CONCEPTION ET REALISATION D'UN OBSERVATEUR DE PROTOCOLES POUR LA SURVEILLANCE EN LIGNE DES SYSTEMES DISTRIBUES

Jocelyne Noubel

N° d'ordre : 775

Année 1981

THESE

présentée

DEVANT L'UNIVERSITE PAUL SABATIER DE TOULOUSE

pour l'obtention

du Diplôme de DOCTEUR INGENIEUR

par

Jocelyne NOUBEL

Ingénieur I.N.S.A.T.

CONCEPTION ET REALISATION D'UN OBSERVATEUR DE PROTOCOLES POUR LA SURVEILLANCE EN LIGNE DES SYSTEMES DISTRIBUES

Soutenue le 5 novembre 1981, devant la Commission d'Examen :

MM.	A. COSTES	*Président*
	F. ANCEAU P. AZEMA C. BETOURNE A. DE FERRY M. DIAZ D. LANCIAUX	*Examinateurs*

AVANT - PROPOS

Le travail présenté dans ce mémoire a été réalisé dans le cadre de la Division "Structures des Systèmes de Commande Automatique" du Laboratoire d'Automatique et d'Analyse des Systèmes du C.N.R.S.

Je tiens à remercier :

- Monsieur A. COSTES, Professeur à l'Institut National Polytechnique de Toulouse,
- Monsieur F. ANCEAU, Professeur à l'E.N.S.I.M.A.G.,
- Monsieur P. AZEMA, Maître de Recherche au C.N.R.S.,
- Monsieur C. BETOURNE, Professeur à l'Université Paul Sabatier de Toulouse,
- Monsieur A. DE FERRY, Responsable du Laboratoire d'Informatique au GIER-SCHLUMBERGER,
- Monsieur M. DIAZ, Chargé de Recherche au C.N.R.S.,
- Monsieur D. LANCIAUX, Ingénieur, Chargé de Mission à l'Agence de l'Informatique,

qui, malgré leurs nombreuses obligations, ont accepté de participer à la Commission d'Examen.

Je suis très reconnaissante envers tous les membres de l'Equipe "Logiciel et Communication" pour leurs conseils et leur soutien constant tout au long de mon séjour parmi eux.

Je remercie également les membres du Service "M.I.T.R.I." pour leur aide aussi amicale qu'efficace.

Enfin, mes remerciements vont à tous ceux, qui, à divers titres, ont permis la réalisation de ce mémoire.

INTRODUCTION

L'évolution rapide de la technologie informatique a permis d'envisager la conception de systèmes de très grande complexité et ayant une structure distribuée, tels que les réseaux d'ordinateurs ou les systèmes de commande et de contrôle de processus industriels.

Bien évidemment, la taille de ces systèmes et le haut degré de parallélisme entre les différents processeurs impliquent de les concevoir en prenant en compte certains critères fondamentaux, et parmi ceux-ci, la composante sûreté de fonctionnement nous paraît être extrêmement importante.

Les problèmes rencontrés en vue d'assurer la sûreté de fonctionnement sont nombreux, variés et se rapportent aux aspects suivants : conception correcte et validée, caractérisation et prévention des fonctionnements critiques, choix de l'architecture, test et maintenance. Ainsi, il est apparu nécessaire, lors de la réalisation d'un système, de disposer d'une méthodologie basée sur une approche en trois étapes principales :

- phase de définition des besoins ou cahier des charges,
- phase de spécification,
- phase d'implémentation.

Chaque phase de la conception doit être prouvée correcte ; ceci sera d'autant plus facile que la phase précédente l'a été et que le passage d'une phase à l'autre est plus direct, par exemple, la phase de spécification représentera effectivement les besoins définis à la phase précédente si le modèle de représentation permet une réécriture systématique des besoins.

Un point particulièrement important est la vérification de la phase d'implémentation qui consiste tout d'abord à vérifier statiquement le lien spécification-implémentation. S'il est possible ensuite de contrôler dynamiquement la conformité du lien implémentation-exécution, cela permettra la détection en ligne des fautes matérielles, logicielles ou humaines pouvant survenir lors du fonctionnement normal. C'est ce dernier problème qui fait l'objet de notre travail.

Le problème du contrôle en ligne de l'exécution d'un système a été largement étudié par plusieurs auteurs. La résolution de ce problème conduit à concevoir le système global de façon à ce que les fautes agissent de manière non identique sur deux parties différentes du système à comparer : le concept présenté ainsi est appelé "distinction". Il a été utilisé tout d'abord pour des réalisations matérielles [1]-[2], puis logicielles [3]-[4]-[5] afin de mettre en évidence la présence d'erreurs résultant de pannes matérielles, de fautes dans le logiciel ou d'événements imprévus.

La fiabilité du matériel s'étant fortement améliorée au cours de ces dernières années, la fiabilité du logiciel ne doit plus être négligée et peut même devenir un point prépondérant. Cette remarque nous a conduit à être très attentifs au problème de sûreté du logiciel.

Généralement, dans les travaux présentant différentes techniques d'auto-test, les structures considérées ne sont pas des structures totalement distribuées et le contrôle en ligne est implémenté au niveau de chaque processus du système. Ainsi, les vérifications liées aux communications interprocesseurs dans un système distribué ne sont pas envisagées, il n'y a donc pas de détection de pannes au niveau global.

Notre démarche s'insère dans le cadre d'une étude, développée au L.A.A.S., d'un réseau local de microcalculateurs, sûr de fonctionnement, support de recherche d'un système distribué plus général de commande et contrôle de procédés industriels.

Pour cette raison, il nous a paru nécessaire de nous attacher aux problèmes de test dans les systèmes distribués temps réel et de développer une étude qui propose une méthodologie de conception d'un logiciel redondant permettant le contrôle en ligne des systèmes communicants. La réalisation d'un tel système d'autotest est le but du travail présenté dans ce mémoire.

Le Chapitre I présente les deux types de vérification nécessaires lors de la conception si l'on veut disposer d'un système sûr de fonctionnement : la vérification statique, permettant la validation des phases de définition des besoins et de spécification, et la vérification dynamique

de l'implémentation ou contrôle en ligne. Ensuite, plusieurs techniques de test du logiciel de processus séquentiels, basées sur l'utilisation de la redondance, sont décrites. Après avoir souligné la carence des méthodes de test utilisables dans les applications distribuées fonctionnant en temps réel, nous exposons le principe de base de la méthode proposée pour la conception d'un logiciel d'autotest dans un système distribué, ne possédant ni mémoire commune, ni processeur fonctionnel commun et dans lequel les seuls liens entre les différents processeurs sont des messages circulant sur un support de communication.

Ce logiciel se compose de deux parties distinctes : un exécutant et un observateur.

L'exécutant est constitué de l'ensemble des tâches fonctionnelles du système, implémentées de façon classique. L'observateur implémente une représentation abstraite, éventuellement simplifiée, des spécifications fonctionnelles du système, également appelée diagramme de mission. La redondance ainsi obtenue, qui résulte de la présence de l'observateur, permet la vérification du système pendant son fonctionnement réel.

Il faut néanmoins souligner que l'introduction de cette redondance logicielle doit permettre d'atteindre certains objectifs tels que :

a) elle doit rendre possible la sélection d'un niveau adéquat de vérification, d'un mécanisme particulier,
b) la fiabilité du système non redondant doit être conservée ou affectée de façon négligeable,
c) le fonctionnement du système global doit être très peu perturbé par les nouvelles interactions possibles.

Le Chapitre II définit la structure de l'observateur et les liens entre l'exécutant et l'observateur.

Nous introduisons ainsi le modèle sélectionné en fonction des critères précédents : les réseaux de Petri, ainsi que ses propriétés dynamiques et structurelles et donnons une première approche de la modélisation du diagramme de mission.

Le logiciel permettant d'implémenter ce modèle est également détaillé ; il se compose d'un simulateur de réseaux de Petri et d'une structure de données décrivant le diagramme de mission. L'évolution de l'observateur ne peut se faire qu'après la réception d'événements faisant évoluer l'exécutant ; nous avons choisi de contrôler le bon fonctionnement de l'exécutant de façon invisible pour lui, l'observateur est alors à l'écoute du support de communication.

La méthodologie de modélisation du diagramme de mission de l'observateur est exposée au Chapitre III et appliquée dans le réseau REBUS, réseau local de microcalculateurs, pour contrôler le protocole de gestion d'une ligne de transmission par anneau virtuel robuste et reconfigurable. Cette méthodologie tient compte de la structure distribuée des systèmes à contrôler et des pertes de messages qui peuvent survenir dans le système. Nous verrons que l'incidence de ces pertes de messages est différente selon que le message a été perdu par le processeur destinataire ou par le processeur observateur.

Enfin, le Chapitre IV traite de l'implémentation de l'observateur dans le système REBUS, un système distribué de commande en temps réel développé au Laboratoire. Il définit le support matériel puis le support logiciel et décrit ensuite les différentes fonctions permettant de réaliser un système redondant de contrôle en ligne captant des messages sur un support de communication et dialoguant avec un opérateur.

Son schéma d'implémentation est ensuite présenté ; chaque fonction est structurée en tâches concurrentes synchronisées à l'aide d'un moniteur temps réel multitâches RMX/80 . Les fonctions réalisées par les différentes tâches sont alors développées. Deux exemples de systèmes distribués temps réel dans lesquels l'observateur a été implémenté (le système REBUS) ou est en cours de mise en oeuvre (le système MODUMAT développé par la compagnie SCHLUMBERGER) sont présentés et montrent sa facilité d'implémentation dans des systèmes communicants.

Une annexe explicite alors les liens entre les différentes tâches de l'observateur et les phases de fonctionnement : phase d'initialisation, de traitement d'un message et de suivi de l'observation

CHAPITRE I

UN LOGICIEL REDONDANT

-

INTRODUCTION

La complexité inhérente aux systèmes distribués introduit de nouvelles contraintes tant au niveau de la conception qu'au niveau de la réalisation.

La conception de systèmes d'une telle complexité requiert alors une approche systématique en plusieurs étapes permettant d'aboutir à une réalisation fiable.

La première étape est l'étape de définition d'un cahier des charges ; elle est constituée par l'inventaire des composantes du système et des relations qui les lient. Le recensement de l'ensemble des besoins permet de définir le système de façon informelle.

La deuxième étape est l'étape de spécification formelle ; elle constitue l'expression des besoins définis à l'étape précédente en un langage particulier et permet de lever les ambiguïtés pouvant exister dans le cahier des charges. Afin de ne prédéterminer en aucune façon la nature de la réalisation, la description est uniquement fonctionnelle. Le modèle formel obtenu à l'aide d'une représentation explicite est le résultat d'une série de transformations du modèle informel [6].

Après ces deux étapes, le concepteur dispose donc d'une spécification informelle associée à une spécification formelle lui permettant de décrire de façon non ambiguë le système à développer.

Dans les systèmes distribués, avant de passer à l'étape suivante, une étape intermédiaire de conception globale est nécessaire. Elle permet de définir les fonctions à réaliser, leur implantation géographique, matérielle et logicielle.

L'étape suivante, la plus classique, est l'étape d'implémentation logicielle et matérielle. Les choix à ce niveau concernent la structure du matériel et du logiciel et en particulier le langage de programmation.

Malgré tous les soins apportés alors par le concepteur, il est bien évident que des erreurs peuvent être introduites à chaque étape de la réalisation : erreurs de spécification, de conception et de codage.

Afin d'assurer la sûreté de fonctionnement du système, il est alors nécessaire de prouver que le comportement du système est bien celui décrit par le cahier des charges et que le système est correctement réalisé.

La vérification de l'ensemble de la conception implique alors la vérification de chacune des différentes étapes de la conception. Une première vérification, que l'on peut qualifier de statique, est décrite dans le paragraphe I.1., elle concerne la fiabilité avec laquelle le logiciel est conçu et construit.

La deuxième vérification, appelée vérification dynamique, concerne le contrôle en ligne du fonctionnement du logiciel ; elle est présentée au paragraphe I.2.

Plusieurs méthodes de vérification en ligne de l'intégrité des données dans des processus séquentiels [7]-[8] ou de vérification de l'intégrité du contrôle dans un système avec centralisation du contrôle [4] sont exposées.

Nous proposons ensuite une méthode de vérification dynamique du contrôle dans un système temps réel, basée sur l'utilisation d'une redondance logicielle.

I.1 VERIFICATION STATIQUE

La vérification statique concerne la fiabilité avec laquelle les spécifications formelles reflètent le cahier des charges et le degré de conformité de ces spécifications vis-à-vis de l'implémentation actuelle [9].

La spécification formelle est obtenue à partir du cahier des charges par transformation des spécifications informelles [6]. Ainsi, la conformité des spécifications au cahier des charges est d'autant plus aisée à vérifier que l'outil de formalisation a permis une reécriture systématique de l'ensemble des relations particulières entre objets et que le modèle formel obtenu peut être validé de façon automatique. De nombreuses analyses ont été réalisées en vue de déterminer un tel outil de formalisation [10].

Parmi différents modèles graphiques possibles (machines à états, SADT, réseaux de Petri,...) il est apparu que les réseaux de Petri sont particulièrement appropriés. En effet :

- d'une part, ils permettent de représenter de façon non ambiguë le comportement attendu du système, de détecter automatiquement des spécifications incohérentes ou incomplètes vis-à-vis du cahier des charges, par analyse des propriétés classiques et étude des invariants du réseau,
- d'autre part, leur pouvoir de description leur permet de représenter des opérations élémentaires (séquencement d'opérations internes à un processus séquentiel ou communes à des processus parallèles, exclusion mutuelle, allocation de ressource) de façon systématique.

Après avoir validé l'étape de spécification, il est nécessaire de détecter les erreurs de conception et de codage introduites lors de l'étape d'implémentation. Cette vérification est généralement réalisée par application de variables de test aux entrées du système et par comparaison des résultats obtenus avec ceux dérivés des spécifications. Malheureusement, elle ne peut être que partielle car il n'est pas réalisable dans un système complexe de tester toutes les entrées et/ou d'exécuter tous les chemins possibles : le nombre de cas à envisager est très grand, la vérification par application de séquences choisies par le concepteur ne permet de tester que les parties qui lui sont familières.

Une méthode de génération automatique de données de test d'un programme a été développé par RAMAMOORTHY [11], mais l'application de ces séquences de test ne permet qu'une validation partielle d'un programme de taille moyenne.

Ainsi, l'application de séquences de test permet de détecter certaines erreurs, mais il n'existe aucune stratégie de test qui garantisse que le programme est sans erreur. Il peut donc rester quelques erreurs de codage ou de conception non détectées lors de l'exécution des tests, mises en évidence au cours du fonctionnement en temps réel et pouvant devenir dangereuses pour le système.

En plus de ces erreurs latentes, des défectuosités non prévisibles [illegible] peuvent induire des mauvais fonctionnements tels que des [illegible] de mémoire et rendre le système peu fiable.

Ainsi pour des systèmes temps reel, tels que la commande et le contrôle de processus industriel, dans lesquels une grande sûreté de fonctionnement est requise, les remarques précédentes montrent qu'il est nécessaire de vérifier dynamiquement le comportement du système, ceci en vue de détecter, isoler ou corriger dès leur manifestation, les effets de ces erreurs non éliminées.

I 2. VERIFICATION DYNAMIQUE

Le contrôle en ligne du fonctionnement du logiciel a pour but de détecter l'incidence éventuelle des fautes matérielles, logicielles et humaines sur l'execution correcte.

Pour tenir compte de l'existence présumée de pannes d'origine matérielle ou d'erreurs de conception, et pour les détecter dès leur apparition, le concept de "distinction" a été largement utilisé.

La distinction est définie comme la manière de réaliser une tâche par des materiels séparés et/ou des logiciels différents. Ainsi, une faute dans le systeme global conduira à des resultats différents sur deux parties distinctes du système.

Le contrôle de la discordance resultante est appelé l'auto-test.

Par la suite, notre démarche, surtout orientée vers le logiciel, reposera essentiellement sur l'hypothese que deux programmes ou deux ensembles de programmes ont un comportement différent en présence de pannes.

La vérification en temps reel de la correction du système se réduira donc à prouver que deux programmes restent, d'une certaine manière, équivalents au cours du fonctionnement normal. Ainsi, nous nous sommes principalement attachés à examiner les différentes techniques de test introduisant la redondance logicielle dans les processus séquentiels, puis à définir une méthode d'approche qui permette de realiser la détection d'erreurs de conception dans un système distribué temps réel.

Les techniques d'utilisation de la redondance logicielle peuvent être classées en différentes catégories en fonction des aspects mis en valeur et de la structure du système étudié [12].

1.2.1. *Processus séquentiel*

Le cas de processus sequentiels ou de processus parallèles dans lesquels la fonction de détection d'erreurs est sise au niveau de chaque tâche a eté le plus largement étudié par plusieurs auteurs, certains s'attachant plus particulièrement à vérifier la cohérence des données ou celle du contrôle.

1.2.1.a. *Approche intégrité des données*

Une approche développée par RANDELL [8]-[13] permet de vérifier la cohérence des structures de donnees et des valeurs de données. Son but est de faciliter la détection et le recouvrement d'erreurs (ces erreurs sont principalement des erreurs de codage résiduelles et non des mauvais fonctionnements occasionnels du matériel).

Elle contient une solution générale au problème de la réparation des dommages causés par une composante erronée et du transfert des commandes vers un composant de remplacement approprié. Elle fournit aussi une méthode de structuration explicite du logiciel.

Le logiciel de l'ensemble du processus séquentiel est composé de blocs de programme : les blocs de recouvrement ("recovery blocks") exécutant l'algorithme à réaliser. De façon générale, un bloc de programme est un ensemble d'instructions ordonnées tel que son exécution commence par la première instruction, se termine par la dernière, toutes les instructions étant exécutées dans un ordre donné.

Une syntaxe possible pour les blocs de recouvrement est la suivante :

bloc de recouvrement :: = ensure ⟨test d'acceptation⟩ by
⟨algorithme primaire⟩ else by
⟨algorithmes alternatifs⟩ else error

Ainsi, chaque bloc de recouvrement est constitué d'un bloc conventionnel déroulant l'algorithme primaire, de zéro ou plusieurs blocs logiciels de remplacement exécutant le(s) algorithme(s) alternatif(s) et d'un mécanisme de détection d'erreur : le test d'acceptation.

C'est donc une unité autonome de détection d'erreur et de recouvrement. Son fonctionnement est le suivant : le système exécute d'abord le programme conventionnel décrit par l'algorithme primaire, un test d'acceptation est alors évalué sur ses sorties. Si les valeurs de sortie de l'algorithme précédent ne sont pas acceptées par le test, l'algorithme alternatif est exécuté à son tour, le processus étant remis dans l'état où il se trouvait avant d'entrer dans l'algorithme primaire (point de reprise). Si l'algorithme primaire (ou un algorithme alternatif) passe le test, les autres algorithmes alternatifs sont ignorés et le système continue son exécution de façon normale. Par contre, si le dernier algorithme alternatif ne passe pas le test d'acceptation, le bloc entier est considéré en panne.

Alors que l'algorithme primaire est destiné à être utilisé normalement pour accomplir la fonction désirée, les algorithmes alternatifs sont conçus pour réaliser la même fonction d'une manière différente, mais plus simple.

Un aspect important dans cette méthode réside dans la production de tests d'acceptation efficaces et dans la définition des états à sauvegarder dans les points de reprise.

De façon idéale, le test doit assurer que les programmes des blocs de recouvrement ont satisfait tous les aspects de la spécification ; malgré tout, il faut éviter que le volume d'informations à stocker à chaque point ne soit trop important.

Cette méthode peut être utilisée pour les systèmes constitués de processus indépendants ; dans ce cas, le rétablissement de l'état du processus est purement local au processus en panne. Par contre, pour les processus interagissants (système à multiprogrammation, système à processeurs fortement couplés avec partage d'une mémoire commune,...), lorsqu'un processus ne passe pas le test d'acceptation, il peut y avoir propagation de l'erreur.

En effet, le processus détecté en panne doit revenir à son plus récent point de reprise. S'il y avait eu échange d'information avec un autre processus, cette information ne doit pas être prise en compte par le processus communicant, qui doit donc également revenir à son point de reprise. Ce phénomène peut se reproduire pour d'autres processus et, dans le cas le plus défavorable, peut conduire tous les processus à revenir dans leur état initial. Cet effet a été appelé "effet domino" par RANDELL. Pour éviter cette propagation de l'erreur d'un processus aux autres, il faut alors coordonner les actions de recouvrement en englobant les interactions entre processus dans des "conversations". Ainsi, tous les processus engagés dans une conversation devront satisfaire leur propre test d'acceptation avant que l'un d'entre eux ne soit autorisé à sortir de la conversation. La non-acceptation des valeurs de sortie d'un processus le conduit, ainsi que les autres processus de la conversation, à revenir dans un état initial cohérent. Le concepteur se doit alors de structurer avec soin les programmes exécutés par des processus parallèles de façon à ce que la longueur de la chaîne de recouvrement la plus importante n'excède pas une limite tolérable ; ceci est en général une tâche très difficile.

Tout ceci ne peut s'appliquer que dans les systèmes avec contrôle centralisé ou au moins avec contrôle de recouvrement centralisé.

En effet, un problème nouveau se pose dans les systèmes totalement distribués : comment connaître, à un instant donné, l'état global des données dans le système ? Comment définir alors des tests d'acceptation ?

Ainsi, on n'a aucun moyen de contrôle de l'intégrité du système, certaines parties peuvent tomber en panne pendant que le reste du système continue à fonctionner normalement, n'ayant reçu aucune notification de la panne. Dans ce cas, la détection et le recouvrement des erreurs peuvent être réalisés de façon indépendante sur chaque processeur mais la surveillance du fonctionnement global du système ne peut pas être effectuée.

Alors que dans l'approche proposée par RANDELL les programmes structurés en blocs sont organisés d'une manière identique à la technique de redondance dynamique du matériel (modules de réserves), une autre

approche de test en ligne proposée par CHEN et AVIZIENIS [7] est également basée sur la notion de blocs mais la technique de redondance utilisée est identique à la technique de redondance statique de matériel (réplication et vote).

Cette méthode de test en ligne proposée dans [7] vise à masquer les effets des erreurs résiduelles de logiciel. Elle propose de générer $N \geq 2$ programmes indépendants, fonctionnellement équivalents, à partir de la même spécification initiale. En comparant les valeurs obtenues par les N versions, on peut détecter et corriger les erreurs d'implémentation. Pour ce faire, il faut définir un programme superviseur capable de décider lors du développement de chaque version si ses performances satisfont des criteres d'acceptation, de coordonner l'exécution des N versions, de faire un vote sur les résultats générés par chaque version pour $N > 2$ (ou de les comparer pour $N = 2$) puis d'entreprendre les actions nécessitées par ce vote : continuation, arrêt d'une ou plusieurs versions, continuation après modification de l'état local d'un ou plusieurs programmes.

Un point positif pour cette méthode est qu'elle permet de masquer les effets des erreurs logicielles à la sortie de chaque module. Malgré tout, cette méthode possède plusieurs limitations.

Tout d'abord, elle n'est pas applicable pour de grands systèmes car elle est très coûteuse, elle nécessite la réalisation de N versions par N groupes ou individus distincts. De plus, elle ne peut être utilisée que pour des programmes séquentiels. Ensuite, elle est très difficile à appliquer dans les cas particuliers où il n'existe pas de chemin unique pour obtenir la solution du problème et un vote sur les (ou une comparaison des) différents résultats intermédiaires obtenus ne peut pas être utilisé comme critère définissant un fonctionnement correct.

1.2.1.b. *Approche intégrée du contrôle*

Une autre approche, introduite par plusieurs auteurs [4]-[11]-[14]-[15] afin de détecter les erreurs de conception et d'implémentation, est celle qui vérifie la cohérence du flot de contrôle d'un programme, en temps réel, par rapport à sa définition initiale.

Elle comprend deux phases : la première phase consiste à construire une base de données contenant l'information utile pour la vérification du contrôle ; la deuxième phase conduira à insérer du code dans le programme en vue d'accomplir la vérification.

La base de données contient toutes les informations sur les chemins obtenues à partir de la structure détaillée du programme. Pour cela on définit tout d'abord des blocs de programme. Puis, dans le but d'obtenir tous les chemins possibles, le programme est partitionné en intervalles. Un intervalle est un sous-graphe dont les sommets sont les blocs de programme, ne contenant pas de boucle et ayant une seule entrée. Le principe de la méthode consiste à associer à chaque bloc dans un intervalle un nombre premier distinct. Chaque chemin dans un intervalle est alors identifié de façon unique par le produit de tous les nombres premiers associés à chaque bloc du chemin et chaque intervalle est identifié par un nombre distinct.

Ainsi, la base de données est constituée par l'identificateur du chemin, l'identificateur de l'intervalle suivant et le prédicat du chemin.

Nous montrons sur la Figure I.2. la construction de la base de données, à l'aide d'un exemple illustratif donné sur la Figure I.1., dans le cas de l'intervalle 3.

A partir du bloc d'entrée V1 de l'intervalle 3, deux chemins possibles sont caractérisés par les prédicats $j>n$ et $j\leqslant n$.

Le chemin caractérisé par $j>n$ conduit du bloc V1 (nombre premier associé = 2) au bloc V6 (nombre premier associé = 13), V6 est un bloc permettant de sortir de l'intervalle 3 pour entrer dans l'intervalle suivant 7. L'identificateur de ce chemin est le produit des nombres premiers associés aux blocs V1 et V6 du chemin, il est donc égal à 2 x 13 soit 26.

De la même façon, le chemin caractérisé par $j\leqslant n$ conduit du bloc V1 au bloc V2 (nombre premier associé = 3). A partir de ce bloc V2, un chemin possible permettant de sortir de l'intervalle 3 est celui caractérisé par le prédicat $j = i+1$ conduisant au bloc V7 (nombre premier associé 17) puis V4 (7) et enfin V5 (11). L'identificateur associé à ce chemin caractérisé par les prédicats $j\leqslant n$ et $j = i+1$ est égal à 2 x 3 x 17 x 7 x 11 soit 7854.

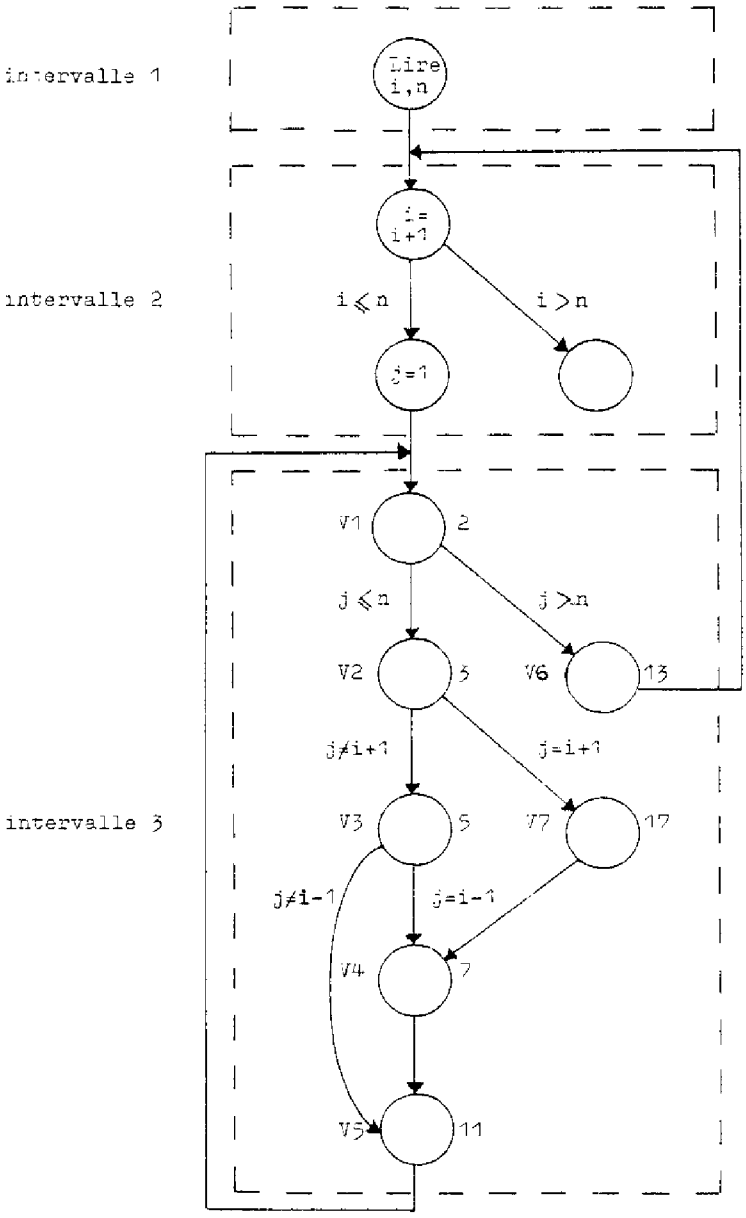


FIGURE I.1. Exemple illustratif

Prédicat du chemin			Identificateur du chemin	Identificateur de l'intervalle suivant
j > n			2.13 = 26	2
j ⩽ n	j=i-1		2.3.17.7.11 = 7854	3
j ⩽ n	j≠i+1	j=i-1	2.3.5.7.11 = 2310	3
j ⩽ n	j≠i+1	j≠i-1	2.3.5.11 = 330	3

FIGURE I.2. Base de données de l'intervalle 3

La deuxième phase conduit à insérer à l'intérieur de chaque bloc des instructions en vue d'accomplir la vérification de l'intégrité du contrôle. En cours d'exécution, on calcule l'identificateur du chemin ; on vérifie que l'identificateur de l'intervalle est celui attendu et, avant de quitter un bloc, on s'assure que l'identificateur calculé existe dans la base de données, toute divergence indique une erreur de contrôle. On cherche ensuite l'identificateur de l'intervalle suivant.

Puisque la base de données est obtenue à partir de la structure du programme, cette méthode permet de détecter en temps réel, les erreurs se caractérisant par une disconcordance entre les valeurs des identificateurs obtenus lors de la phase de conception et celles obtenues lors de l'exécution. Dans le cas de pannes multiples, la majorité des erreurs peut être détectée car, habituellement, toute panne conduit à un comportement erroné différent ; il y a peu de risques de voir deux pannes dans un système se compenser de façon telle que les identificateurs calculés soient en accord avec les identificateurs prévus par la phase de conception.

De la même façon que le modèle vérifiant l'intégrité des données, cette approche n'est applicable que pour les programmes séquentiels ou pour les tâches des programmes parallèles. En effet, pour des programmes parallèles, le contrôle de flux dépend de l'interaction des différents programmes, ce qui n'est pas prévisible lors de la conception.

De plus, même pour un programme séquentiel, il est très coûteux en temps d'exécution et en place mémoire de stocker tous les chemins de contrôle possibles si ce programme comporte : un nombre élevé de blocs et d'intervalles de petite taille, beaucoup de chemins dans un intervalle...

Conclusion : Nous avons présenté plusieurs techniques de contrôle en ligne du fonctionnement d'un système utilisant la redondance. Cette redondance peut permettre la vérification de l'intégrité des données ou de l'intégrité du contrôle. Dans tous les cas exposés, les structures considérées ne sont pas des structures distribuées, ce sont des processus séquentiels ou des processus parallèles avec centralisation du contrôle. Ainsi, la fonction d'observation du comportement est distribuée entre les différents processus

et les mécanismes de détection sont implémentés au niveau de chaque tâche. Il n'existe pas de méthode de vérification des communications interprocesseurs, c'est-à-dire que les pannes ou mauvais fonctionnements au niveau global de la synchronisation ne sont pas détectés.

1.2.2. *Processus distribués : le concept d'observateur*

C'est pourquoi nous nous sommes attachés à étudier le problème du contrôle en ligne dans le cas de systèmes temps réel distribués dans lesquels les différents processus mis en jeu interagissent lorsqu'ils communiquent ou lorsqu'ils sont en compétition pour l'attribution d'une ressource. Les caractéristiques de ces systèmes sont un haut degré de décomposition de tâches et une forte complexité résultant des communications et compétitions entre ces tâches.

La synchronisation a pour but d'organiser ces interactions. D'une manière générale, on peut distinguer dans un processus d'un système distribué deux parties distinctes : la partie interne qui effectue les traitements propres au processus et la partie synchronisation qui permet les échanges avec les autres processus.

Dans la conception d'un logiciel autotestable, comme il a été vu dans le paragraphe précédent, deux aspects complémentaires peuvent être considérés, à savoir la vérification de la consistence des données ou celle du contrôle. La définition et l'utilisation des données dépendant très fortement du type de l'application, nous avons choisi de vérifier le contrôle, c'est-à-dire le bon fonctionnement du séquencement des programmes ou des interactions entre les tâches.

Le mécanisme de synchronisation paraît être le point le plus important lorsque l'on considère des systèmes géographiquement distribués ; il s'agira par exemple de préciser qu'une section critique ne contient qu'un seul processus à un instant donné.

Pour cette raison, notre étude utilise une méthodologie de conception d'un logiciel redondant permettant la vérification en ligne des programmes parallèles s'exécutant sur des sites différents et communicants [5].

Les systèmes distribués présentent souvent une hiérarchie dans les fonctions à réaliser. En conséquence, un objectif essentiel à atteindre par cette méthode est la possibilité de sélection d'un niveau de vérification ou d'un mécanisme spécifique particulièrement intéressant, par exemple le mécanisme d'allocation d'une ressource commune. En effet, en général, l'observation complète de toutes les actions d'un système n'est d'une part pas forcément intéressante et d'autre part peut s'avérer très difficile sinon impossible. Souvent, comme nous l'avons rencontré, un mécanisme donné, significatif d'une vue globale, est suffisant pour contrôler le comportement d'un système. Dans ce cas, le diagramme de mission représente ainsi une description bien moins détaillée que celle utilisée par l'exécutant, ce qui est un autre intérêt important comme nous le verrons plus avant.

De plus, le point qui nous paraît fondamental est la satisfaction du critère de sûreté de fonctionnement. La redondance introduite dans le système ne doit pas affecter sa fiabilité : le système doit être aussi fiable avec la redondance que sans ; la partie redondante du logiciel doit, en conséquence, être extrêmement fiable, elle ne doit donc pas introduire d'erreurs dans le système. Sa fonction première est de détecter les fautes dues à des fonctionnements erronés de matériel, à des erreurs de conception du logiciel, à des événements imprévus et éventuellement de les corriger.

I.2.2.a. Structure de base

La complexité des mécanismes permettant d'assurer la cohérence du contrôle d'un système et sa reconfiguration en cas de comportement erroné rend nécessaire l'élaboration d'un modèle décrivant les mécanismes mis en jeu. Ce modèle appelé diagramme de mission du système devra spécifier les règles de séquencement et les contraintes de synchronisation ; par exemple, il peut indiquer que la procédure 2 doit toujours être exécutée après la procédure 1. Il définit à un haut niveau, c'est-à-dire sans donner la réalisation détaillée, les spécifications formelles, obtenues à partir des besoins du système ou de leur formalisation au niveau des fonctions (graphe de contrôle, invariants,...) et des protections (modèle de panne, priorité des tâches,...) à assurer au système.

Nous appellerons "observateur" le programme qui implémente le diagramme de mission sélectionné, qui permet de le faire évoluer.

L'observateur peut donc être vu comme un programme abstrait qui implémente à un haut niveau, les relations de séquencement entre actions élémentaires ou procédures. Le détail des primitives de synchronisation est écartée au profit de la représentation formelle.

D'un autre côté, le cahier des charges est analysé et codé, au besoin par d'autres personnes pour obtenir le logiciel constituant les programmes réels.

Nous appellerons "exécutant" l'ensemble des programmes développé de façon classique.

Dans le cas général, notre propos est de vérifier que le comportement du système est bien celui décrit par les spécifications. Il s'agira donc de détecter, en ligne, un comportement erroné des programmes réels à travers le programme abstrait.

Le logiciel redondant d'auto-test est donc composé du couple exécutant-observateur.

La première partie, l'exécutant, est formée par l'ensemble des processus qui réalisent de manière classique les tâches fonctionnelles du système prévues par le cahier des charges.

La deuxième partie, l'observateur, conçue et réalisée de façon distincte, implémente un modèle qui définit à un niveau donné les spécifications fonctionnelles du système global.

Les liens entre les différentes phases de la conception d'un logiciel redondant d'auto-test sont représentés sur la Figure I.3.

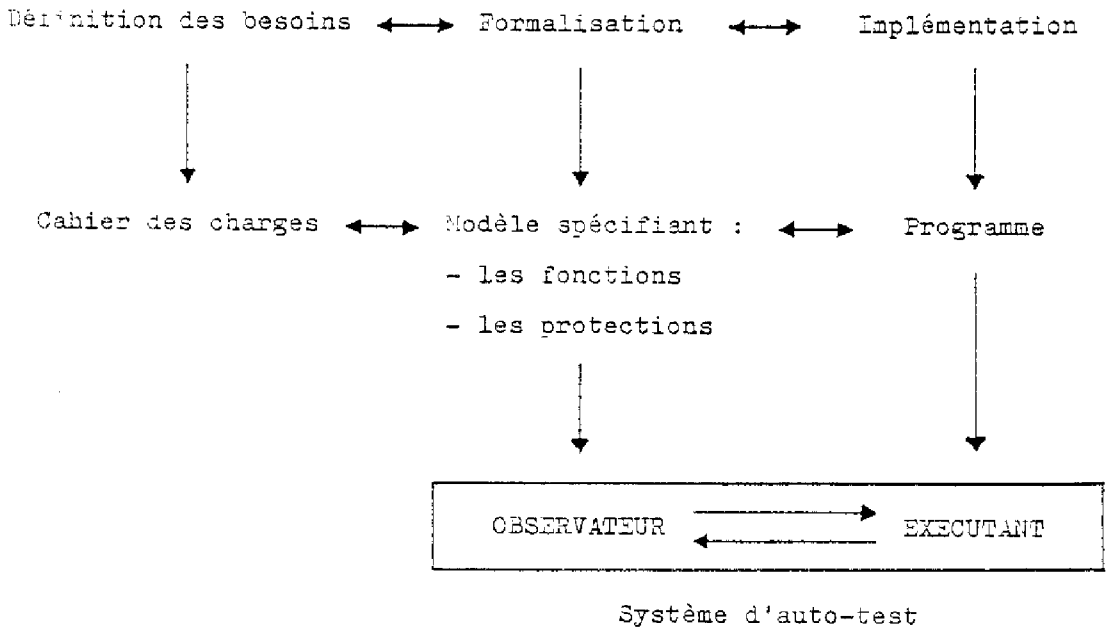


FIGURE I.3. Schéma de conception d'un système d'auto-test

I.2.2.b. Comparaison exécutant-observateur

Lors de l'utilisation du système, l'observateur est la partie du logiciel auto-testable qui réalise la détection et le diagnostic de l'erreur, pour cela, il vérifie que les évolutions de l'exécutant satisfont les contraintes définies par le diagramme de mission. Ceci n'est possible que si le système a des sorties observables, c'est-à-dire qui sont signalées au monde extérieur au système, qui laissent des traces.

Soit O l'ensemble des séquences d'événements observables, qui sont fournies par l'exécution des programmes réalisant les tâches fonctionnelles du système.

Le modèle fonctionnel du contrôle représente, pour un niveau donné, un préordre sur les actions de l'exécutant et donc il exprime d'une manière très compacte l'ensemble de toutes les séquences qui suivent les spécifications, séquences légales d'exécution des programmes de l'exécutant, soit L cet ensemble.

Ainsi,

Exécution des programmes	Modèle de la mission
⇓	⇓
O = { séquences d'événements observables }	L = { séquences légales d'exécution }

En conséquence, l'observateur détectera un comportement adéquat du système vis-à-vis du mécanisme observé si les séquences d'exécution du système sont en accord avec les règles de séquencement spécifiées par le diagramme de mission ; il détectera un comportement erroné du système dans le cas contraire et délivrera un diagnostic d'erreur.

Comportement adéquat de l'exécutant $\Leftrightarrow$ $O \subseteq L$

Comportement erroné de l'exécutant $\Leftrightarrow$ $O \not\subseteq L$

Lorsque le diagnostic du test est un diagnostic de comportement erroné, l'observateur devra activer une procédure d'erreur qui peut permettre soit d'alerter l'opérateur, soit d'arrêter le système, soit encore d'effectuer une reprise pour corriger l'erreur. Cette reconfiguration du système est facilitée par le fait que l'erreur est détectée dès que le comportement du système a divergé par rapport à sa référence (c'est-à-dire l'observateur).

De plus, une partie de l'historique du système jusqu'à la panne a pu être conservée de manière très condensée dans la mesure où l'observation ne concerne que l'activation d'un mécanisme abstrait de haut niveau.

I.2.2.c. *Lien entre l'exécutant et l'observateur*

L'observateur pour réaliser son diagnostic a besoin de recevoir des informations observables. Ce seront soit le résultat de la prise en compte par l'exécutant d'événements le faisant évoluer, soit ces événements eux-mêmes.

- Synchronisation par l'exécutant

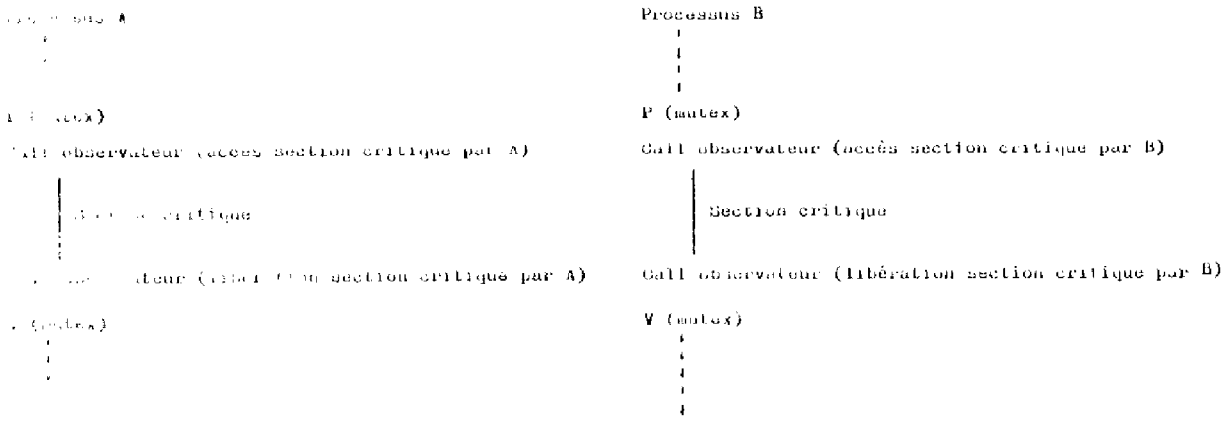
Dans le premier cas, l'exécutant doit coopérer avec l'observateur. Il envoie, à des points précis de son exécution, un compte-rendu de son état courant. Ces points seront appelés points de test et seront généralement associés au début ou à la fin d'une action pour indiquer respectivement soit que cette action va commencer soit qu'elle vient de se terminer.

L'insertion de ces points de test dans l'exécutant a été abordée mais non totalement étudiée dans [5]. Elle concerne la localisation, dans l'exécutant, d'appels à l'observateur qui permettront la vérification.

Le comportement d'un processus séquentiel est représenté par un ensemble d'événements totalement ordonnés ; dans ce cas, les appels à l'observateur sont facilement insérés en début et en fin d'action de la procédure à observer. Par contre, l'insertion de points de test dans un ensemble de processus parallèles est beaucoup plus délicate car le système est défini par un ensemble d'événements partiellement ordonnés en fonction des contraintes de synchronisation. L'insertion des points de test dans un tel ensemble doit donc se faire en accord avec l'ordre dans lequel les actions de synchronisation sont exécutées et modélisées dans le diagramme de mission. Deux règles permettent de respecter cette contrainte :

1) dans le cas d'un événement ou d'un message de synchronisation issu d'un processus après une section de code séquentiel et synchronisant un processus externe, l'appel à l'observateur est inséré avant l'événement,
2) dans le cas d'un événement ou d'un message de synchronisation reçu par un processus, l'appel est inséré après la réception de cet événement.

Nous donnons un exemple de l'utilisation de ces deux règles permettant d'insérer les points de contrôle, dans le cas de l'exclusion mutuelle de deux processus.



Par rapport à la règle 1, V(mutex) est un événement dirigé vers l'extérieur (libération d'une ressource ou de la zone critique), l'appel à l'observateur sera donc inséré avant l'événement. Par rapport à la règle 2, P(mutex) correspond à la réception d'un événement de synchronisation (accès à la ressource ou à la zone critique), l'appel à l'observateur est inséré après.

Dans cet exemple, l'observateur est implémenté par une procédure "observateur" appelée par l'exécutant, ayant pour paramètre d'appel l'événement à prendre à compte et signalant en fin d'exécution le résultat de l'observation. L'exécutant est l'ensemble des processus à synchroniser ; après l'appel à l'observateur, il attend de celui-ci un signal de fin d'exécution.

Les liens entre l'exécutant et l'observateur sont conceptuellement ceux représentés sur la figure I.4. et il y a donc échange explicite d'information entre l'exécutant et l'observateur. L'étude correspondante est à l'heure actuelle en cours de développement.

Remarque : Notons que, quelle que soit l'implémentation de l'observateur, un mécanisme d'accès en exclusion mutuelle par les composants de l'exécutant est nécessaire afin d'assurer que l'observateur est toujours dans un état cohérent. Cette méthode ne peut donc pas s'appliquer dans un système à contrôle distribué.

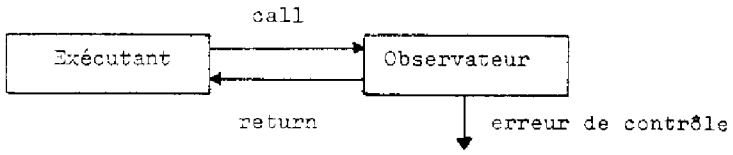


FIGURE I.4.

B. Synchronisation de l'observateur sur les événements observés eux-mêmes

Par contre, dans le second cas, l'observateur réagit à partir d'événements faisant évoluer l'exécutant et contrôlera alors le comportement de l'exécutant de façon indirecte ; en conséquence, celui-ci peut éventuellement ignorer la présence de l'observateur. Ce cas a été abordé dans [16] - [17] et paraît bien adapté pour la réalisation d'une structure redondante de contrôle dans un système distribué. Dans de tels systèmes, la synchronisation

des differents processeurs se fait par l'envoi et la réception de messages circulant sur un support de communication et les échanges d'informations se font également sur ce support.

Ainsi, on peut concevoir que les événements faisant évoluer l'état de l'exécutant et signalant à l'observateur la fin ou le début d'une action sont les messages circulant sur le support de communication. On peut alors definir une structure redondante d'auto-test (Figure I.5.) dans laquelle l'exécutant est distribué sur les N processeurs connectés sur le support de communication et l'observateur est implémenté sur un processeur chargé de recevoir les informations observables. C'est ce cas que nous considèrerons dans notre travail.

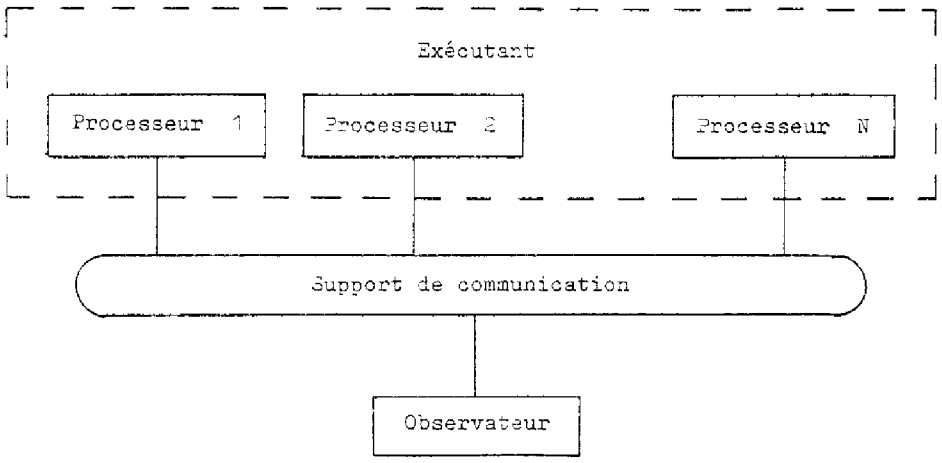


FIGURE I.5. Structure redondante d'auto-test

Soulignons qu'il n'y a pas d'échange explicite d'information entre l'exécutant et l'observateur.

Dans le premier cas, l'exécutant doit être modifié pour permettre l'observation, c'est-à-dire modifié par rapport à ce qu'il était sans les points de test, sans l'observateur. Cette méthode, sans modifier pour autant le comportement global du système, sera écartée au profit de la deuxième méthode dans laquelle l'introduction de l'observateur dans le système et son action sont dans un premier temps complètement transparentes à l'exécutant.

Nous faisons donc le choix d'un observateur contrôlant l'exécutant. D'ores et déjà, la structure du logiciel redondant ainsi définie permet d'atteindre deux des objectifs que l'on s'était proposé de réaliser : l'observation et le test du système global, la sélection possible d'un niveau de vérification ou d'un mécanisme particulier.

L'observateur est à l'écoute du support de communication, il peut donc capter l'ensemble des informations observables qui transitent sur ce support. Les informations ainsi reçues, provenant des divers processus du système, permettent de faire évoluer le modèle de l'observateur et de définir une part de l'état global. Parmi ces informations, les messages de synchronisation entre processeurs permettent de vérifier les mécanismes de synchronisation au niveau global de la communication.

De plus, la sélection d'un niveau particulier de vérification est rendue possible par la réception sélective des informations observables.

Ainsi, à un événement observé correspond, dans l'exécutant, une portion de programme (et éventuellement un appel à l'observateur) et dans l'observateur, une évolution du modèle par l'exécution du programme abstrait.

La satisfaction du troisième objectif : sûreté de fonctionnement du système non affectée par la redondance introduite conduit à concevoir la partie observateur du logiciel redondant de façon très fiable.

Il apparaît donc que le diagramme de mission représentant les spécifications du système et la réalisation de l'observateur doivent être tres fiables.

I.2.2.d. *Propriétés requises par le modèle*

Le modèle représentant l'observateur doit requérir certaines propriétés afin que l'ensemble du système puisse être fiable.

Un des points clefs pour le modèle réside dans sa possibilité de traduire explicitement les règles de séquencement des opérations dans un processus et les mécanismes de synchronisation.

Un organigramme ou des instructions classiques de langage de programmation structurée (if-then-else, do-while,...) suffiraient pour traduire les règles de séquencement d'un processus séquentiel.

Or, nous nous attachons à l'observation de systèmes temps réel formés de processus distribués ayant un haut degré d'interaction. Dans ce cas, les liens entre les processus doivent être spécifiés par un modèle qui permette une représentation aisée et systématique de leurs interactions. Plusieurs modèles sont disponibles : réseaux de Petri [18], CSP [19], diagrammes d'états finis [20], expression des chemins [21].

Un second point crucial pour le modèle est qu'il présente des propriétés telles qu'il permette une analyse statique. En effet, bien que le modèle exprime des spécifications d'un haut niveau, il pourraît être erroné. Le fait que le modèle soit erroné par rapport à l'exécutant n'est pas grave dans la mesure où, lors du test en ligne, le système observateur-exécutant détectera une situation d'erreur ; il y a problème en cas d'erreur dans le modèle et dans l'exécutant, cette erreur ne pouvant alors pas être détectée car l'évolution de l'exécutant est en accord avec celle spécifiée par le modèle de l'observateur.

Nous avons présenté les deux propriétés importantes que doit posséder le modèle afin de se garantir de certaines erreurs de conception et d'assurer que l'ensemble du système aura un comportement fiable. Nous indiquerons au chapitre suivant quel est le modèle de représentation du comportement du système qui a été choisi.

I.3. CONCLUSION

Ce chapitre nous a permis de définir une méthodologie de conception d'un logiciel redondant autotestable dans un système distribué. L'autotest est réalisé par une structure redondante : un observateur et un exécutant.

Nous avons choisi de nous intéresser ici à unestructure distribuée dans laquelle tous les processeurs fonctionnels sont connectés à un support de communication ; l'ensemble des programmes implémentés sur ces processeurs constituera l'exécutant et l'observateur sera implémenté sur un processeur spécialisé connecté lui aussi au support de communication.

L'observation des événements circulant sur le support permet le test en ligne du système par identification des séquences d'événements observables aux séquences légales d'exécutions définies par le diagramme de mission du système.

Nous avons souligné que cette méthodologie permet de sélectionner l'observation d'un niveau particulier de vérification et que l'introduction de la redondance dans le système n'a pas d'influence sur le comportement global ; ces deux points constituaient des objectifs à atteindre.

L'objectif de sûreté de fonctionnement du logiciel redondant sera considéré dans le chapitre suivant.

CHAPITRE II

CONSTRUCTION D'UN LOGICIEL AUTO-TESTABLE

-

INTRODUCTION

Le principe et les caractéristiques d'un logiciel redondant ayant été exposés au chapitre précédent, nous allons maintenant examiner les différentes composantes de cette structure.

Le paragraphe II.1. présente de façon très brève la conception de l'exécutant, cette partie du logiciel d'auto-test réalisant de manière classique les tâches fonctionnelles du système.

La redondance est introduite dans le système par la présence de l'observateur qui implémente un modèle abstrait décrivant les spécifications du système à observer. La conception de l'observateur présentée au paragraphe II.2. nécessite donc de définir un modèle de représentation des besoins du système et un logiciel permettant de faire évoluer ce modèle. Le choix d'un modèle est guidé par la satisfaction d'un certain nombre de propriétés énoncées au chapitre précédent ; il s'est porté sur les réseaux de Petri et de ce fait, le logiciel vérifiant le bon séquencement des événements observés est centré autour d'un simulateur de réseaux de Petri.

Après avoir présenté les méthodes de conception des deux parties du logiciel redondant, nous donnons dans le paragraphe II.3. les liens entre ces deux parties. Une règle d'évolution du modèle en fonction des événements observés est énoncée, puis dans le cadre d'une structure distribuée, nous suggérons deux types de modélisations, dans l'observateur, du couple émission-réception d'un message et faisons le choix de l'un d'entre eux.

II.1. CONCEPTION DE L'EXECUTANT

Dans la structure redondante définie précédemment, la redondance est située au niveau de l'observateur et l'exécutant est constitué de l'ensemble des programmes qui réalisent les tâches du système prévues par le cahier des charges. La conception de l'exécutant est alors réalisée de manière classique.

Les efforts de programmation peuvent être supportés au besoin par des personnes différentes de celles qui réaliseront l'observateur, ceci afin d'assurer que les erreurs de conception seront détectées par un comportement différent de l'exécutant et de l'observateur.

La conception de l'exécutant dépend de la structure du système, du type de l'application, des outils de développement et des langages de programmation disponibles.

Ainsi, dans le cas d'un système distribué sur une ligne série, on peut concevoir un exécutant composé suivant les différents niveaux de protocole définis par la norme ISO [22] : niveau physique, ligne, transport,... et utilisateur ; les processus correspondants seront, par exemple, écrits en langage ASSEMBLEUR, PASCAL, PLM (Intel),...

II.2. CONCEPTION DE L'OBSERVATEUR

L'observateur est composé de deux parties : le diagramme de mission décrivant à un certain niveau de détail les spécifications du système global et le programme permettant de faire évoluer ce modèle.

Un aspect important dans la conception de l'observateur est donc le choix d'un modèle puisqu'il est la base de l'approche.

Parmi tous les choix possibles, énumérés au chapître précédent, les réseaux de Petri semblent être les plus aptes à satisfaire les propriétés requises par le modèle. En effet, ils permettent de spécifier de façon non ambiguë la synchronisation entre processus et le parallélisme. La représentation de certaines configurations de base permettant la synchronisation de processus concurrents (la signalisation, l'allocation de ressource et la mutuelle exclusion) peut être réalisée de façon systématique ; le passage de la représentation informelle à la représentation formelle en est ainsi grandement facilité .

En plus de leur potentialité de représentation de la communication, les réseaux de Petri peuvent être analysés de façon statique ; il est alors possible, à travers leurs propriétés analytiques, d'assurer que l'observateur a un nombre fini d'états, qu'il est sans blocage et que certaines propriétés spécifiques au système sont vérifiées (par recherche des invariants). Ainsi, une spécification incomplète ou erronée peut être détectée.

Leur analyse peut être exécutée automatiquement, en particulier par un Outil Graphique Interactif de VErification (OGIVE) développé au LAAS [23] [24]

Ainsi, l'utilisation des réseaux de Petri apporte deux avantages essentiels : les possibilités d'analyse du modèle et sa facilité d'emploi.

II.2.1. Les réseaux de Petri [15]-[25]

II.2.1.a. Définitions

Définition 1 : réseau de Petri.

Un réseau de Petri est un quintuplet (P, T, E, S, M_o) où :

* P est un ensemble fini de places ($P \neq \emptyset$) ; chaque place est représentée par un cercle. L'ensemble P caractérise l'état du système.

* T est un ensemble fini de transitions ($T \neq \emptyset$ et $P \cap T = \emptyset$) ; chaque transition est représentée par un trait et symbolise un changement d'état du système.

* α est la fonction d'entrée des transitions définie par

$P \times T \xrightarrow{\alpha} \mathbb{N}$ et telle que :

il existe un arc liant la place p_i à la transition t_k si et seulement si $\alpha(p_i, t_k) = n_{ik} \neq 0$

L'arc correspondant est étiqueté avec la valeur n_{ik} (appelée poids de l'arc) si $n_{ik} > 1$

* β est la fonction de sortie des transitions définie par

$P \times T \xrightarrow{\beta} \mathbb{N}$ et telle que :

il existe un arc liant la transition t_l à la place p_j si et seulement si $\beta(p_j, t_l) = n_{jl} \neq 0$

De la même façon que pour α, l'arc est étiqueté avec n_{jl} si $n_{jl} > 1$

* M_o est le marquage initial tel que

$P \xrightarrow{M_o} \mathbb{N}$

Un marquage M est une fonction : $P \longrightarrow \mathbb{N}$ telle que à chaque place est associé un certain nombre $n : M(p) = n \geqslant 0$, symbolisé par la présence de n marques ou "jetons" dans la place p . Le marquage d'un réseau est la représentation de l'état global d'un système.

Définition 2 : règles d'évolution

a) Transition sensibilisée

Une transition t d'un réseau de Petri est sensibilisée ou tirable par un marquage M si et seulement si

$$\forall p \in P, \quad M(p) \geqslant \alpha(p,t)$$

Une transition sensibilisée peut alors être tirée, ce tir entrainant l'évolution du marquage.

b) Tir d'une transition t

Soit $M(p)$ le marquage qui sensibilise la transition t, le tir de la transition t conduit à un nouveau marquage $M'(p)$, symbolisant le nouvel état du système, tel que $\forall p \in P, \quad M'(p) = M(p) - \alpha(p,t) + \beta(p,t)$

Nous noterons : $M(p) \xrightarrow{t} M'(p)$

c) séquence de tir d'une transition

Soit σ une séquence finie de transitions de l'ensemble T $(t_1, t_2, \cdots, t_k)$. σ est une séquence de tir tirable à partir de M_i si et seulement si il existe des marquages $M_{i+1}, \ldots\ldots M_{i+k+1}$ tels que :

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

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

Définition 3 : graphe des marquages conséquents de M_0

a) classe des marquages conséquents de M_0

La classe des marquages conséquents de M_0, notée $\overrightarrow{M_0}$, est l'ensemble de tous les marquages M_i accessibles depuis M_0 , c'est-à-dire pour lesquels il existe au moins une séquence de tir σ_i tirable depuis M_0 et conduisant à M_i

$$M_i \in \overrightarrow{M_0} \Leftrightarrow \exists \sigma_i / M_0 \xrightarrow{\sigma_i} M_i$$

Exemple :

Soit le réseau représenté par la Figure II.1.

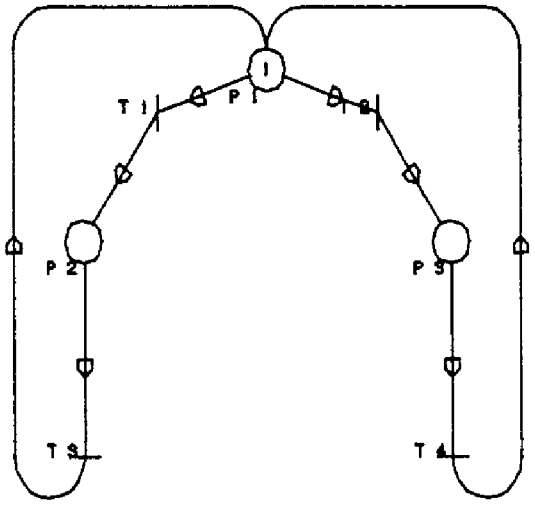


FIGURE II.1. Exemple

La classe des marquages conséquents de M_o = (100) est $\overrightarrow{M_o} = \{M_o, M_1, M_2\}$

avec M_1 = (0 10)

M_2 = (0 01)

Pour un autre marquage initial M'_o, la classe des marquages conséquents $\overrightarrow{M'_o}$ est différente.

$\overrightarrow{M_o}$ représentera en général un ensemble d'états du contrôle possibles pour un système.

b) graphe des marquages conséquents de M_o

Le graphe des marquages conséquents de M_o, noté G $(\overrightarrow{M_o})$, est le graphe orienté $(\overrightarrow{M_o}, A)$ tel que :

. $\overrightarrow{M_o}$ est l'ensemble des sommets

. A est l'ensemble des arcs (M_i, M_j) défini par

$$M_i \in \overrightarrow{M_o},\ M_j \in M_o,\ \exists t \ /\ M_i \xrightarrow{t} M_j \Leftrightarrow (M_i, M_j) \in A$$

Chaque arc est étiqueté par la transition tirée t.

Le graphe des marquages conséquents du réseau de la figure II.1. est représenté figure II.2.

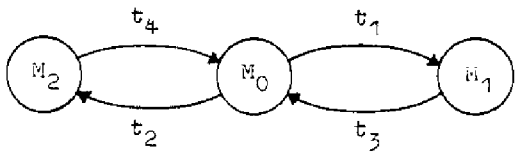


FIGURE II.2. Graphe des marquages conséquents de M_o

Après avoir donné les définitions générales des réseaux de Petri, nous allons rappeler quelques propriétés qui sont utilisées pour la validation des systèmes parallèles. Nous pouvons distinguer les propriétés qui dépendent du fonctionnement du système, les propriétés dynamiques,de celles qui ne dépendent que de la structure du réseau représentant le système, les propriétés structurelles.

II.2.1.b. *Propriétés dynamiques*

Ces propriétés sont définies en fonction d'un marquage initial M_o et les seules méthodes de vérification disponibles nécessitent toutes l'élaboration du graphe des marquages conséquents de M_o. Elles concernent le fonctionnement du réseau et ont été définies de telle sorte que leur vérification induise un comportement correct du système.

Propriété 1 : réseau borné - réseau sauf

Un réseau de Petri est borné par n pour un marquage initial M_o si et seulement si

$$\forall p \in P,\ \exists n \in \mathbb{N} / \forall M \in \vec{M_o},\ M(p) \leqslant n$$

Un réseau de Petri non borné indique, en général, une erreur de spécification ou de description du système modélisé, par exemple, une accumulation de jetons dans une place peut impliquer un débordement de capacité d'une ressource et sera donc à éviter.

Un réseau de Petri est sauf pour un marquage initial si et seulement si $\forall p \in P, \quad n = 1$

Le réseau de la figure II.1. est sauf pour M_o; pour tous les marquages de $\overrightarrow{M_o}$, chacune des places est bornée par n =1.

Par contre, pour M'_o = (1 0 1), ce réseau est non sauf, il est borné par n=2.

Le graphe des marquages conséquents de M'_o est alors (figure II.3).

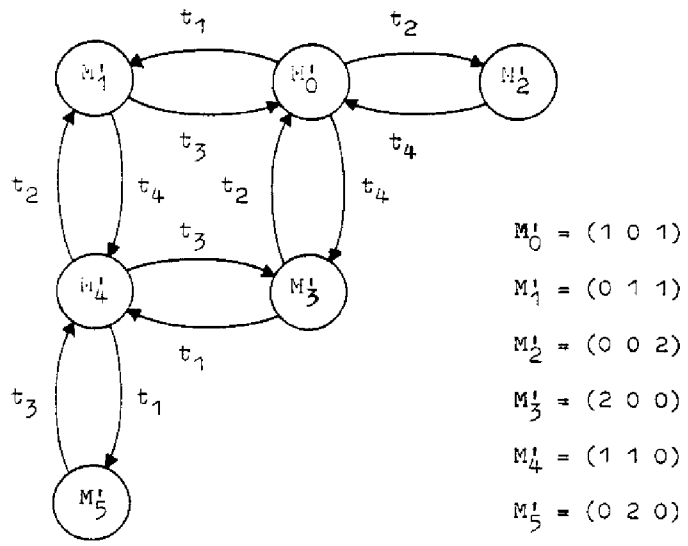


FIGURE II.3. Graphe des marquages conséquents de M'_o

Propriété 2 : réseau réinitialisable.

Un réseau de Petri est réinitialisable pour M_o si et seulement si

$$\forall M \in \overrightarrow{M_o}, \quad \exists \sigma / M \xrightarrow{\sigma} M_o$$

Cette propriété traduit la faculté pour le système modélisé par le réseau de Petri de revenir dans son état initial à partir de n'importe quel état.

Propriété 3 : Réseau vivant.

Un réseau de Petri est vivant pour M_o si et seulement si

$$\forall t \in T, \forall M_i \in \overrightarrow{M_o}, \exists \sigma, M_j / M_i \xrightarrow{\sigma} M_j \wedge \sigma \supset t$$

Ainsi, dans un réseau vivant, toute transition peut être tirable après une séquence de tir partant de n'importe quel marquage du graphe des marquages conséquents de M_o et donc toute partie du système modélisé par le réseau de Petri est accessible.

Cette propriété permet de vérifier que le système représenté est sans blocage. Un blocage dans un système à processus parallèles est une situation dans laquelle une partie du système n'est plus activable, par exemple en attente d'un message qui n'est jamais reçu ; dans le réseau correspondant, il existe alors un marquage à partir duquel certaines transitions ne peuvent plus être sensibilisées. Cette défaillance du système est à éviter lors de la conception.

II.2.1.c. Propriétés structurelles

Ces propriétés sont définies indépendamment du marquage et en particulier indépendamment du marquage initial. Elles fournissent des relations d'invariance permettant de vérifier des propriétés spécifiques au réseau étudié. Les assertions ainsi énoncées sont valides pour chaque marquage du réseau, c'est-à-dire pour toute évaluation du système. De plus, elles permettent de retrouver les propriétés dynamiques du réseau.

Propriété 1 : invariant de places.

Un invariant de places I est un vecteur de dimension

NP (NP : nombre de places du réseau) tel que

$$\forall M,\ M \cdot I = M_o \cdot I \qquad M : \text{marquage courant}$$

Il traduit des propriétés spécifiques au système.

Propriété 2 : invariant de transitions.

Un invariant de transitions est l'élément duál de l'invariant de places.

Il nous fournit des séquences cycliques de tir du réseau, mais pas toujours tirables à partir du marquage initial.

Les propriétés structurelles permettent d'établir un diagnostic sur les propriétés dynamiques du réseau [23] [26] (borné, réinitialisable, vivant)

II.2.1.d Réseaux de Petri étiquetés

Un réseau de Petri étiqueté est un réseau de Petri dans lequel à chaque transition est associée une étiquette.

Soit (q_ρ | O_ρ) l'étiquette associée à la transition t_ρ , elle est telle que :

* q_ρ est une condition (un prédicat)

* O_ρ est une liste d'actions.

Lorsque q_ρ est toujours vraie, elle est omise, si O_ρ est vide, elle est omise.

Pour pouvoir être tirée, la transition étiquetée doit être sensibilisée et validée, c'est à dire que la condition associée q_ρ doit être vraie.

Le tir d'une transition provoque l'exécution de l'action puis l'évolution des marquages.

L'analyse d'un réseau de Petri étiqueté est effectuée en analysant d'abord le réseau de Petri non étiqueté puis en prenant en compte les prédicats et les actions.

II.2.2. Modélisation du diagramme de mission

Après avoir choisi un modèle permettant de représenter les spécifications du système, nous nous attachons maintenant à donner une première étape dans la modélisation du diagramme de mission.

Dans le contexte de systèmes distribués, après la phase de spécification informelle de l'ensemble du système, le concepteur peut décider de modéliser séparément les processus et les communications interprocessus [21].

D'une part, chaque processus peut être considéré comme une entité du système ; il est alors modélisé par un réseau de Petri dont les transitions sont étiquetées soit par des actions internes au processus, soit par des actions externes (mécanisme de synchronisation, émission ou réception de messages circulant sur le support de communication). Dans le cas d'actions externes, les étiquettes indiquent les liens à effectuer entre les processus [27][28] ; deux transitions de processus différents seront couplées si un protocole (ou, plus simplement, un mécanisme) de communication existe entre elles.

Pour deux processus A et B communiquant pour des échanges d'informations ou de synchronisation, la transition correspondant à l'envoi d'un message α à un processus B est étiquetée par B! α ; celle correspondant à l'attente d'un message α envoyé par le processus A est étiquetée par A?α (Figure II.4.).

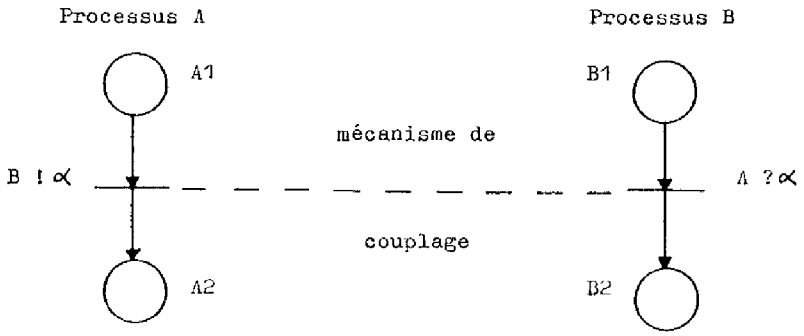


FIGURE II.4.

D'autre part, les mécanismes de synchronisation et/ou les communications sont également modélisés par des réseaux de Petri. Ainsi, le mécanisme de couplage entre un processus émetteur et un processus récepteur (!-?) d'un message devra être explicité [28] selon la réalisation effectuée.

Dans le cas qui nous intéresse ici, il pourra être très simple et représenter par exemple l'émission-réception simultanée qui correspond au niveau global à la fusion des transitions couplées (Figure II.5.a.) ou décrire explicitement le transit du message dans le support de communication par l'existence d'une place partagée qui, si elle est marquée, indique la présence d'un message circulant entre l'émetteur et le récepteur (Figure II.5.b.).

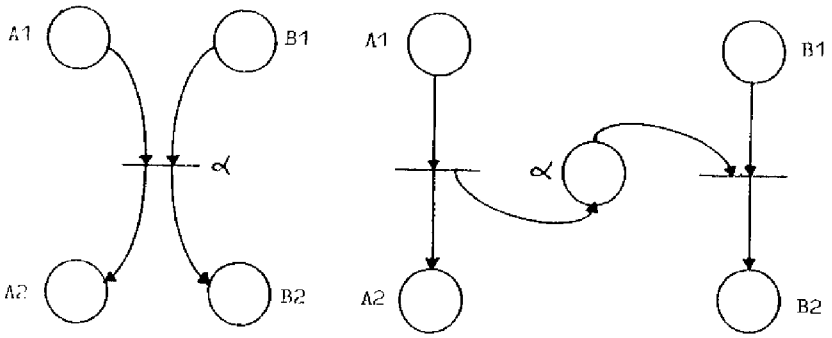


FIGURE II.5.

Remarque : Le mécanisme de couplage par appel-réponse et sa modélisation ne sont pas envisagés dans cette étude car le mode de fonctionnement ainsi décrit n'est pas généralisable pour tout protocole et ne nous permettrait pas d'établir une méthodologie d'obtention du diagramme de mission.

Ainsi, le choix d'un outil homogène pour représenter à la fois les processus et leurs interactions permet de réunir de façon systématique dans un seul modèle l'ensemble du système. L'outil interactif OGIVE est en cours d'extension afin de permettre de fusionner certaines places significatives des différents réseaux de Petri modélisant les processus et les communications ; ceci conduira à la création automatique du réseau représentant le fonctionnement global d'un système distribué. Il est alors possible de vérifier la cohérence des spécifications et certaines propriétés structurelles du système.

La modélisation du système global est donc effectuée en trois phases :

a) modélisation des différents processeurs communicants par des réseaux de Petri étiquetés, les étiquettes explicitant en particulier les envois et les attentes de messages,
b) construction, de façon explicite, des interactions entre les processeurs émetteur et récepteur d'un message selon le mécanisme de couplage choisi,
c) analyse du réseau global.

La première phase constitue la base de l'implémentation de l'observateur.

Le diagramme de mission intermédiaire dérivé de cette phase est obtenu en juxtaposant dans un seul réseau tous les réseaux modélisant un processeur de l'exécutant ; les fusions de ces différents réseaux suivant le protocole de communication choisi ne sont pas réalisées. Ainsi, l'ensemble des places (respectivement l'ensemble des transitions) du diagramme de mission est la réunion de l'ensemble des places (respectivement l'ensemble des transitions) de chaque réseau.

De cette façon, le diagramme de mission obtenu donc par juxtaposition des réseaux de l'exécutant contient une modélisation explicite des différents processeurs du système.

Bien que les liens entre les processeurs (réseaux) ne soient pas représentés, une modélisation particulière des réceptions de message, présentée plus loin (paragraphe II.3.) permettra de connaître le mode d'évolution des processeurs dans l'observateur : émission-réception simultanée ou non.

Le modèle global du système obtenu dans la seconde phase sera validé dans le troisième. S'il est trouvé logiquement correct, alors le fonctionnement de l'observateur correspondant au diagramme de mission sera, lui aussi, logiquement correct.

Il faut noter que la modélisation obtenue dépend à la fois des contraintes de synchronisation et du type de fautes impliquées par ces contraintes; par exemple, dans le cas de la synchronisation par envoi d'un message, la perte probable de ce message par le processeur destinataire doit être envisagée et modélisée.

Dans un premier temps, la modélisation des différents processeurs ne tient pas compte de ces fautes possibles. Après l'analyse du réseau global du système sans panne, il faudra donc revenir à la phase a) pour introduire dans les réseaux, puis dans le réseau global et le diagramme de mission, les transitions modélisant les fonctionnements induits par ces fautes.

II.2.3. *Logiciel de l'observateur*

Nous nous intéressons maintenant à définir la tâche de l'observateur.

Le but de l'observateur est de vérifier que l'évolution de l'exécutant satisfait aux règles définies par le diagramme de mission.

Le diagramme de mission modélisé par un réseau de Petri décrit le contrôle à un niveau donné. Il représente un préordre sur les actions de l'exécutant et en conséquence, il exprime de manière très compacte l'ensemble de toutes les séquences légales de l'exécutant ; ceci car le concepteur fait correspondre aux transitions du réseau certaines exécutions de l'exécutant à un niveau donné. Ainsi, le système présentera un comportement adéquat si ses propres séquences de calcul sont en accord avec

les règles de séquencement spécifiées par le réseau. Dans le cas d'un observateur implanté sur un processeur à l'écoute du support de communication, une erreur est détectée dès qu'il capte un message de l'exécutant ne correspondant pas à une transition sensibilisée.

II.2.3.a. Simulateur de réseaux de Petri

La tâche de l'observation est de faire évoluer le modèle en fonction des événements observés ; elle consiste donc en une simulation du réseau de Petri en effectuant les étapes suivantes :

1) après l'acquisition d'un événement de l'exécutant, l'observateur vérifie que la transition du diagramme de mission correspondant à cet événement est sensibilisée,
2) si elle est sensibilisée, il exécute le tir de la transition et fait donc évoluer le marquage,
3) si elle n'est pas sensibilisée, il active une procédure d'erreur,
4) retour à l'étape 1.

L'algorithme correspondant est représenté Figure II.6.

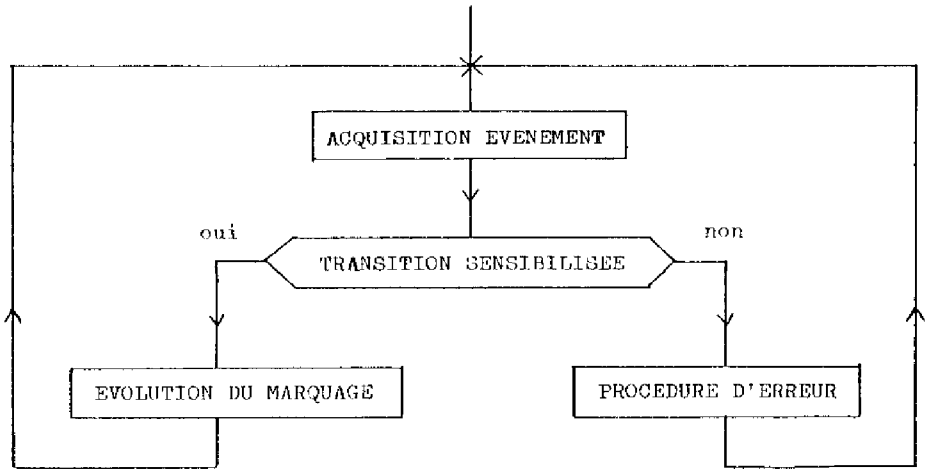


FIGURE II.6. Simulateur de réseaux de Petri

Le coeur du simulateur consiste donc en un bloc de programme vérifiant que la transition est tirable et en un bloc faisant évoluer le marquage si la transition est tirable.

Ceci peut être implémenté en PL/M par le logiciel suivant :

```
TEST=FAUX; /*variable  de test */

/*NP :nombre de places                 */
/*MARQUAGE (NP) :marquage courant */
/*ALPHA (NP) :vecteur d'entree d'une transition  *
/*BETA  (NP) :vecteur de sortie d'une transition *

/* transition TI sensibilisee ? */
J=0;
DO WHILE (J(NP) AND (MARQUAGE(J) >= ALPHA(J))) ;
    J=J+1;
END;
IF J=NP THEN
DO; /*transition sensibilisee */
    N=DPAR;
    TEST=VRAI;
END;

IF TEST=VRAI THEN DO;
/* il existe une transition sensibilisee
   correspondant au message M */
    DO J=0 TO NP-1;
      MARQUAGE(J) = MARQUAGE(J)-ALPHA(J)+BETA(J);
    END;

    END;
ELSE DO;
/* il n'existe pas de transition sensibilisee
   correspondant au message M */
    CALL ERREUR;
    END;
```

Nous voyons ici un intérêt supplémentaire à l'utilisation, dans le logiciel redondant, des réseaux de Petri pour modéliser le diagramme de mission. Les blocs implémentant le simulateur de réseaux de Petri sont extrêmement simples puisqu'ils ne réalisent que des comparaisons, des sommes et des différences de deux vecteurs. Ils sont, de plus, indépendants du réseau de Petri à simuler.

Ainsi, le coeur du simulateur peut être rendu très fiable.

Pour exécuter le simulateur de réseaux de Petri, l'observateur a besoin d'une base de données.

II.2.3.b. *Base de données*

La base de données sur laquelle va s'effectuer cette simulation est constituée de la description du réseau de Petri représentant le diagramme de mission et des marquages courant et initial du réseau.

La description du réseau de Petri est celle fournie par l'outil de dessin et de validation des réseaux de Petri (OGIVE) et utilisée pour la validation du diagramme de mission, ou bien est obtenue par transformation simple de la base de données fournie. La base de données du simulateur est donc elle aussi extrêmement fiable.

Ainsi, la base du logiciel de l'observateur (simulateur-structure de données) est extrêmement fiable : l'introduction de l'observateur dans le système à tester ne modifie pas la sûreté de fonctionnement.

Remarque : Lorsque les événements observables sont les messages transitant sur le support de communication du système, une ou plusieurs transitions peuvent être associées à un message. En effet, lors du fonctionnement d'un processeur, un même message peut être reçu à partir de plusieurs états différents. Il faudra tenir compte de cette remarque lors de l'implémentation réelle de l'observateur dans un système distribué.

II.3. CONNEXION OBSERVATEUR-EXECUTANT DANS UN CONTEXTE DISTRIBUE

II.3.1. Problèmes inhérents à la distribution

Afin de réaliser son diagnostic, l'observateur a besoin d'acquérir des informations observables.

Il peut alors :

- soit coopérer avec l'exécutant par insertion de points de test dans l'exécutant, cas non encore développé,
- soit contrôler, de façon invisible, l'exécutant.

Comme il a été expliqué au chapitre précédent (§ I.2.2.c.B.), c'est le deuxième cas qui est développé dans cette étude, le schéma d'implantation étant celui donné Figure I.5. ; l'exécutant est implémenté sur les processeurs connectés au support de communication ; l'observateur, implémenté sur un autre processeur, est à l'écoute du support de communication, les informations observables sont les messages circulant entre les différents processeurs.

D'où la règle d'évolution de l'observateur :

A chaque message α transmis correspond, dans l'exécutant, au moins un couple émetteur-récepteur de ce message ($!\alpha$ - $?\alpha$) et donc, dans le diagramme de mission, il existe au moins deux transitions étiquetées $!\alpha$ et $?\alpha$ que le simulateur va essayer de tirer simultanément si cela est possible. Comme nous le verrons par la suite, ces étiquettes n'auront peut-être pas la signification usuelle d'envoi et de réception de message.

Dans un contexte distribué, le fonctionnement de base du système est le suivant :

Lorsqu'un message fonctionnel à observer transite du processeur i vers le processeur j, l'observateur capte ce message et fait évoluer, si le message est en accord avec son état propre, les états respectifs des processeurs i et j dans le modèle.

Soulignons que, dans l'exécutant, l'état du processeur i évolue dès l'envoi du message, celui du processeur j n'évoluera qu'après la réception du message.

Nous allons maintenant approfondir l'influence de la modélisation des processus émetteur et récepteur sur l'évolution du diagramme de mission.

A. Lorsque l'émission-réception est considérée comme simultanée (Figure II.5.a.), le modèle global de l'exécutant est obtenu par fusion des deux transitions $!\alpha$ et $?\alpha$; l'observateur devra alors faire évoluer simultanément les états respectifs des processus émetteur et destinataire, l'évolution supposée de ces deux processus est celle décrite par l'observateur.

B. Par contre, lorsque le mécanisme de couplage est plus sophistiqué : fusion par une place partagée représentant un événement ou un protocole abstrait de communication (Figure II.5.b.), les processus évoluent de façon concurrente, ils sont uniquement synchronisés par le mécanisme décrit. Ainsi, à partir des états courants de chaque processeur caractérisés par les marquages $M_{li} = (1,0)$ et $M_{lj} = (1,0)$, (Figure II.7.), deux évolutions possibles sont à envisager (Figure II.8.a. et Figure II.8.b.).

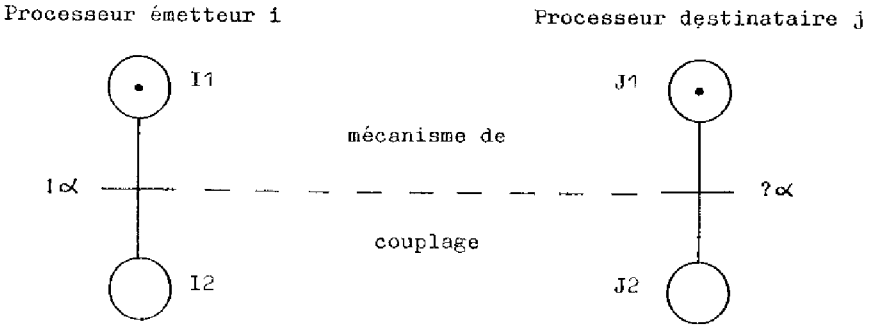


FIGURE II.7.

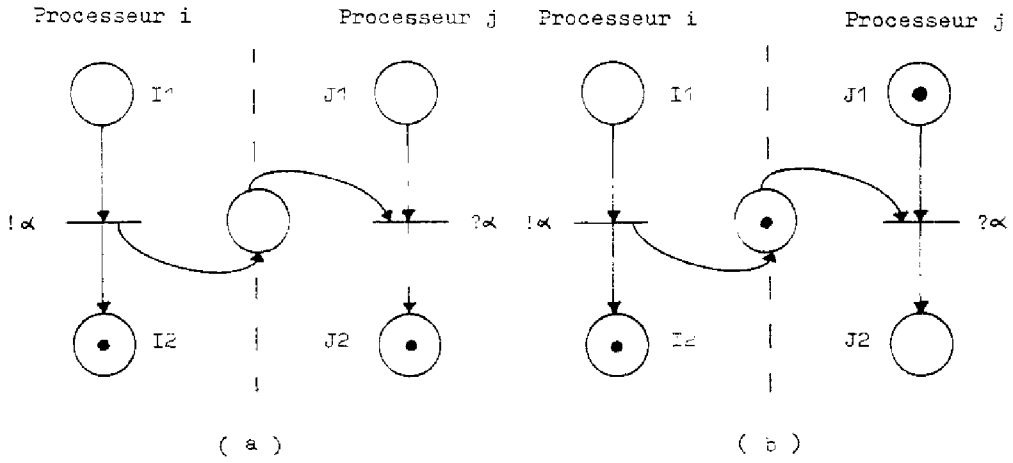


FIGURE II.8. Deux évolutions possibles

Le cas a) indique que la communication entre i et j a été correctement effectuée, le cas b) que la communication a été initialisée mais ne s'est pas terminée, l'état du processeur destinataire n'a pas évolué.

Notons que, quelle que soit l'évolution du processeur destinataire, le processeur émetteur passe toujours de l'état I1 (prêt à envoyer) à l'état I2 dans lequel le message a été envoyé.

L'observateur introduit étant à l'écoute du support de communication, son comportement vis-à-vis d'une communication entre deux processeurs doit prendre en considération le fait que l'état du processeur émetteur a forcément évolué lorsque le message est reçu par l'observateur, alors que l'évolution du destinataire n'est pas connue car, a priori, elle ne peut être déduite directement du passage du message sur le medium ; en effet, par exemple, un message peut être perdu entre sa réception par l'observateur et par le récepteur.

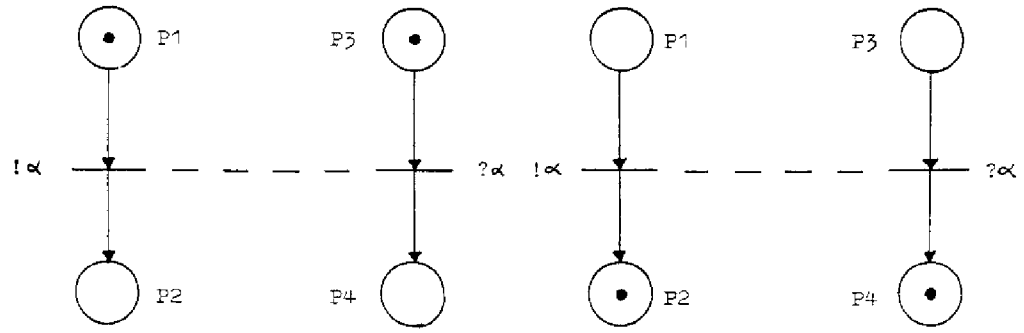
Deux possibilités existent alors pour l'observateur : il peut faire évoluer simultanément ou non les états respectifs des processeurs communicants induisant par là-même un facteur d'anticipation ou de retard sur l'état réel des processeurs. Notons que ce point extrêmement important découle de l'impossibilité de connaître l'état instantané d'un système distribué. Ceci nous a amené à concevoir deux modélisations possibles du diagramme de mission représentant ces deux modes d'évolution.

D'après les remarques précédentes, la partie du modèle correspondant au processeur émetteur est identique dans les deux cas, seule la partie correspondant au processeur destinataire est différente.

1. Dans le premier modèle (Figure II.9.), les étiquettes $!\alpha$ et $?\alpha$ associées aux transitions expriment réellement l'envoi et la réception du message α . En utilisant la règle d'évolution, ce modèle conduit à une anticipation de l'état de l'exécutant.

En effet, toute réception par l'observateur du message α fait évoluer le marquage courant $M_1 = (1,0,1,0)$ (Figure II.9.a.) vers le marquage $M_2 = (0,1,0,1)$ (Figure II.9.b.). Ainsi, quel que soit l'état réel du processeur destinataire dans l'exécutant (état J1 ou J2), on considère que l'état de ce processeur est l'état dans lequel le message α a été reçu : on préjuge ainsi de l'état réel du processeur. Il faudra alors tenir compte de ce fait nouveau dans la modélisation du diagramme de mission ; par exemple, à partir de la place P4, le modèle devra représenter le fonctionnement du processeur sans panne et celui du processeur avec perte du message α (ce problème sera développé dans le chapitre suivant).

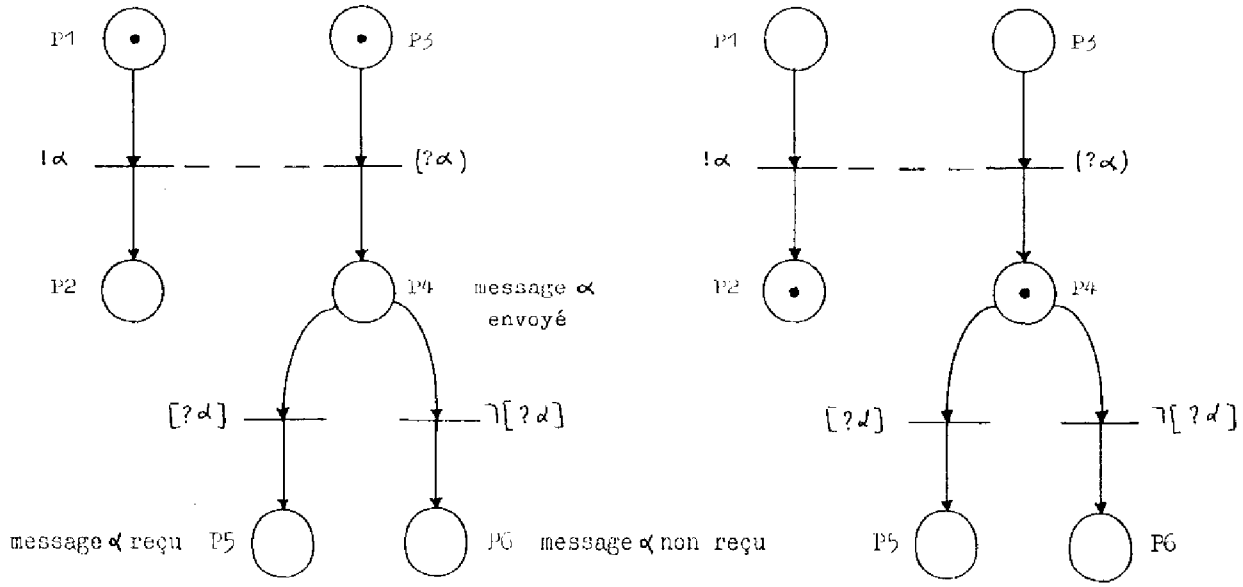
2. Dans le deuxième modèle, il sera nécessaire de distinguer entre la réception par l'observateur de α , notée $(?\alpha)$ et la confirmation ou la non-confirmation de la réception noté $[?\alpha]$ ou $\neg[?\alpha]$ sur la Figure II.10.



a) avant réception de α par l'observateur

b) après réception de α par l'observateur

FIGURE II.9. Anticipation de l'état de l'exécutant



a) avant réception de α par l'observateur

b) après réception de α par l'observateur

FIGURE II.10. Retard sur l'état de l'exécutant

L'utilisation de la règle d'évolution conduit du marquage courant $M'_1 = (1,0,1,0,0,0)$ (Figure II.10.a.) au marquage $M'_2 = (0,1,0,1,0,0)$ (Figure II.10.b.), après réception du message α par l'observateur. Mais, dans ce modèle, le tir de la transition étiquetée $(?\alpha)$ n'exprime pas la réception du message α, il exprime le fait que le message α a été envoyé par le processeur émetteur, ce dont on est sûr. L'observateur ne saura si le message a été reçu correctement, ou non, qu'après avoir capté le message suivant, et donc après avoir tiré soit la transition $[?\alpha]$ soit la transition $\neg[?\alpha]$ lui permettant de sensibiliser la transition correspondant au message suivant circulant dans le système.

Cette modélisation met l'accent sur la nature distribuée du système et sur la décomposition de la communication entre processeurs en deux actions distinctes : l'émission d'un message puis sa réception. Ainsi, on semble faire évoluer les états respectifs des deux processeurs communicants de façon séparée et il existe alors un facteur de retard sur l'état de l'exécutant, alors que dans la première méthode, l'évolution des deux processeurs semble simultanée.

II.3.2. *Comparaison de ces deux modélisations et choix*

Dans le cas où l'émission-réception d'un message est modélisée par une place partagée dans l'exécutant, nous avons vu qu'il pouvait y avoir deux modélisations possibles, dans l'observateur, du couple émetteur-récepteur conduisant à deux évolutions possibles des processeurs communicants. Nous nous proposons de comparer ces deux modélisations et de faire le choix de l'une d'entre elles.

Dans les deux cas, le séquencement des messages est identique.

Dans la modélisation par anticipation de l'état de l'exécutant, (Figure II.9.) à partir de la place P4 où le message α est supposé être reçu par le récepteur, les transitions de sortie de cette place doivent représenter aussi bien le fonctionnement du processeur avec réception effective du message que celui avec perte du message dans le support de communication. Ainsi, toute place du réseau ne représente pas un état réel du processeur.

Dans la modélisation induisant un facteur de retard sur l'état réel du système (Figure II.10.), chaque place a une signification propre (message envoyé, message reçu, message non reçu) ; toute évolution du réseau conduit à un état réel du processeur correspondant. En contrepartie, toutes les transitions du réseau de Petri n'ont pas la même signification. Les transitions étiquetées ($?\alpha$) correspondent à la réception par l'observateur d'un message circulant sur le support de communication ; les transitions étiquetées $[?\alpha]$ et $\neg[?\alpha]$ ne correspondent pas à un message, elles permettent de confirmer ou infirmer la réception du message α . La modélisation et le logiciel de l'observateur sont alors rendus plus complexes. En effet, en plus du simulateur de réseaux de Petri, le logiciel doit comporter une procédure qui, pour pouvoir faire évoluer le processeur récepteur à son état suivant, doit déterminer, à partir du nouveau message capté, si le message α a été effectivement reçu ou non, c'est-à-dire si c'est la transition $[?\alpha]$ ou $\neg[?\alpha]$ qui doit être tirée pour pouvoir sensibiliser la transition correspondant au nouveau message capté. Cette procédure devrait être exécutée après chaque message capté et alourdirait considérablement la tâche de l'observateur. La structure de données dans ce cas est également beaucoup plus volumineuse.

De plus, puisque le but de l'observateur est de vérifier le séquencement des actions de l'exécutant et non de déterminer les états de chacun des processeurs composants l'exécutant, la première modélisation paraît mieux adaptée car plus simple et suffisante.

Ainsi est retenue la modélisation mettant en jeu un facteur d'anticipation sur l'état de l'exécutant et faisant évoluer simultanément l'émetteur et le destinataire d'une communication.

II.4. CONCLUSION

Ce chapitre, après avoir décrit les logiciels de l'exécutant et de l'observateur, a ébauché la méthode d'obtention du diagramme de mission qui va être développée au chapitre suivant.

Il a, de plus, mis l'accent sur l'évolution concurrente des processeurs émetteur et récepteur d'un message dans un système distribué et sur l'implication sur l'observateur de la non-connaissance de l'état global. Ces deux points devront être pris en considération de façon systématique lors de la modélisation, comme cela va être vu maintenant.

CHAPITRE III

MODÉLISATION DU DIAGRAMME DE MISSION

-

INTRODUCTION

Dans un système distribué où les fonctions à réaliser sont implantées sur différents sites, le seul lien physique entre les divers processeurs constituant le système global est le médium de communication.

Quelle que soit la nature du système de communication, il est indispensable de supposer qu'il n'est pas parfait, c'est-à-dire qu'il peut y avoir modification du contenu de l'information lors de la transmission et ainsi non-reconnaissance de l'information reçue en tant que message du système ; cela peut, par exemple, être dû à du bruit sur une ligne pendant la transmission ou à une panne matérielle dans le processeur émetteur... De plus, d'autres cas de pannes peuvent être envisagés (et sont réalistes) tels que : le processeur émetteur peut être "muet", le processur destinataire peut être "sourd" ou "non prêt" lors de l'arrivée d'un message.

Sans développer outre mesure les causes de ces pannes, on peut regrouper ces trois cas de pannes car ils ont le même effet apparent qui est la perte, par le destinataire, de l'information transmise.

Ainsi le concepteur d'un système distribué doit absolument tenir compte de cette perte possible lors de l'écriture du cahier des charges. De même, l'introduction d'une unité d'observation à l'écoute du système de communication dans le système distribué conduit à de nouvelles possibilités de perte de messages dont l'influence doit être modélisée dans le diagramme de mission.

Le choix des réseaux de Petri comme modèle de représentation du diagramme de mission de l'observateur permet de s'assurer que le modèle obtenu répond bien aux spécifications du cahier des charges.

Il nous a paru indispensable de définir une méthodologie d'obtention du diagramme de mission à partir des spécifications, de façon à prendre en compte systématiquement tous les cas possibles de pertes de messages dans le système. Ceci est décrit dans le paragraphe III.1.

Le paragraphe III.2. nous permet de présenter le système REBUS, système distribué de commande en temps réel ayant servi de support à notre étude,et de donner un exemple illustratif d'application de la méthodologie d'obtention du diagramme de mission.

III.1. METHODOLOGIE D'OBTENTION DU DIAGRAMME DE MISSION

Un modèle formel basé sur les réseaux de Petri peut être obtenu, de façon systématique, à partir du modèle informel que sont les spécifications, en utilisant la méthodologie proposée par [6] . Ce modèle décrit le fonctionnement général du système à un certain niveau de détails.

Le diagramme de mission de l'observateur définit, quant à lui, les mécanismes de synchronisation, les communications dans le système global.

Dans un premier temps, il sera obtenu à partir du modèle formel précédent.

Dans un deuxième temps, le concepteur devra tenir compte de la nature distribuée de l'application et introduire, lors de la modélisation des différents processeurs, des pannes ayant pour effet la perte de message.

Dans la structure distribuée étudiée, représentée figure III.1., tous les processeurs sont connectés au système de communication, le processeur observateur est en écoute permanente : il capte tout message émis par un processeur émetteur E vers un processeur destinataire D.

En conséquence, dans le cas général, deux processeurs doivent être en mesure de recevoir le message émis : le destinataire et l'observateur. La perte de ce message par l'un et/ou l'autre de ces processeurs conduit ainsi à différents fonctionnements erronés qui doivent être traités et dont l'influence doit apparaître lors de la modélisation du diagramme de mission :

* ni le destinataire D, ni l'observateur O ne reçoivent le message.

* l'observateur reçoit le message et D ne le reçoit pas, cas que nous avons abordé précédemment.

* le destinataire reçoit le message et O ne le reçoit pas, cas qui sera commenté plus tard et qui complique encore le problème.

Avant d'étudier l'influence de ces trois cas de pannes sur la modélisation du diagramme de mission ou sur le logiciel de l'observateur, nous allons d'abord nous attacher à obtenir le diagramme de mission dans le cas d'un système de communication parfait, c'est-à-dire ne perdant pas de message.

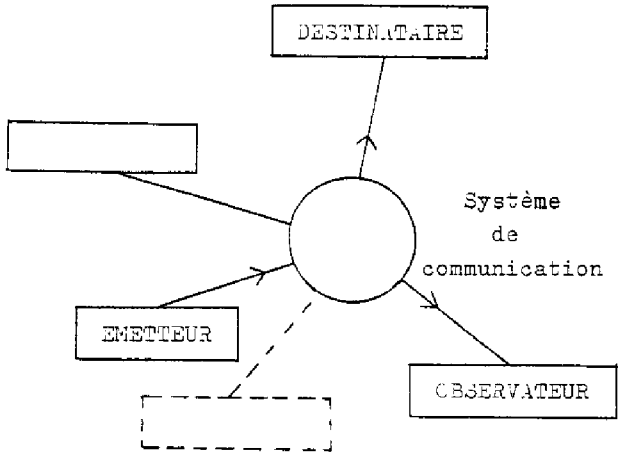


FIGURE III.1. Structure étudiée

III.1.1. *Système de communication parfait*

Comme il a été expliqué au paragraphe II.2.2., le diagramme de mission est obtenu à partir des spécifications formelles des différents processeurs du système global par juxtaposition des différents réseaux de Pétri, en ne prenant en compte que l'aspect communication interprocessus et en considérant un certain niveau de détails.

Nous allons expliciter plus en détails la méthode d'obtention du diagramme de mission à partir d'un exemple.

Soient deux processus A et B communicants modélisés par les réseaux de la figure III.2. représentant l'appel à distance (par A) d'une procédure dans B.

Les processus A et B communiquent par les messages "valeurs" et "résultats", le processus A (respectivement B) effectue l'action locale "traitement" (respectivement "calcul").

Selon la contrainte de synchronisation choisie entre les deux processeurs (émission-réception simultanée ou circulation d'un message), le réseau modélisant le fonctionnement global du système est celui de la figure III.3.a ou celui de la figure III.3.b. Néanmoins, quel que soit le mode de fusion des transitions, les deux messages "valeurs" et "résultats" transitent sur le support de communication dans l'ordre donné par le réseau global, c'est-à-dire "valeur, résultats".

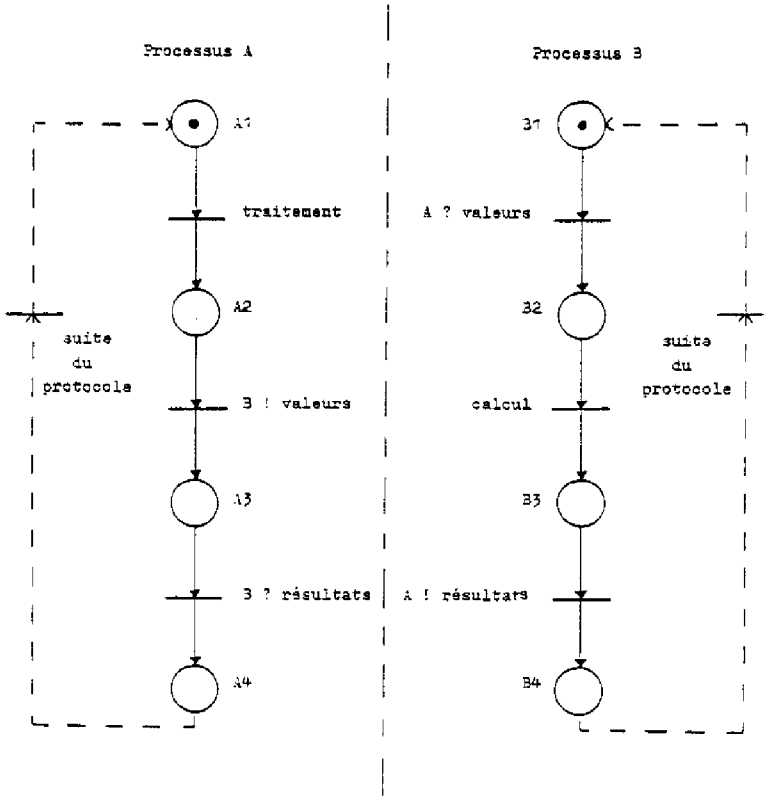
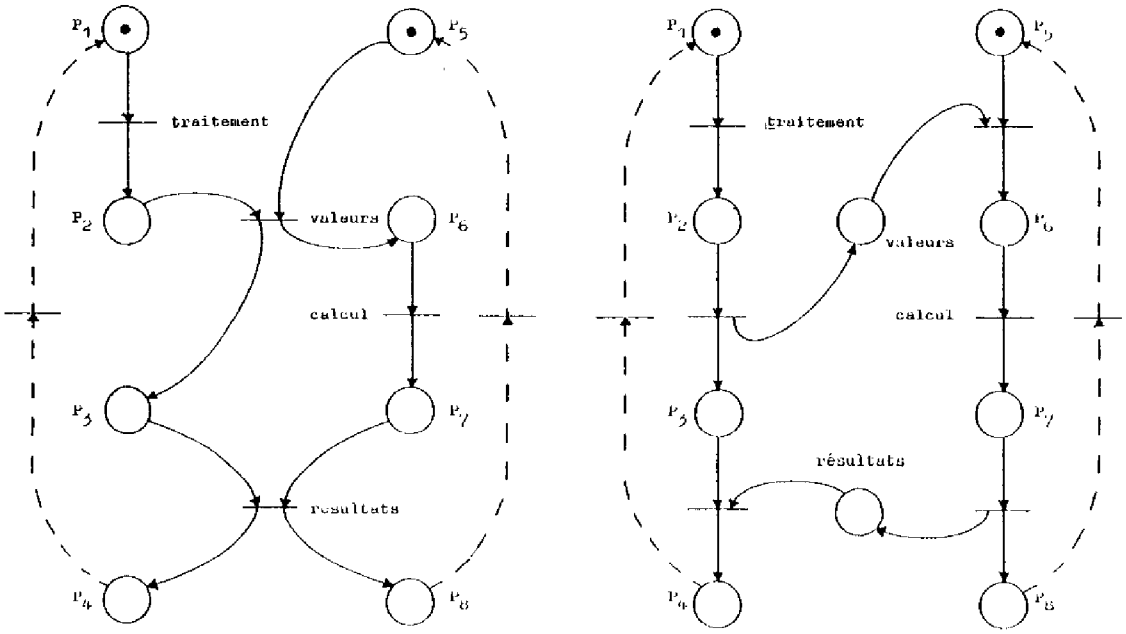


FIGURE III.2. Modélisation des deux processus A et B



a) émission-réception simultanée

b) place partagée

FIGURE III.3. Réseau global

Le diagramme de mission ne prenant pas en compte les fautes du système doit représenter le séquencement des messages. Il peut alors être décrit selon la figure III.4., en adoptant la modélisation permettant de faire évoluer simultanément les processeurs communicants et induisant par là même un facteur d'anticipation sur l'état de l'exécutant.

La représentation des actions locales est écartée au profit de la représentation des mécanismes de synchronisation et/ou des communications inter-processus.

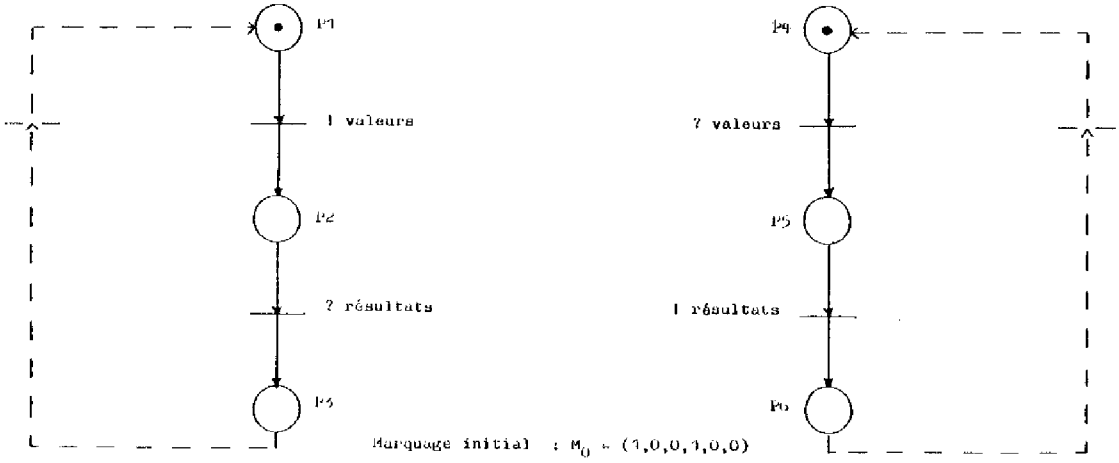


FIGURE III.4. Diagramme de mission dans un système parfait

III.1.2. Système de communication non parfait

1 - Nous allons maintenant introduire un type de panne dans ce système, la perte de messages dans le support de communication, et étudier son incidence sur la représentation du modèle de l'observateur.

Le protocole de communication entre les processus A et B doit alors être plus élaboré pour prendre en compte cette panne possible : dans les cas de non-réception du message "valeurs" par B ou du message "résultats" par A, le processeur A est en attente de "résultats". La reprise de la communication sera la même dans les deux cas et consistera généralement en une réémission du message "valeurs" par A vers B un nombre fini de fois.

Les réseaux simplifiés représentant uniquement les communications inter-processus dans l'exécutant avec reprise sur erreur sont alors donnés sur la figure III.5.

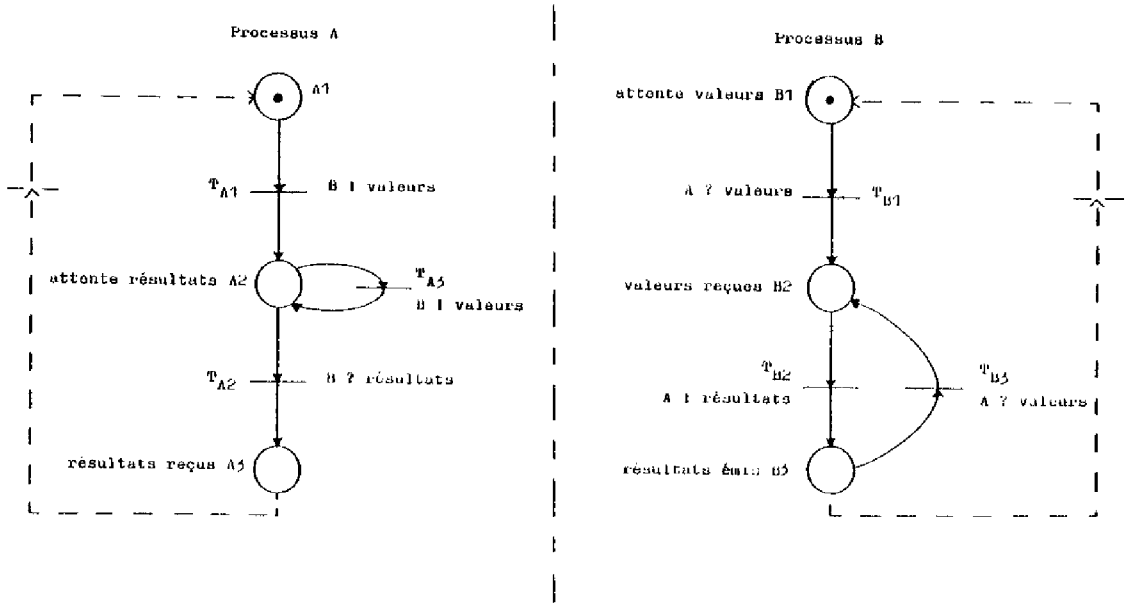


FIGURE III.5. L'exécutant avec reprise sur erreur

La transition TA3 modélise le fonctionnement suivant : après avoir envoyé "valeurs" à B, le processus A est dans l'état "attente résultats". Si au bout d'un certain temps, il ne reçoit pas le message "résultats" à cause d'une perte de "valeurs" ou de "résultats", il envoie à nouveau à B le message "valeurs". La transition TB3 indique, quant à elle, que le message "résultats" a été émis par le processeur B vers le processeur A, mais que, après la perte de ce message, le processeur A a décidé de réémettre le message "valeurs".

Dans un premier temps, un diagramme de mission intermédiaire de l'observateur modélisant l'exécutant avec reprise sur erreur est celui décrit figure III.6. ; il est déduit directement des réseaux figure III.5.

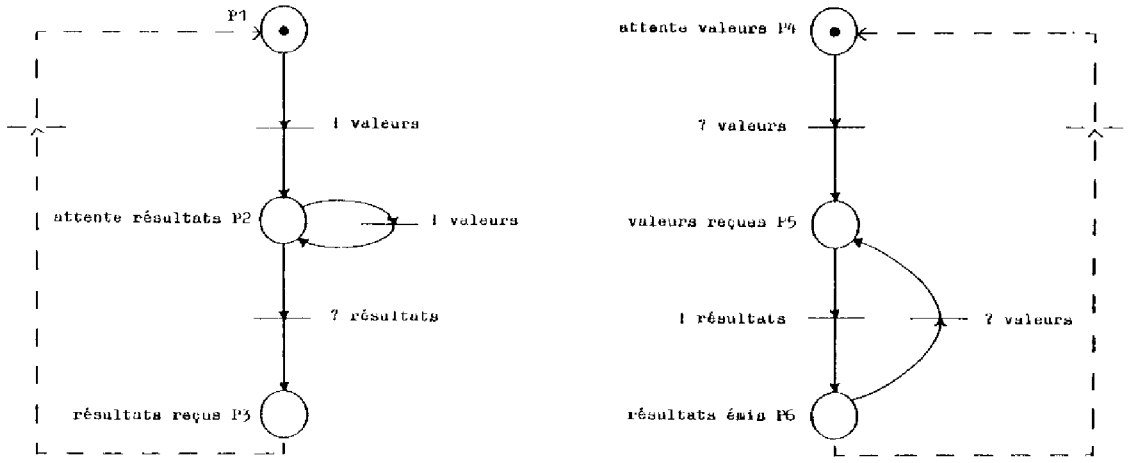


FIGURE III.6. Diagramme de mission intermédiaire

2 - Dans un deuxième temps, ce modèle doit être modifié pour tenir compte, en plus de la possibilité de perte dans le médium, des trois types de fonctionnements erronés décrits précédemment (la modélisation adoptée sera toujours celle induisant un facteur d'anticipation sur l'état de l'exécutant).

* Cas 1 : perte de message par un processeur fonctionnel.

Le premier type de faute résulte de la perte d'un message par un processeur fonctionnel.

Prenons, par exemple, le message "valeurs" envoyé par A vers B. L'observateur fait évoluer en conséquence les états respectifs de A et B si la modélisation des communications est l'émission-réception simultanée. L'observateur pense donc que A est dans l'état "attente résultats" et B dans l'état "valeurs reçues".

Supposons que "valeurs" soit reçu par l'observateur mais ne soit pas reçu par B ; la technique de reprise consiste en une réémission du message "valeurs" par A au bout d'un temps fini. La prise en compte de cette possibilité dans la modélisation du diagramme de mission doit donc se faire en considérant B dans l'état "valeurs reçues" et non dans l'état "attente valeurs" qui est l'état réel du processus B. Ceci se traduira par l'introduction de la transition T8 (figure III.7) dans le réseau de l'observateur.

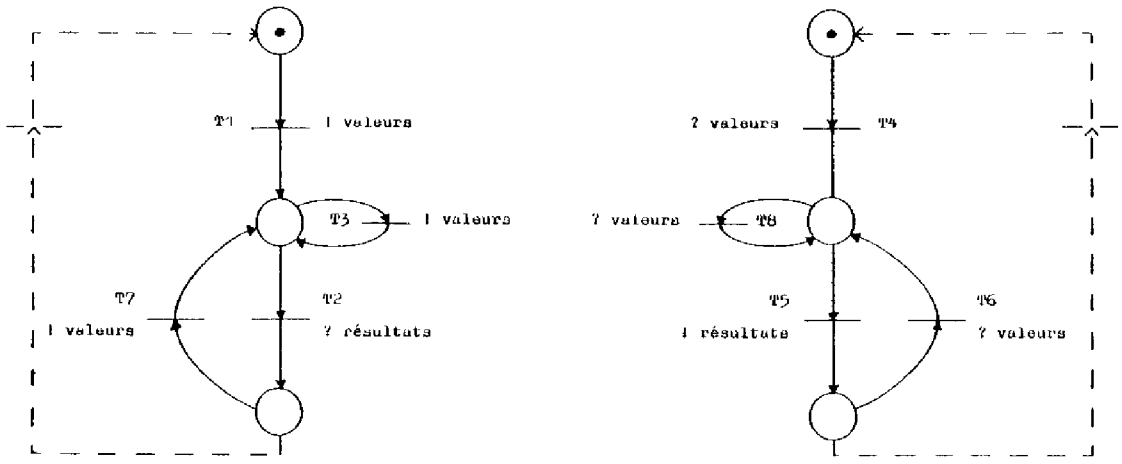


FIGURE III.7. Diagramme de mission final

De la même façon, le message "résultats" peut être perdu sur le support de la communication par le processeur destinataire A. Ne recevant pas de réponse à son message "valeurs", le processus A va émettre à nouveau le message "valeurs" au bout d'un certain temps.

Malgré tout, l'observateur aura capté le message "résultats" et fait évoluer l'état du processus A (respectivement B) vers "résultats reçus" (respectivement "résultats émis"). L'état du processus A récepteur de"résultats" n'est pas son état réel et dans ce cas, la représentation, sur le diagramme de mission, de la réémission du message "valeurs" par A est réalisée par la présence de la transition T7 sur la figure III-7.

A partir de cet exemple, nous voyons que l'introduction dans le fonctionnement du système de la perte de message par un processeur fonctionnel conduit à une amélioration du modèle servant de base à l'observateur par l'introduction de transitions (et de places)représentant la reprise sur erreur.

Il est à noter de plus que ce type de faute peut être pris en compte de façon systématique et donc que la modélisation de la reprise sur erreur peut être également systématique.

Ainsi, dans le modèle de l'observateur obtenu à partir de la représentation de l'exécutant avec reprise sur erreur, à chaque transition étiquetée ?α, le concepteur doit considérer deux cas :

. le message a été correctement reçu, le modèle est alors inchangé,

. le message n'a pas été reçu, le modèle doit être amélioré pour permettre la représentation de la reprise. C'est à partir de la place de sortie de la transition ?α que le concepteur modélise la reprise puisque l'observateur aura fait évoluer les états des processeurs émetteur et destinataire après la réception du message α .

* Cas 2 : perte de message par l'observateur.

Etudions maintenant l'influence de la perte de message par l'observateur sur le fonctionnement du système.

Il est bien évident que la non-réception d'un message par l'observateur doit être évitée avec le plus grand soin car elle résulterait d'une mauvaise conception de la partie réception de l'observateur.

Malgré tout, il se peut que le message ne puisse pas être reçu par l'observateur auquel cas l'observateur ne captant pas de message n'est pas activé bien que l'état des processus dans l'exécutant ait évolué.

Supposons que l'observateur dans l'état initial n'ait pas capté le message "valeurs" et que, dans l'exécutant, le message "valeurs" ait été correctement émis et reçu. L'observateur ne fait pas évoluer le modèle et reste dans l'état initial, alors que le processus A (respectivement B) a évolué vers l'état "attente résultats" (respectivement "valeurs reçues").L'exécutant fonctionnant sans panne, l'observateur va capter le message suivant c'est-à-dire "résultats". A partir de l'état initial le simulateur de réseaux de Petri ne trouvera pas de transitions tirables correspondant au message "résultats" et va donc déclarer le message "résultats" incorrect, diagnostic qui est erroné.

Afin d'éviter une telle erreur de diagnostic, le logiciel de l'observateur doit être amélioré ; nous proposons de lui permettre de rechercher un état cohérent à partir duquel le message "résultats" est correct. Ceci peut se faire en explorant les états qui suivent l'état actuel des processus (par rapport à l'observateur) et en essayant de tirer, à partir de ces états, une transition correspondant au message. S'il existe un état suivant pour lequel la transition est tirable, le message sera supposé correct, sinon il sera incorrect.

La partie test du logiciel de l'observateur est composée alors du simulateur de réseaux de Petri et d'une procédure de recherche des états suivant l'état acutel. L'algorithme de test est le suivant (figure III-8) :

1) Après l'acquisition du message, l'observateur teste s'il existe une transition sensibilisée correspondant au message,

2) S'il en existe une, il exécute le tir de la transition et fait donc évoluer le marquage,

3) S'il n'existe aucune transition sensibilisée correspondant au message, il lance la procédure de recherche des états suivant l'état actuel,

- si , à partir d'un de ces états, une transition correspondant au message est sensibilisée, l'observateur effectue le tir de cette transition, faisant évoluer le marquage,

- sinon, il active une procédure d'erreur.

4) retour à la première étape.

La détermination des états suivants de l'état actuel consiste à construire le graphe des marquages conséquents du marquage courant caractérisant l'état actuel. Avant de rechercher un état cohérent du système, il faudra faire une hypothèse sur le nombre maximum de messages consécutifs qu'il est raisonnable de supposer non captés par l'observateur. Ce nombre détermine la longueur maximale de la séquence de tir tirable à partir de l'état actuel et conduisant dans un état où la transition correspondant au message est tirable. L'absence de cette hypothèse conduirait à constuire à nouveau tout le graphe des marquages conséquents du marquage initial, obtenu lors de la phase d'analyse du modèle. Dans ce cas, l'analyse ayant prouvé le caractère vivant du réseau, toute transition pourraît être tirée, après une séquence de tir, à partir de tout marquage conséquent du marquage initial ; ainsi, tout message serait détecté correct et notre méthode s'avèrerait parfaitement inadéquate pour réaliser un diagnostic de message incorrect. Il apparaît alors que cette hypothèse doit être suffisamment restrictive pour permettre de détecter les erreurs, elle dépend en particulier du nombre de messages différents caractérisant le protocole observé et de l'hypothèse de pannes du système. La valeur sélectionnée sera fonction de la réalisation effective de l'observateur et du système lui-même.

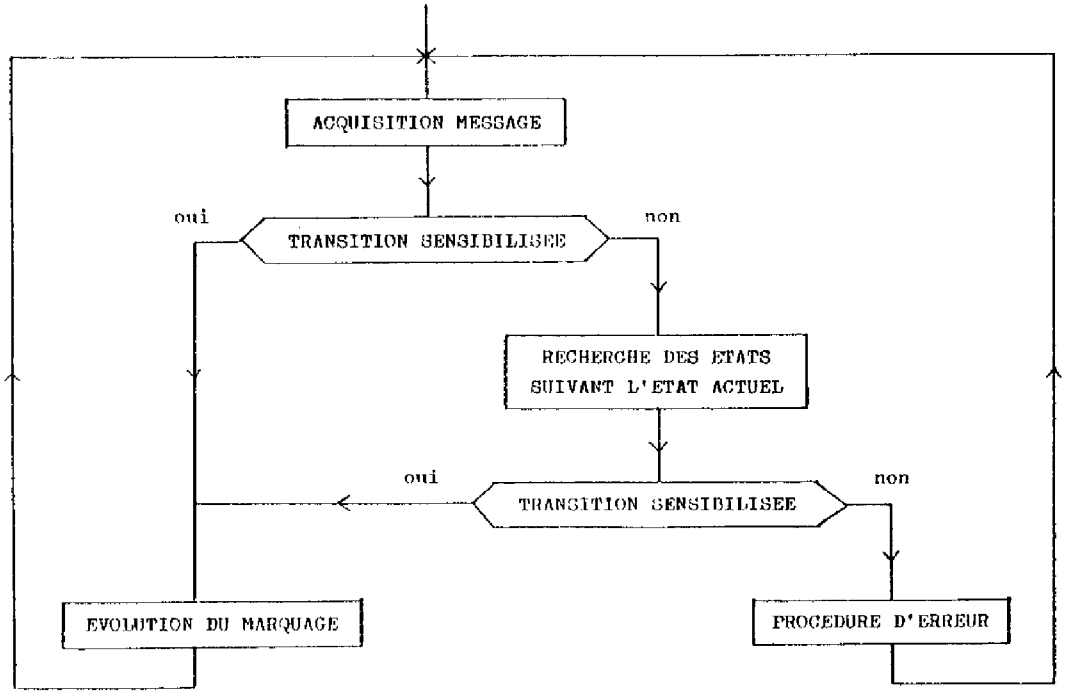


FIGURE III.8. Algorithme de test

Avec cette amélioration du logiciel, le diagnostic d'erreur est repoussé après la recherche d'un état cohérent permettant de prendre en compte ce cas de panne. Ainsi, le test d'un message dans l'observateur est affiné et conduit aux résultats suivants :

message correct <=> il existe une transition tirable à partir du marquage courant, correspondant au message capté.

message correct après une séquence de tir <=> il existe, à partir du marquage courant, une séquence de transitions tirables, de longueur maximale fixée, conduisant à un état à partir duquel une transition correspondant à ce message est sensibilisée.

message incorrect <=> il n'existe aucune séquence de tir, dans la limite que nous nous sommes fixés, permettant à partir du marquage courant de tirer la transition correspondant à ce message.

Remarque :

Nous avons montré les conséquences de la perte d'un message par l'observateur dans le cas d'un système ne comportant que deux processeurs communicants. Lorsque le système distribué est composé de plus de deux processeurs communicants, le perte d'un message par l'observateur rend la recherche d'un état cohérent beaucoup plus complexe. Tout en faisant évoluer simultanément, lorsque cela est possible, les états respectifs des deux processeurs en communication, la recherche d'un état cohérent doit être réalisée sur chaque processeur de façon indépendante.

Ce cas sera développé plus en détail dans le chapître IV.

* Cas 3 : perte de message par le destinataire et l'observateur.

Enfin, étudions le cas où le message n'est reçu ni par le destinataire, ni par l'observateur.

Dans ce cas,si à partir de l'état initial le message "valeurs" a été envoyé par A, ni l'observateur ni le destinataire B dans l'exécutant ne recevant ce message ne feront évoluer leurs états respectifs. L'observateur reste dans l'état initial ; dans l'exécutant, le processeur A est dans l'état "attente résultats" et le processeur B dans l'état "attente valeurs"(figure III-5). Ne recevant pas le message "résultats" en réponse au bout d'un certain temps, A va alors réémettre ce message "valeurs". L'observateur captant ce nouveau message tirera les transitions T1 pour l'émetteur et T4 pour le récepteur (figure III-7). Dans ce cas, ni l'observateur ni le destinataire n'ont été activés et la reprise de la communication a déjà été modélisée dans le diagramme mission lors d'une première étape (figure III-6).

En conséquence, la perte du message par le destinataire et l'observateur n'entraine aucune modification tant au niveau de la représentation du diagramme de mission qu'au niveau du logiciel de l'observateur.

III.1.3. Conclusion

Nous avons étudié à l'aide d'un exemple l'influence, sur le modèle et le logiciel de l'observateur, de pannes pouvant intervenir dans le système, pannes qui se traduisent par la perte de messages et donc par la non-réception de ces messages par un équipement à l'écoute sur le support de communication.

Nous pouvons en déduire une méthodologie d'obtention du diagramme de mission.

. Méthodologie

Tout d'abord, à partir du cahier des charges du système, on peut obtenir les réseaux de Petri modélisant les différents processeurs de l'exécutant dans le cas d'un fonctionnement normal. Après avoir choisi un modèle de communication inter-processeurs, le réseau global obtenu en fusionnant les différents réseaux est analysé afin de vérifier la cohérence globale des spécifications et de déduire les propriétés structurelles du système.

La représentation du réseau global ne se fera que dans des cas restreints où le modèle final n'est pas trop complexe. L'étape de fusion et celle d'analyse peuvent être réalisées de façon automatique par l'outil graphique interactif OGIVE [23]

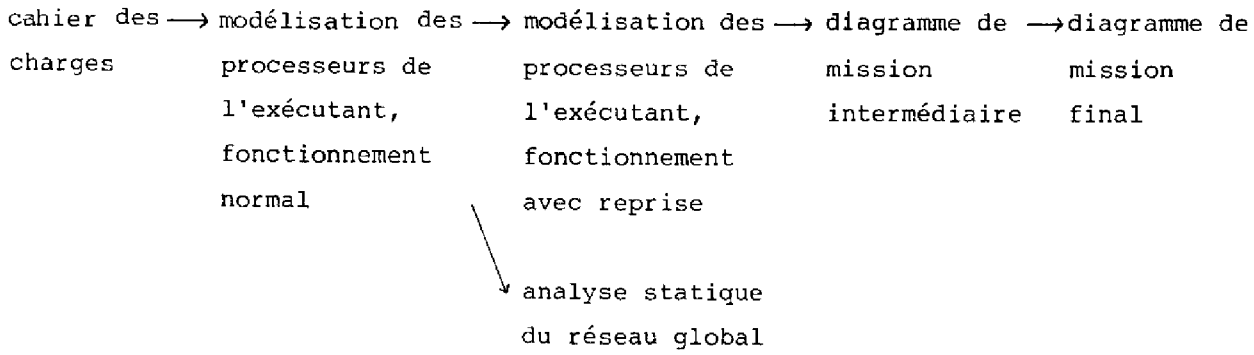
Ensuite, est introduite dans le système la possibilité de perte de message et la représentation de la reprise en cas de panne apparaît dans les différents réseaux.

Un premier modèle appelé diagramme de mission intermédiaire est alors dérivé de l'ensemble des réseaux de l'exécutant par juxtaposition de tous les réseaux en un seul en ne gardant que les communications interprocesseurs et en écartant la représentation des actions locales.

A partir de ce modèle, à chaque transition indiquant la réception d'un message ? α , le concepteur modifiera le diagramme de mission de l'observateur en vue de représenter la reprise du système après la perte de ce message α .Cela conduira dans la plupart des cas à rajouter des transitions correspondant aux messages de la reprise. On obtient ainsi le diagramme de mission final.

Bien entendu, à chaque étape de l'obtention du diagramme de mission les réseaux résultants seront validés afin de prouver les propriétés : borné, réinitialisable et vivant.

On a ainsi les étapes suivantes



Remarque :

Le type des fautes considérées et le mode d'évolution de l'observateur ont une influence sur la modélisation finale obtenue.

. Mode d'évolution.

Le mode d'évolution du modèle, choisi ici, est l'émission-réception simultanée d'un message. Ceci signifie que lorsque l'observateur, à l'écoute du support de communication, capte un message qui y transite, il fera évoluer simultanément les réseaux de l'émetteur et du récepteur correspondant, ce qui induira un facteur d'anticipation sur l'état de l'exécutant.

. Portabilité.

Ainsi, lorsque la structure distribuée est celle représentée figure III-1, le logiciel auto-testable peut facilement être implémenté sur différents systèmes.

Seul le diagramme de mission, donc la base de données de l'observateur, doit être modifié lorsqu'on veut implémenter l'observateur dans un autre contexte distribué ou lorsqu'on veut vérifier un autre mécanisme ou niveau de protocole. De plus, afin d'assurer une bonne fiabilité à la structure de données représentant le réseau, nous utiliserons une structure de données dérivée de façon systématique de celle sur laquelle l'outil de vérification OGIVE a fait son analyse.

III.2. EXEMPLE ILLUSTRATIF DANS LE SYSTEME REBUS

Afin de mieux illustrer la méthode d'obtention du diagramme de mission, nous allons l'appliquer au système REBUS, système de communication, sûr de fonctionnement, pour un réseau local de microcalculateurs. Ce système développé au L.A.A.S., en collaboration avec le Centre de Recherche GIER-SCHLUMBERGER est l'une des bases d'étude du système MODUMAT 800, système distribué temps réel de commande et contrôle de procédés industriels, développé et commercialisé par SCHLUMBERGER-EUROPE.

III.2.1. Présentation du support de l'étude : le système REBUS [29][30][31]

L'introduction d'une structure numérique multi-microprocesseurs en vue de réaliser la commande et le contrôle de procédés industriels permet de supporter, avec une technologie unique, l'ensemble des fonctions de ces systèmes : régulation de boucle - optimisation locale - commande et contrôle de procédé - observation de la production,... et permet la souplesse d'utilisation nécessaire dans de tels systèmes.

Il convient alors de définir la structure de l'ensemble, les connexions entre les différents processeurs d'application et les communications dans cette structure.

Le choix entre toutes les solutions possibles sera guidé par la prise en compte, dès les premiers stades de la conception, de trois contraintes importantes que doit vérifier le système : accepter la complexité, permettre l'extensibilité et assurer la sûreté de fonctionnement.

.Complexité

Pour des systèmes de commande et contrôle de procédés industriels, un aspect important est la complexité d'un processus à réguler (nombre élevé de boucles de régulation, de points de mesures, de capteurs, d'actionneurs,...). Il paraît donc opportun de distinguer plusieurs niveaux de fonctions plus ou moins sophistiqués : fonction d'acquisition des données, fonction de régulation, fonction de décision,... Ces fonctions devront pouvoir être réalisées par la commande.

.Extensibilité

Pour de tels systèmes, les besoins des utilisateurs, même s'ils sont bien définis au début de l'étude, peuvent évoluer du fait de nouvelles technologies, de nouveaux outils mis à leur disposition et de nouvelles contraintes d'exploitation. Il faut adopter une méthode globale de conception du logiciel de communication qui permette aussi bien l'extensibilité du matériel (retrait ou ajout d'un processeur fonctionnel) que du logiciel (modifications des fonctions, des lois de commande,...).

.Sûreté de fonctionnement

Ce critère de sûreté est le point le plus important pour un système assurant la commande et le contrôle automatique d'un procédé. Des pannes matérielles ou des erreurs logicielles peuvent intervenir et perturber le fonctionnement normal du système ; en aucun cas, ces pannes ne doivent conduire à la possibilité de perte de la surveillance ou du contrôle, ou à l'arrêt de la commande. Il faut donc introduire des mécanismes de détection de fautes, éviter la propagation d'une panne dans tout le système et éliminer les effets de pannes rendant le système commandé incontrôlable.

De nombreux mécanismes de détection de pannes et de reconfiguration existent afin de satisfaire les critères précédents.

L'analyse des différentes possibilités de pannes et de leurs conséquences, dans un contexte où la contrainte de sûreté de fonctionnement est extrêmement importante, a conduit à étendre l'hypothèse classique de la panne unique aux systèmes distribués et communicants : une panne est localisée dans un processeur unique et les différents processus situés dans ce processeur doivent évidemment être considérés comme pouvant fonctionner de façon incorrecte. De plus, après une panne dans un processeur, on ne considère pas que ce processeur s'arrête dans un état "mort" ; plus généralement, nous supposerons que ses actions peuvent être erronées et, par exemple, qu'il peut envoyer des messages corrects suivant une séquence imprévue à n'importe quel instant.

Le processeur en panne pouvant avoir un comportement erroné quelconque, il faudra le détecter au plus tôt afin d'éviter la propagation de l'erreur à tout le système.

En conséquence, une structure permettant de contrôler et commander un procédé industriel tout en assurant une bonne sûreté de fonctionnement a été définie ; elle repose sur la distribution réalisée tant au niveau des fonctions (hiérarchie du traitement) qu'au niveau de l'implémentation (distribution entre processeurs indépendants).

Un système distribué est, de façon classique, décomposé en deux parties : les processeurs fonctionnels et un système de communication entre ces différents processeurs. Dans notre cas, le schéma de la structure générale du système global est donné figure III.9.

L'ensemble des processeurs fonctionnels d'un tel système général de commande-contrôle industriel est formé :

- des équipements situés dans une salle de contrôle comprenant : des processeurs de visualisation banalisés ou spécialisés supportant le dialogue homme-machine, des imprimantes pour journal de bord,...
- des équipements situés dans des salles techniques comprenant : des régulateurs multiboucles programmables, des acquéreurs de données, des automates programmables,...

Les régulateurs multiboucles programmables déroulent les algorithmes de régulation pour au maximum huit boucles et effectuent une régulation dégradée (monoboucle) de secours en cas de panne de leur unité centrale.

Le système de communication relie l'ensemble des équipements précédents ; il comprend au niveau matériel un support de communication et des cartes interface bus série (IBS) indépendantes des équipements fonctionnels connectés.

L'interface bus série assure la connexion des équipements sur le bus (réception - émission des messages et gestion des protocoles). Elle est bâtie autour d'un microprocesseur ZILOG 80 et est composée de trois parties : une partie interface avec le bus, une partie traitement des messages et une partie interface avec le processeur fonctionnel arrière.

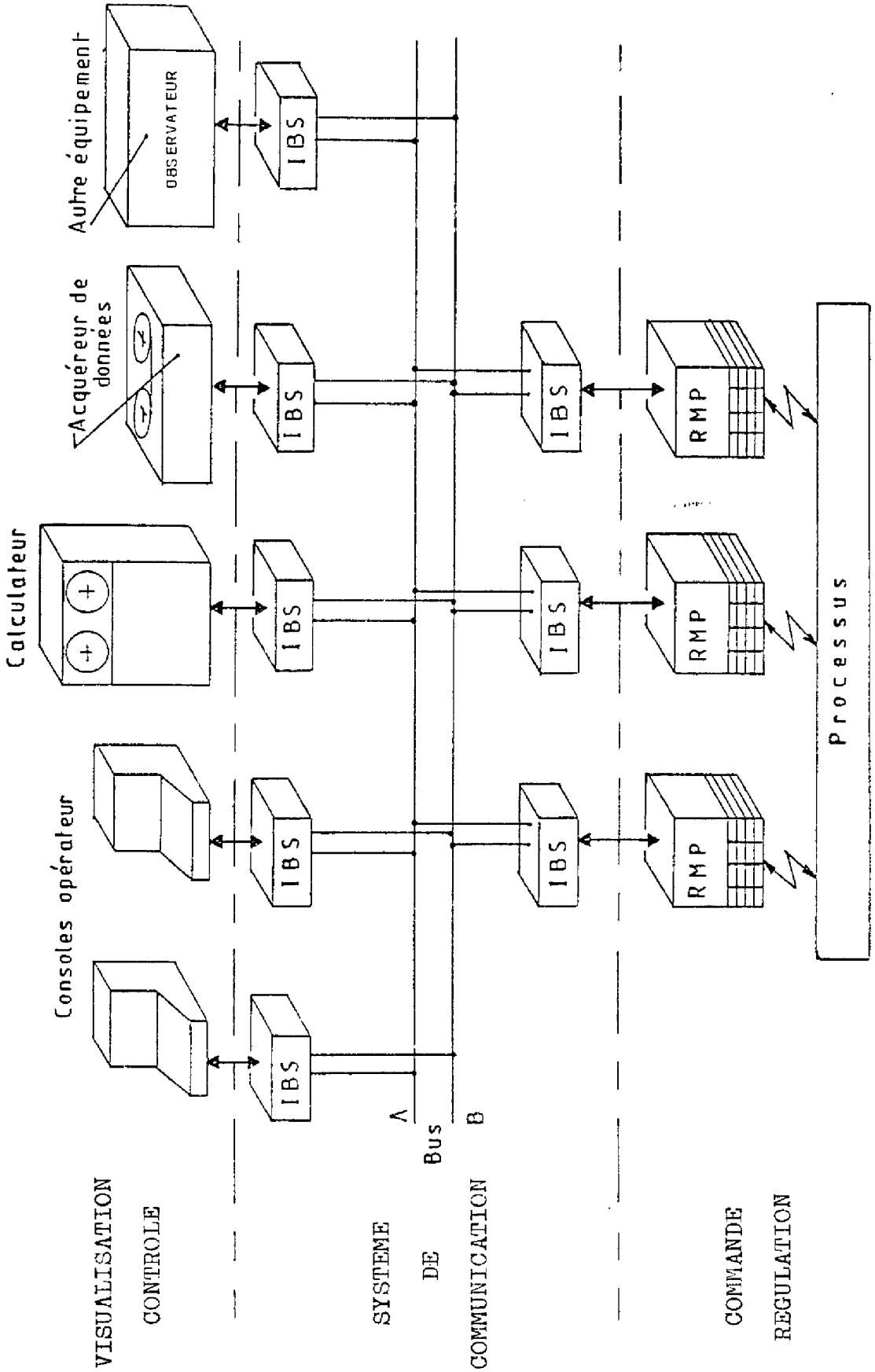


FIGURE III.9. Structure générale

Le support de communication est un élément très important du système de communication : il doit être sûr et fiable. Il est constitué ici d'un double bus série, de longueur inférieure à 1,5 km, travaillant à 512 Kbauds et utilisant un codage biphase et la trame HDLC.

Le logiciel de base du système de communication a pour but de gérer l'ensemble des messages transitant dans le système. La complexité de sa conception, résultant du parallélisme des processeurs et de la contrainte de sûreté de fonctionnement, a conduit à utiliser deux outils formels de conception : le langage C.S.P. (Communicating Sequential Processes) [19] pour étudier une méthode de conception progressive des systèmes parallèles [32] ; les réseaux de Petri [18] pour exprimer, analyser et implémenter les protocoles [33].

Dans ce contexte, un point crucial est apparu ; il est extrêmement délicat de mettre au point, mesurer, détecter les erreurs fonctionnelles dans les systèmes distribués et les difficultés rencontrées semblent rendre indispensable l'utilisation de mécanisme de détection en ligne à deux niveaux :

- au niveau local, sur chaque IBS, par vérification sur une "liste de bienvenue" que le message reçu par le processeur peut effectivement l'être,
- au niveau global, sur un processeur spécialisé, par utilisation de l'observateur connecté au système de communication.

Pour illustrer notre méthode d'obtention du diagramme de mission de l'observateur, parmi tous les choix possibles et afin d'avoir un mécanisme significatif, nous avons choisi de traiter l'aspect de base de la communication dans REBUS, à savoir le partage robuste de l'accès au bus par anneau virtuel reconfigurable. En effet, pour communiquer entre-eux, tous les processeurs fonctionnels utilisent le support de communication, et de ce fait, l'établissement d'une discipline de partage du bus par les processeurs est un des aspects de base du système.

III.2.2. Protocole de passage du statut maître : cahier des charges

Nous allons présenter l'algorithme distribué défini pour le réseau REBUS afin de gérer l'accès du bus en exclusion mutuelle, à la fois pour un système sans faute et dans le cas d'une panne simple.

Comme dans le mode synchrone de HDLC, la notion de station primaire et station secondaire a été utilisée. Une unité primaire, à un instant donné, a le contrôle des transferts physiques sur la ligne, appelé statut primaire, et de la reprise sur erreur ; toutes les autres unités sont secondaires, c'est-à-dire ne peuvent utiliser le bus que sur autorisation de l'unité primaire. L'unité primaire initialise la communication avec une autre unité secondaire, lui envoie des messages. Elle aura ensuite le droit d'autoriser le secondaire à répondre. Elle pourra alors envoyer d'autres requêtes lorsqu'elle aura reçu les réponses attendues ou lorsqu'elle doit entreprendre une action de reprise sur erreur dans le cas d'acquittement non reçu.

Dans REBUS, une unité primaire a le contrôle du bus pour un temps fini et elle transmet son privilège de parole à une autre unité lorsqu'elle n'a plus de messages à transmettre ou lorsque son temps de parole est terminé.

Le problème général à résoudre est de décider quelle station doit être primaire à un instant donné.

Dans ce but, plusieurs disciplines d'allocation de la ligne sont disponibles (contention, hiérarchie temporelle, multiplexage,...) mais n'ont pas été retenues pour diverses raisons (exclusion voulue, pas de centralisation,...) [34].

Aussi, l'algorithme d'allocation de la ligne utilise l'idée suivante : le seul lien de communication entre stations étant la ligne, le passage du statut primaire se fera par messages sur la ligne. Le mécanisme de passage du statut primaire doit satisfaire les trois conditions suivantes :

- il ne peut y avoir perte définitive du statut primaire,
- il ne peut y avoir plus d'une unité primaire à un instant donné,
- une unité ayant le droit d'être primaire doit pouvoir l'être au moins une fois dans un intervalle de temps donné.

Le passage du statut primaire d'une unité à une autre doit se faire suivant un ordre bien défini. Parmi plusieurs organisations possibles des unités (centralisée avec polling, par anneau,...) [32][34], l'organisation logique de l'ensemble des unités d'interface (IBS) sur un anneau virtuel a été retenue (Figure III.10.). De cette façon, une unité primaire ne peut relâcher son statut primaire qu'en faveur de l'unité suivante dans l'anneau. Par contre, les messages fonctionnels circulent librement d'une unité à une autre.

Afin de satisfaire les trois critères précédents assurant une bonne sûreté de fonctionnement et de prendre en compte les pertes éventuelles de messages sur le support de communication (et donc les pertes de statut primaire), un protocole plus robuste et tolérant aux pannes est ajouté au protocole simple de passage d'un message "statut primaire".

Ce protocole contrôlera la discipline d'allocation de la ligne, mais il est relativement long, ainsi les stations pouvant avoir le privilège d'être primaire seront divisées en deux classes.

La plupart des unités sont "ouvriers" : elles peuvent acquérir ou relâcher le statut primaire suivant le protocole de passage par un message "statut primaire".

Les autres unités appelées "patrons" peuvent également obtenir le statut primaire mais sont, en plus, responsables du passage du statut primaire et ont la charge de reconfigurer l'anneau virtuel en cas de panne.

Un seul patron est actif à un instant donné et est appelé "maître", les autres patrons vérifient collectivement le comportement du maître. Le passage du statut primaire (appelé alors statut maître) entre les différents patrons se fait à travers un anneau virtuel de patrons selon un protocole robuste permettant la reconfiguration du système en cas de perte du statut primaire. De même, les ouvriers sont organisés en anneaux virtuels d'ouvriers. Une boucle d'ouvriers part d'un patron et lui revient en fin de boucle (Figure III.11.).

Remarque : un ouvrier se trouve sur deux anneaux afin d'être accédé par deux maîtres et de ne pas être isolé en cas de panne de l'un d'entre-eux.

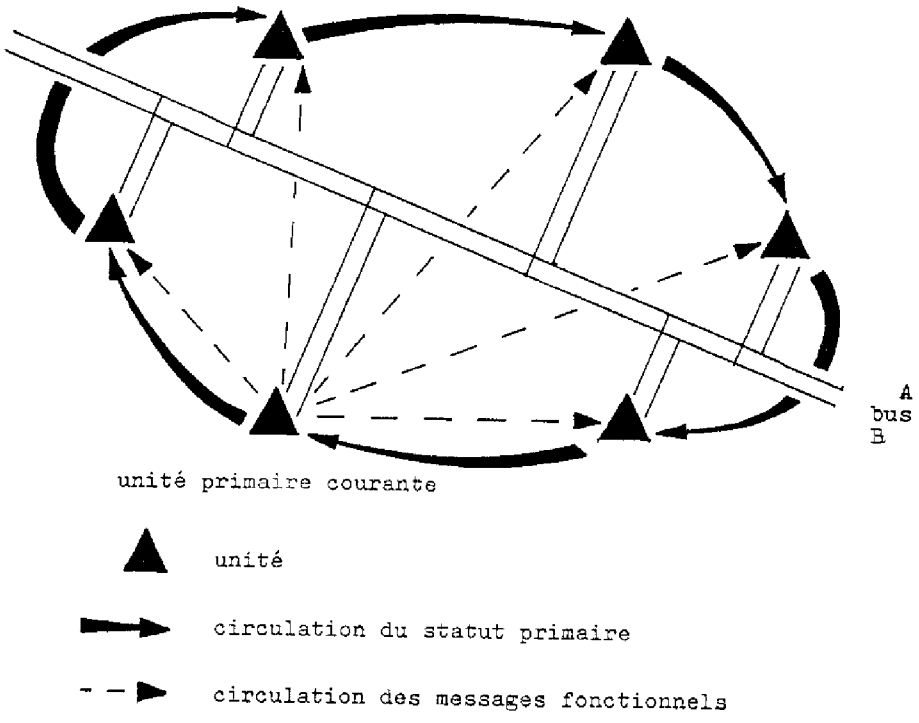


FIGURE III.10. Anneau virtuel de patrons

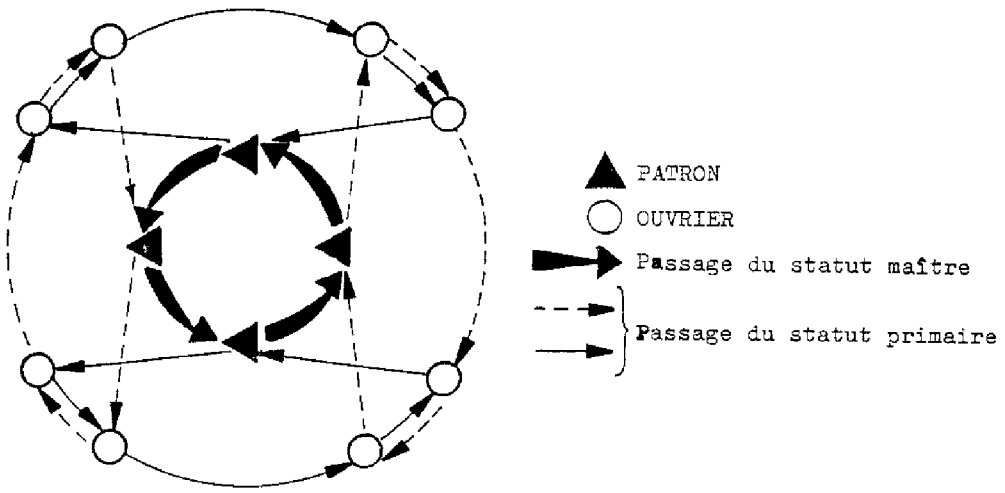


FIGURE III.11. Configuration à plusieurs patrons : une boucle de patrons et quatre boucles d'ouvriers

Lorsqu'un patron reçoit le statut maître, il devient en même temps primaire. Il peut relâcher son statut primaire en faveur d'un ouvrier sur un anneau virtuel tout en restant l'unité maître c'est-à-dire qu'il vérifie le fonctionnement correct de ses ouvriers. Lorsque le statut primaire lui revient, il peut alors envoyer le statut maître (perdant ainsi également le statut primaire) à son suivant dans l'anneau virtuel de patrons. De plus amples détails sur les différents états possibles d'un patron sont donnés dans [32][34].

Résumé :

a) le statut primaire est le privilège de l'accès au bus et du contrôle des transferts physiques,
b) le statut maître est le privilège du contrôle d'allocation de la ligne et de sa reconfiguration.

A un instant donné, une seule unité patron doit avoir le statut maître. Un patron est une unité capable de devenir maître à un instant donné ; toute unité ayant le statut maître est capable de :

- déléguer son statut primaire aux ouvriers dans l'anneau,
- faire une reprise du statut primaire en cas de panne dans un ouvrier,
- participer collectivement au contrôle du passage du statut maître,
- reconfigurer le système en présence de panne simple dans un patron.

Un ouvrier peut recevoir le statut primaire, mais n'a aucune possibilité de reconfiguration.

III.2.3. *Protocole de passage du statut maître : modélisation*

III.2.3.a. *Modèle informel - modèle formel*

Nous devons maintenant aborder la description fonctionnelle du protocole de passage du statut maître qui nous permettra alors d'obtenir un modèle formel associé à un modèle informel.

Un certain nombre de solutions ont été envisagées [32][34][35] ; nous retiendrons celle proposée par [34]. Elle repose sur l'utilisation de trois messages échangés entre deux unités interface IBS (Figure III.12.).

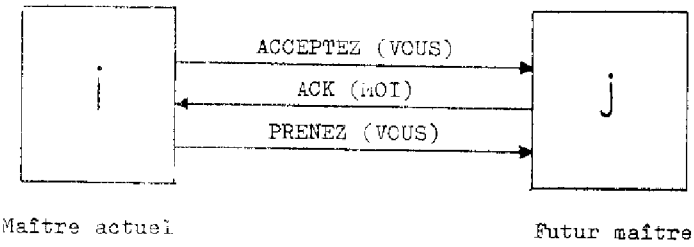


FIGURE III.12. Protocole de passage du statut maître

Son fonctionnement est le suivant :

Soit i le maître actuel et j le futur maître sur l'anneau virtuel :

- i envoie ACCEPTEZ (vous) à j,
- j acquitte à l'aide d'un message diffusé ACK (moi) ; c'est bien lui le futur maître et il avertit les différents processeurs. i a toujours le privilège et peut répéter ACCEPTEZ (vous) s'il trouve la réponse de j incorrecte,
- après réception de l'accord, i envoie à j PRENEZ (vous), il perd alors le statut maître.

Supposons maintenant que le statut maître soit perdu entre i et j. Le suivant de j dans l'anneau, k, doit reconfigurer, mais il ne peut le faire ni seul (pour prévenir une possible action erronée de sa part), ni avec i ou j qui sont susceptibles d'être en panne. Il demande alors à son suivant dans l'anneau, l, l'autorisation de reconfigurer par le message DONNEZ. Si l est en accord avec k, le protocole précédent à trois messages peut débuter ; l est alors le pseudo-patron et k deviendra le nouveau maître (Figure III.13.).

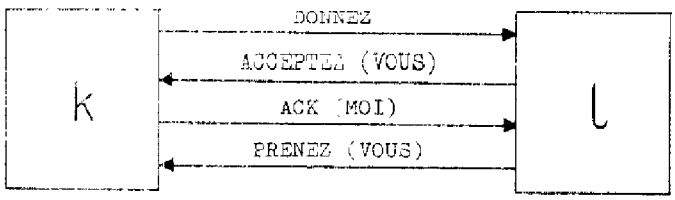


FIGURE III.13. Reconfiguration du statut maître

A partir de la description du fonctionnement du protocole de passage de statut maître et des principales possibilités de reconfiguration, un modèle informel (Figure III.14.a.) associé à un modèle formel (Figure III.14.b.) a été obtenu.

Les spécifications du passage du statut maître sont représentées par les transitions T1 à T6 et quelques cas de reconfiguration par les transitions T7 à T13.

Remarque : la modélisation du diagramme de mission est faite en pensant à l'évolution des processeurs dans le simulateur de réseaux de Petri (évolution de deux processeurs simultanément) ; ainsi, la diffusion du message ACK(moi) à tous les processeurs patrons contrôlant le passage du statut maître n'est pas modélisée. Sa représentation conduirait à modéliser les conditions et les actions internes à chaque processeur (mise à jour d'une variable et armement d'un time-out) et permettrait de vérifier que le patron qui reçoit le statut maître est bien le suivant du maître actuel dans l'anneau.

III.2.3.b. Modèle global

Dans ce système distribué, la communication est effectuée par transfert de messages entre le processeur émetteur et le processeur récepteur, elle est donc modélisée par une place partagée entre les deux processeurs.

Le réseau modélisant la logique du passage du statut maître entre quatre patrons est donné sur la figure III.15. Chaque cycle représente un processeur, les places reliant les cycles représentent les messages ACCEPTEZ , ACK et PRENEZ circulant entre les différents processeurs. Par souci de clarté, la reprise en cas de perte de ces messages n'est pas modélisée.

I.- PASSAGE DU STATUT MAITRE

1. Si une unite est non-patron et reçoit le message "acceptez" (?), elle peut accepter le statut maître.
2. Le statut maître étant accepté elle l acquitte (!ack) et se met en attente de confirmation du statut.
3. L'attente de confirmation se termine par la réception du message "prenez" ; l'unité devient patron
4. L'unité perd son état de patron en envoyant le message "acceptez" et attend un acquittement (!ack).
5. L'attente se termine par la réception de l'acquittement.
6. L'unité ayant reçu l'acquittement confirme par "prenez" et redevient non patron.

II.- RECONFIGURATION DU STATUT MAITRE

7. Une unité en attente de confirmation du statut maître peut recevoir le message "acceptez" et revenir dans l état d'acceptation du statut maître.
8. Une unité en attente d'acquittement pour prendre l'initiative de relâcher à nouveau son statut maître en envoyant le message "acceptez".
9. Une unité etant non-patron peut prendre l initiative de demander le statut maître par un message "donnez" elle attend alors la transmission du statut maître.
10. L'attente du statut maître se termine par la réception du message "acceptez" ; l'unité est dans l'état d'acceptation du statut maître.
11. Une unité en attente de confirmation du statut maître peut prendre l'initiative d'envoyer le message "donnez" et aller dans l'état d'attente de transmission du statut de maître.
12. Une unité non patron peut être sollicitée par la réclamation du statut maître ("donnez") elle passe alors dans l'état de pseudo-patron.
13. Une unité étant pseudo-patron envoie le message "acceptez" et se met en attente d'acquittement du message

III.- ETATS INITIAUX

Places marquées
Une unité : patron
Toutes les autres : non-patron

FIGURE III.14.a. Protocole de passage du statut maître. Modèle informel

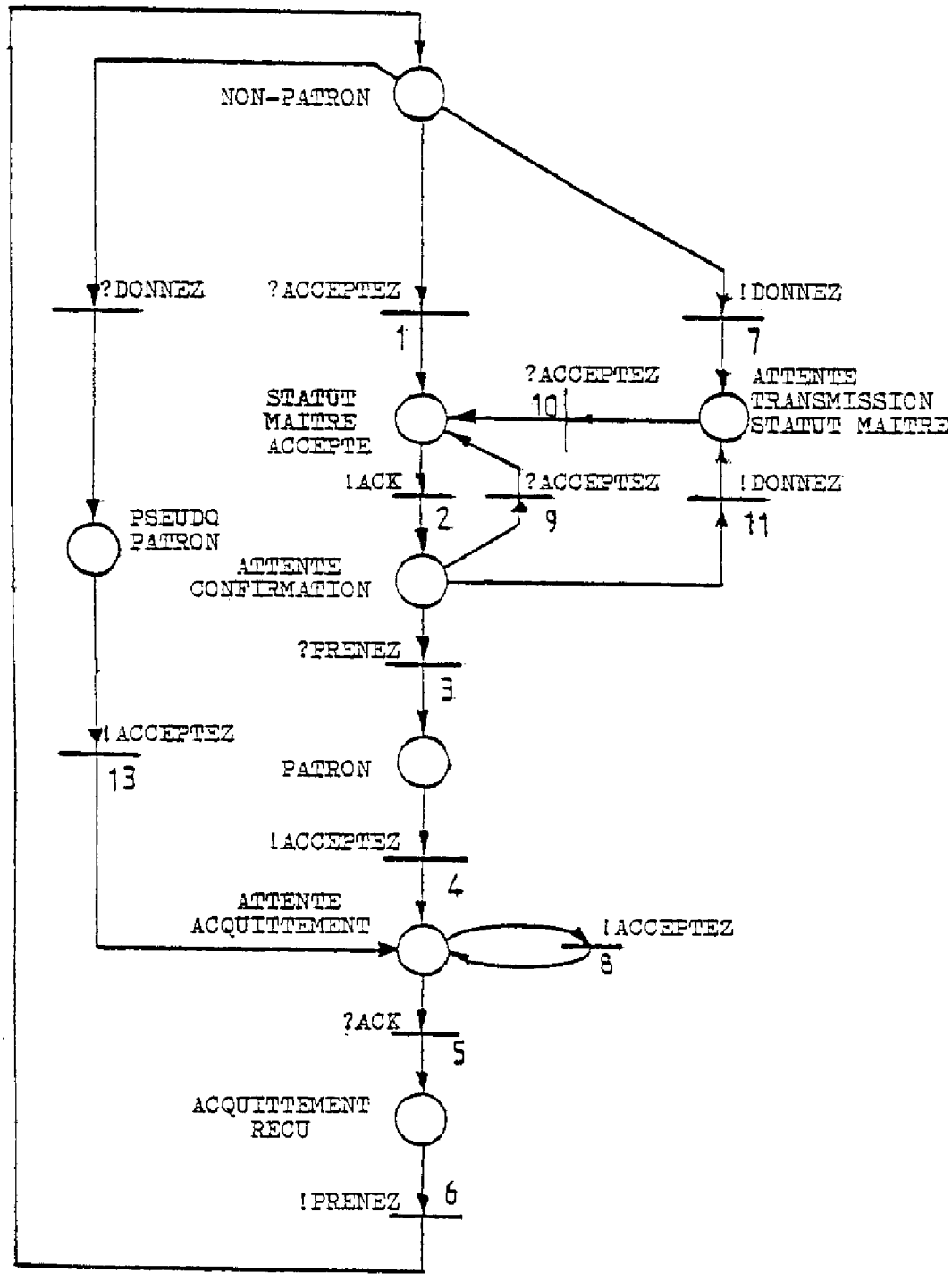


FIGURE III.14.b. Protocole de passage du statut maître. Modèle formel

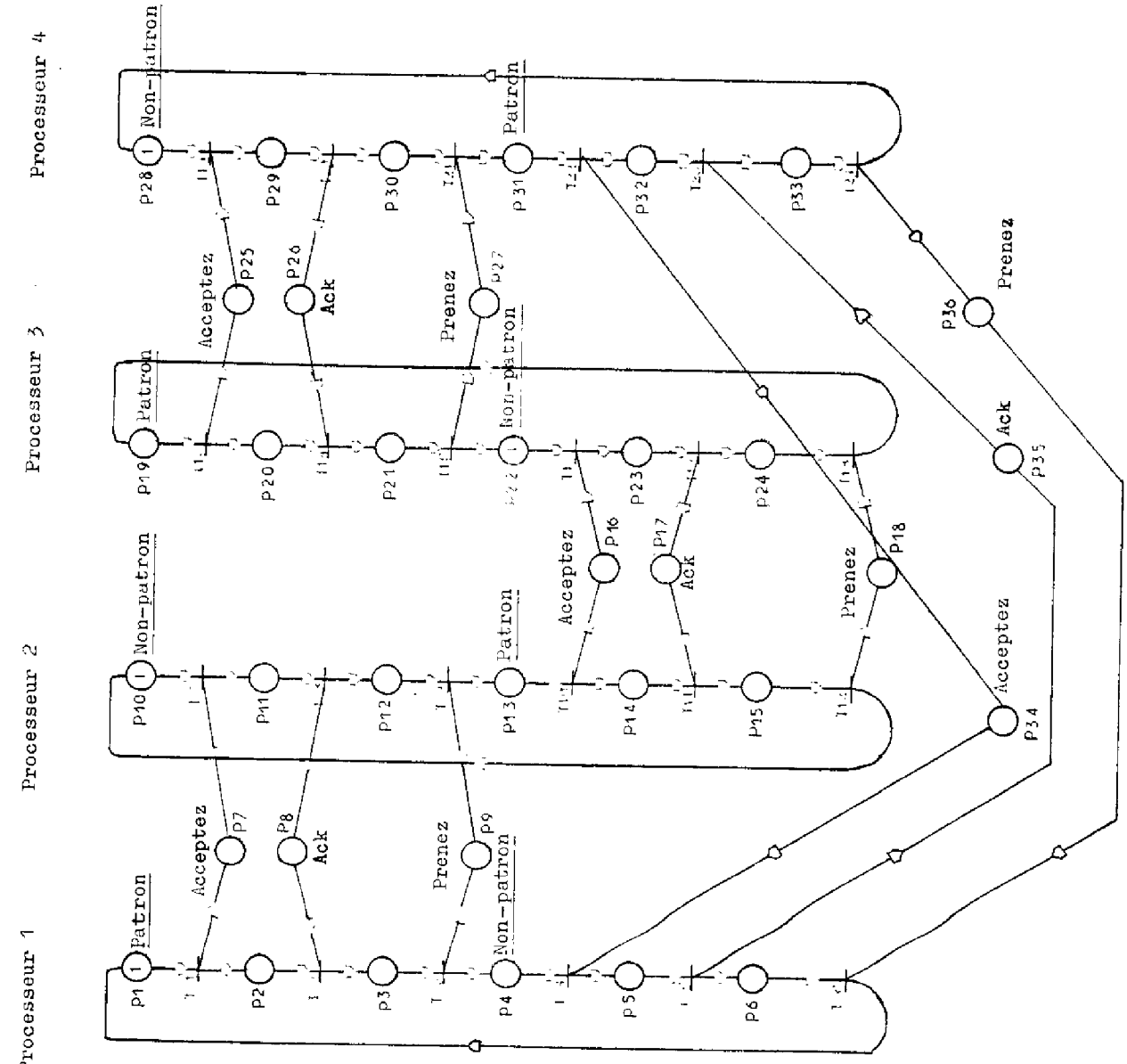


FIGURE III.15. Passage du statut maître entre quatre patrons

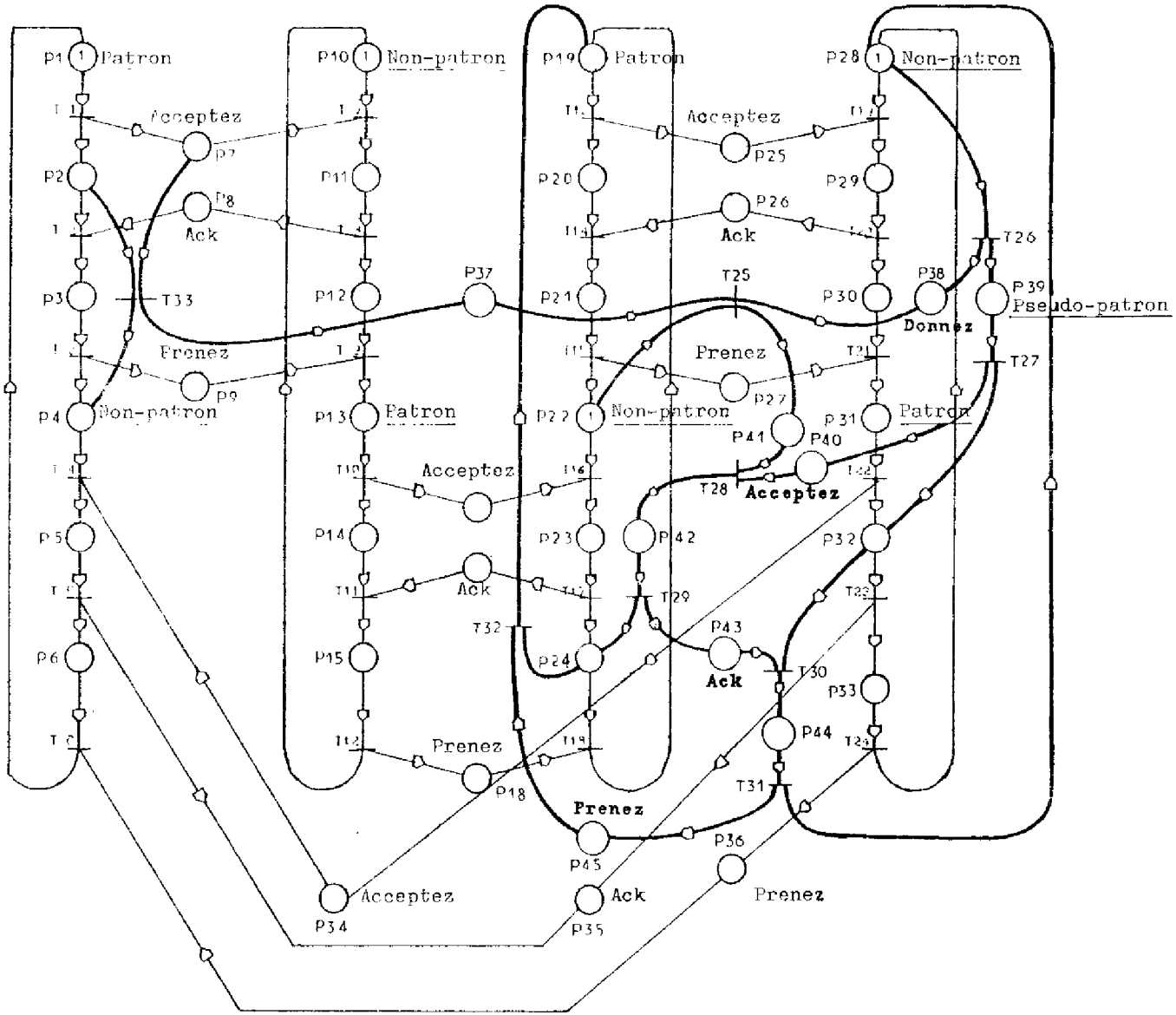


FIGURE III.16. Reconfiguration du deuxième patron

La figure III.16. représente le réseau précédent auquel a été ajouté la représentation de la reconfiguration du deuxième processeur (en traits plus épais). Cette reconfiguration est initialisée par le patron 3 par envoi du message DONNEZ au patron 4, après perte du message ACCEPTEZ émis par le processeur 1.

Ces deux réseaux ont été prouvés saufs et vivants par OGIVE.

III.2.4. *Diagramme de mission*

En utilisant la méthodologie proposée, le diagramme de mission de l'observateur est obtenu à partir du modèle formel et en considérant que tout message envoyé par un processeur est capté par l'observateur mais peut ne pas avoir été reçu par le processeur destinataire. Ainsi, depuis une place suivant une transition étiquetée ?α , on considère de façon systématique le cas où le message a effectivement été reçu et celui où il n'a pas été reçu, ce qui conduit à introduire dans le modèle de nouvelles transitions.

Le modèle global de chacun des processeurs est donné sur la Figure III.17. Il est obtenu à partir du réseau de la Figure III.14.b. auquel on a ajouté les transitions T14 à T16, modélisant le mauvais fonctionnement possible du destinataire d'un message.

La transition T14 résulte de la non-réception par l'unité non-patron du message ACCEPTEZ et de la réémission par l'unité patron de ce message. La perte du message ACK par l'unité dans l'état "attente d'acquittement" et sa réception par l'observateur conduit à modéliser la réémission du message ACCEPTEZ par la transition T15. L'unité dans l'état "attente confirmation" n'ayant pas reçu le message PRENEZ envoyé par son précédent dans l'anneau peut décider de reconfigurer et d'envoyer le message DONNEZ à son suivant ; ceci est représenté par la transition T16.

Remarque : Nous avons fait l'hypothèse de la panne unique dans un système distribué. Or, une panne dans un processeur conduit à la perte d'un message du protocole normal et donc à l'activation du mécanisme de reconfiguration du statut maître. Les autres processeurs impliqués par cette reconfiguration seront donc supposés sans panne, le message DONNEZ ne peut alors pas être perdu.

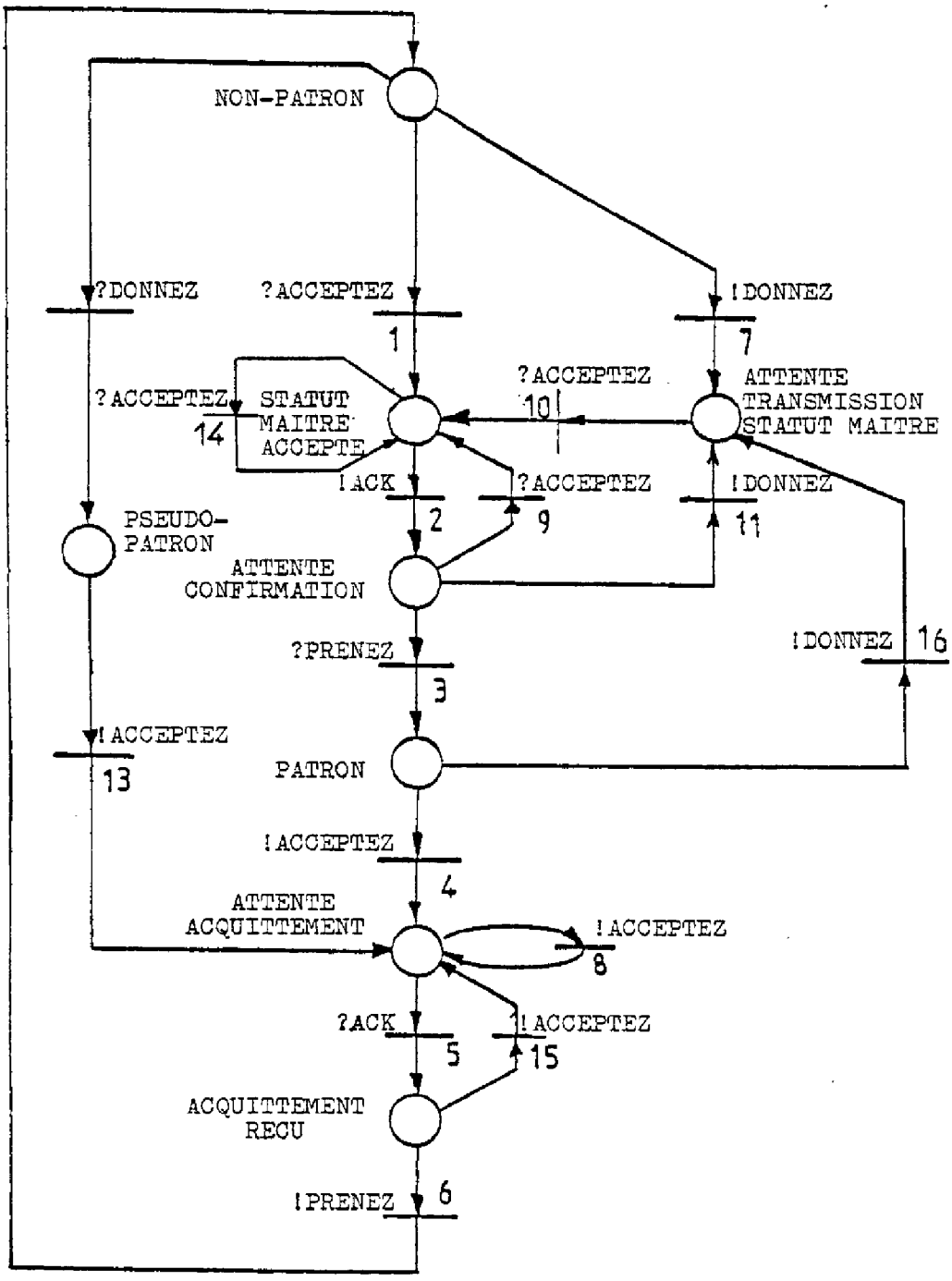


FIGURE III.17. Modèle global d'un processeur

Ainsi, il existe virtuellement un tel réseau par unité connectée au support de communication.

En règle générale, le diagramme de mission global de l'observateur est obtenu par la réunion dans un seul réseau de l'ensemble des réseaux modélisant les différents processeurs. Les transitions !message et ?message sont virtuellement connectées par un protocole : c'est la modélisation de la réception d'un message qui spécifie le protocole choisi et le mode d'évolution dans l'observateur. La solution la plus simple, choisie ici, est l'émission-réception simultanée, c'est-à-dire que lorsque l'observateur, à l'écoute du bus, capte un message qui y transite, il fait évoluer simultanément les réseaux de l'émetteur et du récepteur.

Dans ce système distribué et pour ce mécanisme observé, tous les réseaux étant identiques, il n'est, bien entendu, pas nécessaire que le diagramme de mission les contiennent tous. Un seul réseau et le marquage courant de chaque processeur sont suffisants pour caractériser l'ensemble des états.

III.3. CONCLUSION

A partir d'un premier exemple simple, mais pourtant significatif, nous avons étudié la répercussion que pouvait avoir la perte d'un message, par un processeur de l'exécutant et/ou par le processeur observateur, sur la modélisation du diagramme de mission ou sur le logiciel de l'observateur.

La perte de message par le processeur fonctionnel destinataire se traduit par l'ajout, dans le modèle, de transitions représentant les messages dûs à la reprise sur erreur.

De plus, une extension du simulateur de réseaux de Petri par une procédure de recherche des états suivants nous permet de prendre en compte la perte possible d'un message par l'observateur et en conséquence de rechercher un état cohérent du système global.

Il a été également montré que la perte d'un message par un processeur et l'observateur n'avait aucune influence ni sur la modélisation ni sur le logiciel de l'observateur.

Enfin, nous avons présenté un système distribué de commande en temps réel, REBUS, et son protocole de gestion et d'allocation de ligne. Nous nous sommes ensuite appliqués à obtenir le diagramme de mission de ce mécanisme en utilisant la méthodologie proposée.

CHAPITRE IV

IMPLÉMENTATION

-

INTRODUCTION

Dans le contexte distribué du réseau REBUS présenté au chapitre précédent, la nécessité de sûreté de fonctionnement conduit à considérer d'une part que tout processeur peut devenir défectueux et avoir un comportement imprévu et d'autre part que l'environnement extérieur peut agir de façon incorrecte sur le système. Ceci pourraît, par exemple, conduire un processeur à recevoir et traiter un message alors que son état n'était pas adéquat : le nouvel état serait alors totalement inconnu et les actions conséquentes absolument imprévisibles. Le système global serait en danger car sa cohérence serait détruite.

De plus, l'état d'un système distribué ne peut être connu de façon exhaustive, et donc, dans un tel système, il est délicat de mettre au point des programmes, de détecter des erreurs fonctionnelles. La résolution de ces différentes difficultés explique que des mécanismes de détection en ligne au niveau global sont indispensables et nous a conduit à implémenter, dans le réseau REBUS, la méthode d'autotest présentée dans les chapitres précédents.

Un observateur implémenté sur un processeur spécialisé connecté au système de communication peut réaliser cette détection en ligne au niveau global. Il devra tout d'abord assurer le traitement des messages et donc leur réception, leur test et leur stockage et ensuite permettre le dialogue avec l'opérateur, pour la maintenance par exemple.

Le paragraphe IV.1. présente le support matériel sur lequel la partie redondante du logiciel d'auto-test, l'observateur, est implantée.

Le développement d'un logiciel de taille importante et sa structuration en différentes tâches concurrentes ont conduit à choisir un support logiciel particulier (langage, moniteur multitâches) que nous présentons au paragraphe IV.2.

La présentation des différentes tâches et de leurs fonctions est effectuée dans le paragraphe IV.3. Nous avons décomposé le logiciel de l'observateur en quatre modules : le simulateur de réseaux de Petri, l'interface avec le système de communication, l'interface avec l'opérateur et la partie de stockage.

Enfin, deux exemples de système distribué,sur lequel l'observateur a été implémenté ou est en cours de développement,sont présentés au paragraphe IV.4. : le réseau REBUS et le système MODUMAT 800.

IV.1. SUPPORT MATERIEL

L'observateur est un équipement spécialisé de contrôle du système distribué. Contrairement aux autres équipements du système, il n'émet pas de messages sur le support de communication, il est simplement à l'écoute du bus et capte soit un sous-ensemble donné de messages, soit tous les messages qui y transitent en vue de les traiter.

Ainsi, sa partie réception de messages doit être très fiable pour ne pas avoir d'influence sur la sûreté de fonctionnement de l'ensemble du système.

IV.1.1. Carte interface bus série et carte unité centrale SBC 80/30

Il nous est apparu nécessaire de séparer sur deux matériels distincts la partie réception de messages et la partie traitement de ces messages. La structure du matériel ainsi définie est la suivante (Figure IV.1.):

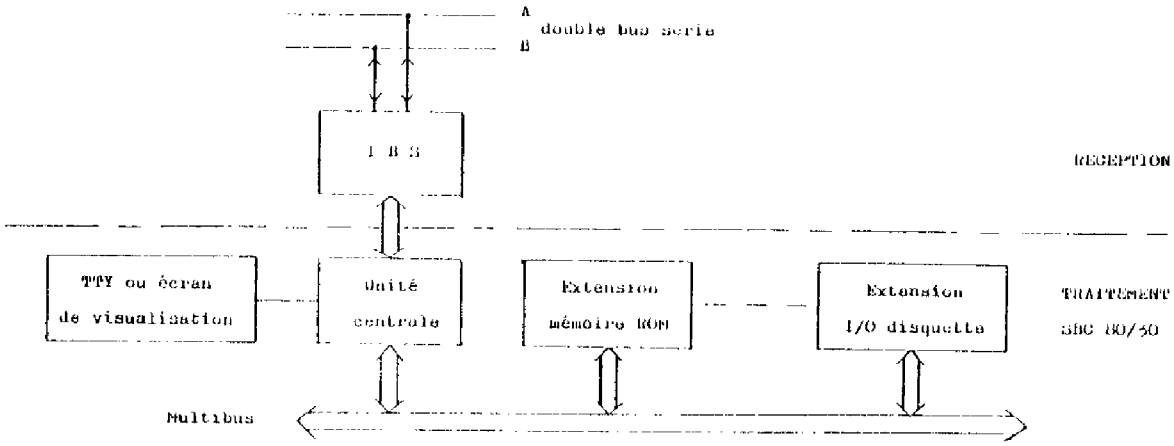


FIGURE IV.1. L'observateur - support matériel

IV.1.1.a. La carte Interface Bus Série

Un Interface Bus Série supportera la communication de l'observateur sur le bus. Il a la même structure que les IBS standard mais les fonctions implémentées seront réduites : réception de tous les messages circulant sur le bus, stockage dans un buffer circulaire en vue d'une transmission au processeur fonctionnel.

IV.1.1.b. La carte unité centrale SBC 80/30

La partie traitement de l'observateur sera implantée sur :

- une carte unité centrale SBC 80/30 INTEL construite autour d'un microprocesseur 8085 A INTEL, de 16 K octets de RAM,...,
- une carte extension mémoire ROM de 64 K octets,
- un bus interne parallèle permettant de connecter les différentes cartes : le Multibus.

De plus, un télétype connecté à la carte unité centrale assurera l'interface avec un opérateur.

Cette structure peut être élargie en fonction des besoins de l'utilisateur : grâce au Multibus il est possible d'ajouter une carte extension d'entrée/sortie sur disquette pour stockage des informations pour un journal de bord.

Ce module supporte plusieurs fonctions :

- réception des messages venant de l'IBS,
- simulation du réseau de Petri,
- dialogue avec l'opérateur : sélection d'un niveau de protocole , des équipements à tester, affichage des messages incorrects, suivi du traitement,...,
- traitement statistique et stockage des résultats.

IV.1.2. Communication carte IBS et carte SBC 80/30

La communication entre la carte IBS et la carte unité centrale SBC 80/30 est effectuée sur un bus parallèle 8 bits.

Le transfert des messages de l'interface vers l'équipement observateur est régi par la structure du matériel de l'interface processeur de l'IBS. Un rappel de cette structure et le protocole de transfert utilisé sont développés en annexe.

IV.2. SUPPORT LOGICIEL

IV.2.1. Langage de programmation : PL-M/80

Nous avons choisi d'écrire le logiciel de l'observateur dans un langage de programmation évolué, PL-M/80 d'INTEL [37]. C'est un langage structuré de haut niveau qui permet d'écrire de façon aisée, et lisible pour d'autres utilisateurs, des programmes complexes. Ainsi, on peut écrire un programme par affinements successifs des différents blocs qui le compose. Le simulateur de réseaux de Petri de l'observateur peut être écrit de la façon suivante :

```
SIMUL : PROCEDURE
initialisation
if transition sensibilisée then évolution du marquage ;
else erreur ;
end ;
```

Dans une première étape, après développement de chaque partie, la procédure SIMUL est donnée Figure IV.2.

IV.2.2. Moniteur temps réel multitâches

L'observateur à implémenter doit répondre, dans un certain temps, aux sollicitations du monde extérieur : réception de messages de priorités différentes, de demandes de l'opérateur, traitement de ces messages en parallèle. Ceci nous a conduit à structurer notre système en tâches s'exécutant de façon concurrente. Cette décomposition en tâches nous permet de plus de satisfaire au critère d'extensibilité du réseau REBUS ; on peut ainsi insérer ou supprimer une tâche, définir de nouvelles relations entre tâches en fonction de nouveaux besoins, sans modifier profondément tout le système.

La synchronisation des différentes tâches et la gestion de leur interaction sont réalisées à l'aide d'un moniteur temps réel multitâches RMX/80 développé par INTEL [38] pour le type de matériel que nous utilisons SBC 80/30.

Le noyau de ce moniteur est basé sur l'utilisation de messages comme mécanisme de synchronisation des tâches, les messages et les tâches en attente de message étant liés sur un concentrateur nommé l'échange.

```
SIMUL : PROC PUBLIC;
    DO FOREVER;
    /****ENTREE DU MESSAGE  M  A TIRER****/
    /* NP nombre de places du reseau */
    /* MARQUAGE(NP) marquage courant */
    /* ALPHA (NP) vecteur d'entree d'une transition  */
    /* BETA  (NP) vecteur de sortie d'une transition */

    TEST=FAUX;
    DO N=0 TO DPAR-1;
        TI=MESS(M).TRANS(N);/* calcul du numero de la transition
                    a tester */
        IF TI=0 THEN DO; /* il n'y a plus de transition a tester */
            N=DPAR;
        END;
        ELSE DO;
            TI=TI-1;
            DO J=0 TO NP-1 ; /*remise a zero de ALPHA et de BETA */
                ALPHA(J) = 0;
                BETA(J) = 0;
            END;

        positionnement sur la liste des places d'entree et de sortie

        reconstitution du vecteur ALPHA

        reconstitution du vecteur BETA

        /* transition TI sensibilisee ? */
            J=0;
            DO WHILE (J<NP) AND (MARQUAGE(J) >= ALPHA(J)) ;
                J=J+1;
            END;
            IF J=NP THEN
            DO; /*transition sensibilisee */
                N=DPAR;
                TEST=VRAI;
            END;
        END;
        IF TEST=VRAI THEN DO;
        /* il existe une transition sensibilisee
           correspondant au message M */
            DO J=0 TO NP-1;
              MARQUAGE(J) = MARQUAGE(J)-ALPHA(J)+BETA(J);
            END;

        END;
        ELSE DO;
        /* il n'existe pas de transition sensibilisee
           correspondant au message M */
            CALL ERREUR;
        END;
    END;
    END;
    END SIMUL;
```

FIGURE IV.2. Procédure SIMUL

Nous allons maintenant détailler ces trois types d'objets.

.La tâche

La tâche est une portion de code exécutée de façon séquentielle. Le système global est composé de plusieurs tâches concurrentes et à chacune d'elles est associée une priorité statique. La tâche la plus prioritaire des tâches prêtes est celle qui sera exécutée.

.Le message

Le message est une zone mémoire permettant à deux tâches de communiquer. Il contient soit de l'information, soit des événements nécessaires à la synchronisation entre les tâches.

Un message manipulé par le noyau du moniteur RMX/80 doit se conformer à une certaine structure. Il doit avoir au moins un en-tête de 3 à 7 octets, le reste du message (optionnel) est de longueur quelconque et contient l'information à transmettre d'une tâche à une autre.

L'en-tête est constitué de :

- un champ LINK (utilisé par le noyau),
- un champ LENGTH (longueur du message entier,en octets),
- un champ TYPE (type du message),
- un champ HOME$EXCHANGE (adresse de l'échange sur lequel le message doit être envoyé s'il n'est plus utilisé dans le système),
- un champ RESP$EXCHANGE (adresse de l'échange sur lequel une réponse à ce message doit être envoyée).

En fait, ce ne sont pas les zones mémoire qui circulent entre les tâches mais les pointeurs de ces zones.

.L'échange

L'échange est une sorte de boîte aux lettres entre les tâches et les messages. Une tâche, envoyant un message à une autre tâche, le dépose sur un échange ; une tâche attend, sur un échange, un message provenant d'une autre tâche.

Ainsi, un échange se comporte comme une double file d'attente : une file de messages attendant d'être reçus par des tâches et une file de tâches en attente de message. Lorsqu'un message arrive sur un échange où plusieurs tâches sont en attente, c'est la tâche en attente la première sur cet échange qui le consommera, indépendamment des priorités des différentes tâches de la file. De la même façon, lorsqu'une tâche arrive sur un échange où plusieurs messages attendent, elle prendra le premier message arrivé sur cet échange.

Un message ne peut être envoyé que sur un seul échange ; une tâche ne peut se mettre en attente que sur un seul échange. Ainsi, un message ne peut être reçu que par une seule tâche.

Le noyau de RMX/80 fournit un ensemble de primitives manipulant ces objets et facilitant la programmation de l'ensemble des tâches du système. Parmi celles-ci, les plus importantes sont les primitives RQSEND et RQWAIT correspondant respectivement à l'envoi et à l'attente de messages.

.Primitive d'envoi RQSEND

Elle envoie un message à un échange, les adresses du message et de l'échange sont les paramètres d'appel. Le message est chaîné dans la file de messages si aucune tâche n'est en attente sur l'échange et la tâche en cours continue son exécution. Si une tâche ou plusieurs sont chaînées, la première tâche reçoit le message, est retirée de la file et rendue prête ; elle prendra le processeur si elle a unepriorité supérieure à celle de la tâche émettrice.

.Primitive d'attente RQWAIT

Elle permet à une tâche de se mettre en attente sur un échange pendant un temps déterminé. L'adresse de l'échange et la valeur du temps d'attente sont les paramètres d'appel ; si cette valeur est 0 le temps d'attente est indéfini. Si un message est chaîné dans la file de messages de l'échange, la tâche reçoit le message et continue son exécution, le message étant retiré de la file et l'adresse du message étant fournie comme résultat de l'exécution de la fonction RQWAIT. S'il n'y a pas de message, la tâche est chaînée à la fin de la file d'attente des tâches, son exécution est interrompue. Si le temps d'attente est limité, la tâche est également chaînée sur une file interne au noyau.

Lorsque la tâche est en attente, elle ne pourra à nouveau être prête à reprendre son exécution (donc être retirée de la file d'attente) qu'après un RQSEND sur son échange par une tâche (cf. paragraphe précédent) ou après expiration du temps limite d'attente par réception d'une interruption de l'horloge temps réel. Dans ce dernier cas, la tâche sera également retirée de la file des tâches en attente d'un délai et l'adresse du message reçu identifiera une condition de time-out.

La combinaison de ces deux primitives permet de réaliser un certain nombre de fonctions :

- suspension volontaire d'une tâche pendant un temps donné, par exécution d'un RQWAIT temporisé sur un échange où aucune tâche n'envoie de message ,
- activation volontaire par une tâche moins prioritaire, d'une tâche plus prioritaire en attente sur un échange, par envoi d'un message sur cet échange,
- accès en exclusion mutuelle d'une zone mémoire par plusieurs tâches par RQSEND et RQWAIT, sur un échange donné, de messages ne contenant pas d'information. Ces actions sont équivalentes aux traditionnelles opérations P et V sur un sémaphore [39].

D'autres primitives sont disponibles dans le noyau du moniteur, en particulier, on utilisera des primitives permettant de créer ou de supprimer des tâches ou des échanges (RQCTSK, RQDTSK, RQCXCH, RQDXCH), de prendre un message s'il est disponible sur un échange et continuer l'exécution (RQACPT). Pour de plus amples détails sur les fonctions de ces primitives, se reporter à [38].

Autour du noyau du moniteur multitâches peuvent être greffées différentes extensions telles que les systèmes de gestion des entrées-sorties avec un terminal, de gestion dynamique de la mémoire, de gestion des fichiers d'un système à disquette. Ces extensions répondent à la plupart des exigences des utilisateurs d'un système temps réel et sont ajoutées aux tâches application pour composer l'ensemble du système.

Le système de gestion des entrées-sorties (terminal handler) avec un terminal opérateur a pour fonction principale la lecture et l'écriture d'une ligne de caractère. C'est un ensemble de tâches communiquant avec les tâches utilisateurs principalement à travers deux échanges RQINPX et RQOUTX. Des messages de requête de lecture (écriture) sont envoyés par l'utilisateur vers le terminal handler sur l'échange RQINPX (RQOUTX). Les requêtes sont satisfaites (lecture terminée/écriture réalisée) dès que le demandeur reçoit un message sur l'échange de réponse spécifié dans le champ RESP$EXCHANGE de la requête.

La tâche réalisant la gestion dynamique de la mémoire permet d'allouer une zone mémoire à une tâche utilisateur ou de libérer cette zone après utilisation. La communication entre cette tâche et les tâches utilisateurs se fait suivant le même principe que précédemment (requêtes sur deux échanges particuliers, réponses sur des échanges propres à chaque tâche).

Dans le cadre de la réalisation de l'observateur, seule l'extension permettant la gestion des entrées/sorties avec un terminal opérateur est utilisée.

Remarque : Le système local de communication implanté sur chaque carte interface IBS est basé lui aussi sur un moniteur temps réel multitâches compatible avec RMX/80 d'INTEL et développé au L.A.A.S. [40] afin de s'adapter au matériel IBS (le noyau manipule les mêmes types d'objets, avec des primitives ayant les mêmes fonctions). Un système de gestion dynamique de la mémoire a aussi été ajouté au noyau car les messages peuvent séjourner un temps indéfini dans l'IBS avant d'être consommés.

Alors que l'utilisation d'un moniteur multitâches temps réel apporte une plus grande simplification dans la conception et l'implémentation du logiciel de communication des divers équipements, elle n'est pas du tout adéquate pour l'implémentation du logiciel de communication de l'observateur, vu le nombre réduit de fonctions à réaliser. De plus, le critère de fiabilité de l'observateur ne peut plus être satisfait. En effet, d'une part, la fiabilité de l'observateur repose principalement sur la confiance que l'on peut accorder à son mécanisme de réception des messages circulant sur le bus ; l'autre partie de l'observateur, le simulateur de réseaux de Petri, est simple, identique quel que soit le réseau observé et donc très fiable. D'autre part, la réception d'un message est gérée par un circuit spécialisé, le SIO, qui engendre une interruption après réception du début du message et une autre interruption à la fin de la réception mais le traitement de ces messages est réalisé à l'aide des primitives de synchronisation du moniteur qui

sont ininterruptibles. Ainsi, si un message circule sur le bus alors que le processeur est dans une zone protégée, il ne peut pas être reçu et la fiabilité de l'observateur s'en trouve donc diminuée.

C'est pourquoi, afin de rendre improbable, dès la conception, la non-réception d'un message par l'IBS de l'observateur, le traitement devenu séquentiel a été limité à la seule réception et au transfert vers le SBC 80/30 : chaque message reçu est traité complètement avant de passer au suivant. L'augmentation du temps d'exécution due à cette nouvelle architecture du logiciel a été compensée par l'écriture de toutes les procédures en langage assembleur.

IV.3. STRUCTURE DE L'OBSERVATEUR

Par défaut, nous appellerons observateur la partie traitement du logiciel d'autotest, c'est-à-dire que nous exclurons le logiciel d'acquisition des trames dans l'interface bus série.

Comme il a été vu au chapitre II, la fonction principale de l'observateur est la simulation d'un réseau de Petri en vue de réaliser une détection d'erreur et un diagnostic de fonctionnement du système.

Autour de cette tâche viennent se greffer toutes les tâches permettant de réaliser un système de test en ligne autonome, pouvant être implanté sans trop de modifications sur différents systèmes distribués temps réel.

Un des objectifs de notre méthode de test est la possibilité de sélection d'un niveau de vérification ou d'un mécanisme particulier, l'observateur doit donc permettre de réaliser cet objectif.

Afin d'en faire un outil de test en ligne plus complet, il doit, de plus, permettre de demander l'impression de résultats en fin de traitement ou activer (et désactiver) le suivi du test à n'importe quel moment au cours de l'exécution.

Ainsi, l'observateur doit supporter plusieurs modules :

- le simulateur de réseaux de Petri,
- l'interface avec le système de communication, chargé de recevoir les messages circulant sur le bus et transmis par l'IBS, de les formater et les filtrer ensuite, afin de ne prendre en compte que ceux correspondant au protocole à observer,
- l'interface avec un opérateur, chargé d'interpréter les commandes de l'opérateur et de lui permettre de sélectionner un des trois modes de dialogue possible : sélection de la configuration du système, traitement des résultats, suivi de l'observation. Il reçoit soit des commandes de l'opérateur, soit des valeurs de paramètres et lui transmet soit des résultats, soit des demandes de valeurs,
- le traitement statistique des résultats et leur stockage.

Le terminal handler est utilisé pour supporter la fonction d'interface avec un opérateur. De plus, une tâche particulière regroupant toutes les initialisations de variables est ajoutée à l'ensemble et une procédure réalisant la gestion des entrées-sorties avec l'IBS permet l'interface avec le système de communication.

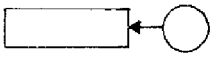
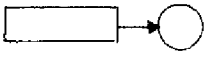
La structure modulaire du logiciel de l'observateur est représentée Figure IV.3. et les interactions entre les différents modules sont données Figure IV.4.

Nous allons maintenant nous attacher à décrire les différentes tâches dans chaque module, les fonctions réalisées.

IV.3.1. Interface opérateur

Tous les échanges entre l'opérateur et les différentes tâches constituant l'interface opérateur sont supportés par le terminal handler (T.H.) de RMX/80.

La structure de l'implémentation propre à l'interface opérateur est la suivante (Figure IV.5.).

Les tâches sont symbolisées par des rectangles, les échanges par des cercles. Le transfert de messages d'une tâche à un échange ou d'un échange à une tâche est représenté par une flèche. Ainsi, une tâche exécutant un RQWAIT sur un échange est symbolisée par  , une tâche exécutant un RQSEND par : 

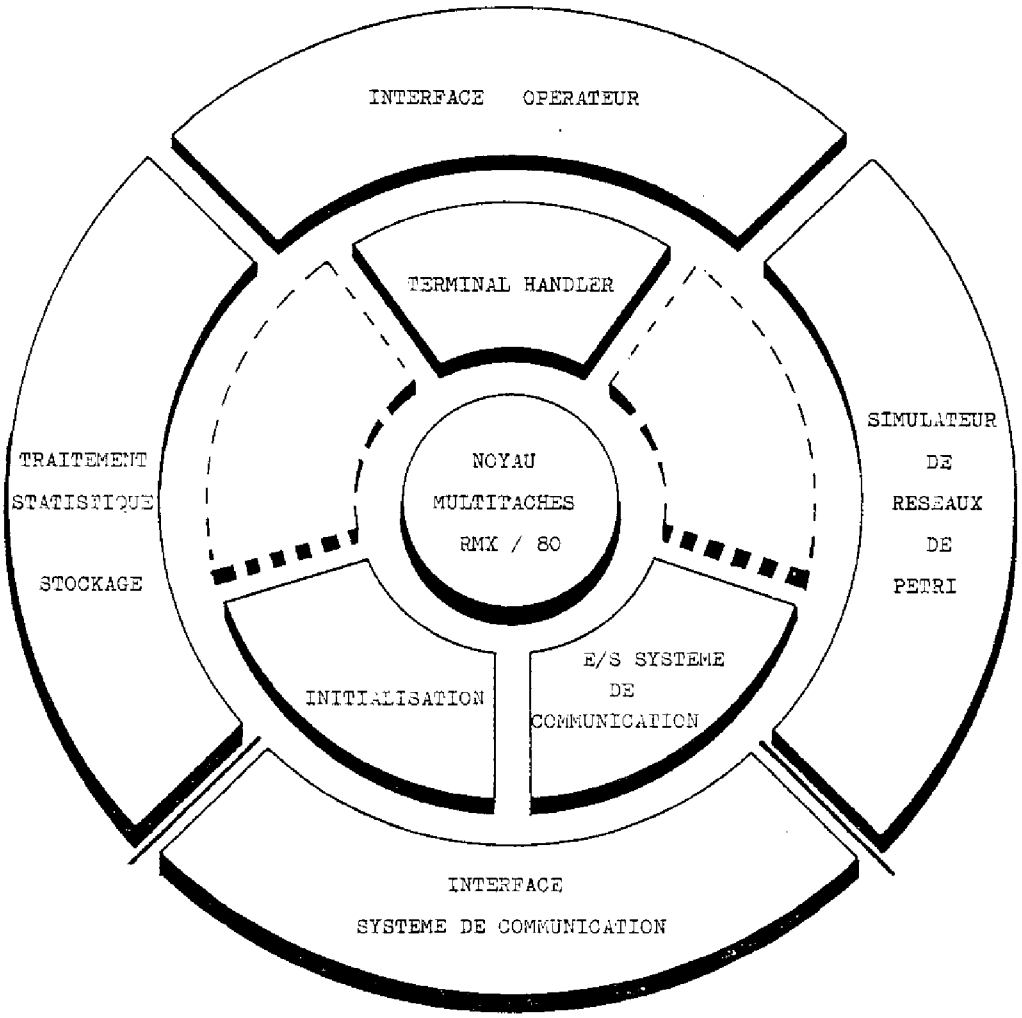


FIGURE IV.3. Structure de l'observateur : logiciel

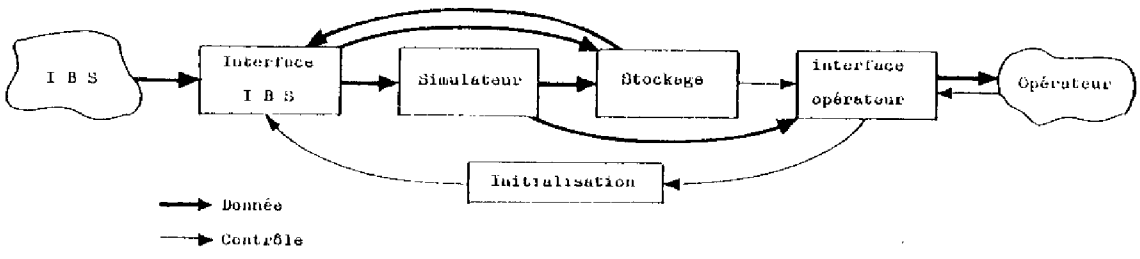


FIGURE IV.4. Interactions entre les différents modules

Les flèches en trait plein indiquent un flux de messages contenant de l'information,celles en trait pointillé un flux de messages de contrôle (ne contenant que l'en-tête nécessaire pour le noyau).

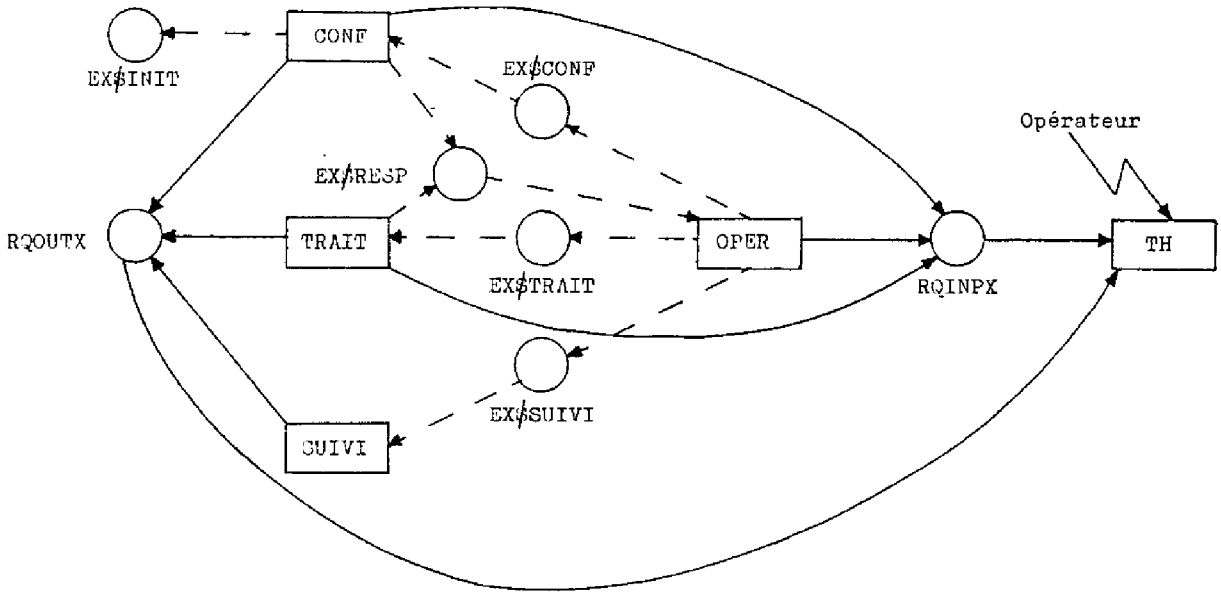


FIGURE IV.5. Interface opérateur

La tâche opérateur OPER a pour fonction d'aiguiller les commandes de l'opérateur vers les différentes tâches CONF, SUIVI ou TRAIT qu'elle rend alors actives (ou en attente).

Les commandes admises par la tâche opérateur sont :

C[ONFIGURATION]	demande de configuration
T[RAITEMENT]	demande de traitement des résultats
S[UIVI]	ordre de début ou de fin de suivi

Ces commandes peuvent être envoyées à n'importe quel moment en cours de fonctionnement.

La tâche CONF est une tâche conservationnelle permettant de définir la configuration du système global par sélection du niveau de protocole ou du mécanisme à observer, du nombre de processeurs actifs et de leur liste.

La tâche TRAIT fournit à l'opérateur, après lecture de la requête opérateur, soit une partie de l'historique du fonctionnement du système, les messages traités par l'observateur étant stockés dans un buffer circulaire par une autre tâche, soit des données statistiques sur le fonctionnement global. Bien évidemment, cette tâche ne sera pas activée souvent, elle le sera, par exemple, en fin de journée pour la composition d'un journal de bord du système global.

Le dialogue entre l'opérateur et la tâche TRAIT est :

```
soit   T[RAITEMENT]
       requête: H[ISTORIQUE]
       historique du fonctionnement

soit   T[RAITEMENT]
       requête: S[TATISTIQUE]
       statistiques
```

La tâche SUIVI est chargée de procurer à l'opérateur l'information utile contenue dans les messages captés, testés et stockés dans une zone mémoire, au fur et à mesure de leur stockage. Cette tâche peut être désactivée par l'opérateur par la même commande que celle qui la rend active.

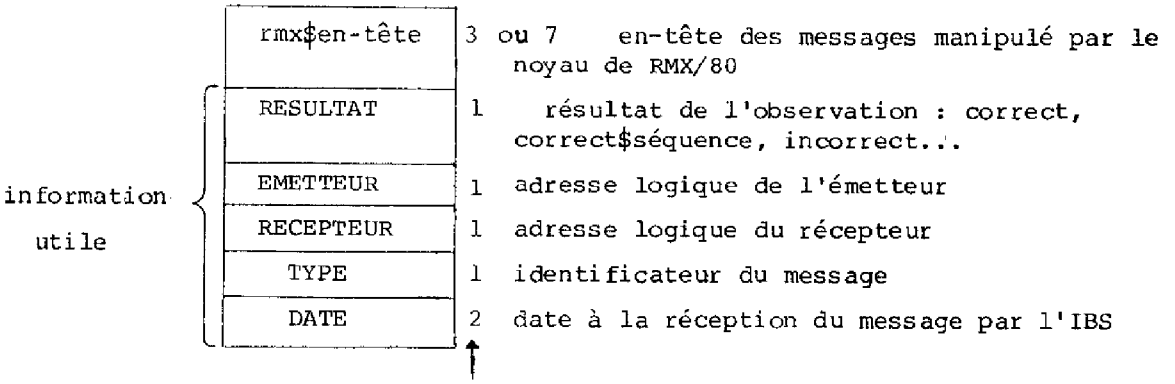
Ainsi, les tâches OPER, CONF et TRAIT font des demandes de lecture de paramètres ou de requête à travers le terminal handler. Afin d'être sûr que toute information demandée par une tâche est bien reçue par cette tâche, l'accès en lecture du terminal opérateur se fait en exclusion mutuelle à l'aide de l'échange de réponse EX$RESP. Il n'y a pas de problème particulier posé par l'accès en écriture ; toute tâche peut écrire quand elle le désire sur le terminal opérateur.

IV.3.2. Initialisation

La tâche d'initialisation INIT est chargée d'initialiser toutes les variables de l'observateur, et en particulier, la structure de données servant de base au simulateur du réseau de Petri et le marquage initial des différents processeurs actifs. De plus, elle crée des descripteurs de messages qui sont chaînées sur l'échange POOL$EX.

IV.3.3. *Interface avec le système de communication*

Les seuls messages, transitant entre tâches et échanges, qui contiennent de l'information utile pour l'observateur sont les "messages observateurs" et ont le format suivant :



Les messages captés par l'IBS sur le bus sont rangés dans deux buffers circulaires distincts selon que leur réception s'est effectuée de façon correcte ou non. Si la trame est correcte, seul l'en-tête du message qui nous intéresse dans le cadre de l'observation est rangé dans le buffer de messages corrects et à la structure suivante :

adresse logique du récepteur
identificateur du message
adresse logique de l'émetteur
longueur du message en octets
type application
type message

en-tête du message reçu par l'IBS

Lorsque la trame reçue est détectée incorrecte, seul le champ "status" indiquant la cause de la mauvaise réception : erreur de CRC, de recouvrement, buffer de réception plein,... est stocké dans le buffer de messages incorrects.

Ces messages sont transmis par l'IBS vers l'observateur.

La tâche DONNEE a pour fonction de recevoir tous les messages captés sur le bus et transmis par l'IBS vers l'observateur et de constituer, à partir des informations reçues, les messages observateurs. Ils peuvent être regroupés en deux classes :

- messages dont la réception par l'IBS a été correcte et qui sont envoyés à la tâche FILTRE en vue du test,
- messages dont la trame a été détectée incorrecte par l'IBS ou par la carte unité centrale (à cause d'un erreur de CRC par exemple) et qui sont envoyés sans être davantage traités à la tâche STOCK à travers l'échange EX$MESS$TOUS collectant tous les messages observateurs circulant dans le système.

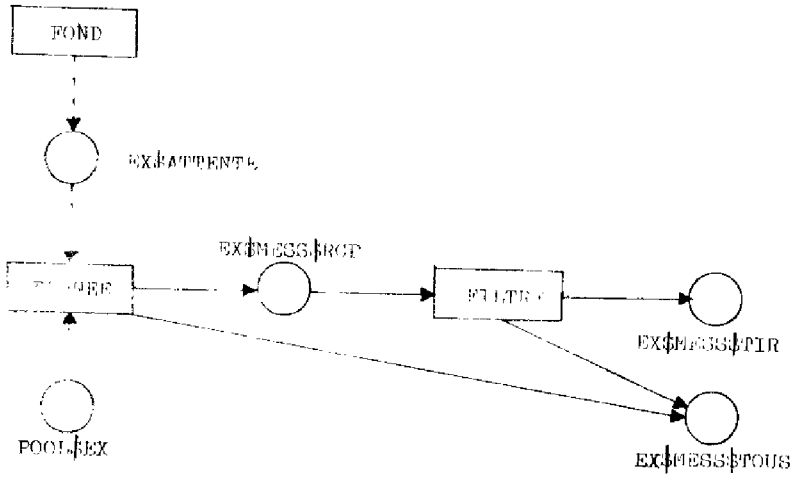


FIGURE IV.6. Interface avec le système de communication

La réception et la constitution d'un message ne peuvent se faire qu'après réception par la tâche DONNEE d'un descripteur de messages sur l'échange POOL$EX.

Le schéma d'implémentation de l'interface avec le système de communication est représenté sur la Figure IV.6.

La fiabilité de l'observateur reposant sur celle de sa partie réception et les pannes matérielles étant imprévisibles, il faut tout d'abord éliminer les causes de réception incorrecte dues à la conception

même de l'ensemble et en particulier, il faut éviter la saturation du buffer de réception de messages corrects de l'IBS. La tâche DONNEE doit donc être capable de recevoir les messages issus de l'IBS dès leur arrivée dans le registre de transfert interface-équipement.

Malheureusement, on ne peut pas disposer d'interruption dans un système construit autour du moniteur temps réel RMX/80 (primitives de synchronisation ininterruptibles). C'est pourquoi nous avons été amenés à exécuter un traitement cyclique des entrées-sorties, la tâche DONNEE est alors réactivée périodiquement par la tâche FOND afin de recevoir les messages issus de l'IBS.

Tous les messages reçus ne sont pas forcément destinés à l'observation, seuls sont soumis au test les messages du protocole choisi pour l'observation et mettant en communication des processeurs déclarés actifs par l'opérateur dans CONF. Ainsi, la tâche FILTRE réalise le filtrage des messages ; les messages à prendre en compte pour l'observation sont envoyés sur l'échange EX$MESS$TIR en vue de leur traitement par le simulateur de réseaux de Petri, les autres sont chaînés sur EX$MESS$TOUS et leur champ RESULTAT est mis à la valeur "non pris en compte". Cette tâche va, à l'initialisation, mettre à jour le marquage courant des différents processeurs après réception d'un message particulier du protocole.

IV.3.4. Simulateur de réseaux de Petri

C'est à ce niveau que le test proprement dit est exécuté. La tâche SIMUL implémente un simulateur de réseaux de Petri et fait appel à une procédure GRAMAR chargée d'affiner le test par la recherche d'un état cohérent.

Au chapitre précédent, nous avons choisi une modélisation de l'émission-réception qui, en appliquant la règle d'évolution, fait évoluer de façon simultanée les états respectifs des deux processeurs impliqués dans la communication. Nous avons montré, dans le cas de deux processeurs communicants, quelle était l'influence de la perte d'un message par l'observateur, une méthode de recherche d'un état cohérent était alors proposée.

Dans le réseau REBUS, plusieurs processeurs sont reliés au support de communication. Dans ce contexte, la perte par l'observateur d'un message α circulant par exemple entre deux processeurs P1 et P2 conduit à deux cas possibles.

.Si le message suivant, β , reçu par l'observateur est un message allant de P2 vers P1 (on se trouve dans le cas présenté au § III.1.2.), la communication semble n'avoir lieu qu'entre ces deux processeurs.

.Par contre, si ce message suivant reçu par l'observateur est un message du processeur P2 vers le processeur P3, l'observateur va le détecter "incorrect" alors qu'il est correct pour le processeur P3 et qu'une recherche de l'état cohérent pour le processeur P2 le rendrait "correct après une séquence de tir" pour ce processeur. De la même façon, lorsque le processeur P1 sera à nouveau en communication avec un processeur PN, le diagnostic rendu par l'observateur devra être "message correct après une séquence de tir pour le processeur P1".

Ainsi, afin de ne pas introduire d'erreurs supplémentaires dans le diagnostic et d'être plus proche du fonctionnement réel du système global, il paraît judicieux de rendre indépendante l'évolution du processeur émetteur de celle du processeur récepteur, d'autant plus que chaque processeur est caractérisé par son marquage courant.

Cela nous a conduit à tester puis à faire évoluer l'état de chaque processeur indépendamment de l'autre. L'exécution de la tâche SIMUL consiste donc à dérouler en parallèle pour chaque entité communicante l'algorithme de test présenté au Chapitre III, Figure III.8.

Le diagnostic rendu par l'observateur sur l'état global du système est une combinaison des diagnostics partiels obtenus sur chaque processeur.

En faisant cette séparation, le test d'un message dans l'exécutant fournit alors des diagnostics résultant des différents fonctionnements possibles des processeurs émetteur, récepteur ou de l'observateur.

Ces fonctionnements peuvent être regroupés en deux classes : fonctionnement CORRECT et fonctionnement INCORRECT.

Dans chaque classe, les différents diagnostics sont les suivants :

correct

- correct ⟺ les processeurs émetteur et récepteur fonctionnent correctement
- correct$seqSrcp ⟺ il y a incertitude sur le message précédent qui transitait entre le récepteur du message testé et un autre processeur
- correct$seqSem ⟺ il y a incertitude sur le message précédent qui transitait entre l'émetteur du message testé et un autre processeur
- correct$seq ⟺ il y a incertitude sur le message précédent qui transitait entre l'émetteur et le récepteur du message testé

incorrect

- incorrect$rcp ⟺ le processeur récepteur du message se comporte de façon erronée
- incorrect$em ⟺ le processeur émetteur du message se comporte de façon erronée
- incorrect ⟺ les processeurs émetteur et récepteur du message se comportent de façon erronée.

Ces résultats de test sont rangés dans le champ RESULTAT du message observateur.

Quel que soit le résultat de l'observation, les messages sont envoyés sur l'échange EX$MESS$TOUS pour être stockés dans une zone mémoire.

Les liens entre les différentes tâches du module simulateur de réseaux de Petri sont représentés sur la Figure IV.7.

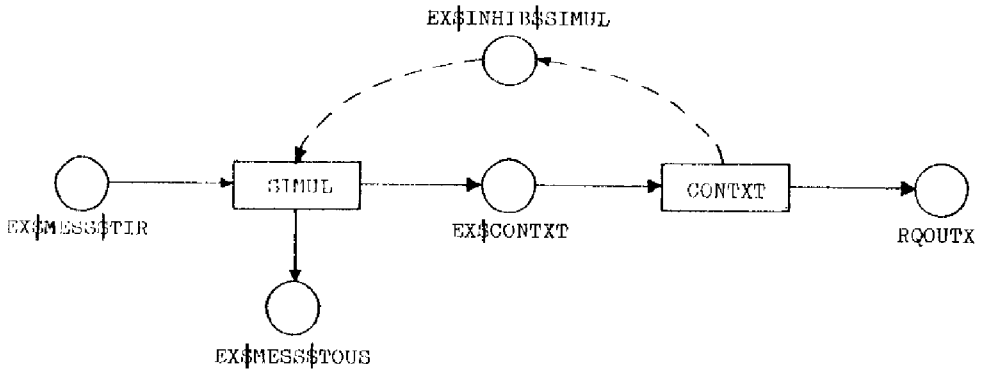


FIGURE IV.7. Simulateur de réseaux de Petri

Lorsque l'observateur diagnostique un fonctionnement erroné des processeurs émetteur et/ou récepteur, il se doit d'avertir l'opérateur d'une erreur dans le système.

C'est la tâche CONTXT qui est chargée d'alerter l'opérateur en cas d'erreur, de lui fournir le contexte de l'erreur : historique immédiat précédant (et si besoin, suivant) l'erreur, état du système,... et de mettre le processeur détecté en panne (ou tous les processeurs) dans un état cohérent.

Fournir à l'opérateur l'historique immédiat consiste, pour la tâche CONTXT, à écrire sur le terminal opérateur un certain nombre (définissant la "largeur de la fenêtre" du contexte) de messages précédant et suivant le message incorrect.

IV.3.5. Le stockage des messages

Le stockage de toute l'information utile contenue dans les messages reçus par l'observateur est effectué dans une zone mémoire de 1K octets, gérée en buffer circulaire.

Il est réalisé par une seule tâche, STOCK, qui tient une comptabilité des messages reçus en fonction de leur champ RESULTAT.

IV.4. EXEMPLES

IV.4.1. REBUS reconfiguration

Une implémentation des différentes tâches constituant l'observateur a été réalisée, dans le cadre de la recherche effectuée au L.A.A.S., sur le système distribué de communication, sûr de fonctionnement REBUS.

Parmi l'ensemble des protocoles possibles, nous avons choisi, dans un premier temps, d'observer l'aspect de base de la communication, c'est-à-dire le protocole de passage du statut maître défini au chapitre précédent.

Dans un deuxième temps, notre système permettra de mettre au point un autre protocole, le protocole du niveau transport, qui est en cours d'étude.

Le temps moyen de traitement d'un message, depuis la prise d'un descripteur de message jusqu'à sa libération est de 15 ms.

Dans l'état actuel, l'occupation mémoire du système est de :

		Code	Donnée
	Module de configuration	0,2 K	1,5 K
OBSERVATEUR	INIT	4	
	DONNEE	0,7	
	FILTRE	0,8	2 K
	CONF	2,2	
	FOND	0,1	
	SIMUL	1,8	1 K
	SUIVI	0,7	
	STOCK	0,4	
	TRAIT	0,4	1,3 K
	CONTXT	1	
	OPER	0,2	
Noyau RMX/80		1,9	0,2
Terminal handler		3,1	0,9
		17,3 K	6,9 K

Dans une première approche du problème, la tâche SIMUL réalisant la fonction de base de l'observateur : le simulateur de réseaux de Petri, présentée Figure IV.2., paraît simple. Mais, par la suite, l'adjonction de nombreuses tâches indispensables pour la réalisation un système complet de surveillance en ligne rend cette tâche de simulation relativement peu importante (1/10ème du système total).

Constatations après utilisation.

Lorsqu'il y a incertitude sur un message transitant sur le bus, le temps d'exécution de la procédure GRAMAR est relativement long par rapport au temps de traitement normal d'un message. Si le temps entre deux réceptions de messages est inférieur au temps de traitement de cette procédure, il y a un risque d'avoir saturation du buffer de réception de l'interface IBS. Pour éviter ce travers, il est possible d'envisager deux solutions :

- filtrer les messages au niveau de l'IBS ; seul un sous-ensemble de messages est observé et par voie de conséquence, on augmente le temps d'arrivée des messages "utiles" pour l'observateur,
- réécrire en assembleur la procédure GRAMAR actuellement écrite en PL-M/80 ; le gain sur le temps de traitement devrait être supérieur à 2.

La tâche SUIVI ne devra être activée que pendant de courts intervalles de temps afin de ne pas perturber le fonctionnement de l'observateur. En effet, lorsqu'elle est activée, son exécution et le mécanisme de synchronisation avec STOCK décrit dans l'annexe augmente considérablement le temps de traitement de chaque message dans l'observateur, entraînant ainsi une non-réception possible de messages.

La structure modulaire de l'ensemble de l'observateur rend possible la transformation de cette tâche pour permettre, par exemple, un suivi de l'observation à des intervalles de temps réguliers.

IV.4.2. Modumat 800

Une deuxième version est actuellement en cours de développement pour le système distribué temps réel de contrôle et commande de processus industriel MODUMAT 800 commercialisé par la Société SCHLUMBERGER. La communication dans ce réseau est implanté sur le même matériel que REBUS et le système a la même structure.

Le protocole de gestion et d'allocation de ligne est plus simple que dans REBUS : le statut maître est centralisé dans un seul patron appelé ici contrôleur (ce contrôleur peut tomber en panne et avoir un comportement erroné ; il existe un autre contrôleur qui le surveille ; en cas de panne, l'opérateur est averti et commute les deux contrôleurs). Le contrôleur constitue le mécanisme de séquencement global du système.

Un cycle contrôleur démarre chaque 500 ms. Tout d'abord, le contrôleur scrute tous les autres processeurs (SURVM0). En réponse, chaque processeur indique s'il a des événements à faire circuler sur le bus (indication de changement d'état de régulation, d'alarmes) et le temps de parole qui lui est nécessaire (SURVM1). Le contrôleur acquitte alors ce message (SURVM2) s'il contenait une demande de temps.

Après cette phase de scrutation, une phase de compte rendu permet au contrôleur d'envoyer à certains équipements le compte rendu des événements (CREM0) et éventuellement d'en attendre une réponse (CREM1).

Ensuite, la phase de délégation peut débuter. Le contrôleur délègue son statut primaire à un ouvrier l'ayant demandé pendant la phase de scrutation et devient secondaire (DELEGM0). L'ouvrier devenu primaire dialogue sur le bus avec les autres stations (toutes secondaires) suivant une procédure d'appel-réponse. Quand il a reçu les réponses à toutes ses requêtes ou que son temps de parole alloué pour la délégation est terminé, il renvoie la délégation au contrôleur et redevient secondaire (FDELEGM0). Le contrôleur envoie, à nouveau, la délégation à l'ouvrier suivant sur l'anneau. En fin de cycle contrôleur ou lorsque tous les équipements l'ayant demandé sont devenus primaires, le statut primaire est rendu au contrôleur en attente d'un nouveau cycle. Un cahier des charges plus détaillé de ce système est présenté dans [41].

La structure hiérarchique de l'ensemble est donnée Figure IV.8.

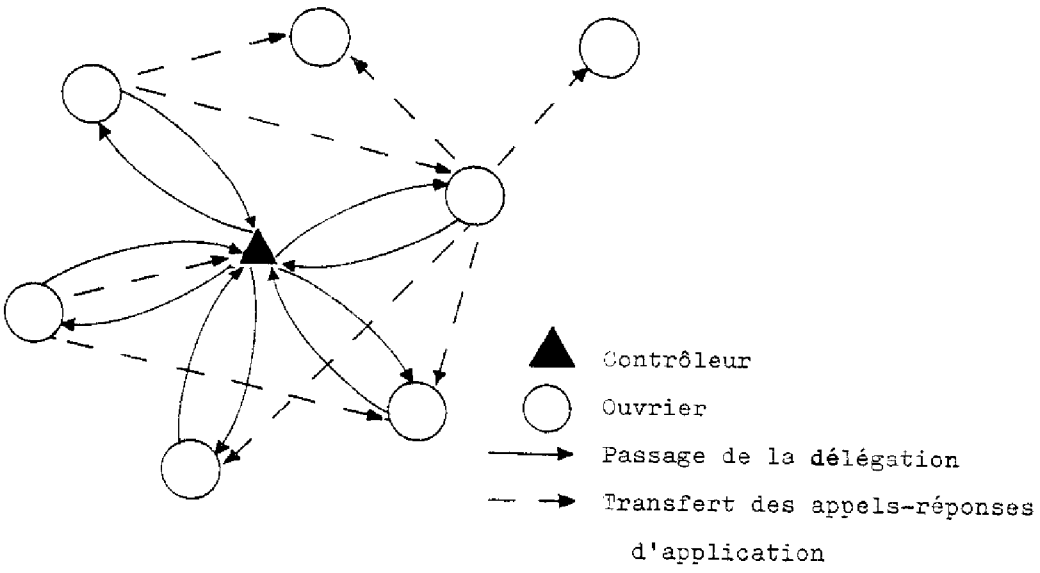


FIGURE IV.8. Structure hiérarchique

A partir de ces spécifications, le réseau de Petri modélisant le diagramme de mission de l'observateur est représenté Figure IV.9.

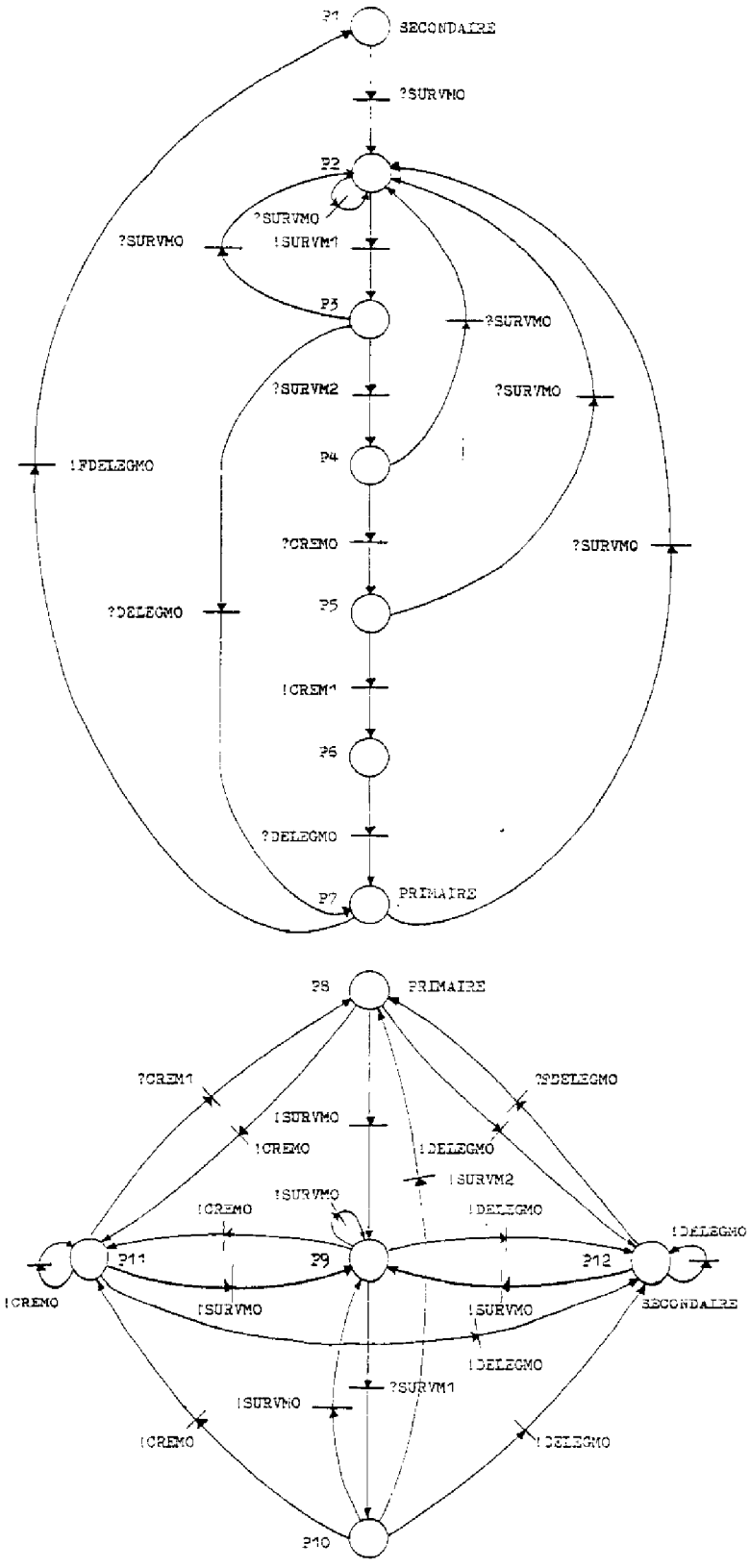


FIGURE IV.9. Diagramme de mission de l'observateur dans le système MODUMAT 800

Ce réseau permet de vérifier le séquencement des messages dans le système MODUMAT 800. Il est caractérisé par un nombre peu élevé d'états pour le contrôleur et l'équipement mais, par contre, par un nombre important de transitions ; ceci est dû au fait que certaines phases (de compte rendu ou de délégation) peuvent ne pas avoir lieu (si l'équipement n'a pas demandé de compte rendu ou la délégation lors de la phase de scrutation).

Afin d'adapter l'observateur de REBUS au système MODUMAT 800, quelques modifications devront être apportées.

En effet, dans ce système, le séquencement des messages se fait à une cadence très élevée (environ un message par milliseconde). Ainsi, un problème important à résoudre est la réception et le traitement d'un message pendant cet intervalle de 1 ms.

Pour s'affranchir de la perte de temps due au transfert de l'information entre l'IBS et l'observateur, on utilisera le transfert par DMA ; ainsi, les deux processeurs seront rendus indépendants et plus aptes à recevoir tous les messages circulant sur le support de communication.

De plus, les différentes tâches seront écrites en langage assembleur afin de réduire le temps global de traitement d'un message.

IV.5. CONCLUSION

Ce chapitre nous a permis de donner des détails sur l'implémentation du logiciel d'auto-test.

Le choix d'un moniteur temps réel multitâches pour coordonner l'ensemble des tâches réalisant les fonctions du système s'est avéré très judicieux car il permet, lors de la phase de mise au point, soit de modifier les fonctions d'une tâche sans modifier les autres tâches, soit de définir de nouvelles relations entre les tâches et de changer les priorités sans affecter les fonctions de chaque tâche.

Les études effectuées en vue d'implanter l'observateur sur deux systèmes distribués a permis de mettre en évidence que, tout en étant centré autour du même simulateur de réseaux de Petri, l'observateur pouvait devoir réaliser des fonctions différentes au niveau par exemple du traitement statistique de messages, de leur stockage.

Nous avons donné la description des fonctions désirées dans le cadre de l'observation du réseau REBUS ; elles dépendent de besoins de l'utilisateur et du système.

L'observateur a été utilisé pour mettre au point le protocole d'allocation de ligne dans le réseau REBUS. Dans le système MODUMAT 800, il doit permettre de détecter en ligne les pannes intervenant dans le système mais dont les effets ne sont pas visibles car le système s'est reconfiguré de lui-même de façon correcte.

CONCLUSION

Nous avons présenté, dans ce mémoire, une approche permettant la détection en ligne, dans les systèmes distribués, des erreurs dues à des fautes matérielles, logicielles ou humaines et affectant le fonctionnement normal du système. Elle repose sur l'utilisation de deux logiciels distincts constituant un couple exécutant-observateur. L'exécutant réalise de manière classique les tâches fonctionnelles du système, l'observateur implémente un simulateur de réseaux de Petri.

Une étude des différentes techniques existantes permettant l'auto-test du fonctionnement d'un système a mis en évidence,dans le chapitre I,leurs limitations dans des structures distribuées temps réel. Ceci nous a conduit à proposer une méthode de conception d'un logiciel redondant d'auto-test dans les systèmes distribués temps réel.

Après avoir choisi et présenté un modèle de représentation du diagramme de mission de l'observateur, les réseaux de Petri, le chapitre II a défini les relations entre les deux composantes du logiciel redondant : l'exécutant et l'observateur.

Le chapitre III a proposé une méthodologie de modélisation du diagramme de mission de l'observateur qui tienne compte de la nature distribuée du système et donc de la perte possible d'un message par un équipement devant le recevoir. Elle a ensuite été appliquée au système distribué de communication, REBUS.

L'implémentation de l'observateur dans ce système a été détaillée dans le chapitre IV ; nous avons ainsi défini les supports matériel et logiciel et décrit les différentes tâches composant l'observateur et leurs relations.

Plusieurs caractéristiques de l'approche que nous avons développée sont intéressantes. D'une part, l'effort de modélisation du diagramme de mission est réduit au minimum dans la mesure où le modèle est obtenu à partir de la phase de formalisation, phase nécessaire pour une meilleure compréhension du système lors de la conception. D'autre part, lorsque

l'observateur ne fait que contrôler, de façon invisible, l'exécutant, ce qui est le cas que nous avons étudié dans ce travail; le logiciel redondant introduit dans le système ne réduit pas la fiabilité de l'ensemble car il est lui-même extrêmement fiable :

- le programme réalisant le coeur de l'observateur est un simulateur de réseaux de Petri, simple et identique quel que soit le réseau à observer,
- le réseau de Petri modélisant le diagramme de mission du système a été analysé et prouvé correct (sauf ou borné, vivant) avant d'être utilisé,
- la structure de données sur laquelle se déroule le simulateur est dérivée, de façon systématique et simple, de celle qui a été utilisée lors de l'étape de validation,
- l'introduction, dans le système, du logiciel redondant n'affecte pas le logiciel de l'exécutant et donc sa fiabilité, ce qui n'avait jamais été réalisé dans les approches existantes.

Enfin, un autre intérêt essentiel de cette approche réside dans le fait que, conceptuellement, l'observateur réalisé peut être implanté sur toute structure distribuée.

L'approche présentée a été appliquée pour mettre au point un protocole de gestion et d'allocation de ligne de REBUS, un système distribué sûr de fonctionnement reposant sur une structure à bus série ; elle devra permettre la mise au point du protocole du niveau transport dans ce réseau ; elle est en cours de réalisation dans le système MODUMAT 800 pour l'observation en ligne du comportement de ce système industriel commercialisé par SCHLUMBERGER-EUROPE.

La structuration en tâches concurrentes du logiciel de l'observateur permet son évolution aisée en fonction des exigences des utilisateurs. On peut penser, par exemple, rajouter une carte permettant les entrées-sorties sur un système à disquette en vue d'un stockage plus important des informations reçues et traitées. Un système de traitement pouvant avoir accès aux informations stockées sur les disquettes permettrait de faire des mesures sur les probabilités de panne dans le système.

Un développement intéressant de ce logiciel d'auto-test serait de prendre en compte la diffusion des messages lors de la modélisation du diagramme de mission et lors du test. Le simulateur de réseaux de Petri ne ferait alors pas évoluer 2 processus mais N processus. On peut également concevoir de modéliser le comportement global du système dans un seul diagramme de mission. Les transitions ne seraient alors plus étiquetées !message et ?message mais l'étiquette contiendrait de l'information sur le message, le processeur émetteur et le processeur récepteur ; le simulateur de réseaux de Petri ne ferait alors plus évoluer qu'un seul réseau.

Ces deux cas, appliqués au protocole d'allocation et de gestion de ligne du réseau REBUS permettraient tous les deux contrôler la place de chaque patron dans l'anneau virtuel et non plus uniquement le séquencement des messages entre les différents processeurs. Dans le cas général, ceci nous conduirait à faire des diagnostics sur des mécanismes particuliers du protocole observé. Il est enfin possible de penser à de nombreuses extensions au niveau du traitement qui, si les contraintes adéquates de rapidité étaient remplies, permettraient d'étendre l'observateur afin d'en faire un outil complet de surveillance et de mesures des protocoles dans les systèmes distribués.

ANNEXES

. COMMUNICATION CARTE IBS - CARTE SBC 80/30

. EXEMPLES D'UTILISATION

-

ANNEXE 1. COMMUNICATION CARTE IBS ET CARTE SBC 80/30

La communication entre la carte IBS et la carte unité centrale est effectuée sur un bus parallèle 8 bits.

Le transfert des messages de l'interface vers l'unité fonctionnelle observateur est régi par la structure du matériel de l'interface processeur arrière de l'IBS.

Nous rappelons brièvement cette structure, développée largement dans [29], afin d'expliquer le fonctionnement de la communication (Figure A.1.).

Deux registres de 8 bits se comportent comme des registres d'entrée et de sortie. Un registre d'état est associé à chacun de ces registres, trois de ses bits indiquent l'autorisation de lecture, d'écriture dans le registre et si la valeur est de type commande ou donnée.

Le fonctionnement de base du transfert de données est le suivant : tout processeur (interface ou équipement connecté) doit avoir l'autorisation de lecture (écriture) avant de pouvoir lire (écrire dans) un registre. Il autorise alors l'autre processeur à écrire (lire) dans ce même registre.

Les algorithmes de lecture et d'écriture sont les mêmes pour un équipement que pour l'interface (Figure A.2.).

En général, l'équipement et l'interface communiquent de façon asynchrone, il faut donc assurer la synchronisation entre ces deux processeurs. Nous avons ainsi été amenés à définir un protocole de communication.

Toute communication entre deux processeurs peut se décomposer en trois phases : phase d'initialisation de la liaison, phase de transfert de l'information, phase de déconnexion [36]. Ceci nous a conduit à implémenter la communication de chaque processeur en différentes tâches : les tâches handler de réception et handler d'émission et la tâche synchronisation des transferts (Figure A.3.).

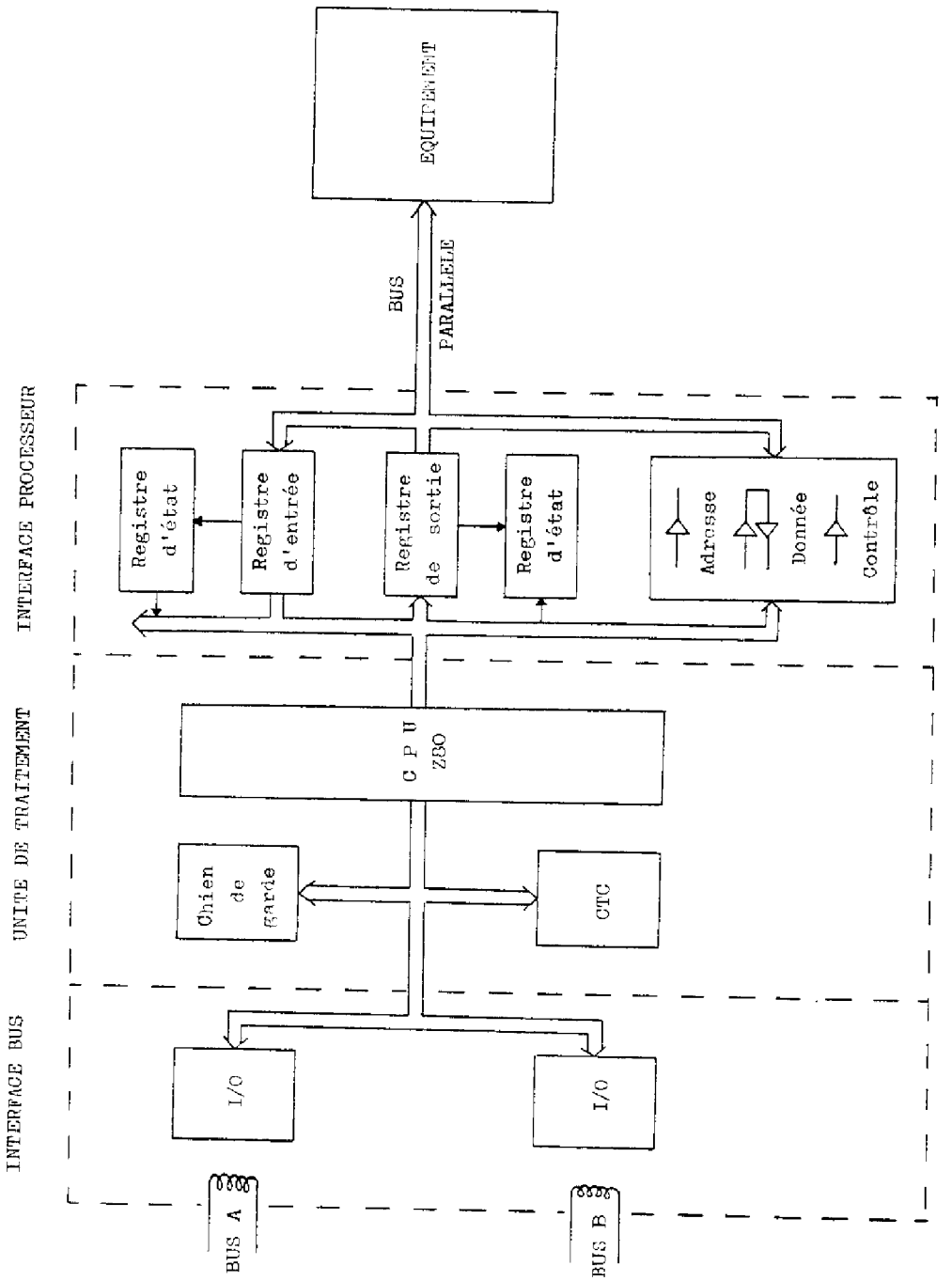


FIGURE A.1. Schéma de la carte Interface Bus Série

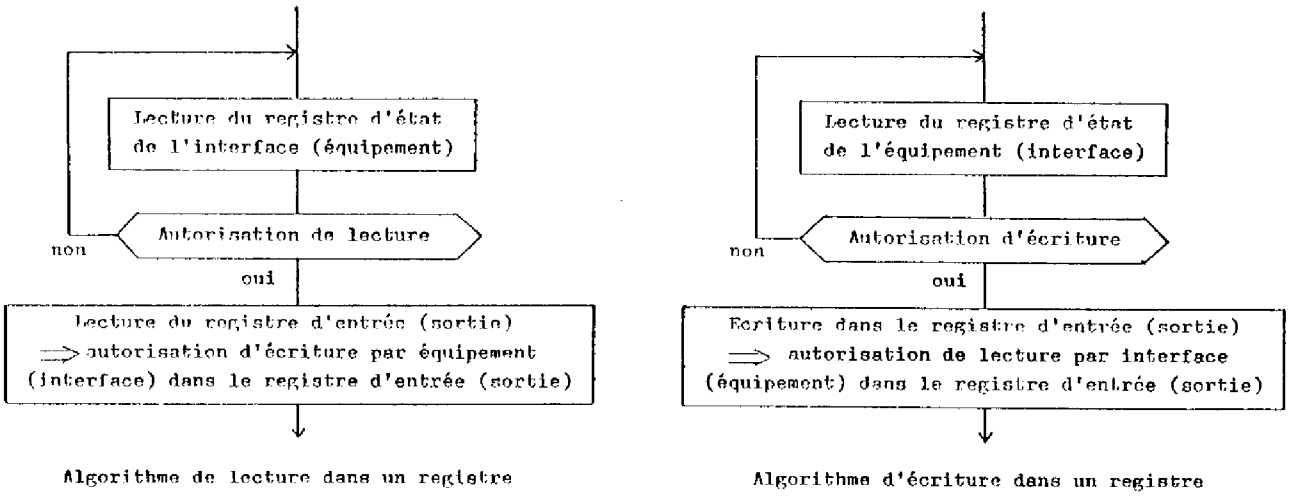


FIGURE A.2. Algorithme de lecture et d'écriture

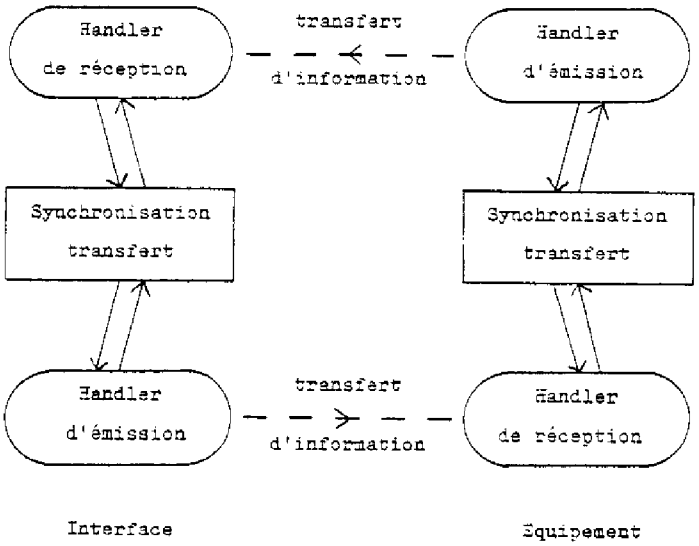


FIGURE A.3. Communication Interface - Equipement

Le handler d'émission signale à la tâche synchronisation qu'il a des informations disponibles à émettre, indique leur type afin que le handler de réception associé pour la communication décide s'il est capable de recevoir ces informations et de les stocker en mémoire. Le handler d'émission a ensuite la charge d'émettre l'information vers l'entité communicante, puis de recevoir un acquittement (ou un non-acquittement) en cas de réception correcte (ou incorrecte).

La tâche du handler de réception consiste, tout d'abord, à signaler à la partie synchronisation sa disponibilité à recevoir puis, bien évidemment, consiste à recevoir ces informations, les ranger en mémoire en vue du traitement et envoyer un acquittement en cas de réception correcte.

La tâche de synchronisation des transferts implémente, quant à elle, le protocole de communication entre les deux processeurs ; son rôle est de permettre les échanges d'informations de l'un vers l'autre, c'est-à-dire d'initialiser la communication et de la terminer, compte tenu du fait que l'équipement a la priorité du transfert d'informations en cas de simultanéité de demande de transfert par les deux processeurs.

La synchronisation des deux processeurs est réalisée par émission/réception de mots de contrôle. Ceux-ci sont inspirés des mots de contrôle utilisés dans la structure HDLC.

DMTF : demande de transfert
CMRJ : rejet de la commande
DISC : déconnexion (fin de transfert)
UA : acceptation (unnumbered acknowledgment)

Lorsque DMTF est envoyé à une entité, celle-ci détermine si elle est prête à recevoir ou non.

DMTF et DISC sont émis par une entité primaire vers une entité secondaire ; CMRJ et UA sont les réponses.

Le réseau de Petri modélisant la synchronisation des transferts est donné Figure A.4.

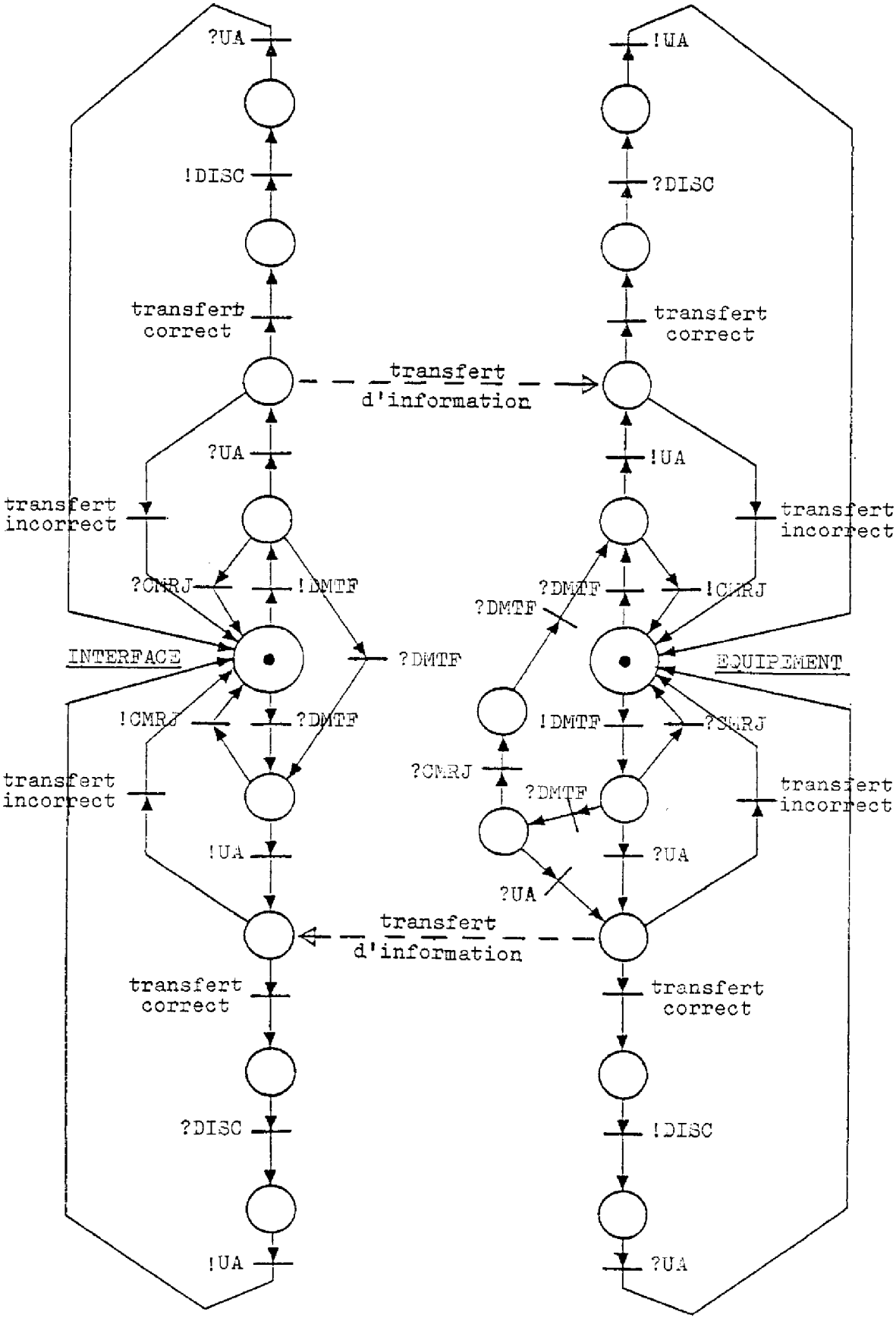
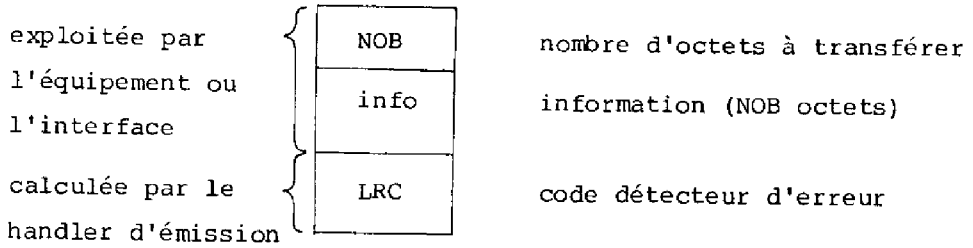


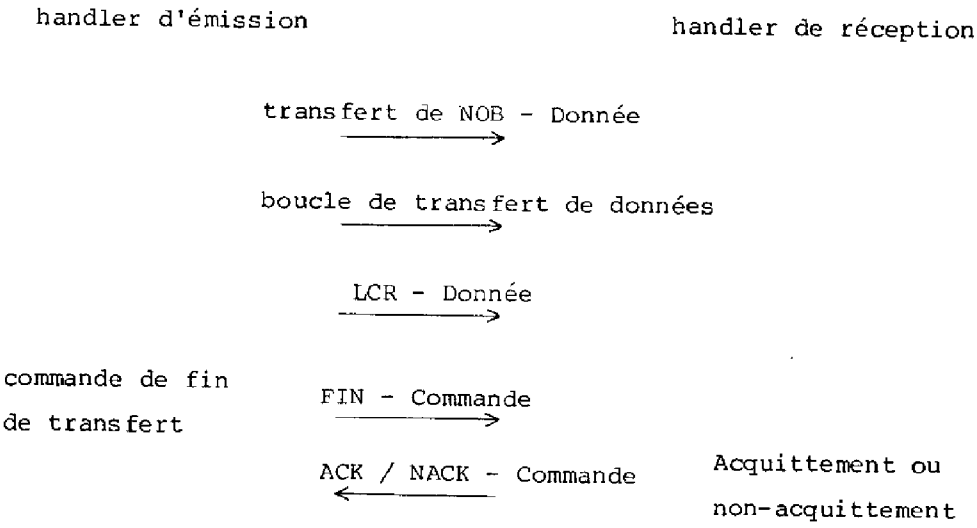
FIGURE A.4. Synchronisation des transferts

De façon générale, le format de l'information à transférer est le suivant :



Tous ces octets sont des mots de données.

Les handler d'émission et/ou de réception ont la charge d'assurer le transfert de l'information d'un processeur à un autre. La procédure d'échange est la suivante :



FIN, ACK et NACK sont des mots de commande, FIN avertit le récepteur de la fin du transfert. Si le transfert a été correct pour le handler de réception, celui-ci l'acquitte en envoyant ACK et le transfert est terminé ; sinon il répond NACK et le transfert est réitéré jusqu'à trois fois. Au bout de trois fois, on engage une procédure d'erreur.

Remarque : Un mot de commande n'est pas nécessaire pour indiquer le début de transfert d'information car la synchronisation a déjà été réalisée par DMTF - UA ; on sait après cela que le prochain mot à recevoir est un mot de donnée.

Dans le cas particulier où l'unité observateur ne nécessiterait que des échanges dans le sens interface-équipement, le protocole de communication pourraît devenir très simple.

Les handler d'émission et de réception se simplifient afin de satisfaire le schéma de transfert suivant :

transfert de NOB - commande →
boucle de transfert - donnée →
LRC - donnée →

L'octet NOB est alors un octet de type commande et c'est le seul.

De cette façon, la synchronisation du transfert est réalisée, à l'initialisation, par l'attente de réception d'un mot de type commande.

Il n'y a ensuite plus besoin de procédure élaborée de synchronisation car la synchronisation est maintenue par la structure même du matériel : l'écriture par l'interface dans le registre de transfert n'est permise que lorsque la lecture par l'équipement a été effectuée.

La sûreté des transferts est assurée à la fois par la comparaison du nombre d'octets reçus à NOB et par la concordance des codes détecteurs d'erreur, reçu et calculé, par l'équipement observateur.

ANNEXE 2. EXEMPLES D'UTILISATION

A.2.1. Structure de la base de données

Un des objectifs de notre méthode est la possibilité de sélection d'un niveau particulier de protocole. La base de données de l'observateur doit permettre la réalisation de cet objectif. Elle contient la configuration maximale du système, c'est-à-dire le nombre maximal de processeurs connectés au support de communication et pouvant communiquer entre eux. Elle contient de plus le nombre de niveaux de protocoles différents pouvant être observés, à chaque protocole est associé un réseau de Petri.

Chaque réseau est caractérisé par les variables suivantes :

NP	nombre de places
NT	nombre de transitions
NPE	nombre de places d'entrée d'une transition
NPS	nombre de places de sortie d'une transition
NPR	nombre de processeurs impliqués dans le protocole
NM	nombre de messages du protocole
DPAR	degré de parallélisme du système, c'est-à-dire la valeur maximum, pour tous les messages du protocole, du nombre maximum de transitions étiquetées !message et ?message pour chaque message. Cette variable permet de retrouver pour chaque message les numéros des transitions ! et ?

La description du réseau est réalisée grâce à deux matrices : MATE(NT,NPE) matrice d'entrée d'une transition et MATS (NT,NPS) matrice de sortie, telles que à chaque transition est associé respectivement le numéro des places d'entrée et le numéro des places de sortie ; ces matrices sont complétées avec des zéros lorsqu'il n'existe plus d'arc reliant une place à une transition.

De plus, à chaque matrice MATE et MATS est associée respectivement une matrice MATE2 et MATS2 de même dimension et indiquant le poids des arcs reliant les places aux transitions.

Afin d'assurer une plus grande fiabilité à l'observateur, ces structures sont obtenues, de façon simple, à partir de la structure de données sur laquelle l'analyse a été faite, dans OGIVE.

Pour chaque niveau de protocole, chaque message est caractérisé par un identificateur et les listes des transitions correspondant à l'envoi de ce message et à sa réception.

L'identificateur d'un message contient à la fois une indication sur le protocole dont il fait partie (appelé TYPE) et un numéro l'identifiant dans le protocole.

Exemple : pour le protocole de gestion de ligne, TYPE=60H le message ACCEPTEZ a pour indice 1 dans le protocole, ainsi son identificateur est 61H.

Ainsi, avant toute observation, il faudra définir la configuration du système et pour cela, sélectionner le type du protocole que l'on désire tester, le nombre de processeurs réellement actifs, leur indice.

A.2.2. *Description du fonctionnement général*

La description des fonctions réalisées par les différentes tâches a été donnée dans le Chapitre IV. Nous allons maintenant donner une description des principales phases de fonctionnement de l'observateur.

Le schéma d'implémentation du fonctionnement général est celui de la figure A.5.

Les échanges MUTEX et MUTEX2 permettent l'accès en exclusion mutuelle respectivement à la zone mémoire de stockage des messages reçus, par STOCK, SUIVI, TRAIT et CONTXT et aux variables du système (marquages courants,...) par FILTRE, SIMUL, CONF et CONTXT.

A.2.2.a. *Phase d'initialisation ou de réinitialisation*

A l'initialisation, toutes les tâches sont en attente sur leurs échanges de réponse.

L'opérateur, après avoir synchroniser le terminal opérateur et la carte SBC 80/30 par une série de U tapée sur la console opérateur peut, à travers la tâche OPER, activer la tâche CONF.

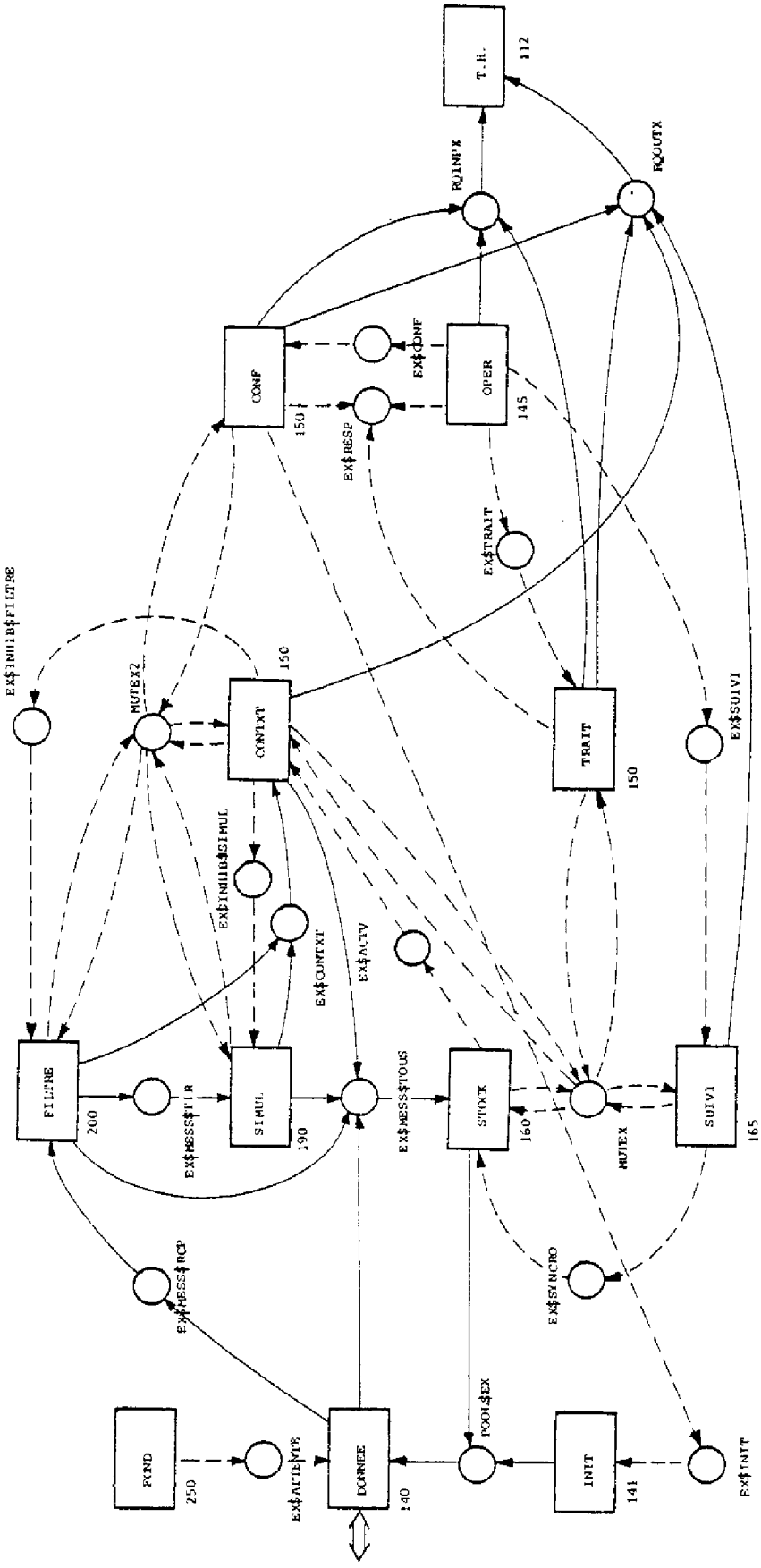


FIGURE A.5. Schéma d'implémentation général

La tâche CONF, après s'être exécutée, va à son tour activer la tâche INIT en attente sur EX$INIT, réactiver la tâche OPER en attente sur EX$RESP et se remettre en attente d'activation par OPER sur l'échange EX$CONF. Un exemple de configuration du système est :

```
type de message: G
nombre de processeurs actifs: 3
processeur actif (1): 1   identificateur du premier processeur actif
processeur actif (2): 2
processeur actif (3): 3
```

La tâche INIT crée des descripteurs de message qui sont chaînés sur l'échange POOL$EX.

A.2.2.b. *Trajet d'un message*

Dès que la tâche DONNEE reçoit un descripteur de messages sur POOL$EX, elle peut recevoir un message issu de l'IBS et constitue alors un message observateur qu'elle envoie sur l'échange correspondant EX$MESS$RCP (message dont la trame est correcte, tel que RESULTAT=ØFFH) ou EX$MESS$TOUS (message dont la trame a été refusée, tel que RESULTAT=status de l'erreur,EMETTEUR=RECEPTEUR=TYPE=Ø).

Le message envoyé sur EX$MESS$RCP est filtré par la tâche FILTRE puis chaîné, soit sur EX$MESS$TIR pour activer le simulateur de réseaux de Petri, soit sur EX$MESS$TOUS pour être stocké dans une zone mémoire.

La tâche SIMUL en attente sur EX$MESS$TIR va tester le message reçu et l'envoyer à la tâche STOCK en attente sur EX$MESS$TOUS quel que soit le résultat du test.

La tâche STOCK va alors ranger ce message dans la zone mémoire de stockage et libérer le descripteur de message en le chaînant sur POOL$EX.

A.2.2.c. *Suivi de l'observation*

Le suivi de l'observation est réalisé par la tâche SUIVI activée par une commande 'S'. La tâche SUIVI fait une lecture de la zone mémoire de stockage, alors que la tâche STOCK écrit dans cette zone mémoire. L'accès en exclusion mutuelle est réalisé à l'aide de l'échange MUTEX ; pour permettre l'affichage d'un message après son stockage, un flag indique à la tâche STOCK de se mettre en attente illimitée sur l'échange EX$SYNCRO après le stockage, elle est alors activée de nouveau par la tâche SUIVI après l'affichage. Lorsque la tâche SUIVI n'est pas activée, la tâche STOCK ne se met pas en attente sur EX$SYNCRO.

Les informations fournies par l'observateur lors du suivi sont les champs RESULTAT, EMETTEUR, RECEPTEUR et TYPE du message observateur testé. Pour faciliter leur lecture, le champ RESULTAT est décodé et écrit en langage naturel, les résultats possibles sont :

correct, correct$séquence, correct$séquence$émetteur, correct$séquence$récepteur,
incorrect, incorrect$émetteur, incorrect$récepteur,
non pris en compte (n'appartient pas au protocole à observer),
contexte (pour le contexte suivant l'erreur),
incorrect$réception (trame refusée par l'IBS ou par le SBC).

Les champs EMETTEUR et RECEPTEUR sont écrits en base 10, le champ TYPE en base 16.

Exemple : CORRECT résultat
S 01 émetteur
S 02 récepteur
S 61 type

S indique que la valeur affichée est une valeur fournie par la tâche SUIVI.

Les codages des différents messages du protocole de gestion de ligne sont les suivants :

ACCEPTEZ	61H
ACK	62H
PRENEZ	63H
DONNEZ	64H

La tâche SUIVI peut être désactivée à tout moment par la même commande 'S'.

A.2.2.d. Affichage du contexte de l'erreur

Si le message reçu par la tâche SIMUL est de la classe INCORRECT, SIMUL signale ce fonctionnement incorrect à la tâche CONTXT par envoi d'un en-tête de message sur l'échange EX$CONTXT. Ainsi activée, CONTXT attend de STOCK un message de synchronisation lui permettant de connaître l'emplacement mémoire dans lequel le message a été stocké et donc d'afficher sur un terminal opérateur le contexte immédiat précédant l'erreur.

Lorsque le champ RESULTAT du message est "incorrect", les messages circulant entre les deux processeurs déclarés en panne et suivant le message incorrect sont fournis à l'opérateur pour constituer l'historique immédiat suivant l'erreur et les marquages courants de tous les processeurs actifs sont réininitalisés. Dans ce cas, l'observation est suspendue pendant toute la sortie du contexte, le déroulement des tâches SIMUL et FILTRE a été modifié : les messages circulant dans l'observateur ne sont pas traités, ils sont uniquement chaînés sur l'échange EX$MESS$TOUS (leur champ RESULTAT est alors égal à ØFFH).

L'observation ne reprend qu'après la fin de l'écriture du contexte, les tâches SIMUL et FILTRE peuvent à nouveau traiter les messages, et cela après la réception d'un message de synchronisation envoyé par CONTXT respectivement sur l'échange EX$INHIB$SIMUL et EX$INHIB$FILTRE.

Lorsque le champ RESULTAT du message est "incorrect$rcp" ou "incorrect$em", seul le marquage courant du processeur en panne est réinitialisé, l'observation des messages n'est pas interrompue.

La largeur de la fenêtre de sortie du contexte a été définie lors de la configuration du système. Par exemple, si "largeur$fenêtre" est égal à 2, la tâche CONTXT affichera les deux messages précédents du message détecté incorrect, le message incorrect, et le cas échéant, les deux messages suivants ce message.

Afin de ne pas augmenter le temps d'exécution de la tâche CONTXT, le champ RESULTAT est écrit en décimal pour les messages précédant l'erreur, le codage des différents résultats possibles est le suivant :

```
/*************VALEURS DE RESULTAT ****************/

DCL AVORTEMENT LIT '10000000B',
ERR$CRC     LIT '01000000B',
ERR$RECOU   LIT '00100000B',
LGR$INCOH   LIT '00010000B',
PARASITE    LIT '00001000B',
BUF$RCP     LIT '00000100B',
RCP$SBC    LIT '00000010B',

CORRECT  LIT '01000001B',
CORRECT$SEQ$RCP    LIT '01010011B',
CORRECT$SEQ$EM     LIT '01001011B',
CORRECT$SEQ        LIT '01011011B',
INCORRECT$RCP      LIT '01010101B',
INCORRECT$EM  LIT '01001101B',
INCORRECT  LIT '01011101B',
NON$PRIS$COMPTE    LIT '01000111B';
```

Pour le champ RESULTAT des messages INCORRECT, on affiche CONTEXTE.

Exemple :

```
01000001          correct
01
02
61                ACCEPTEZ du processeur 1 vers le processeur 2

01000001          correct
02
01
62                ACK du processeur 2 vers le processeur 1

01011101          incorrect
03
02
64                DONNEZ du processeur 3 vers le processeur 2

CONTEXTE
01
02
63                PRENEZ du processeur 1 vers le processeur 2

CONTEXTE
02
03
61                ACCEPTEZ du processeur 2 vers le processeur 3
```

BIBLIOGRAPHIE

---

1 W.C. CARTER, P.R. SCHNEIDER, "Design of dynamically checked computers", IFIP Congress 1968, Edinburgh, Scotland

2 M. DIAZ, P. AZEMA, J.M. AYACHE, "Unified design of self-checking and fail-safe combinational circuits and sequential machines", IEEE Trans. on Computers, vol C-28, n° 3, pp 276-281, Mars 1979

3 M.A. FISCHLER, O. FIRSCHEIN, D. L. DREW, "Distinct software : an approach on reliable computing", Second U.S.A., JAPAN Comp. Conf.

4 S.S. YAU, F.C. CHEN, "An approach to concurrent control flow checking", I E E E Trans. on Software Engineering , vol 6, n° 2, Mars 1980

5 J.M. AYACHE, P. AZEMA, M. DIAZ, "Observer : a concept for on-line detection for control errors in concurrent systems", 9th Int. Symp. on Fault Tolerant Computing, Madison, Juin 1979

6 J.M. AYACHE, M. DIAZ, R. VALETTE, "A methodology for specifying control in electronic switching systems", Int. Switching Symposium, PARIS, Mai 1979

7 L. CHEN, A. AVIZIENIS, "N - version programming : a fault-tolerance approach to reliability of software operation", 8 th Int. Symp. on Fault Tolerant Computing, TOULOUSE, Juin 1978

8 B. RANDELL, "System structure for software fault tolerance", I E E E Trans. on Software Engineering, vol 1, n° 2, Juin 1979

9 D.L. PARNAS, "The use of precise specifications in the development of software", IFIP 1977

10 H.KONBER, "Contribution à la conception des protocoles de communication dans les autocommutateurs électroniques", Thèse de Docteur-Ingénieur, U.P.S. TOULOUSE, Décembre 1980

11 C.V. RAMAMOORTHY, "On the automated generation of program test data", I E E E Trans. on Software Engineering, vol 2, n° 4, Décembre 1976

12 K.H. KIM, "Error detection, reconfiguration and recovery in distributed processing systems", 1st Int. Conf. on Distributed Computing Systems, Huntsville Alabama, 1-5 Octobre 1979

13 B. RANDELL, "Software fault-tolerance", Computing Systems Reliability, TOULOUSE, 10-21 Septembre 1979

14 J.R. KANE, S.S. YAU, "Concurrent software fault detection", I E E E Trans. on Software Engineering, vol 1, n° 1, Mars 1975

15 S.S. YAU, F. CHEN, K.H. YAU, "An approach to real time control checking", COMPSAC, Novembre 1978

16 J.M. AYACHE, M. DIAZ, J. NOUBEL, "Conception d'un logiciel d'autotest par systèmes temps réel distribués", Congrès AFCET, NANCY, Septembre 1980

17 J.M. AYACHE, M. DIAZ, J. NOUBEL, J.L. MASSIEU, B. POTIN, "Test en ligne et surveillance dans les systèmes locaux à diffusion série", Projet Pilote SURF, Bilan et Perspectives, PARIS, Janvier 1981

18 J.L. PETERSON, "Petri Nets", Computing Surveys, Septembre 1977

19 C.A.R. HOARE, "Communicating Sequential Processes", Com. of the A.C.M., vol 21, n° 8, Août 1978

20 G.V. BOCHMANN, "Finitestate description of communication protocols", Proc. Symp. on Computer Network Protocols, LIEGE, Février 1978

21 P.E. LAUER, E. BEST, M.W. SCHIELDS, "On the problem of achieving adequacy of concurrent programs", Tech. Report 103 Univ. de NEWCASTLE Upon Tyne, Juin 1977

22 H. ZIMMERMANN, "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection", I E E E Trans. on Communications, vol 28, n° 4, Avril 1980

23 B. CHEZALVIEL - PRADIN, "Un outil graphique interactif d'aide à la vérification des systèmes à évolution parallèle décrits par réseaux de Petri", Thèse de Docteur-Ingénieur, U.P.S. TOULOUSE, Décembre 1979

24 B. PRADIN, B. BERTHOMIEU, P. AZEMA, M. DIAZ, S. BACHMAN, "OGIVE : un outil graphique interactif de vérification des systèmes parallèles décrits par des réseaux de Petri", revue MICADO, n° 35, Septembre 1980

25 C.A. PETRI, "Kommunikation mit Automaten", Univ. BONN, 1962

26 B. BERTHOMIEU, "Analyse structurelle des réseaux de Petri : méthodes et outils", Thèse de Docteur-Ingénieur, U.P.S. TOULOUSE, Septembre 1979

27 G.V. BOCHMANN, "A general transition model for protocols and communication services", I E E E Trans. on Communications,vol 28, n° 4, Avril 1980

28 M. DEVY, M. DIAZ, "Multilevel specification and validation of the control in communication systems", 1st Int. Conf. on Distributed Computing Systems, Huntsville Alabama, Octobre 1979

29 M. DIAZ, B. POTIN, B. CARRICHON, M. DEVY, M. SHAPIRO, O. APPEL, J.M. AYACHE, "Système de communication adapté à la commande coordonnée des processus industriels", Rapport de fin de contrat D.G.R.S.T., Décision d'aide n° 78.8.0302, Décembre 1979

30 J. M. AYACHE, B. CARRICHON, M. DEVY, M. DIAZ, B. POTIN, M. SHAPIRO, "A distributed control system for industrial plants", Proceedings of the EUROMICRO, LONDRES, Septembre 1980

31 J. M. AYACHE, B. CARRICHON, J. P. COURTIAT, M. DIAZ, B. POTIN, M. SHAPIRO, "Fault-tolerance in REBUS, a distributed system for industrial real-time control", Fault-Tolerant Computing Symposium, Portland, Maine, USA, Juin 1981

32 M. SHAPIRO, "Une méthode de conception progressive des systèmes parallèles utilisant le langage C.S.P.", Thèse de Docteur-Ingénieur, U.P.S. TOULOUSE, Septembre 1980

33 M. DEVY, "Modélisation et validation des protocoles multiniveaux dans les systèmes localement distribués", Thèse de Docteur-Ingénieur, U.P.S. TOULOUSE, Septembre 1980

34 M. DEVY, M. DIAZ, M. MENASCHE, M. SHAPIRO, "A fault-tolerant virtual ring algorithm for bus allocation", Rapport interne LAAS-LC n° 80-I-35, Novembre 1980

35 J. M. AYACHE, J. P. COURTIAT, M. DIAZ, "Conception et validation de protocoles - Application au réseau REBUS", journées PARDI-AFCET, PARIS, 22-23 Juin 1981

36 C. MACCHI, J. F. GUILBERT, "Téléinformatique. Transport et traitement de l'information dans les réseaux et systèmes téléinformatiques", Dunod Informatique 1979

37 INTEL, "PL/M Programming Manual" 1977

38 INTEL, "RMX/80 user's guide manual" 1978

39 E. W. DIJKSTRA, "The structure of the THE multiprogramming system", Comm. of the A.C.M., vol 13, n° 4, Avril 1968

40 O. APPEL, J. M. AYACHE, M. DIAZ, "Etude et développement d'un moniteur temps réel multitâches pour microordinateurs", note interne LAAS-LC n° 70.I.53, Novembre 1979

41 GIER - SCHLUMBERGER, "Test en ligne et reconfiguration dans les systèmes localement distribués pour la commande et le contrôle industriel", Rapport partiel de première phase, Juin 1981.

TABLE DES MATIERES

INTRODUCTION 1

CHAPITRE I. UN LOGICIEL REDONDANT 7

INTRODUCTION 9

I.1. VERIFICATION STATIQUE 10

I.2. VERIFICATION DYNAMIQUE 12
I.2.1. Processus séquentiel 13
I.2.1.a. Approche intégrité des données 13
I.2.1.b. Approche intégrité du contrôle 16
I.2.2. Processus distribués : le concept d'observateur 20
I.2.2.a. Structure de base 21
I.2.2.b. Comparaison exécutant-observateur 23
I.2.2.c. Lien entre l'exécutant et l'observateur 24
I.2.2.d. Propriétés requises par le modèle 28

I.3. CONCLUSION 29

CHAPITRE II. CONSTRUCTION D'UN LOGICIEL AUTO-TESTABLE 31

INTRODUCTION 33

II.1. CONCEPTION DE L'EXECUTANT 34

II.2. CONCEPTION DE L'OBSERVATEUR 34
II.2.1. Les réseaux de Petri 35
II.2.1.a. Définitions 35
II.2.1.b. Propriétés dynamiques 38
II.2.1.c. Propriétés structurelles 40
II.2.1.d. Réseaux de Petri étiquetés 41
II.2.2. Modélisation du diagramme de mission 41
II.2.3. Logiciel de l'observateur 44
II.2.3.a. Simulateur de réseaux de Petri 45
II.2.3.b. Base de données 46

II.3. CONNEXION OBSERVATEUR-EXECUTANT DANS UN CONTEXTE DISTRIBUE 47
II.3.1. Problèmes inhérents à la distribution 47
II.3.2. Comparaison de ces deux modélisations et choix 52
II.4. CONCLUSION 53

CHAPITRE III. MODÉLISATION DU DIAGRAMME DE MISSION 55

INTRODUCTION 57
III.1. METHODOLOGIE D'OBTENTION DU DIAGRAMME DE MISSION 58
III.1.1. Système de communication parfait 59
III.1.2. Système de communication non parfait 61
III.1.3. Conclusion 69
III.2. EXEMPLE ILLUSTRATIF DANS LE SYSTEME REBUS 71
III.2.1. Présentation du support de l'étude : le système REBUS 71
III.2.2. Protocole de passage du statut maître : cahier des charges 76
III.2.3. Protocole de passage du statut maître : modélisation 79
III.2.3.a. Modèle informel - modèle formel 79
III.2.3.b. Modèle global 81
III.2.4. Diagramme de mission 86
III.3. CONCLUSION 88

CHAPITRE IV. IMPLÉMENTATION 91

INTRODUCTION 93
IV.1. SUPPORT MATERIEL 94
IV.1.1. Carte interface bus série et carte unité centrale SBC 80/30 94
IV.1.1.a. La carte Interface Bus Série 95
IV.1.1.b. La carte unité centrale SBC 80/30 95
IV.1.2. Communication carte IBS et carte SBC 80/30 95

IV.2. SUPPORT LOGICIEL 96
IV.2.1. Langage de programmation : PL-M/80 96
IV.2.2. Moniteur temps réel multitâches 96
IV.3. STRUCTURE DE L'OBSERVATEUR 102
IV.3.1. Interface opérateur 103
IV.3.2. Initialisation 106
IV.3.3. Interface avec le système de communication 107
IV.3.4. Simulateur de réseaux de Petri 109
IV.3.5. Le stockage des messages 112
IV.4. EXEMPLES 112
IV.4.1. REBUS reconfiguration 112
IV.4.2. Modumat 800 114
IV.5. CONCLUSION 117

CONCLUSION 119

ANNEXES 125

BIBLIOGRAPHIE 143