qu'elle évite tout phénomène de saturation des requêtes sur une mémoire. Il est à noter aussi que cette stratégie permet de répartir la charge des utilisateurs sur l'ensemble des mémoires valides.

4.2.3. Cohérence :

I) cohérence forte; elle implique une exécution de toutes les mises à jour avant d'autoriser de nouvelles transactions.

II) cohérence faible; au contraire elle permet l'exécution en parallèle de plusieurs transactions sachant qu'au bout d'un temps fini (mais bien souvent indéfini) l'unicité des données sera réalisée.

D'après ces deux définitions une remarque s'impose. En effet, plus le degré de parallèlisme est élevé, plus le degré de cohérence est faible, et au contraire un algorithme aura une cohérence forte si son granule est la copie entière.

Dans notre cas, seules des transactions de lecture peuvent se dérouler en parallèle pendant la mise à jour de mémoires, alors que la granularité de certains algorithmes autorise l'exécution simultanée de plusieurs transactions d'écriture et de lecture . Aussi pour notre algorithme, le degré de cohérence est-il très près de la cohérence forte.

4.2.4. *Résilience* :

Le degré de résilience se définit par le nombre de sites valides nécessaires sur les m sites initiaux pour que le système fonctionne.

Dans notre cas la notion de site n'existe pas, aussi nous définirons le degré de résilience pour les ensembles mémoires, et il est facile de voir que le système fonctionnera tant qu'au moins un ensemble mémoire est valide.

4.3. *Comparaison*

	ELL 77	LEL 78	MIR 79	MUL 75	SEG 79	LAT 80
COHERENCE	FORTE	FORTE	FAIBLE	FORTE	FORTE	(FORTE)
INTRA TRANSACTION	NON	OUI	OUI	OUI	OUI	NON
INTER TRANSACTION	(OUI)	OUI	OUI	NON	NON	(OUI)
GRANULARITE	/	/	ENTITE	COPIE	COPIE	COPIE
CONTROLE	REPARTI	REPARTI	MAITRE	MAITRE	MAITRE	MAITRE
RESOLUTION CONFLITS	ATTENTE	ATTENTE	REJET	VERROU GLOBAL	VERROU GLOBAL	ATTENTE
ALLOCATION RESSOURCES	STATI.	STATI	DYNA	STATI	STATI	STATI
PERTE D'UNICITE	(OUI)	OUI	OUI	/	NON	NON
RESILIENCE	N	N	N	1	N/2	N
RESTAURATION	(OUI)	OUI	OUI	NON	OUI	OUI

Tableau III$_6$

Dans le tableau III$_6$ sont récapitulées les principales caractéristiques de notre algorithme [LAT 80]. Juxtaposées à elles figurent celles d'autres algorithmes qui sont un échantillon de l'ensemble des algorithmes traitant ce sujet.

5. *CONCLUSION*

Nous venons de décrire dans ce chapitre, l'algorithme de synchronisation à implanter dans le processeur de synchronisation pour le système étudié. Cet algorithme permet de maintenir l'intégrité de données dupliquées. Toutefois, nous n'attendons pas que les copies soient toutes identiques pour lancer une transaction.

L'intérêt de ce chapitre, est la preuve formelle de l'absence de blocage du réseau spécifiant les mécanismes de synchronisation malgré la lourdeur de ces derniers.

Pour nous assurer du sens de ce réseau, nous avons vérifié, le plus souvent à l'aide des invariants, que les contraintes définissant l'algorithme étaient bien respectées.

Nous avons aussi essayé de définir les caractéristiques de notre algorithme. Pour cela nous nous sommes servis de critères de classement utilisés pour différencier les algorithmes de bases de données dupliquées. Bien que les domaines d'applications soient différents entre notre système et les bases de données réparties, nous donnons à la fin de ce chapitre un tableau récapitulatif des principales caractéristiques de divers algorithmes. Si nous examinons ce tableau, nous nous apercevons qu'un point faible de notre algorithme est le manque de "granularité" entraînant le verrouillage global de la mémoire pour exécuter une transaction. A part cet inconvénient, nous pouvons remarquer en particulier que nous garantissons par une implémentation centralisée le maintien de l'unicité des données en présence de panne sans diminuer la résilience du système.

CHAPITRE IV

IMPLÉMENTATION

1. INTRODUCTION

Dans ce chapitre, nous allons proposer une implémentation de l'algorithme de maintien de la cohérence de données dupliquées précédemment décrit à l'aide des réseaux de Petri à jetons individualisés.

Dans un premier paragraphe, nous donnerons rapidement les diverses solutions qui sont, à notre connaissance, disponibles à l'heure actuelle pour implémenter un réseau de Petri. Nous indiquerons les raisons qui ont motivé le choix d'une solution logicielle particulière et nous en exposerons les grandes lignes.

Avant d'aborder plus en détail cette réalisation, nous essayerons de resituer les fonctions principales remplies par le processeur de synchronisation, et nous en donnerons l'organisation générale.

Ensuite dans le troisième paragraphe, nous décrirons l'implémentation de notre algorithme en considérant dans un premier temps le cas de m mémoires, puis le cas particulier où il n'y a que deux mémoires. Dans chaque cas, nous proposerons une solution pour optimiser le temps de l'analyse des messages reçu par le processeur de synchronisation.

Enfin dans un dernier paragraphe, nous indiquons comment est réalisé le traitement des alarmes en provenance des processeurs arbitres, ainsi que les conséquences qu'elles ont sur le réseau de Petri.

Cette implémentation est en cours de réalisation. Les programmes ont été écrit en langage PL/M-80 sur le système de développement "INTELLEC". Ils ne pourront être complétement testés que lorsque la structure PASTELS aura été construite.

2. GENERALITES SUR LES REALISATIONS

2.1 Introduction

De nombreux travaux ont donnés le moyen de passer d'une spécification d'un mécanisme décrit par réseau de Petri à son implémentation [TOU 76 - VAL 80 ...].

Ces implémentations correspondent toujours à une simulation du fonctionnement du réseau, simulation qui doit respecter les règles d'évolution avec des contraintes de type "temps réel" plus ou moins fortes.

Les implémentations sont classées en deux grandes catégories :

I) Matérielles [COU 79 - VAL 80], qui se rapportent malheureusement trop souvent aux réseaux de Petri dit "sauf". Seul [COU 79] donne une solution pour les réseaux de Petri généralisés, toutefois la solution proposée est lourde et parait difficile à mettre en oeuvre pour de très gros réseaux.

A la vue de toutes ces restrictions, il nous semble que les applications de ce type de réalisations soient assez limitées. Cependant, elles possèdent le grand avantage d'avoir des temps de réponse très courts.

II) Logicielles [TOU 76 - VAL 79 - VAL 80], qui sont beaucoup mieux adaptées à implémenter les réseaux de Petri généralisés. Cependant leur temps de réponse peut dans certains cas obliger à choisir une solution qui est un compromis entre matériel et logiciel.

Pour notre cas, nous avons été amenés à choisir une implémentation logicielle. En effet, les solutions pour implémenter de façon matérielle les réseaux de Petri à jetons individualisés doivent, à la vue de cellesdes réseaux de Petri généralisés, être trop complexes et trop figées pour être intéressantes. De plus si une solution existe, il n'est pas certain qu'elle soit généralisable à un nombre quelconque d'identificateurs.

2.2. *Réalisations logicielles*

2.2.1 *Moniteurs spécialisés :*

Nous avons déjà présenté le concept de moniteur au chapitre I. Cependant nous allons rappeler le parallèle qui peut être fait entre celui décrit par [KES 77] , et les réseaux de Petri [VAL 79] . En quelques mots, un moniteur est un ensemble de données et de procédures, tandis qu'un réseau de Petri est un ensemble de places et de transitions.

La correspondance entre les deux se fait entre données et places et entre procédure et tir de transition. La structure interne des moniteurs est alors le moyen d'enchaîner des tirs de transitions.

L'implémentation des mécanismes de synchronisation décrits par réseau de Petri à jetons individualisés est réalisable par ce concept de moniteur [VAL 80] . Il suffit pour tenir compte des identificateurs d'introduire des listes de noms à la place des compteurs, et de définir deux opérations "ajoute" et "efface" sur ces listes qui ont même influence que le "+" et "-" sur les compteurs.

Cependant, le concept de moniteur a été conçu pour la spécification et l'implémentation de la synchronisation de systèmes monoprocesseurs multiprogrammés. Son extension aux systèmes multiprocesseurs n'est pas évidente, principalement à cause de la nécessité d'exécuter en exclusion mutuelle ses procédures.

2.2.2. *Automates programmables :*

Pour le cas particulier des automatismes logiques, les réseaux de Petri sont utilisés pour spécifier les séquences de commandes.

Diverses implémentations directes ont été proposées aboutissant à la conception d'automates programmables par réseaux de Petri. Ils sont en général basés sur des microprocesseurs standards qui saisissent les données, simulent le réseau de Petri spécifiant le fonctionnement désiré et envoient les commandes.

Deux approches sont possibles :

I) conduite compilée : Le réseau et ses évolutions sont traduits en code machine, et le programme s'exécute cycliquement en fonction d'événements extérieurs au réseau.
Les événements engendrent des séquences de tir dont le résultat est répercuté sur le monde extérieur du réseau (c'est-à-dire pour nous : la synchronisation à accomplir).

II) conduite tabulaire : Le réseau est traduit en tables manipulées par un programme d'exécution universel qui n'est autre qu'un simulateur de réseau (joueur [TOU 76]).

On peut prévoir une conduite tabulaire qui focalise la recherche par le"joueur" des évolutions possibles du réseau.

Une réalisation de ce type d'automate peut être trouvée dans [SAM 80] et plusieurs exemples de traduction des réseaux sous formes de tables y sont décrits. Le principe de cet automate est le suivant :
Le réseau est décrit sous la forme de tables. Elles permettent de trouver les transitions sensibilisées à partir d'un marquage. Elles sont conçues pour diminuer la place mémoire utilisée et réduire le temps de réponse. Cependant un tel automate n'est applicable que pour des réseaux de Petri "sauf".

L'intérêt majeur que présente ce dernier type d'automates est qu'en cas de modification de réseaux, il suffit simplement de changer les tables, alors que pour un automate à conduite compilée il faut changer tout le programme. Toutefois l'inconvénient de la conduite tabulaire est qu'elle présente souvent dans sa traduction du réseau en données quelques redondances.

Il apparaît clairement que le processeur PS joue un rôle tout à fait analogue à un automatisme logique, la saisie des données est remplacée par la réception des messages en provenance des divers processeurs, alors que l'envoie des commandes correspond aux messages qu'il émet en leurs directions.

Néanmoins, le mécanisme de synchronisation ayant été spécifié par un réseau à jetons individualisés, l'utilisation directe des méthodes précédentes est impossible.

2.3. *Solution retenue*

La solution retenue, est un compromis entre les automates à conduite tabulaire et à conduite compilée.

En effet, le réseau est décrit sous forme de tables, et un simulateur permet de tirer une transition donnée du réseau. Les transitions à tirer sont déterminées par la partie compilée. Celle-ci tire profit du fait que les séquences de tir possibles sont relativement figées pour éviter une scrutation systématique et répétitive de toutes les transitions. Elle permet ainsi de diminuer le temps de réponse.

Il faut souligner que ce que nous appelons la partie "simulateur" n'est en fait que l'embryon d'un simulateur complet. Il ne permet que le tir d'une transition donnée et non d'effectuer une simulation complète du réseau. Il n'est donc pas redondant par rapport à la partie compilée.

3. *DIALOGUE ENTRE LE PROCESSEUR DE SYNCHRONISATION ET LES AUTRES PROCESSEURS*

3.1. *Choix du partenaire*

Nous avons vu que ce processeur de synchronisation est le nerf du système et qu'il articule toutes les liaisons entre les éléments qui le composent. Si sa tâche principale est de gérer l'enchaînement des transactions demandées par les processeurs utilisateurs, il doit tenir compte des échanges qu'il a avec les processeurs arbitres.

Aussi quand il a à traiter une demande de communication en provenance d'un de ces éléments, il établit par l'intermédiaire de ses bus de commande (figure IV_1) la liaison avec son interlocuteur.

En cas de conflit entre les interlocuteurs, les solutions suivantes ont été adoptées.

I) une priorité fixe a été donnée à chaque processeur utilisateur, et arbitre.

II) les processeurs arbitres sont plus prioritaires que les processeurs utilisateurs.

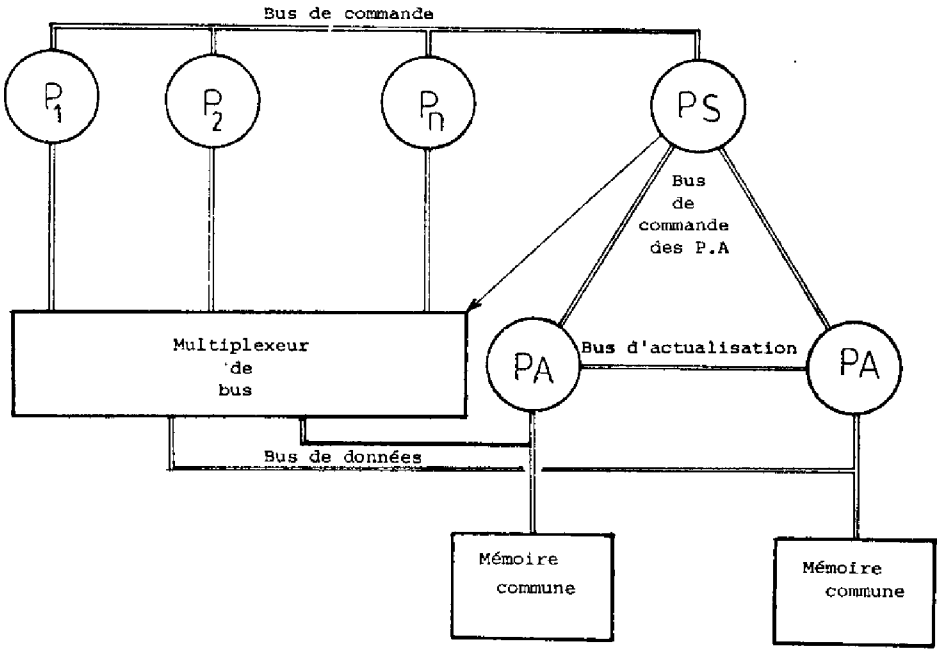


Figure IV_1.

3.2. *Messages échangés avec les utilisateurs*

Si le processeur de synchronisation doit s'occuper d'une demande d'échange en provenance des processeurs utilisateurs, il leur signale le nom du processeur choisi après avoir, le cas échéant, sélectionné le plus prioritaire.

Ce processeur envoie alors sous la forme d'une information codée la nature de son message. Dans tous les cas cette information se rapportera à une des transactions définies au chapitre III. Toutefois nous pouvons les classer en deux catégories : celles qui se rapportent aux demandes de transactions, et celles aux signalements des fins de transactions.

Par le décodage de ce message, le processeur de synchronisation, connaît le tir de la transition (ou la séquence de tirs de transitions) qu'il doit essayer de tirer. Il aura eu auparavant par le choix du processeur, l'initialisation des variables étiquetées du réseau.

Suivant le résultat du tir des transitions de la séquence déclenchée par le message, plusieurs solutions se présentent au processeur de synchronisation comme message réponse.

En fait deux résultats sont possibles, soit la transition (ou la séquence) est tirée, et dans ce cas l'évolution du réseau se traduit par un changement d'état du système, soit la transition est non sensibilisée (ou la séquence de tir n'arrive pas à son terme à cause d'une transition non sensibilisée) et le système ne change pas d'état.

Par l'utilisation des réseaux de Petri à jetons individualisés, cette non-évolution du système a les deux causes suivantes :

I) les conditions de tir sont fausses car les spécifications du problème de synchronisation l'imposent (exclusion mutuelle). Pour le système il correspond dans ce cas à une attente.

II) les conditions de tir sont fausses car la tentative de tir de la (ou d'une) transition résulte d'une fausse manoeuvre (par exemple mauvaise identité du processeur utilisateur).

Il correspond pour ce cas, à un rejet du processeur questionneur.

Un autre facteur influençant le message réponse du processeur de synchronisation est la nature du message qu'il a initialement reçu.

En effet si ce message est :

I) une demande de transaction : les messages réponses possibles sont les suivants :

- accord : si le tir (ou la séquence) a été tirée (ou est arrivée à son terme).

- mise en attente : si le tir (ou la séquence) est interrompue car les conditions de synchronisation ne sont pas vérifiées,

- rejet : si elle est interrompue à cause d'un fonctionnement erroné

II) une fin de message : ce type de message, ne doit pas satisfaire de conditions de synchronisation, et seules des conditions sur l'identité du processeur sont faites. Aussi seul le message de rejet sera envoyé dans ce cas. Le message accord lui peut ou ne pas être envoyé indifféremment, et seule la contrainte temps permet de trancher.

La figure IV_2 récapitule l'ensemble des messages émis et reçus par le processeur de synchronisation.

3.3. Messages échangés avec les arbitres :

Généralement après chaque évolution du réseau de Petri, un message est nécessaire entre le processeur de synchronisation et le processeur arbitre de la mémoire ayant subi l'évolution.

Ainsi en direction de ces éléments les messages codés sont relatifs à la nature et au type de transaction exécutée sur la mémoire. Pour les transactions de lecture et d'écriture figure en plus le nom du ou des utilisateurs. Ceci pour permettre à l'arbitre de surveiller les occupants de la mémoire qui lui est associée.

Dans l'autre sens , les messages seront de deux sortes. Soit pour signaler une fin de transaction automatique (actualisation, restauration partielle) soit pour signaler des anomalies.

Nous examinerons ces cas dans le traitement des alarmes qui est décrit au paragraphe IV_4.

La récapitulation de tous les messages est donnée figure IV_2.

PU	PS	Demande de transaction Fin de transaction
PS	PU	Rejet Mise en file d'attente Accord
PS	PA	Deb. de trans. + (Masque) Fin de trans. +(Masque)
PA	PS	Fin de transaction Alarme

Figure IV_2.

3.4. *Organisation globale*

Le réseau de Petri de la figure IV_3 représente l'enchaînement des tâches du processeur de synchronisation vu au niveau le plus global. On peut remarquer qu'un seul message est traité à la fois. Pour des raisons de commodité nous avons séparé les messages en provenance des processeurs arbitres de ceux en provenance des processeurs utilisateurs. Toutefois dans les deux cas, les grandes lignes du traitement sont les mêmes. Le message est d'abord décodé. Le code nous précise la partie du réseau présenté au troisième chapitre qui doit être simulée, pour élaborer les actions à entreprendre par le processeur de synchronisation. Sur le réseau de la figure IV_3, ce sont les places "analyse" et "recherche des processeurs à réveiller" qui correspondent à la mise en oeuvre de l'algorithme présenté au chapitre précédent.

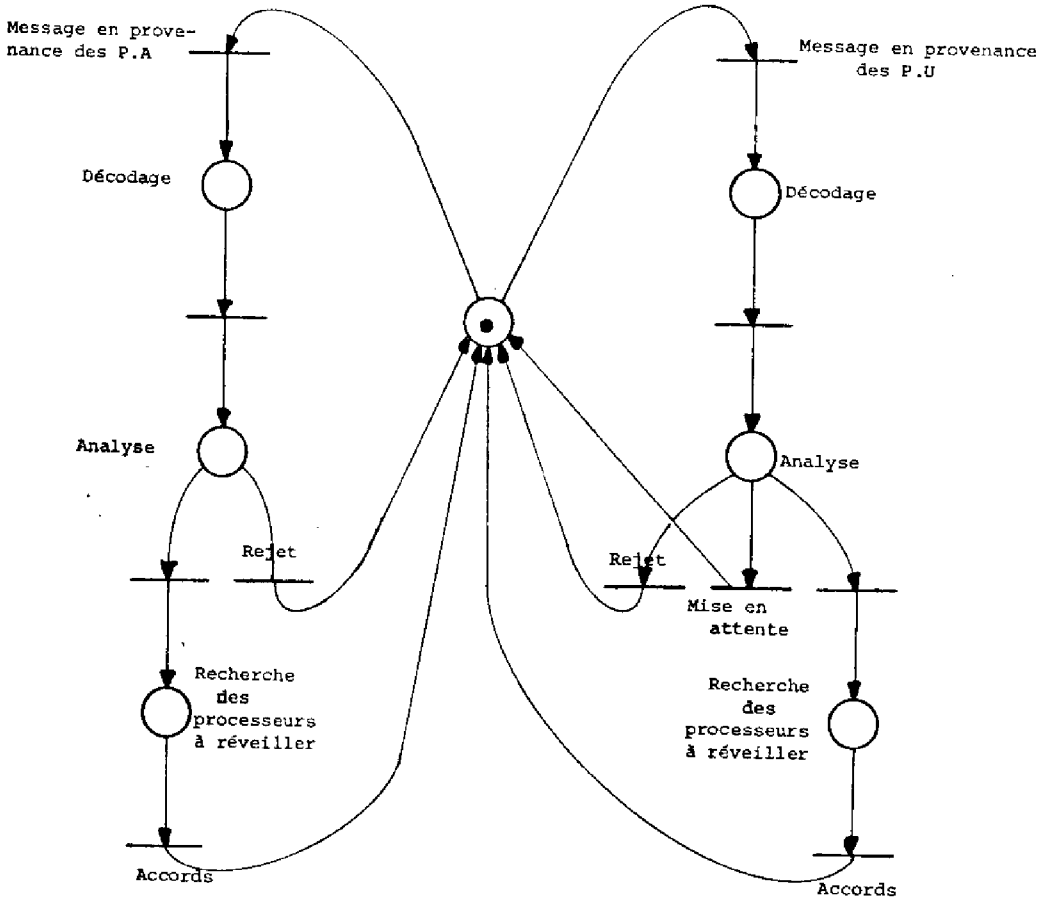


Figure IV_3.

4. *IMPLEMENTATION DE L'ALGORITHME DE SYNCHRONISATION*

4.1. *Principe*

Après décodage le processeur de synchronisation analyse le message reçu. La solution retenue pour cela, compromis entre conduite compilée et tabulaire, se compose de :

- une représentation tabulaire du réseau de Petri et d'un simulateur, (paragraphe $IV_{4.4.}$)

- un ensemble de sous programmes compilés (synchroniseurs) appelant le simulateur pour des séquences de tirs de transitions.

Le choix de la conduite compilée, malgré tous les inconvénients (risque de recompilation....) qu'elle présente, est à notre avis la méthode qui permet de mieux focaliser la recherche des transitions sensibilisées dans un réseau de Petri à jetons individualisés.

En effet, lorsque le processeur reçoit un message (demande ou fin de transaction), il ne doit tirer qu'un nombre très limité de transitions (souvent une seule).

Une implémentation compilée permet de considérer directement les transitions susceptibles d'être tirées en fonction de la nature du message. Ainsi le déroulement de la séquence de tir est relativement figé : par exemple après réception d'une demande d'écriture, il n'est possible que de tirer la transition de préautorisation d'écriture (PAE) puis on a le choix entre l'une des deux autorisations d'écriture (AE_i).

Dans une réalisation à conduite tabulaire, toutes les transitions sont examinées (au plus dans les versions optimisées [SAM 80], toutes celles qui ont une place précédente marquée) d'où un temps d'exécution bien plus long que pour notre solution compilée.

Toutefois lorsqu'une transition susceptible d'être sensibilisée est trouvée par la conduite compilée, la vérification de ces conditions de tir, et son tir éventuel, sont réalisés par simulation du réseau. Pour cela les données correspondant au réseau sont stockées sous formes de tables. Ceci permet de diminuer la place mémoire occupée par les programmes et de mieux structurer le logiciel.

Donc à chaque message reçu par le processeur de synchronisation correspond un programme permettant d'enchaîner correctement les tirs de transitions. En fait ces programmes sont de deux types :

I) Programmes correspondant à une demande de transaction.

Les transitions "demande" étant toujours sensibilisées, un jeton est mis directement dans la place correspondant à la file d'attente associée. Ensuite le simulateur est appelé pour les transitions "autorisations" (pré-autorisation et autorisation dans le cas de l'écriture).

Aucun processeur en attente ne pouvant être débloqué par la mise en attente ou l'autorisation d'un autre, il est inutile de rechercher si d'autres transitions sont tirables.

Les programmes pour ce type de messages reçus par le processeur de synchronisation se terminent par l'envoi d'un message de mise en attente ou d'accord vers le processeur utilisateur.

II) Programme correspondant à une fin de transaction.

Nous avons vu au chapitre III que pour une contrainte de sécurité, le nom des processeurs utilisateurs effectuant une transaction était filtré. Ce filtrage est réalisé par le processeur de synchronisation chaque fois qu'un message "fin" de transaction est reçu pour être certain que le processeur ayant émis ce message a bien reçu auparavant un message d'accord du processeur de synchronisation.

Le simulateur est donc appelé pour les transitions fin associées à chaque bloc. Si aucune n'est sensibilisée un message de "rejet" est envoyé pour signaler l'erreur.

Si une transition a été tirée, il faut à ce moment-là examiner si une transaction en attente peut être libérée car seule une "fin de transaction" change les variables de synchronisation dans un sens favorable.

Ainsi à chaque "fin de transaction", les diverses files d'attente sont explorées et si l'une d'elles n'est pas vide, le simulateur est appelé (en fait une partie du logiciel écrit pour le traitement des messages est réutilisée à ce niveau). Si une transition est tirée (une transaction est libérée), l'exploration des files d'attente continue jusqu'à ce que plus aucune transaction ne soit libérable.

Nous allons maintenant présenter chacun des programmes plus en détail.

4.2. *Détail des algorithmes*

4.2.1. *Début d'écriture* :

Une particularité de l'algorithme correspondant aux demandes d'écriture est qu'il se décompose en deux niveaux. Ceci est dû au fait, qu'il faut tirer deux transitions (pré-autorisation (PAE), et autorisation d'écrire sur un ensemble mémoire (AE_i)) pour que la demande soit autorisée.

Un autre problème a pour cause le fait que le réseau représentant la synchronisation, ne permet pas de différencier au cours d'un cycle d'écriture, la phase d'écriture des phases mises à jour. Or cette information est nécessaire au processeur de synchronisation pour l'envoi des messages réponses.

Une solution simple à ce problème(figure IV_4) consiste à utiliser une variable auxiliaire "a" mise à la valeur "1" lors du tir de la transition pré-autorisation (PAE) et mise à la valeur "0" au tir de la transition fin d'écriture (FE_i). Quand une transition AE_i est tirée, il suffit alors de tester la valeur de "a" pour connaître la phase du cycle.

Il faut souligner que l'introduction de cette variable "a" n'agit pas sur les conditions de tir des transactions, et donc l'analyse faite au chapitre précédent n'est pas remise en cause.

L'enchaînement des tirs des transitions est donné par le réseau de la figure IV_4, l'organigramme du programme associé par la figure IV_5. Dans cet organigramme, si le tir d'une transition autorisation d'écrire sur un ensemble mémoire n'est pas possible, on examine la possibilité de panne de cette mémoire par le tir de EP_i avant de regarder si une autre transition AE_i est sensibilisée. Au bout de m (nombre d'ensembles mémoires) impossibilités de tir sur les transitions AE_i cette procédure est interrompue.

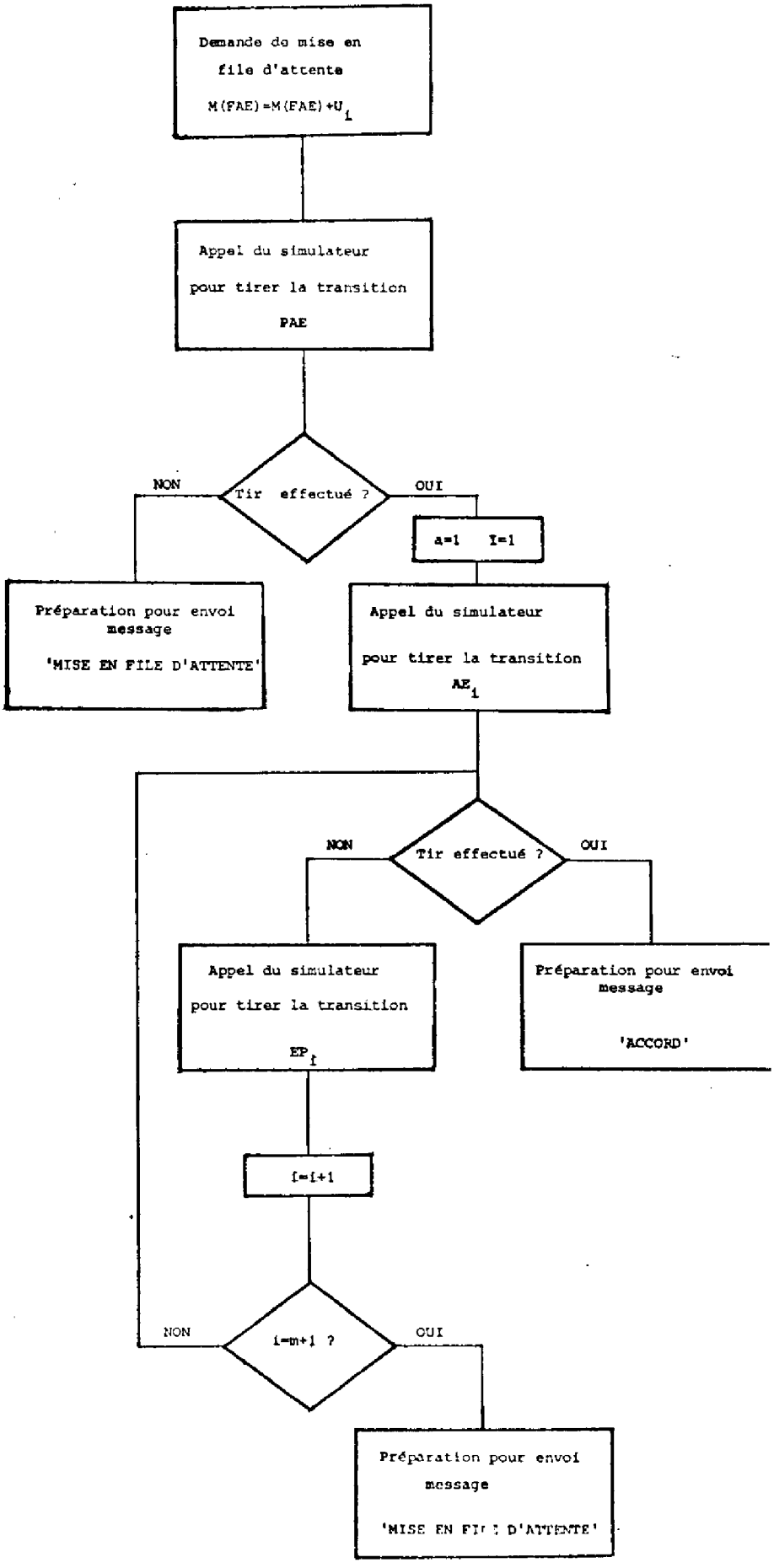

Demande de mise en file d'attente
M(FAE)=M(FAE)+U_i
Appel du simulateur pour tirer la transition PAE
NON
Tir effectué ?
OUI
a=1 I=1
Préparation pour envoi message 'MISE EN FILE D'ATTENTE'
Appel du simulateur pour tirer la transition AE_i
NON
Tir effectué ?
OUI
Appel du simulateur pour tirer la transition EP_I
Préparation pour envoi message 'ACCORD'
I=I+1
NON
i=m+1 ?
OUI
Préparation pour envoi message 'MISE EN FILE D'ATTENTE'

Figure IV_5.

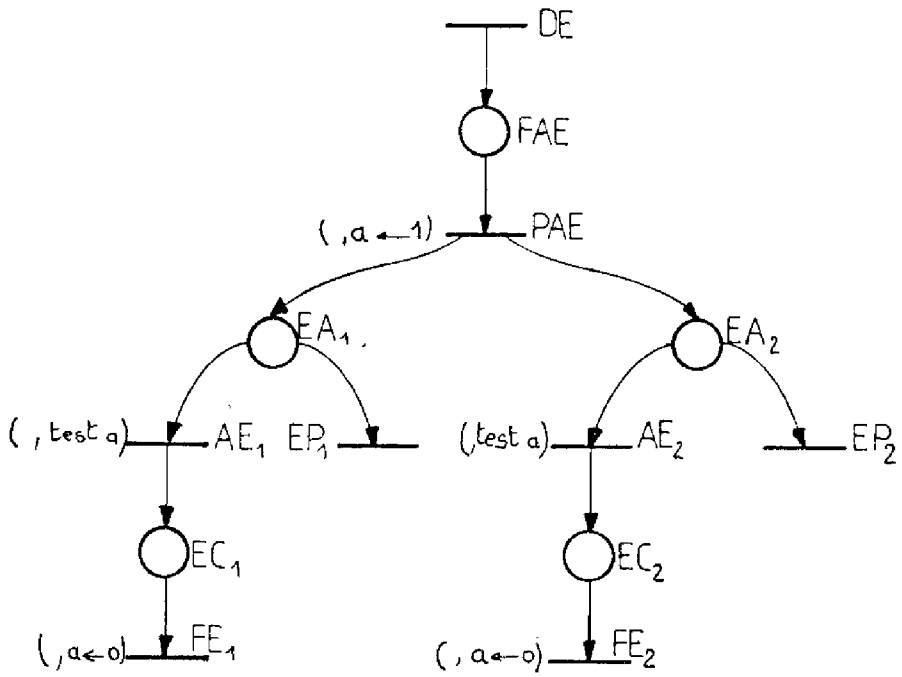


Figure IV_4.

4.2.2. Demande de lecture

L'un des intérêts de cette structure en fonctionnement normal, est de pouvoir répartir la charge des transactions lectures sur les différentes mémoires. Spécifier cette répartition par réseau de Petri implique un graphe trop complexe, pour les informations qu'il apporte. Aussi dans cet état, le réseau de Petri laisse le libre choix de tir entre les diverses transitions d'autorisations de lecture (AL_i).

Implémenter une stratégie pour répartir la charge, c'est donc introduire un algorithme effectuant le choix de la transition à tirer. Evidemment, il faut que cet algorithme n'affecte pas les propriétés du réseau, et de la synchronisation étudiée au chapitre précédent.

Ce choix est réalisé par un compteur (modulo m) incrémenté chaque fois qu'une lecture est autorisée sur une mémoire. La valeur du compteur donnant l'indice de la transition AL_i ayant la préférence. Si la transition n'est pas tirable (mémoire en panne ou dans une phase d'actualisation, ou saturée), le compteur est incrémenté afin de tester la mémoire suivante. Bien sûr en cas de m insuccès , (aucune transition n'est tirable) cette incrémentation est arrêtée.

L'algorithme correspondant à une demande de lecture est alors celui décrit par l'organigramme de la figure IV_6.

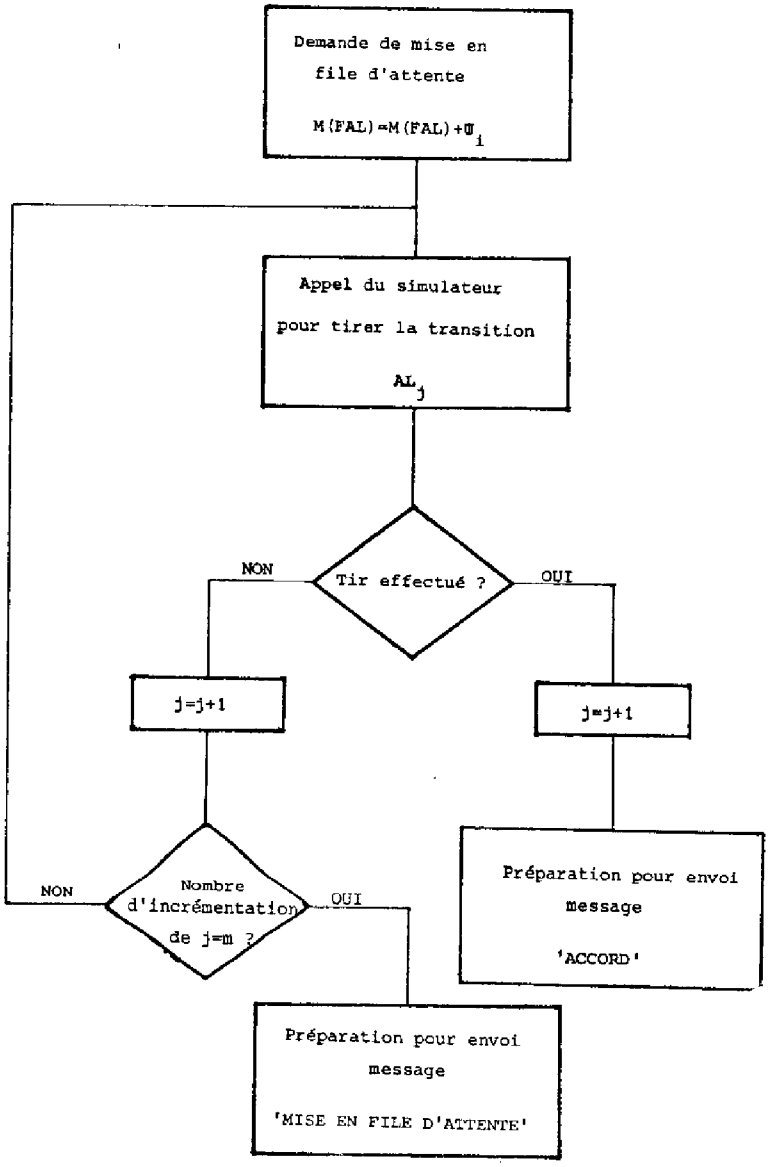


Figure IV_6.

4.2.3. Début de restauration :

L'enchaînement des tirs de transitions pour les deux types de restauration est identique et ne présente pas d'originalité comme les deux précédents. Ainsi, après avoir soit déclaré une mémoire hors service temporairement (restauration partielle) ou soit réparé un ensemble mémoire, le simulateur est appelé pour savoir si cette restauration est faisable ou si elle est mise en file d'attente. Un point différencie cependant ces deux algorithmes : il est relatif aux restaurations totales où en cas d'impossibilité de tir, nous testons que la mémoire était effectivement invalide avant d'envoyer un message de mise en file d'attente (ceci est fait pour pallier une demande de restauration erronée).

4.2.4. Fin de transaction

Nous avons vu (paragraphe 4.1.) le principe des algorithmes associés aux fins de transaction. Donc après avoir tiré la transition correspondant à ce message "fin de transaction", il faut explorer toutes les files d'attente pour savoir s'il y a des transactions succeptibles d'être débloquées (et donc des processeurs réveillés).

L'ordre suivant lequel cette scrutation est effectuée n'est pas indifférent, et le fait qu'une file d'attente soit explorée avant une autre, va automatiquement affecter une certaine priorité aux transactions. De plus par la connaissance d'une transaction en attente, la scrutation des autres files d'attente peut être généralement diminuée et même arrêtée. Ainsi, le temps de réponse du processeur de synchronisation est considérablement diminué.

Un facteur important pour organiser cette exploration des files d'attente est de tenir compte des spécificités du réseau de Petri, qui sont rappelées au tableau de la figure IV_7. Ce tableau est construit à partir des contraintes du problème (vérifiées par les invariants présentés à l'annexe II), et on peut y voir par exemple que si une écriture pré-autorisée, ou une mise à jour est en attente ($\exists\ |M(EA_i)| = 1$), les transactions restaurations partielle et totale ne peuvent se dérouler, ni une nouvelle pré-autorisation d'écriture. Seules dans ce cas des transactions de lectures peuvent avoir lieu.

	Ecriture pré-autorisée ou mise à jour	Restauration totale	Ecriture	Restauration partielle	Lecture
Ecriture pré-autorisée ou mise à jour en attente	I	I	I	I	N
Restauration totale en attente	N	I	N	N	N
Ecriture en attente	N	N	I	N	N
Restauration partielle en attente	I	I	I	I	N
Lecture en attente	N	N	N	N	N

Figure IV_7.

Ce tableau est à interpréter de la manière suivante :

Si la file d'attente indiquée horizontalement est non vide, et après examen de l'autorisation de la transition associée, I signifie qu'il est inutile d'examiner le cas des files d'attente indiquées verticalement, car en aucune manière la transaction correspondante ne pourra être autorisée. Par contre N signifie que cet examen est nécessaire.

A la vue de ce tableau, il apparaît intéressant, pour focaliser un peu plus la recherche des transitions sensibilisées, d'examiner en premier les files d'attente qui ont le plus grand nombre de "I", car en cas de réponse positive la scrutation s'arrête immédiatement.

Nous avons retenu l'ordre d'exploration des files d'attente suivant :

- Ecriture pré-autorisée ou mise à jour (EA_i)
- Restauration partielle
- Restauration totale
- Ecriture (FAE)
- Lecture.

Nous donnons la priorité à l'examen des files d'attente de "l'écriture pré-autorisée ou de mise à jour", par rapport à celle des restaurations partielles, car la première a une plus grande probabilité d'être non vide que la seconde. Par contre pour éviter tout phénomène de famine, la priorité a été donnée à l'examen de la file d'attente des restaurations totales. Le phénomène inverse a très peu de"chance"de se produire si l'on considère la fréquence de ces transactions de restauration. De plus cette transaction est pilotable à des périodes de faible charge du système.

L'algorithme donnant la scrutation des files d'attente qui doit être effectuée à la "fin d'une transaction" est décrit par l'organigramme de la figure IV_8.

Nous pouvons remarquer le parallèle qui peut être fait entre l'exploration des files d'attente que nous proposons, et celle faite dans des moniteurs spécialisés [HOA 74 - KES 77] . En effet l'examen, une à une, de toutes les files d'attente à la fin d'une procédure pour le moniteur décrit par [KES 77] est similaire dans son principe à la scrutation systématique de toutes les transitions ayant une file d'attente comme place d'entrée. En revanche celle exposée dans l'organigramme de la figure IV_8 correspond à l'examen personalisé qu'implique l'utilisation d'un"moniteur Hoare". Il est bien normal que nous en retrouvions, pour ces deux types de simulation, les mêmes inconvénients, que pour ces deux types de moniteurs.

4.3. *Optimisation de la scrutation des files d'attente dans le cas de deux ensembles mémoires.*

Pour le cas de deux mémoires, la scrutation des files d'attente lors du réveil des processeurs peut être optimisée si nous tenons compte de la nature du message "fin de transaction" reçu par le processeur de synchronisation.

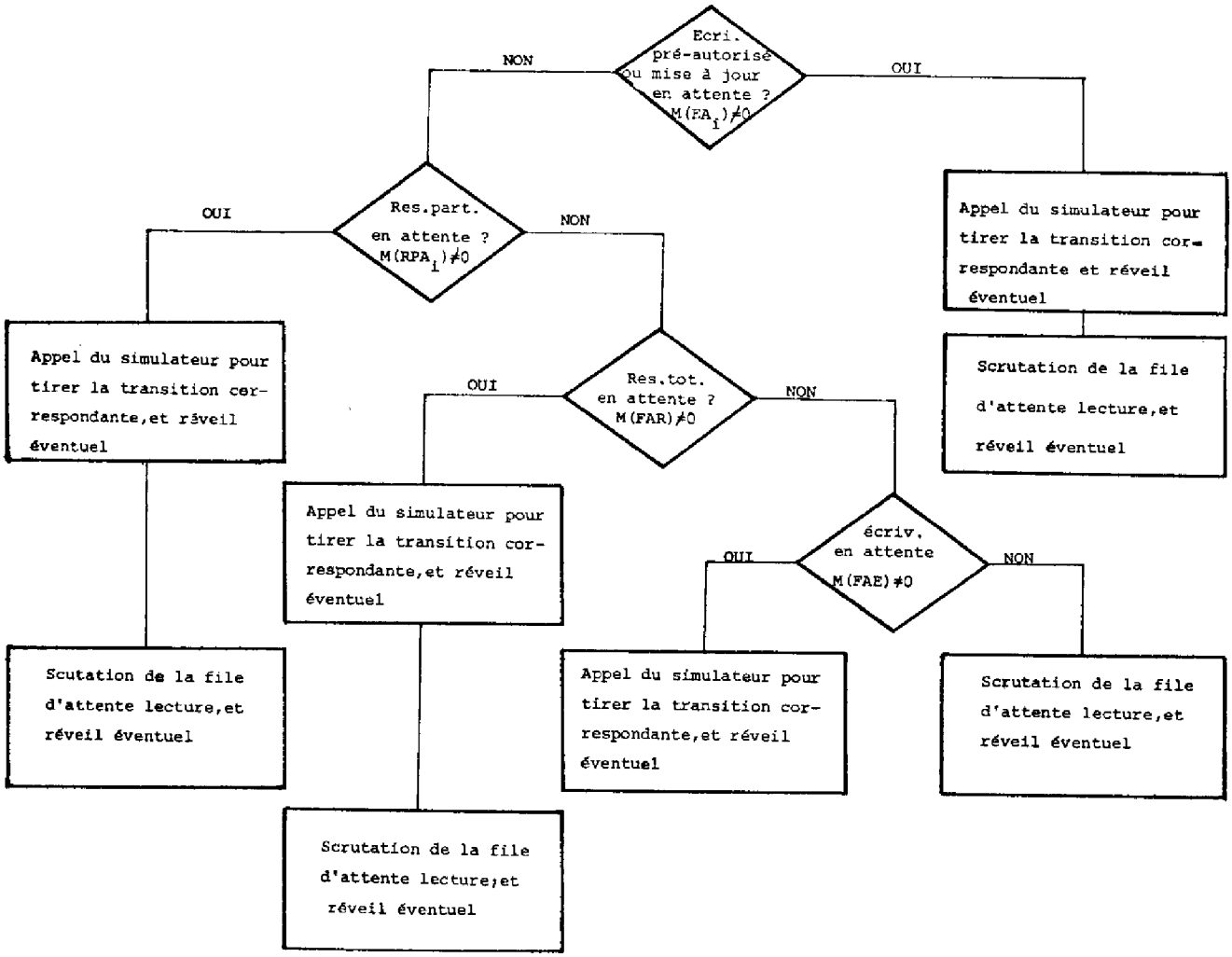


Figure IV_8

Par exemple, considérons que le processeur reçoive comme message "fin d'actualisation", il est inutile alors dans l'exploration des files d'attente d'examiner :

- Celle correspondant aux mises à jour (puisqu'il n'y en a plus)
- Celles correspondant aux restaurations, car par le message reçu les deux ensembles mémoires sont valides.

Ainsi seule la scrutation des files d'attente associées aux écritures et aux lectures est nécessaire dans ce cas.

Cet affinement dans la recherche des transitions tirables est personnalisé d'une part à la nature des messages reçus par le processeur, et d'autre part au cas particulier de deux mémoires.

4.3.1. Fin d'écriture :

Quand le processeur reçoit ce message, cela implique que la phase initiale du cycle d'écriture s'est bien déroulée, et donc que la mémoire n'a pas été déclarée en panne temporairement.

Ainsi dans la scrutation des files d'attente, celle associée aux restaurations partielles est sautée. L'algorithme devient celui représenté par l'organigramme de la figure IV_9.

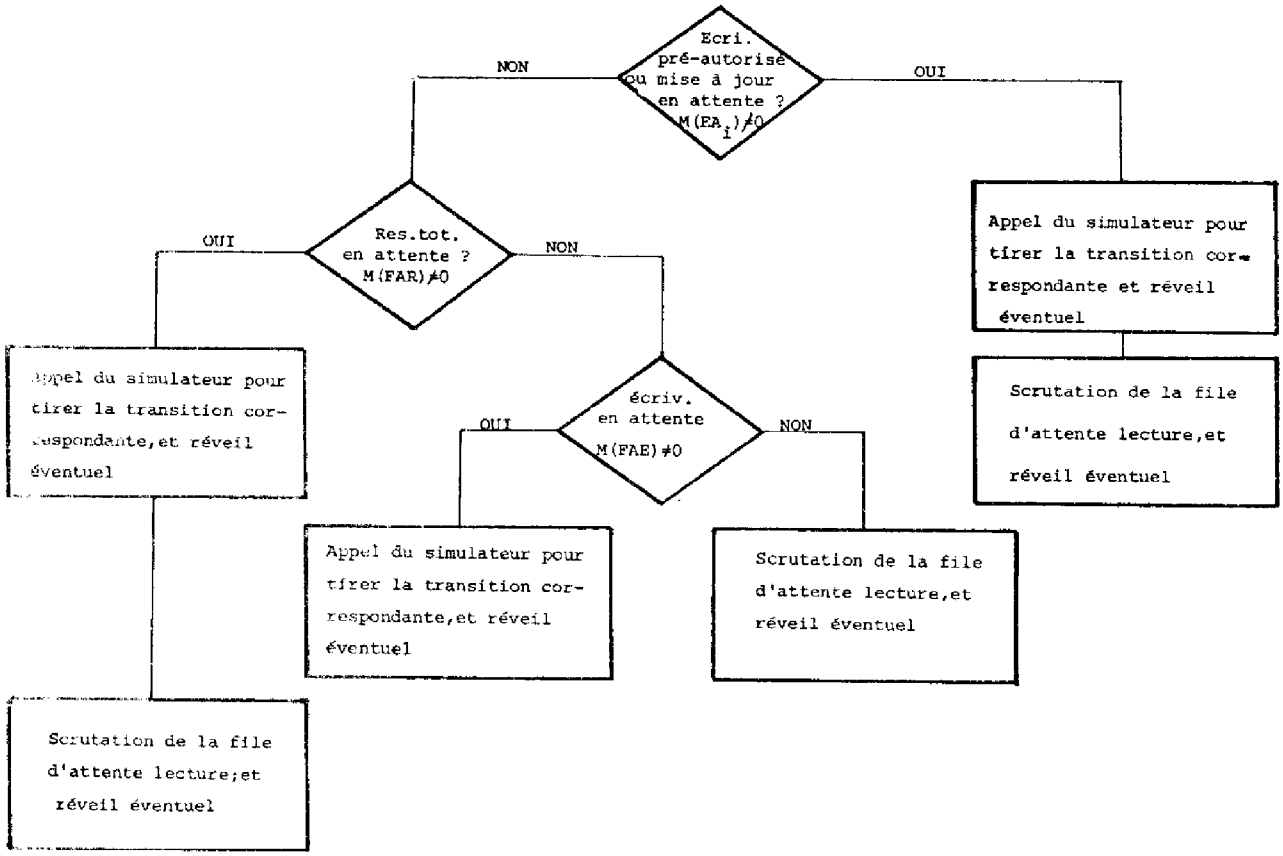


Figure IV_9.

4.3.2. *Fin de lecture* :

La réception de ce message ne remet pas en cause le fait qu'un cycle d'écriture est en cours et il n'a aucune influence sur sa libération.

Ainsi l'exploration de la file d'attente FAE pour ce type de message n'est plus nécessaire, et l'organigramme devient celui de la figure IV_{10}.

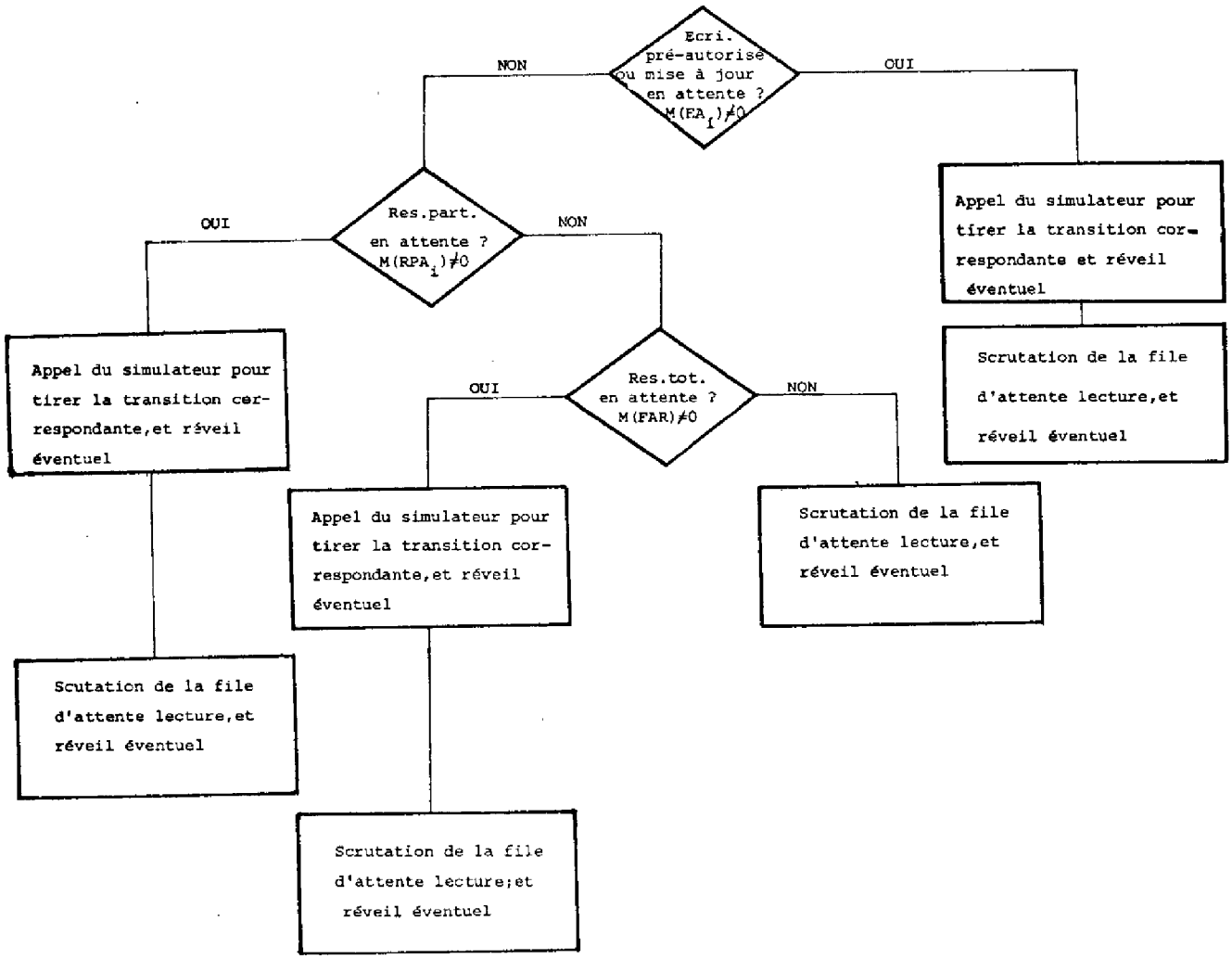


Figure IV_{10}.

4.3.3. Fin d'actualisation et des restaurations :

Pour ces différents messages, la scrutation des files d'attente est identique, et elle est décrite dans l'organigramme de la figure IV_{11}.

Nous avons déjà présenté dans l'introduction de paragraphe 3.3., les causes de la réduction de cette scrutation quand le processeur de synchronisation avait reçu un message de "fin d'actualisation".

En ce qui concerne la scrutation après réception d'un message "fin de restauration", un raisonnement analogue peut être fait pour montrer l'inutilité de certaines explorations.

Aussi une "fin de restauration totale" signifie qu'un cycle d'écriture n'avait pas lieu (exclusion entre cycle d'écriture et restauration), et donc les files d'attente des écritures pré-autorisées, ou mises à jour (places EA_i), et des restaurations partielles (places RPC_U) sont vides.

Pour le cas d'une"fin de restauration partielle" , il est immédiat qu'il ne peut exister de mises à jour. Il en est de même pour les restaurations totales car nous sommes dans le cas où nous n'avons qu'une seule mémoire en panne.

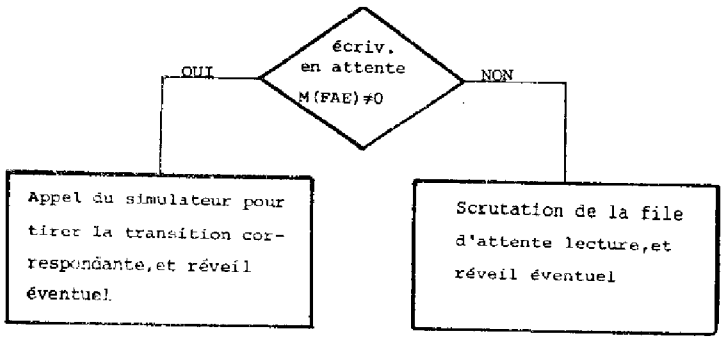


Figure IV_{11}.

4.4. *Présentation du simulateur*

Ce simulateur est une adaptation de celui présenté par [AYA 79] pour tenir compte des jetons individualisés.

4.4.1. *Description tabulaire :*

Le réseau de Petri à jetons individualisés est traduit sous forme de tables qui représentent les matrices d'entrée et de sortie. Un emplacement mémoire est réservé pour traduire le vecteur marquage des places.

De plus le réseau global a été partitionné en sous-réseaux, chacun d'eux étant associé à un des divers messages que peut recevoir le processeur de synchronisation. Ceci dans le but de faciliter l'enchaînement des indices des transitions dans la partie compilée, en cas d'augmentation des modules. Par ce moyen, une modification du nombre de modules nécessitera simplement une nouvelle traduction des réseaux en tables, et un réajustement de la variable indiquant le nombre de modules.

Enfin dans ce simulateur, il nous a fallu tenir compte de l'introduction des jetons individualisés pour le marquage des places. Aussi dans ce réseau, deux catégories de places sont à différencier : les places contenant seulement des jetons banalisés, dont l'implémentation classique du marquage est réalisée par un compteur ; et les places contenant des jetons individualisés. Pour ces dernières, le marquage est implémenté par un vecteur de binons (ou "bits"), où chaque binon correspond à un identificateur. L'inconvénient de cette réalisation est son manque de généralité pour représenter plusieurs jetons de même identificateur. Toutefois, lorsqu'elle est suffisante, elle permet de restreindre le place "mémoire vive" nécessaire.

4.4.2. *Simulation :*

Rappelons que le simulateur est appelé par la partie compilée. Celle-ci lui fournit trois informations :

- le réseau concerné,

- la transition à tenter de tirer sur ce réseau,

- et l'initialisation des variables étiquetant les arcs.

A partir de ces informations, le simulateur examine sur les tables correspondant au réseau, si le marquage des places d'entrée de la transition est suffisant pour respecter la condition de tir (paragraphe $II_{3.3.2.}$). Si cette condition est remplie, la transition est tirée comme nous l'avons défini (paragraphe II 3.3.3.). Si la condition n'est pas respectée, la source de l'impossibilité de tir (marquage des places à jetons banalisés insuffisant, ou marquage des places à jetons individualisés) est détectée.

L'organisation de ce simulateur est décrite dans l'organigramme de la figure IV_{12}.

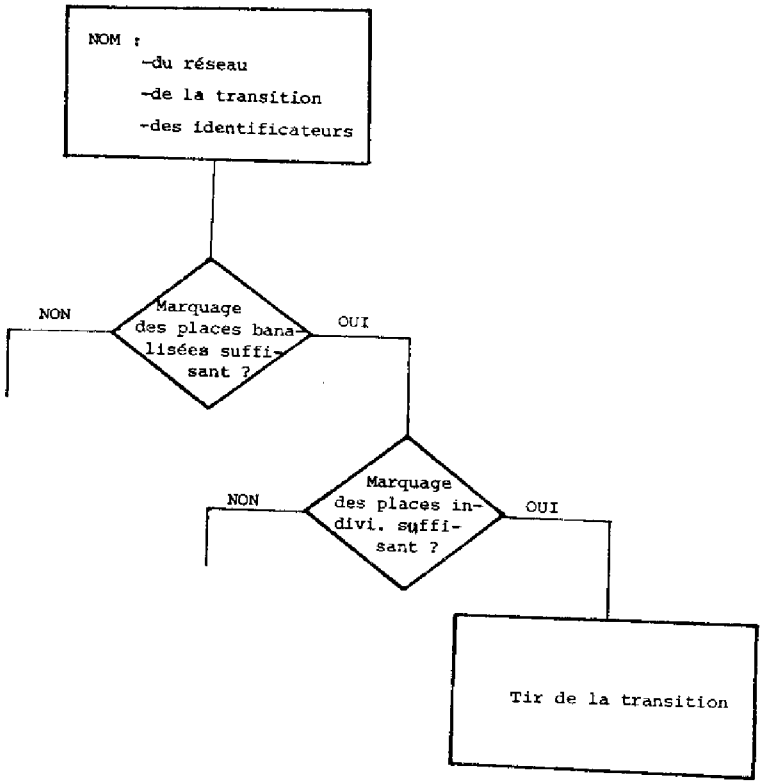


Figure IV_{12}.

5. *TRAITEMENT DES ALARMES*

Afin de détecter les éléments hors service, ou ayant un fonctionnement erroné, plusieurs tests ont été prévus. A l'heure actuelle, les principaux sont concentrés au niveau des processeurs arbitres. Ceux-ci localisent les éléments fautifs et la nature de leur panne. A la détection de l'un d'eux une alarme est émise en direction du processeur de synchronisation. Suivant le type de l'élément et la nature de sa panne, plusieurs sortes d'alarmes ont été définies, et évidemment le processeur réagit en fonction de leur sens.

5.1. *Tentative de viol mémoire*

Un utilisateur, ayant vu sa demande de transaction accordée par le processeur de synchronisation, tente de faire des accès en écriture sur la mémoire. Ces accès filtrés par le processeur arbitre sont contraires aux droits signalés pour cet utilisateur par le processeur de synchronisation, et sont suffisament nombreux pour que le processeur arbitre en réfère au processeur de synchronisation. Celui-ci exclut alors le processeur fautif de la liste des lecteurs et le lui signale par un message de rejet.

5.2. *Dépassement de la capacité mémoire tampon*

Chaque processeur arbitre possède une mémoire tampon où sont stockées les modifications apportées aux données contenues dans sa mémoire lors d'une phase d'écriture. Cette mémoire tampon (buffer), est utilisée lors de phases de mises à jour des autres mémoires pour répercuter les modifications.

Cette mémoire donc, doit être de dimension suffisante pour satisfaire le processus faisant le plus grand nombre de modifications.

Si pour une raison quelconque, un processeur dépasse cette capacité mémoire, l'enregistrement des transformations de données ne peut plus se faire à cause de la perte qu'entraînerait leur concaténation, et une alerte est émise vers le processeur de synchronisation. Celui-ci déclare cet ensemble mémoire en panne temporairement, et rejette le processeur écrivain.

5.3. *Temps d'accès incorrects*

Un autre type de panne grave détecté par les processeurs est celui qui correspond au cas où un processeur a vu sa demande d'accès mémoire accordée, et ne l'utilise pas, soit parce qu'il vient de tomber en panne, soit parce que son bus le liant à la mémoire est hors service.

Cette situation est dangeureuse pour la synchronisation totale du sytème car le processeur utilisateur peut ne pas signaler de fin de transaction. Aussi chaque PA surveille-t-il l'utilisation de sa mémoire. Cette surveillance sert aussi dans le sens opposé à détecter une utilisation abusive par un utilisateur, en chronométrant son temps d'accès.

Donc en présence d'un de ces mauvais fonctionnements une alarme est transmise au processeur de synchronisation qui rejettera le processeur utilisateur, et déclarera la mémoire hors service temporairement si le processeur était un écrivain, car il y a possibilité d'incohérence de données.

5.4. *Divers*

Lors de ses propres procédures de test, le processeur arbitre peut détecter un de ses éléments en panne, et donc qu'il est inutilisable pour le système.

Il signale ce fait pour que le processeur de synchronisation l'isole du système et rejette les utilisateurs de ce bloc.

6. *CONCLUSION*

Nous venons de présenter une possibilité d'implémentation logicielle des réseaux de Petri à jetons individualisés. La solution retenue repose sur le principe des automates programmables par réseau de Petri. Pour des contraintes de temps, le principe de l'automate choisi est un compromis entre les deux types possibles de conduite des évolutions avec toutefois, une grande part pour la conduite compilée.

Nous avons construit notre implémentation de telle manière qu'elle soit la plus générale possible, avec en particulier, la possibilité de modifier le nombre d'ensemble mémoire sans pour autant remettre en cause le principe de la conduite compilée. Cependant, le défaut de cette méthode est qu'elle est liée au problème particulier de synchronisation traité. Quand l'algorithme de synchronisation est modifié, l'implémentation doit être entièrement revue.

Une possibilité est alors de choisir une implémentation par simulation du type de celle proposée par [SAM 80] . Bien sûr comme toutes solutions générales, les performances, telles que le temps de réponse par exemple, sont moins bonnes.

C O N C L U S I O N

- - - -

Dans ce mémoire, nous avons étudié les mécanismes de synchronisation pour un système de commande de processus industriels sûr de fonctionnement.

Ceci nous a amené à définir une structure offrant des possibilités de fonctionnement en mode dégradé. En particulier, pour tenir compte des pannes de mémoire contenant les données, nous avons été amené à les dupliquer. Cette duplication entraîne cependant de nouvelles contraintes au niveau des mécanismes de synchronisation. Ceux-ci devenant relativement complexes, il devient alors nécessaire de les spécifier et de les valider rigoureusement.

Les réseaux de Petri sont à notre avis un très bon outil pour représenter, valider et mettre en oeuvre des systèmes de complexité non négligeable . Toutefois, comme tout outil général, ils sont parfois mal adaptés pour représenter certains phénomènes. C'est le cas quand on considère l'aspect sécurité d'un système, où le réseau représentant sa synchronisation peut devenir rapidement lourd. Nous avons alors proposé d'utiliser les réseaux de Petri à jetons individualisés, qui permettent, sans compliquer outre mesure la preuve, et sans perte de rigueur, de spécifier de façon condensée, les problèmes liés à la sécurité des systèmes.

Nous avons donc utilisé ces réseaux pour spécifier et valider l'algorithme de maintien de la cohérence de données dupliquées pour un système multimicroprocesseur travaillant sur des mémoires partagées et dupliquées. Au cours de sa spécification, nous nous sommes aperçus de la limitation des réseaux, en particulier de l'impossibilité d'aborder le problème dans son ensemble. Notre approche a été alors d'introduire les contraintes progressivement et de les valider au fur et à mesure. Afin de ne pas alourdir inutilement le réseau global, nous n'avons pas représenté certains points secondaires tels que la répartition des charges des transactions de lecture sur les différentes mémoires. En effet, ces mécanismes sont relativement indépendants de la synchronisation globale et nous pouvons nous contenter de les décrire en utilisant des variables auxiliaires testées et modifiées lors du tir de la transition. Ainsi, nous évitons de tout spécifier par un réseau de Petri volumineux.

L'inconvénient de cette approche progressive est qu'à chaque pas, la preuve de l'étape précédente peut être remise en cause.

Ceci est en particulier vrai lorsque les fusions entre réseaux portent aussi bien sur les transitions que sur les places car les invariants linéaires de places ne sont plus nécessairement les mêmes. Un autre problème, est que notre réseau global n'est pas totalement réductible. En conséquence la preuve de l'absence de blocage se fait à partir de l'énumération des marquages et donc avec des variables comme poids sur les arcs déterminées. Ce problème se retrouve de façon plus grave encore lorsque l'on généralise l'algorithme au cas de m mémoires puisque, malgré la modularité des réseaux, la preuve de l'absence de blocage ne peut se faire que pour très peu de mémoires. Il nous faut cependant souligner que les invariants sont, eux, valables quelle que soit la valeur des variables, et quel que soit le nombre de mémoires, mais ils ne sont pas suffisants pour prouver l'absence de blocage.

En ce qui concerne l'implémentation, nous nous sommes tournés vers une simulation en grande partie compilée pour éviter des temps de réponse prohibitifs. Les inconvénients sont alors d'une part la réalisation qui est à reprendre presque entièrement si l'algorithme de synchronisation est modifié. D'autre part, la probabilité de l'introduction d'erreurs lors de l'implémentation n'est plus nulle, et la simple validation des spécifications devient insuffisante.

En conclusion, l'utilisation pratique des réseaux de Petri (avec ou sans jetons individualisés), pour spécifier, valider et implémenter des algorithmes de synchronisation dans des systèmes de commande de complexité moyenne nécessite encore quelques résultats.

Le premier est celui de la preuve de l'absence de blocage pour un réseau de Petri non totalement réductible. Sa solution peut provenir de l'utilisation de règles de réduction plus puissantes, mais elle peut également découler directement de la démarche que nous avons suivie, où les contraintes sont introduites une à une. Il nous faut ajouter toutefois une condition à cette démarche, celle d'utiliser des règles permettant d'obtenir un réseau sans blocage par la fusion de réseau sans blocage. Certains travaux en cours semblent sur ce sujet très prometteurs.

En ce qui concerne le problème spécifique de la construction d'un réseau par la fusion de m modules identiques, l'approche développée par [JEN 79]

préconisant l'utilisation de réseaux de Petri colorés semblent être intéressante. Mais, signalons toutefois que la preuve et la recherche des invariants pour ce modèle n'est pas automatique.

Enfin, la réalisation finale, après validation, a souligné combien il restait encore à faire pour pouvoir produire sans erreur un code efficace à partir d'un réseau de Petri. C'est un problème relativement complexe pour être abordé dans sa généralité et il faut pour le moment se restreindre au développement d'outils et de méthodes adaptés à des cas particuliers.

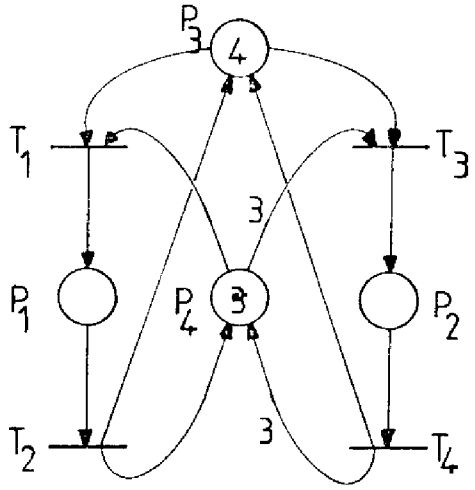
A N N E X E S

- - - - - -

ANNEXE 1

ANALYSE DU RESEAU DE PETRI SPECIFIANT LE PROBLEME DES LECTEURS-ECRIVAINS

RESEAU DE PETRI A ANALYSER



Description du réseau pour OGIVE par le tableau des suivants [CHE 79]

```
00010 ************************************************************
00020                      LECTEURS-ECRIVAINS
00030 ************************************************************
00040   4   4   2           / RESPEC. Nbre de places, de transitions, de suivants /
00050   1   2   3   4       / Numérotage des places /
00060   5   6   7   8       / Numérotage des transitions /
00070   6   0
00080   8   0
00090   5   7
00100   5   7
00110   1   0               / Suivants du sommet i /
00120   3   4
00130   2   0
00140   3   4
00150   1   1
00160   1   1
00170   1   1
00180   1   3
00190   1   1               / Poids des arcs /
00200   1   1
00210   1   1
00220   1   3
00230   0   0   4   3       / Marquage initial /
00240
00250
00260
```

```
00270
00280          ANALYSE DIRECTE DU RESEAU
00290          **************************
00300
00310
00320
00330     NP= 4              NT= 4
00340
00350
00360
00370
00380
00390          BORNE: T
00400                      SAUF: F
00410                                  PROPRE: T
00420                                              VIVANT: T
00430
00440
00450
00460
00470
00480
00490               LE NOMBRE DES MARQUAGES EST :  5
00500     1         0   0   4   3
00510     2         1   0   3   2
00520     3         0   1   3   0
00530     4         2   0   2   1
00540     5         3   0   1   0
00550
00560
00570
00580     LA TAILLE DU GRAPHE DES MARQUAGES EST: 4
00590
00600
00610          1              1   2      3   3
00620          2              1   4      2   1
00630          3              4   1      0   0
00640          4              1   5      2   2
00650          5              2   4      0   0
00660
00670
00680
00690
```

```
00700
00710
00720                    ANALYSE PAR REDUCTION CONSERVANT PROPRE
00730                    ***************************************
00740
00750
00760
00770          RESEAU DE DEPART NON REDUIT
00780
00790          NP= 4
00800                     NT= 4
00810                                NMIN=1
00820
00830
00840
00850                  1 EST PLACE SUBSTITUABLE ********
00860                  2 EST PLACE SUBSTITUABLE ********
00870            1EST T IDENTITE ET SIMPLIFIABLE PAR   2****
00880          LA PLACE  3 EST PLACE IMPLICITE *********
00890          LA PLACE  4 EST PLACE IMPLICITE *********
00900
00910
00920
00930
00940
00950
00960     LE RESEAU A ETE COMPLETEMENT REDUIT
00970          CE RESEAU REDUIT EST DONC BORNE ET VIVANT
00980          LE RESEAU INITIAL EST DONC EGALEMENT PROPRE
00990
01000
01010
01020
01030
01040
01050
01060          COUVERTURE D'INVARIANTS
01070             DE PLACES
01080
01090
01100
01110     1   1   1   0
01120     1   3   0   1
01130
01140
01150
01160
01170
```

A N N E X E 2

ANNEXE 2.1. GENERALISATION DE LA RECONFIGURATION MEMOIRE APRES UNE MISE HORS SERVICE TOTALE

Dans le chapitre III nous avons considéré que le système avait toujours une mémoire valide sur les deux. Avec cette hypothèse, la spécification par réseau de Petri suffisait pour l'algorithme décrit. Si maintenant nous voulons le généraliser dans le cas de m mémoires, il faut nous assurer que la mémoire servant de référence pour la restauration de la mémoire en panne est bien valide

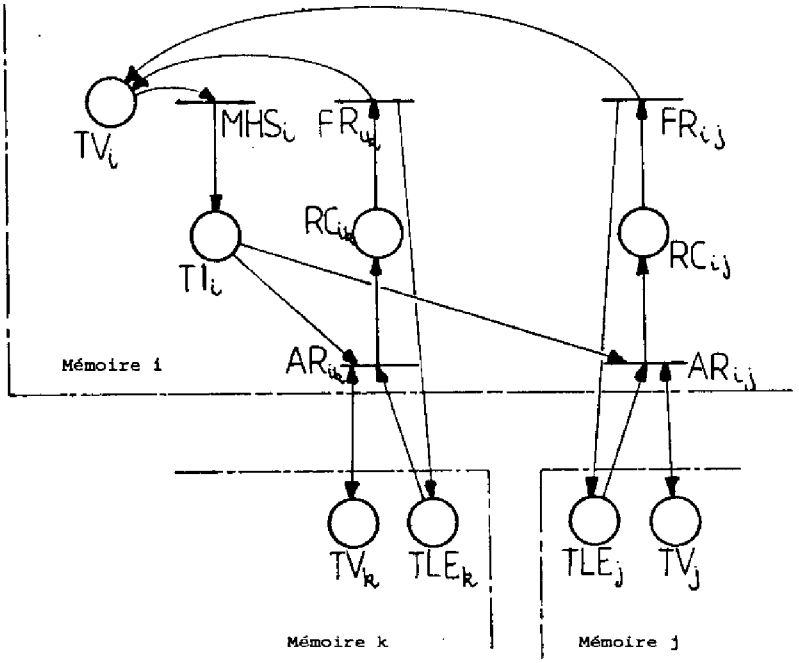


Figure A.2.1.

Ceci nous conduit à introduire cette nouvelle contrainte (test de la validité de la mémoire source) dans la spécification de l'algorithme (figure A.2.1.)

Pour ce réseau, nous avons "dédoublé" pour chaque module l'ensemble des places et transitions spécifiant la transaction restauration totale de telle manière que chaque ensemble AR_{ij}, RC_{ik}, FR_{ik} donne le nom de la mémoire source.

Notons toutefois, qu'en introduisant ce test sur la validité de la mémoire source, le réseau global possède une configuration de marquage qui amène un blocage mortel. Ce marquage est celui obtenu par le tir de toutes les transitions MHS_i. En fait, ce blocage est bien normal car il correspond au cas où toutes les mémoires sont hors d'usage, et le système est sans possibilité de reprise puisque toutes les informations sur le passé ont été irrémédiablement perdues.

En faisant le test de validité sur la mémoire source, les invariants linéaires de places sont alors les suivants :

$$|M(TV_i)| + |M(TI_i)| + \sum_{\substack{j=1 \\ j\neq i}}^{m} |M(RC_{ij})| = 1$$

$$|M(CM)| + \sum_{i=1}^{m} \left[|M(EA_i)| + |M(EC_i)| + m \sum_{\substack{j=1 \\ j\neq i}}^{m} |M(RC_{ij})| \right] = m$$

$$|M(TLE_i)| + |M(LC_i)| + n|M(EC_i)| + \sum_{\substack{j=1 \\ j\neq i}}^{m} |M(RC_{ji})| = n$$

Ces invariants sont respectivement la généralisation des invariants et le terme $\sum_{\substack{j=1 \\ j\neq i}}^{m} |M(RC_{ij})|$ représente le dédoublement de la place "RC_i".

ANNEXE 2.2. GENERALISATION DE LA RECONFIGURATION MEMOIRE APRES UNE MISE HORS SERVICE TEMPORAIRE

Pour ce cas aussi, il nous faut introduire le test sur la validité de la mémoire source. Le réseau est alors celui de la figure A.2.2.

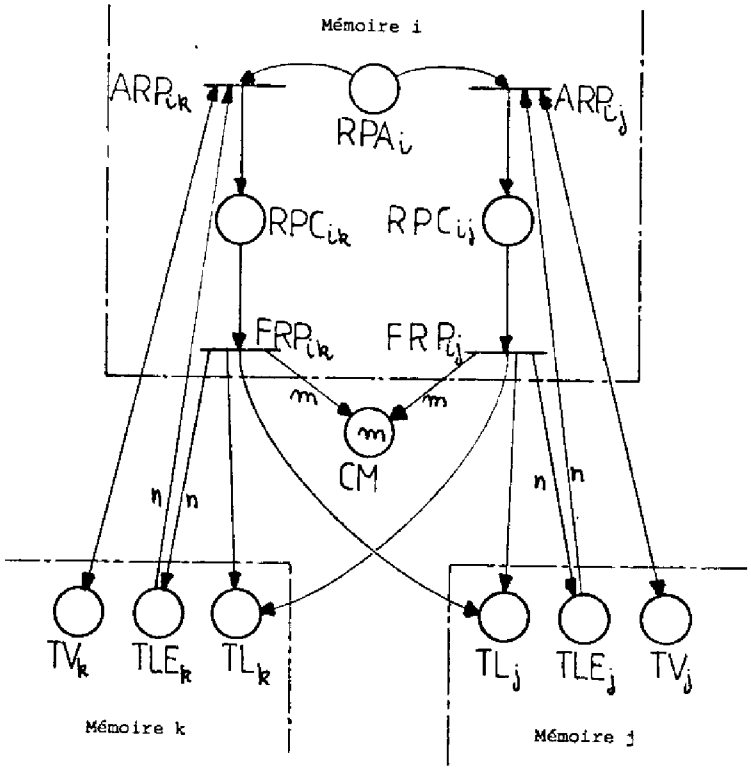


Figure A.2.2.

Pour cette solution aussi, il existe une configuration blocante, elle aussi est très logique. Pour cette solution, les invariants qui ont été modifiés sont alors les suivants :

$$\left|M(TL_i)\right| + \left|M(EA_i)\right| + \left|M(EC_i)\right| + \sum_{\lambda=1}^{m} \sum_{\substack{\delta=1 \\ \delta \neq i}}^{m} \left[\left|M(RPA_{\lambda j})\right| + \left|M(RPC_{\lambda j})\right|\right] = 1$$

$$\left|M(CM)\right| + \sum_{\delta=1}^{m} \left[\left|M(EA_j)\right| + \left|M(EC_j)\right| + \sum_{\substack{\delta=1 \\ \delta \neq i}}^{m} m\left[\left|M(RPA_j)\right| + \left|M(RPC_j)\right| + \left|M(RC_j)\right|\right]\right] = m$$

$$n\left[\left|M(EC_i)\right| + \left|M(RPA_i)\right| + \sum_{\substack{\delta=1 \\ \delta \neq i}}^{m} \left|M(RPC_{ij})\right|\right] + \left|M(TLE_i)\right| + \left|M(LC_i)\right| + \sum_{\substack{\delta=1 \\ \delta \neq i}}^{m} \left|M(RC_{ji})\right| = n$$

B I B L I O G R A P H I E

- - - - - - - - - - - - - -

ADA 78 G. ADAMS, T. ROLANDER
Design motivations for multiple processor microcomputer systems.
Computer Design. March 78.pp 81-89.

ALS 76 P.A. ALSBERG , J.D. DAY
A Principle for Resilient Sharing of Distributed Resources
Proceedings of the Second International Conference on Software Engineering.Oct. 76.

AGE 79 T. AGERWALA
Putting Petri nets to work
Computer, December 79 pp 85-94.

AYA 79 J.M. AYACHE, P. AZEMA, M. DIAZ
Observer : A concept for on line detection of control errors in concurrent systems.
FTCS 9 Madison, Wisconsin June 79 pp 79-86.

AZE 80 P. AZEMA, B. BERTHOMIEU, P. DECITRE
The design and validation by Petri nets of a mecanism for the invocation of remote servers.
Proceedings of IFIP Congress 80, Tokio-Melbourne 1980 pp 599-604

BAN 79 J.S. BANINO, C. KAISER, H. ZIMMERMANN
Synchronisation for distributed systems using a simple broadcast channel.
Ist International Conference on distributed Computing Systems Huntville, Oct. 79.

BERN 79 BA. BERNSTEIN, D.W. SHIPMAN, J.B. ROTHNIE
Concurrency Control in SDD-1 : a system for distributed data-bases ; Part 1. Description. 1
Computer Corporation of America, Cambridge, Massachusetts , Technical Report CCA 03 79 Jan. 79.

BER 77 G. BERTHELOT
Une méthode de vérification des réseaux de Petri
Journées AFCET sur les réseaux de Petri Mars 77 Paris.

BER 78 G. BERTHELOT
Vérification des réseaux de Petri
Thèse de 3e Cycle Université de Paris VI, Paris 78.

BER 79 B. BERTHOMIEU
Analyse structurelle des réseaux de Petri Méthodes et Outils
Thèse de Docteur-Ingénieur U.P.S. Toulouse, Septembre 79.

BOU 78 J. BOUSSIN
Synthesis and analys of logic automation systems
IFAC congress 1978

BRI 72 P. BRINCH - HANSEN
Structured multiprogramming
Communications ACM, July 72 Vol 15 N°7 pp 574-578

CHE 79 B. CHEZALVIEL - PRADIN
Un outil graphique interactif pour la vérification des systèmes à évolution parallèle décrits par réseaux de Petri.
Thèse de Docteur-Ingénieur U.P.S. Toulouse Décembre 79.

COU 71 P.J. COURTOIS, F. HEYMANS, D.L. PARNAS
Concurrent control with "readers" and "writers"
Comm ACM Vol 14 1971 pp 667-668.

COU 79 M. COURVOISIER, J.P. SECK
Réalisation matérielle des réseaux de Petri généralisés
Electronics Letters 22 - NOV 79 Vol 15 N°24.

COU 80 M. COURVOISIER, JC. GEFFROY, J. GOLINSKI, G. LATAPIE JP. SECK, R. VALETTE.
Synchronization and security of a multiprocessor system.
Third International Conference on Fault-tolerant Systems and Diagnostics Sept 80 Katowice pp 187-193.

CRI 77 Z. CRISTIAN RADU
Colored Petri nets : their properties and applications
Ph D University of Michigan 1977.

CRO 73 CROCUS
Systèmes d'éxploitation des ordinateurs
Dunod 73.

ELL 77 C.A. ELLIS
Consistency and Correctness of Duplicate Database Systems
Proceedings of Sixth ACM Symposium on Operating Systems principles. NOV 77 - pp 67-84.

GEN 79 H.J. GENRICH, K. LAUTENBACH
The analysis of distributed Systems by means of Predicate/Transition-Nets
Semantics of Concurrent Computation Evian 79 pp 123-146

GIR 78 C. GIRAULT
Réseaux de Petri et synchronisation de processus
Note IP 78-02 Institut de Programmation Paris VI

GSA 77 Groupe de travail Système logique de l'AFCET
Pour une représentation normalisée du cahier des charges d'un automatisme logique.
Automatique et informatique industrielles
n° 61 Novembre 77 pages 27-32

HOA 72 C.A.R. HOARE
Towards a theory of parallel programming.
Operating Systems Techniques, Academic Press New York, 1972.

HOA 74 C.A.R. HOARE
Monitors : An Operating System Structuring Concept
Comm. ACM - Oct 74 - Vol 17 N°10 - pp 549-557

HOA 78 C.A.R. HOARE
Communicating Sequential Processes
Comm ACM Vol 21 - August 78 N°8.

JEN 80 K. JENSEN
Colored Petri Nets and the Invariant-method
Computer Science Department AARHUS UNIVERSITY
Rapport interne. DAIMI PB-104 - Aug 80.

KAR 69 R.M. KARP, R.E. MILLER
Parallel Programm schemata
JCSS Vol 3 - 1969 - pp 147-195

KES 77 J.L.W. KESSELS
An Alternative to Event Queues for Synchronization in Monitors.
Communication ACM July 77 - Vol 20 - N°7 - pp 500-503

KOT 78 V.E. KOTOV
An Algebra for Parallelism based on Petri Nets
Mathematical Foundations of Computer Science 78 - pp 39-55
Proceedings, 7th Symposium Zakopane, Poland 78

LAG 76 M.L. LAGIER
Conception et réalisation d'un moniteur temps réel pour un système multimicroprocesseur
Thèse Docteur-Ingénieur - Nancy décembre 1976.

LAM 78 L. LAMPORT
Time, clocks and the ordering of events in a distributed system.
Communication ACM - July 78 - Vol 21 - N°7 - pp 558-565

LAT 80 G. LATAPIE, R. VALETTE
An algorithm ensuring data consistency in a fault-tolerant distributed system
Third International Conference on Fault-tolerant Systems and Diagnostics - SEPT 80 - Katowice pp 77-84.

LAU 74 K. LAUTENBACH, H.A. SCHMID
Use of Petri nets for proving correctness of concurrent process systems
IFIP Congress 74 - pp 187-191.

LAU 75 P.E. LAUER, R.H. CAMPBELL
Formal semantics for a class of high level primitives for coordinating concurrent processes.
Acta Informatica - VOL 5 - Fasc 4 75 - pp 297-332.

LEL 78 G. LE LANN
Algorithms for Distributed Data-Sharing Systems which use tickets
Proceedings of the third Berkeley Workshop on Distributed Data Management and Computer Networks, Aug 78 San Francisco pp 259-272.

MAZ 78 G. MAZARE
Structure multi-microprocesseurs problème de parallèlisme, Définition et évaluation d'un système particulier.
Thèse d'état - Grenoble Juin 78.

MEM 77 G. MEMMI
Semiflows and Invariants, applications for Petri net theory
Journées AFCET sur les réseaux de Petri, mars 77 Paris.

MIC 79 C. MICHEL
Ensemble d'outils pour la conception assistée par ordinateur de systèmes numériques à haute performance.
7ème colloque sur le traitement du signal et ses applications. Nice 79.

MIR 79 S. MIRANDA
Etude d'un problème d'interférence dans la mise à jour de données réparties sur un réseau d'ordinateurs
Journées BIGRE, "Méthodes et outils pour la conception des systèmes répartis". Nancy Jan. 79 - pp 76-113.

MOA 79 M. MOALLA, G. SAUCIER, J. SIFAKIS, M. ZACHARIADES
A design tool for the multilevel description and simulation of systems of interconnected modules
3° Annual Symposium on Computer Architecture Tampa USA 1979.

MUL 75 A.P. MULLERY
The distributed control of multiple copies of data.
IBM Thomas J. Watson Research-Center, Yorktown Heights, New York Aug 75.

MUN 78 T. MUNTEAN
Spécification de la synchronisation par contraintes.
Thèse de 3° cycle - Grenoble juin 78.

PET 77 J.L. PETERSON
Petri Nets
A.C.M. Computing Surveys - VOL 9 - N°3 - September 77 pp 223-252.

REE 79 D.P. REED, R.K. KANODIA
Synchronization with eventcounts and sequencers
Comm ACM - Feb 79 - VOL 22 - N°2 pp 115-123.

ROB 77 P. ROBERT, JP. VERJUS
Towards autonomous description of synchronisation module.
IFIP Congress 77 Toronto.

ROS 78 D.J.ROSENKRANTZ, R.E.STEARNS, P.M.LEWIS
System level concurrency control for distributed database systems.
ACM Transactions on Data base Systems - VOL 3 - N°2 June 78.

ROU 78 G.P. ROUCAIROL
Mot de synchronisation
RAIRO Informatique/Computer Science - VOL 12 - N°4 1978.

SAM 80 M. SAMI
Conception et réalisation d'un automate programmable par schémas à réseaux de Petri
Thèse de Docteur-Ingénieur UPS Toulouse Octobre 1980.

SCH 78 M. SCHIFFERS
Analysing program solutions of coordination problems by CP-Nets.
Mathematical Foundations of Computer Science 78 PP 462-473
Proocedings 7th Symposium Zakopane Poland 78.

SEG 79 J. SEGIN, G. SERGEANT, P. WILMS
A majority concensus algorithm for the consistency of duplicated and distributed information.
The 1st international conference on distributed computing systems
Hunstville Alabama - Oct 79 pp 617-624.

SIF 79 J. SIFAKIS
Le contrôle des systèmes asynchrones : concepts, propriétés, analyse statique.
Thèse de Docteur ès-Sciences Grenoble 79.

THO 78 R. THOMAS
A solution to the concurrency control problem for multiple copy data bases.
Digest of papers COMPCON 78 Spring.

TOU 76 L. TOURRES
Une méthode nouvelle d'étude des systèmes logiques et son application à la réalisation d'automatismes programmés.
Revue générale de l'Electricité T-85 N°3 Mars 76.

VAL 76 R. VALETTE
Sur la description, l'analyse et la simulation des systèmes de commande parallèles.
Thèse de Docteur ès-Sciences Toulouse 76.

VAL 79 R. VALETTE, M. DIAZ
A Methodology for easily provable implementation of synchronization mechanisms.
1st European Conference on Parallel and Distributed Processing, Toulouse February 14-17, 1979 pp 156-162.

VAL$_a$ 80 R. VALETTE, M. MENASCHE, G. LATAPIE
Monitors Petrinets and error confinement.
IEEE 10th International Symposium on Fault-Tolerant Computing.
KIOTO Japan OCT. 80.

VAL$_b$ 80 R. VALETTE, M. COURVOISIER
Systèmes de commande temps réel
Editions SCM - OCT. 80.

VER 77 JP. VERJUS, J. MOSSIERE, M. TCHUENTE
Sur l'exclusion mutuelle dans les réseaux informatiques
IRISA Publication interne N°75 - 1977.

WIL 79 P. WILMS
Etude et comparaison d'algorithmes de maintien de la cohérence dans les bases de données réparties.
Thèse de Docteur-Ingénieur - Grenoble - Novembre 1979.

WIL 80 P. WILMS
Qualitative and uantitative comparaison of update algorithms in distributed database
Distributed Data Bases C.Delobel-W.Litwin (eds) North-Holland Publishing Compagny pp 275-294

TABLE DES MATIERES