



HAL
open science

SSH : un outil et des techniques simples à implémenter pour construire et simuler des modèles hiérarchisés de systèmes

Jean Rohmer

► **To cite this version:**

Jean Rohmer. SSH : un outil et des techniques simples à implémenter pour construire et simuler des modèles hiérarchisés de systèmes. Modélisation et simulation. Institut National Polytechnique de Grenoble - INPG, 1976. Français. NNT : . tel-00287086

HAL Id: tel-00287086

<https://theses.hal.science/tel-00287086>

Submitted on 10 Jun 2008

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

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

THESE

présentée à

Institut National Polytechnique de Grenoble

pour obtenir le grade de

DOCTEUR-INGENIEUR
Informatique

par

Jean ROHMER



**SSH : UN OUTIL ET DES TECHNIQUES
SIMPLES A IMPLEMENTER
POUR CONSTRUIRE ET SIMULER
DES MODELES HIERARCHISES
DE SYSTEMES**



Thèse soutenue le 24 juin 1976 devant la Commission d'Examen :

Président : L. BOLLIET
Examineurs : F. ANCEAU
S. GUIBOUD-RIBAUD
C. KAISER
S. KRAKOWIAK

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Président : M. Philippe TRAYNARD

Vice-Président : M. Pierre-Jean LAURENT

PROFESSEURS TITULAIRES

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

PROFESSEUR ASSOCIE

M. ROUXEL Roland Automatique

PROFESSEURS SANS CHAIRE

MM. BLIMAN Samuel	Electronique
BOUVARD Maurice	Génie Mécanique
COHEN Joseph	Electrotechnique
LACOUME Jean-Louis	Géophysique
LANCIA Roland	Electronique
ROBERT François	Analyse numérique
VEILLON Gérard	Informatique Fondamentale et Appliquée
ZADWORNY François	Electronique

MATTRES DE CONFERENCES

MM. ANCEAU François	Mathématiques Appliquées
CHARTIER Germain	Electronique
GUYOT Pierre	Chimie Minérale
IVANES Marcel	Electrotechnique
JOUBERT Jean-Claude	Physique du solide
MORET Roger	Electrotechnique Nucléaire
PIERRARD Jean-Marie	Mécanique
SABONNADIÈRE Jean-Claude	Informatique Fondamentale et Appliquée
Mme SAUCIER Gabrièle	Informatique Fondamentale et Appliquée

MAITRE DE CONFERENCES ASSOCIE

M. LANDAU Ioan

Automatique

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

MM. FRUCHART Robert

Directeur de Recherche

ANSARA Ibrahim

Maître de Recherche

CARRE René

Maître de Recherche

DRIOLE Jean

Maître de Recherche

MATHIEU Jean-Claude

Maître de Recherche

MUNIER Jacques

Maître de Recherche

Je tiens à remercier

Monsieur le Professeur Bolliet, qui m'a fait l'honneur de présider le jury. Depuis 1971, où il m'a permis de travailler dans les équipes grenobloises, j'ai pu apprécier sa disponibilité et sa sollicitude constantes envers les jeunes chercheurs.

Monsieur Anceau, qui est à l'origine des idées contenues dans ce travail, idées qu'il est parvenu malgré quelques lenteurs de ma part à me faire partager. Son amitié encourageante et sa fermeté bienveillante m'ont toujours soutenu.

Monsieur Guiboud-Ribaud, qui a accepté d'être le rapporteur de cette thèse, et qui m'a aidé de ses conseils lors de la phase de rédaction.

Messieurs Kaiser et Krakowiak, qui m'ont fait l'honneur de participer au jury.

Je remercie également mes anciens collègues de l'équipe de Monsieur Anceau, en particulier M.M. Fortier, Poujoulat et Schoellkopf, qui m'ont souvent aidé à résoudre des problèmes matériels dus à mon éloignement géographique de Grenoble.

Enfin ma gratitude va aux services techniques de Madame Theis et de Monsieur Mallet à l'IRIA, de Monsieur Iglésias à l'ENSIMAG, pour la réalisation matérielle de cette thèse.

A mes parents, à Nicole, à Peggy

A la mémoire de Hervé Lucot et de Bernard Sans

TABLE DES MATIERES

AVANT-PROPOS
PLAN

I PREMIERE PARTIE

	Pages
<u>I.1. MOTIVATIONS.....</u>	3
I.1.1. LES BESOINS.....	4
I.1.2. Les solutions actuelles.....	4
I.1.3.1. Un bon modèle.....	6
I.1.3.2. Un bon langage algorithmique.....	8
I.1.3.3. Une implémentation économique et accessible.....	8
<u>I.2. RAPPELS SUR LA TSH.....</u>	9
I.2.1. APPORTS DE TSH A LA NOTION DE HIERARCHIE.....	10
I.2.1.1. Historique des hiérarchies.....	10
I.2.1.2. Formalisation de la hiérarchie des synchronisations.....	11
I.2.2. SEPARATION SYNCHRONISATION/EXECUTION.....	11
I.2.3. UNE DEFINITION SUCCINTE DE LA TSH.....	13
I.2.3.1. La synchronisation dans SSH.....	13
I.2.3.2. Les inductions.....	17
I.2.3.2.1. Induction Directe.....	17
I.2.3.2.2. Induction Inverse.....	18
I.2.3.2.3. Induction de Libération.....	19
I.2.3.2.4. Induction de Réquisition.....	19
<u>I.3. LE PASSAGE DE TSH A SSH.....</u>	19
<u>I.4. LE CAHIER DES CHARGES DE SSH.....</u>	21
I.4.1. CARACTERISTIQUES SEMANTIQUES.....	21
I.4.1.1. Objets manipulés par SSH.....	22
I.4.1.2. 5 aspects principaux dans la sémantique de SSH.....	22
I.4.1.2.1. La structure du modèle.....	22

I.4.1.2.2. Primitives de synchronisation.....	23
I.4.1.2.3. Mécanismes d'induction.....	23
I.4.1.2.4. Mécanismes de contrôle.....	24
I.4.1.2.5. Les activités des noeuds.....	25
I.4.1.2.6. Séparation simulation du temps / description du modèle...	25
I.4.2. CARACTERISTIQUES LINGUISTIQUES.....	25
I.4.2.1. L'implémentation actuelle.....	25
I.4.2.2. Autres possibilités d'implémentation.....	26
I.4.2.3. Adressage de l'environnement d'un noeud.....	28
I.4.2.4. Concision.....	28
I.4.3. CARACTERISTIQUES OPERATIONNELLES.....	28

II DEUXIEME PARTIE

<u>II.1. DEFINITION D'UN PSEUDO-LANGAGE HOTE P.L.....</u>	32
<u>II.2. CONVENTIONS POUR LA CONSTRUCTION DES NOEUDS SIMPLES.....</u>	33
II.2.1 NOTION D'EVENEMENT.....	33
II.2.2. LES DIFFERENTES CLASSES D'ALGORITHMES ATTACHEES AUX NOEUDS...	33
II.2.2.1. La classe de décision.....	34
II.2.2.2. La classe d'activité.....	34
II.2.2.3. La classe d'archivage.....	35
<u>II.3. DECLARATION D'UN NOEUD SIMPLE.....</u>	36
II.3.1. INSTRUCTION PERE.DE.....	36
II.3.2. INSTRUCTION ESPACE.LOCAL.....	36
II.3.3. FONCTIONS DE SERVICE COPIE,DETRUIRE.....	38
II.3.4. INSTRUCTIONS ALGORITHMES.....	39
II.3.4.BIS ECRITURE DES ALGORITHMES DE DECISION-PERE.....	40
II.3.5. ECRITURE DES ALGORITHMES DECISION-FILS.....	41
II.3.6. NOEUDS SIMPLES ACTIFS ET INACTIFS.....	43
II.3.7. ECRITURE DES SOUS-PROGRAMMES D'ACTIVITE.....	44
II.3.8. EXEMPLES D'UTILISATION DE DUREE.....	45
II.3.9. EXTENSION DE DUREE.....	46
II.3.9.BIS INSTRUCTION REPRISE.....	47

II.3.10. NOTE SUR LA PROGRAMMATION DES ALGORITHMES D'ACTIVITE..... 47

II.4. RESUME SUR LES NOEUDS SIMPLES..... 50

II.5. LES NOEUDS SIMPLES STANDARD DE SSH..... 53

II.5.1. LE NOEUD LAMBDA..... 53

II.5.2. LE NOEUD PRIORITAIRE..... 54

II.5.3. LE NOEUD BARILLET..... 55

II.6. PREMIER EXEMPLE D'UTILISATION DES NOEUDS SIMPLES STANDARD..... 55

II.7. DEUXIEME EXEMPLE D'UTILISATION DES NOEUDS SIMPLES STANDARD..... 58

III LES NOEUDS MULTIPLES

III.1. MOTIVATIONS..... 67

III.2. DEFINITION DES NOEUDS MULTIPLES..... 69

III.2.1. INSTRUCTION COMPOSE.DE..... 70

III.2.2. L'ALGORITHME DE DECISION DES NOEUDS MULTIPLES..... 73

III.2.2.1. Instruction AFFECTER..... 73

III.2.2.2. Instruction DESFFECTER..... 74

III.2.2.3. Instruction MUTER..... 74

III.3. POSSIBILITES DES NOEUDS MULTIPLES POUR OBSERVER L'ETAT DE
LEUR FILS..... 75

III.4. PROPAGATION DE L'INDUCTION INVERSE DANS UNE HIERARCHIE DE
NOEUDS MULTIPLES IMBRIQUES..... 77

III.4.1. SOLUTION ADOPTEE..... 77

III.4.1.1. Instruction SOUS-TRAITER..... 78

III.4.1.2. Instruction SUR-TRAITER..... 78

III.4.2. DISCUSSION DES DIFFERENTES SOLUTIONS DE GESTION DES NOEUDS
MULTIPLES..... 79

III.4.2.1. Position du problème..... 79

III.4.2.2. Exemple pratique.....	81
III.4.2.3. Solutions envisageables.....	83
III.4.2.4. Solution préférée.....	85
III.4.3. CONSTRUCTION D'UN NOEUD MULTIPLE STANDARD.....	85
III.4.4. EXEMPLE D'UTILISATION D'UN NOEUD MULTIPLE STANDARD.....	86
<u>III.5. NOEUDS AYANT PLUSIEURS PERES.....</u>	88
III.5.1. LA SOLUTION TSH.....	88
III.5.2. LA SOLUTION AVEC DES NOEUDS MULTIPLES.....	88
III.5.3. LA SOLUTION SIMPLISTE : LES NOEUDS DE TYPE RESSOURCE.....	89

IV QUATRIEME PARTIE

<u>IV.1. LANCEMENT ET ARRET DE LA SIMULATION.....</u>	93
IV.1.1. INSTRUCTION SIMULATION.....	93
IV.1.2. INSTRUCTION STOP.....	93
IV.1.3. EXEMPLES D'UTILISATION.....	93
<u>IV.2. L'INSTRUCTION CONFIE.A.....</u>	94
IV.2.3. EXEMPLES.....	95
IV.2.3.1. Exemple 1.....	95
IV.2.3.2. Exemple 2.....	96
<u>IV.3. L'INSTRUCTION CHANGER.VITESSE.....</u>	97
IV.3.1. MOTIVATIONS.....	97
IV.3.2. DEFINITION.....	99
IV.3.3. VITESSE LOCALE.....	100
IV.3.4. VITESSE GLOBALE.....	101
IV.3.5. MISE EN OEUVRE DE CHANGER.VITESSE.....	102
IV.3.5.1. Généralité de l'implémentation de CHANGER.VITESSE.....	102
IV.3.5.2. Induction des changements de vitesse.....	102
IV.3.5.3. Autres exemples d'utilisation de la vitesse.....	103
<u>IV.6. LE MODE D'EXECUTION DIFFEREE, OU "MODE D".....</u>	105
IV.6.1. INSTRUCTIONS PERMETTANT DE METTRE EN OEUVRE LE MODE "D".....	105
IV.6.2. EXEMPLES D'UTILISATION.....	106

V CINQUIEME PARTIE

<u>V.1. ETUDE DE CAS D'UNE UTILISATION REELLE DE SSH.....</u>	111
V.1.1. LE SYSTEME A SIMULER.....	111
V.1.2. LA SOLUTION SSH.....	115
<u>V.2. TECHNIQUES DE COMMUNICATION UTILISATEUR-MODELE.....</u>	118
V.2.1. OUTILS GENERAUX.....	118
V.2.2. OUTILS SSH.....	118
V.2.2.1. Noeuds espions du temps et des événements.....	119
V.2.2.2. L'utilisateur considéré comme un noeud.....	121
<u>V.3. PROBLEMES DE RECUEIL DES DONNEES ET d'ANALYSE DES RESULTATS....</u>	121
V.3.1. UN PROBLEME DE MESURES POUR LA SIMULATION.....	122
V.3.1.1. Difficulté du problème général.....	122
V.3.1.2. Cas particulier.....	124
V.3.1.3. Développements possibles.....	128
V.3.1.4. Exemple.....	128
<u>V.4. UNE TECHNIQUE POUR INTERPRETER DES RESULTATS DE SIMULATION</u> <u>EN SSH.....</u>	130

VI SIXIEME PARTIE

<u>VI.1. L'ALGORITHME DE SIMULATION DU TEMPS.....</u>	137
VI.1.1. STRUCTURES DE DONNEES DE L'ALGORITHME.....	137
VI.1.2. FONCTIONS DE L'ALGORITHME.....	139
VI.1.3. ALGORITHME DE MISE A JOUR DE L'HEURE.....	140
VI.1.4. AUTRE IMPLEMENTATION POSSIBLE.....	141
VI.1.5. ORGANISATION DE LA GESTION DU TEMPS.....	141
<u>VI.2. L'ALGORITHME DE SIMULATION DES INDUCTIONS.....</u>	143
VI.2.1. L'IMPLEMENTATION DE L'INSTRUCTION CONFIE.A.....	144
VI.2.2. LES INDUCTIONS DANS LES HIERARCHIES DE NOEUDS MULTIPLES IMBRIQUES.....	144
VI.2.3 L'INDUCTION INVERSE DANS L'ARBRE DES NOEUDS SIMPLES.....	145

<u>VI.3. PROGRAMMATION INVERSE DANS L'ARBRE DES NOEUDS SIMPLES.....</u>	146
VI.3.1. PREMIERE SOLUTION : APPEL PROCEDURAL RECURSIF.....	147
VI.3.2. DEUXIEME SOLUTION : UNE "RECURSIVITE FAIBLE".....	148
VI.3.3. TROISIEME SOLUTION : UNE IMPLEMENTATION "AD-HOC".....	150

VII SEPTIEME PARTIE

<u>VII.1. DEVELOPPEMENTS ENVISAGES POUR SSH.....</u>	153
VII.1.1. AJOUT DE FACILITES DE RECUEIL DES RESULTATS.....	153
VII.1.1.1. Instructions sur les types SSH.....	153
VII.1.1.2. Un type de variables "mesurables".....	154
VII.1.2. SIMULATION INTERNE A UN SYSTEME.....	154
<u>VII.2. DEVELOPPEMENTS D'APPLICATIONS DE QUELQUES ASPECTS DE LA TSH..</u>	156
VII.2.1. LA TSH EN TANT QUE RESEAU RENSEIGNE.....	156
VII.2.2. EXPLOITATION DE LA HIERARCHIE DES SYNCHRONISATIONS.....	156
<u>VII.3. CONCLUSION.....</u>	157
<u>BIBLIOGRAPHIE.....</u>	159

ANNEXES

<u>A. PROGRAMMATION DES NOEUDS STANDARD DE SSH.....</u>	162
A.1. CONVENTIONS COMMUNES AUX NOEUDS SIMPLES LAMBDA ET PRIORITAIRE..	162
A.2. ALGORITHMES D'ACTIVITE.....	162
A.3. ALGORITHMES D'ARCHIVAGE.....	163
A.4. ALGORITHMES DE DECISION-FILS STANDARD.....	164
A.4.1. Algorithme de decision-fils lambda.....	164
A.4.2. Algorithme de décision-fils prioritaire.....	165
A.5. DECLARATION DES NOEUDS STANDARD.....	165
A.6. LE NOEUD STANDARD BARILLET.....	166
A.7. L'ALGORITHME DE DECISION DU NOEUD MULTIPLE STANDARD.....	168
A.8. REPRESENTATION GRAPHIQUE DES NOEUDS STANDARD.....	170
<u>B. LES INSTRUCTIONS DE SSH NON STRUCTURE.....</u>	173
<u>C. LE PROGRAMME D'IMPLEMENTATION DE SSH.....</u>	175

AVANT-PROPOS

Ce travail a trois objectifs :

- a) Partant des concepts de la théorie des systèmes hiérarchisés ANCEAU [1] définir un langage et son interpréteur permettant de décrire formellement et d'expérimenter des systèmes à architecture hiérarchisée ou pouvant s'y ramener.
- b) Introduire dans un tel formalisme des dispositifs évolués de simulation du temps, et montrer que les notions de hiérarchie apportent une dimension nouvelle aux langages et techniques de simulation.
- c) Prouver que les points a) et b) sont implémentables avec efficacité et très simplement (quelques centaines d'instructions) dans n'importe quel langage usuel de programmation.

On obtient finalement un outil facilement accessible à tout utilisateur de moyens informatiques, plus puissant et moins coûteux que d'autres produits habituellement utilisés pour la spécification et la simulation des systèmes.

Remarque : Nous avons essayé de définir :

Un langage ayant une "bonne sémantique" vis-à-vis du but recherché, plutôt qu'un "beau" langage. L'étude de ses qualités purement linguistiques aurait certes été intéressante en elle-même, mais il nous a semblé primordial de ne pas transformer une "réalisation d'outil d'aide à un projet" en "projet d'aide à la réalisation d'outils" !

Ce travail a été en grande partie réalisé en 1972-1973, dans le cadre de l'équipe d'architecture des calculateurs de l'ENSIMAG, à l'occasion du contrat CRI N° 72-21/I-41 auquel participaient également les Sociétés TELEMECANIQUE et TITN.

Note de terminologie :

Dans la suite, nous emploierons le sigle "TSH" (Théorie des Systèmes Hiérarchisés) pour désigner les travaux exposés dans ANCEAU [1] et "SSH" (Simulateur de Systèmes Hiérarchisés) pour désigner le langage présenté dans ce travail.

PLAN DE CE TRAVAIL

La première partie rappelle l'essentiel de la Théorie des systèmes hiérarchisés, montre l'intérêt qu'elle présente comme support d'un outil de simulation et indique l'esprit dans lequel le "cahier des charges" de SSH a été établi.

La seconde partie définit la syntaxe et la sémantique des noeuds simples de SSH et donne de nombreux exemples de leur utilisation.

La troisième partie motive le besoin d'un type de noeud supplémentaire : le noeud multiple, et illustre ses particularités.

La quatrième partie présente des instructions plus sophistiquées, introduisant des notions de "vitesse" généralisant l'introduction du temps simulé dans un modèle SSH.

La cinquième partie est consacrée à l'illustration de l'utilisation de SSH, des facilités qu'il offre et de quelques problèmes pratiques liés à la simulation.

La sixième partie décrit le principe et le détail de l'implémentation de SSH.

En Annexe, on trouvera en particulier de nombreux programmes complets décrivant les algorithmes attachés à divers exemples cités dans les parties précédentes.

I PREMIERE PARTIE

Présentation générale :
Motivations et Objectifs

I.1. MOTIVATIONS

I.1.1. LES BESOINS

Les besoins en outils de description et de simulation de systèmes informatiques vont actuellement en croissant pour diverses raisons :

- les informaticiens au sens habituel du terme conçoivent des systèmes de plus en plus complexes et difficiles à maîtriser. Ils découvrent la nécessité de comprendre leurs performances en simulant leur fonctionnement, afin d'améliorer ce qui peut encore l'être,
- les architectes de système veulent commencer par décrire et simuler le fonctionnement logique et intrinsèque du système indépendamment de toute idée préconçue d'implémentation, puis s'inspirent des résultats ainsi obtenus pour s'orienter par étapes successives vers une implémentation finale optimisée. C'est "l'approche descendante" ANCEAU et AL [2], ROHMER, TUSERA [3],
- les théoriciens essaient de définir des notations nouvelles permettant de décrire les interactions entre les entités composant un système, et si possible de déduire directement certains résultats généraux sur le comportement du système,
- enfin, de nouveaux utilisateurs de plus en plus nombreux, atteints par l'informatique grâce aux progrès des micro-composants, éprouvent le besoin de s'aider d'outils automatisés pour choisir et mettre au point leur système, en général très spécialisé.

I.1.2. LES SOLUTIONS ACTUELLES

Les outils usuels de simulation peuvent être classés en deux grandes catégories :

- A) - les "outils-modèles" basés sur une notion de modèle : ils permettent en général de décrire le problème à simuler sous la forme d'un réseau de stations entre lesquelles circulent des entités représentant les usagers de ces stations. Exemple : GPSS [4] ,
- B) - les "outils-langages" basés sur un langage classique de programmation (FORTRAN, ALGOL) auquel sont ajoutées des facilités de simulation de temps. Par exemple, à partir de l'appel simple de sous-programme "CALL SSPRO", sera introduit un appel avec temporisation : "CALL SSPRO AT TIME t". Ainsi, le langage SIMSCRIPT [5.1] est une extension de FORTRAN, et SIMULA [5.2] est dérivé des langages de type ALGOL.

Ces deux catégories d'outils ont des avantages et des inconvénients complémentaires.

Les premiers sont intéressants si l'application à traiter obéit effectivement aux propriétés de base du modèle. Sinon, l'adaptation de l'application au modèle risque d'être très lourde, voire tout à fait impossible. D'autre part, ces langages offrent en général peu de moyens pour affiner la description d'une application, par exemple pour décrire des algorithmes de transition de complexité quelconque d'une station à une autre.

Ceci est par contre facile avec les outils de la seconde catégorie qui, par construction, offrent toutes les possibilités algorithmiques et les structures de données des langages dont ils sont issus. Malheureusement, la contrepartie de cette puissance est souvent l'introduction d'une confusion : on emploie indifféremment les mêmes outils pour décrire ou bien l'évolution du temps, ou bien un algorithme d'aiguillage vers diverses stations, ou bien un générateur de nombres aléatoires, etc...

Alors qu'avec la première solution, la notion de modèle est présente dans la syntaxe et la sémantique (stations, file d'attente), dans la seconde, elle n'existe que dans l'esprit de l'utilisateur et dans l'usage qu'il a fait de notions purement linguistiques (sous-programmes, listes, tableaux) pour traduire les caractéristiques de son application.

Un corollaire de ceci est que les "outils-modèles" offrent en général des facilités de recueil des données et d'analyse des résultats de simulation (valeurs moyennes, écarts, histogrammes, etc...) qui ne peuvent pas par nature exister dans les outils-langages, et sont donc à la charge de l'utilisateur ou de bibliothèques de sous-programmes de service rajoutées au langage.

En résumé, les outils-modèles sont d'emploi plus rapide et plus aisé que les outils-langages, qui eux, sont plus universels. Les deux ont néanmoins un inconvénient commun, qui est leur coût, à l'achat comme à l'exploitation. De ce fait, ils ne sont disponibles que dans quelques grands centres de calcul, alors que, comme on l'a vu, le cercle de leurs utilisateurs potentiels va en s'élargissant bien au-delà de ces limites.

I.1.3. LA SOLUTION PROPOSEE

Partant des remarques précédentes, il est facile de définir sans risques un outil souhaitable.

Il devrait réunir les qualités :

- d'un bon modèle,
- d'un bon langage algorithmique,

et il devrait être économique et accessible à toutes les installations informatiques.

Reprenons ces trois points avec plus de réalisme :

1.1.3.1. Un bon modèle

Nous ne savons pas s'il existe un "meilleur modèle". Nous pensons par contre que la TSH apporte des nouveautés fondamentales dans la formalisation des systèmes. Le but de ce travail étant précisément d'utiliser avec profit la TSH, nous laissons au lecteur le soin de décider à la fin de ces pages si ce but a été atteint. Donnons simplement ici un petit exemple, pour justifier le choix de la TSH comme support de notre outil.

En GPSS, on décrit un système comme un réseau de stations interconnectées entre lesquelles circulent des utilisateurs. Les stations ont une politique de service vis-à-vis des utilisateurs définissant par exemple le temps passé dans une station et vers quelle station ils sont ensuite dirigés. Les stations fonctionnent indépendamment les unes des autres.

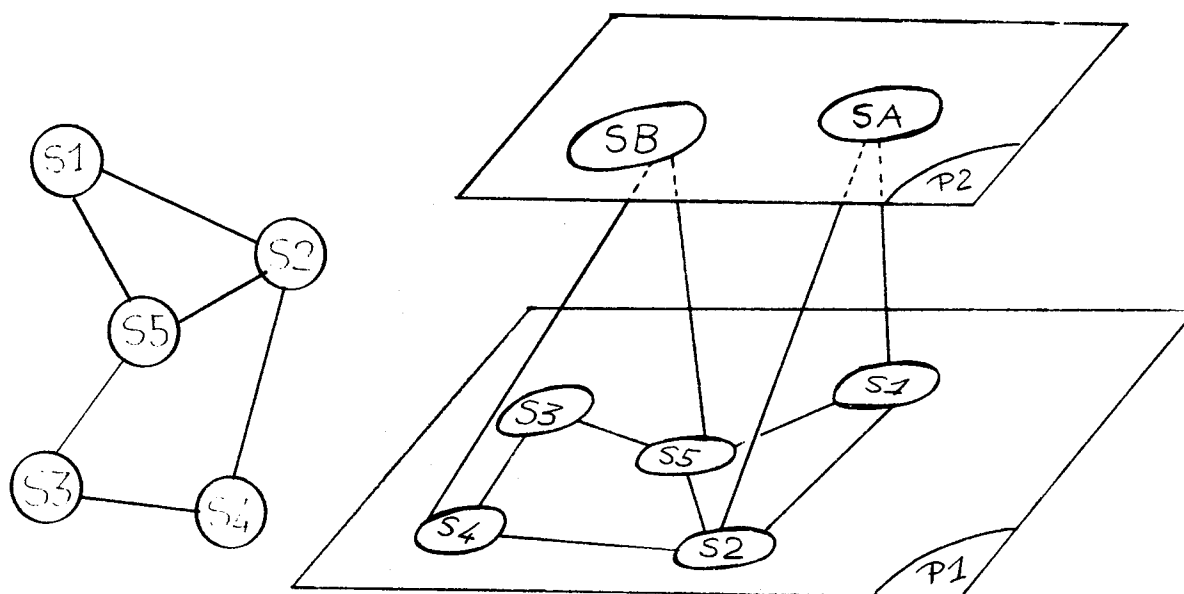


FIGURE 1

Modèle GPSS

FIGURE 2

Modèle TSH

- Un modèle GPSS peut ainsi être représenté par un réseau comme sur la Figure 1.

L'apport essentiel de la TSH est de permettre de spécifier des relations entre les stations elles-mêmes, par un moyen très élégant qui consiste à considérer récursivement les stations comme pouvant être elles-mêmes des utilisateurs d'une autre couche de stations, lesquelles ont bien sûr aussi leur politique de service vis-à-vis de leur "utilisateur-ressource". La TSH parle de relation "utilisateur-ressource".

Par exemple, sur la Figure 2, dans le plan P1, on retrouve exactement le réseau de la Figure 1, mais en plus, il existe un plan P2 comportant des stations SA et SB :

SA est en relation avec S1 et S2, et sa politique peut par exemple être : faire travailler alternativement S1 et S2 pendant Δt ("time-slicing"). SB est en relation avec S5 et S4, et peut par exemple donner toujours la priorité à S5 sur S4 lorsqu'elles veulent travailler ensemble, réalisant au passage une mutuelle exclusion entre S5 et S4. Les stations SA et SB peuvent bien sûr à leur tour être des utilisateurs vis-à-vis d'une couche supérieure et ainsi de suite.

De même, elles peuvent très bien avoir comme utilisateurs, en plus des stations du modèle de la Figure 1, dessinées dans le plan P1, des stations d'un ou de plusieurs autres modèles. On obtient finalement des architectures comme celles de la Figure 3.

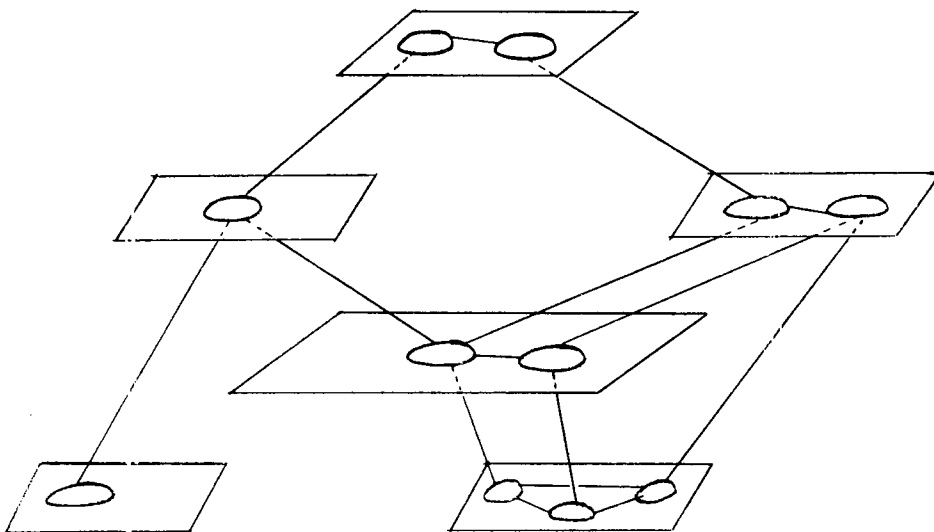


FIGURE 3

Par abus de langage (car la notion de dimension n'est pas adaptée aux réseaux), on peut dire que la TSH permet de passer d'un univers "plan" à un univers "dans l'espace". La TSH rajoute une dimension, ou, plus justement, elle rajoute une relation, qui est la relation existant entre des stations de plans différents.

On entrevoit facilement les applications de cette nouvelle dimension pour formaliser, par exemple, des questions fondamentales dans le domaine particulier de l'architecture des systèmes informatiques :

- synchronisation, ordonnancement des tâches, allocation et partage de ressources, etc...

I.1.3.2. Un bon langage algorithmique

Il est très facile à définir sans longs discours :

"le meilleur langage, c'est le langage que préfère l'utilisateur".

Donc, le meilleur langage, ça peut être n'importe quel langage.

La conséquence stratégique est claire : il doit y avoir autant de SSH que de langages algorithmiques existants.

Les conséquences tactiques rejoignent les nécessités d'une implémentation économique et accessible, expliquées dans l'alinéa suivant.

I.1.3.3. Une implémentation économique et accessible

SSH se présente sous la forme d'un ensemble de sous-programmes à partir desquels l'utilisateur peut - moyennant certaines conventions - construire, simuler et analyser le système qu'il veut étudier.

Personnellement, nous avons écrit trois versions de SSH ou de sous-ensembles de SSH dans les langages suivants : AED, FORTRAN, APL.

Mais - et c'est le plus important - nous donnons dans ce travail la description complète de la programmation des sous-programmes constituant SSH dans un "pseudo-langage" PL très simple et très facilement transcrivable dans un langage usuel de programmation, y compris en assembleur.

Cette description comporte seulement quelques centaines d'instructions, et est très facilement compréhensible.

L'utilisateur désirant se servir de SSH est donc invité à suivre la procédure suivante :

- 1°) lire les spécifications du pseudo-langage,
- 2°) Comprendre le fonctionnement des fonctions SSH décrites dans le pseudo-langage,
- 3°) Choisir les seules facilités de SSH qui l'intéressent,
- 4°) Réécrire dans le langage de son choix les seules parties des programmes du pseudo-langage concernées par 3.

Cette manière de faire est réaliste et avantageuse pour les raisons suivantes :

- la taille du programme en PL est faible. Donc l'utilisateur comprendra la sémantique de SSH beaucoup mieux - et presque aussi vite - en le lisant qu'en consultant seulement des spécifications "en prose", toujours plus ou moins floues et incomplètes.

- les points 3 et 4 sont particulièrement importants. Si l'utilisateur le désire, il peut ainsi supprimer certains tests ou certains calculs inutiles pour son cas particulier qui n'utilise qu'un sous-ensemble de SSH, et gagner en efficacité. (On fait des langages extensibles, mais bien souvent l'utilisateur préférerait des langages "rétrécissables" !).

De plus, il pourra réécrire certaines parties de l'algorithme suivant la technique qui aura sa préférence : tri d'une liste, d'un tableau, remplacer des tests par un aiguillage, etc... pour gagner en temps d'exécution ou en place mémoire.

A notre avis, un tel travail peut être fait en quelques journées, et le temps ainsi investi sera largement récupéré lors de l'utilisation proprement dite de SSH.

I.2. RAPPELS SUR LA TSH

La TSH se place historiquement dans la continuation de deux grandes tendances en informatique :

- la tendance à la hiérarchisation,
- la tendance à la séparation de la synchronisation d'un système du reste de son activité.

1.2.1. APPORTS DE LA TSH A LA NOTION DE HIERARCHIE

1.2.1.1. Historique des hiérarchies

L'introduction de la notion de hiérarchie en informatique n'est pas nouvelle. Plusieurs notions de hiérarchies ont été mises en oeuvre dans différents systèmes :

- hiérarchies de substitution : c'est la règle de recopie, la macrogénération, les appels de sous-programmes,
- hiérarchies des mécanismes d'adressage et de protection GUIBOUD [6],
- hiérarchies sémantiques : c'est la hiérarchie des interpréteurs : chaque couche d'un système réalise les requêtes de la couche qui lui est extérieure, et est elle-même réalisée sur la couche qui lui est intérieure.

Ces hiérarchies ont été introduites pour simplifier le travail d'écriture des programmes et des systèmes, en procédant de manière modulaire : l'utilisateur situé à un niveau donné de la hiérarchie ne connaît que le nom et les propriétés sémantiques de l'entité du niveau supérieur qu'il utilise. Il ne connaît rien de l'implémentation de cette entité. On dit communément que "c'est transparent".

La transparence est une propriété agréable pour celui qui est chargé d'écrire une couche particulière du système, en ce sens qu'elle minimise ses connaissances nécessaires sur le reste du système, tout en lui garantissant le "service sémantique" qu'il demande. L'exemple le plus simple est celui des instructions d'une machine.

Par contre, la transparence est une propriété qui, à elle seule, ne saurait satisfaire l'architecte d'un système. Celui-ci veut en effet, non seulement assurer la correction sémantique du système, mais encore il veut s'intéresser à l'efficacité de son système, c'est-à-dire à son coût et ses performances. Il ne peut se contenter d'une vision couche par couche, mais doit avoir une vision du comportement de l'ensemble, et en particulier, des interactions entre les couches.

Les problèmes se posent alors en termes économiques, quantitatifs ou temporels, notions inconnues dans les hiérarchies précédentes. Il existe certes dans les hiérarchies sémantiques des opérations de synchronisation, comme les primitives P et V sur les sémaphores DIJKSTRA [7], mais c'est précisément de la constatation de leur insuffisance à maîtriser le problème général de l'allocation des ressources - qui est la clé du comportement quantitatif d'un système - qu'est née la TSH.

I.1.1.2. Formalisation de la hiérarchisation des synchronisations

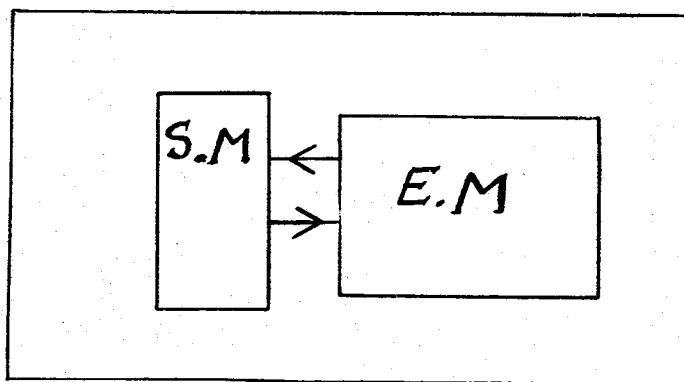
Les primitives P et V, introduites par la même école que celle qui s'intéresse aux hiérarchies sémantiques d'interpréteurs, et à la programmation structurée des systèmes, sont en fait dépourvues de toute valeur sémantique et de toute notion de structure. De même que l'on reproche justement à l'instruction ALLER A X de ne pas indiquer d'où l'on vient et pourquoi on va à X, de même on peut reprocher aux P et V de ne pas indiquer qui les exécute et pour quoi faire. On peut ainsi interpréter un P, V comme un GOTO émis non pas d'une instruction vers une autre, mais émis d'une couche d'un système vers une autre. Et, comme les P et V ne colportent que peu de sémantique, la couche réceptrice ne peut en faire grand chose, (arrêter ou démarrer une tâche).

Le point de départ de la TSH a donc été de faire de la synchronisation structurée entre les différentes couches, en introduisant des mécanismes appelés inductions, permettant de réaliser des synchronisations complexes à partir de niveaux de réalisation élémentaires du genre P et V.

I.2.2. LA SEPARATION SYNCHRONISATION/EXECUTION

La TSH se situe d'autre part, dans la ligne générale qui consiste à séparer clairement dans un système la partie exécution de la partie synchronisation. Nous insisterons un peu sur ce point car l'expérience montre que l'utilisateur a presque toujours tendance, par la suite, à mélanger abusivement ces deux parties.

Il faut considérer un système comme composé de deux machines reliées entre elles : une machine de synchronisation (ou S.Machine) et une machine d'exécution (E.Machine).



La machine d'exécution émet des demandes de synchronisation (souvent appelées primitives de synchronisation) vers la S. Machine.

En réponse, la S. Machine peut envoyer à la E. Machine des ordres que celle-ci interprétera.

Au niveau le plus bas, ces ordres se traduisent par une modification impérative de l'état de la E. Machine : écrire un registre, positionner un signal d'interruption.

Exemple :

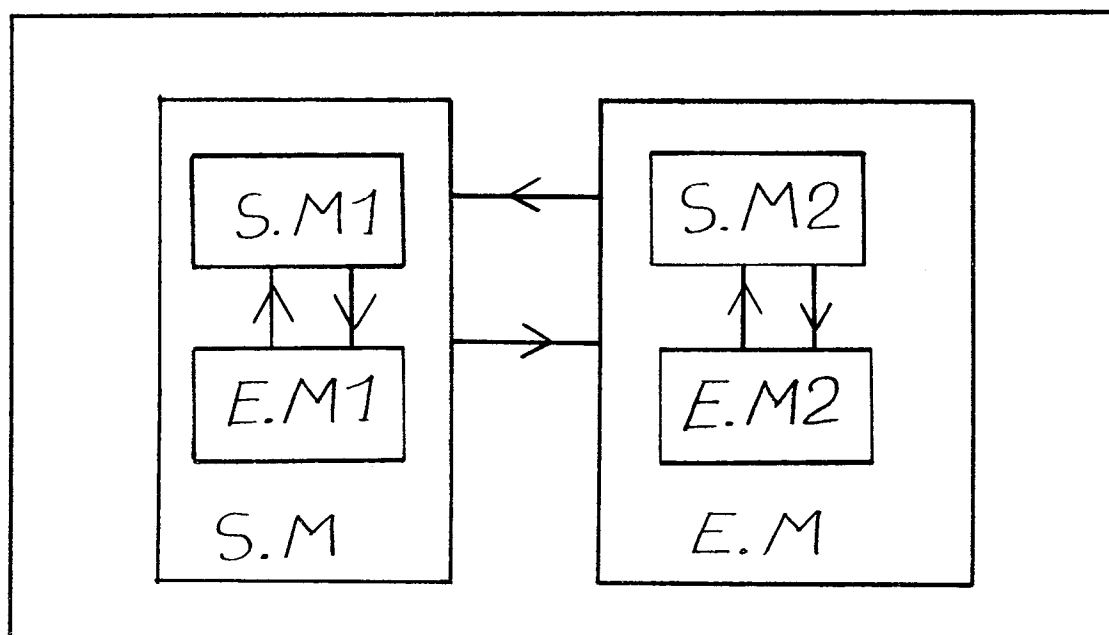
La E. M. envoie vers la S. M. l'information : "la tâche T est terminée".
En réponse, la S. M. va

- choisir une tâche T' à activer,
- charger l'adresse de T' dans un registre de E. M.

N.B. : Tout ceci suppose donc la préexistence d'un mécanisme quelconque de synchronisation entre S. M. et E. M.

Remarque :

On peut, dans un tel schéma, décomposer récursivement un S. M. où une E. M. en un système composé et ainsi de suite :



Exemple :

S. M. est un superviseur.

- SM1. est le gestionnaire des interruptions dans le superviseur.
- EM1. est l'ensemble des programmes de traitement des interruptions.

EM est l'ensemble des partitions gérées par SM.

Une de ces partitions est un système de télétraitement dont

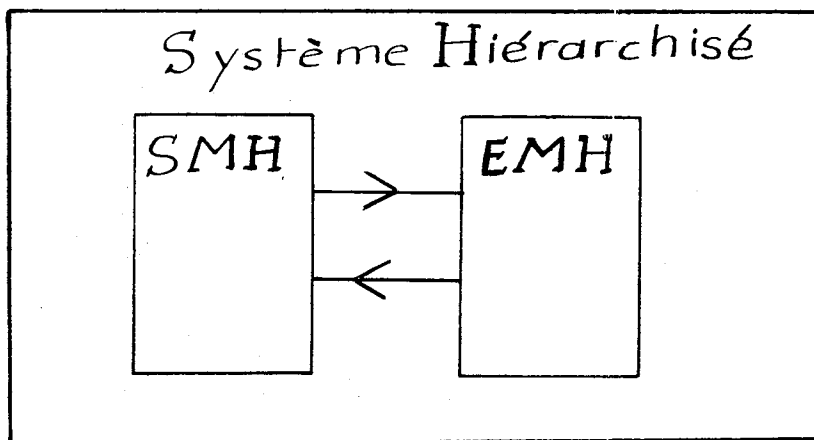
- SM2 est le moniteur,
- EM2 est l'ensemble des programmes des transactions.

C'est le cas par exemple du moniteur CICS sous le système DOS/VS sur les machines IBM 370.

I.2.3. UNE DEFINITION SUCCINCTE DE LA TSH

Il est difficile de définir complètement la TSH en quelques pages. Nous allons tenter d'en donner une définition formelle concise, et d'introduire les notions plus intuitives d'induction.

I.2.3.1. La TSH introduit d'abord un mécanisme particulier de synchronisation



La E. Machine est constituée d'un ensemble d'entités appelées articulations $\{a,b,\dots,n\}$ qui sont supposées vouloir se synchroniser entre elles deux à deux.

La S. Machine possède un ensemble d'entités appelées SDS (structures de synchronisation) : $\{\Delta_1,\dots,\Delta_P\}$.

La seule chose que peut faire la E. Machine est d'émettre vers la S. Machine une demande de la forme :

$$S(\Delta,a)$$

qui a la signification suivante :

l'articulation a demandé à la SDS de l'associer à un partenaire si elle n'en a pas encore, ou de changer son partenaire si elle est déjà associée.

La S. Machine a deux actions possibles envers la E. Machine :

ASSOCIER(a,b).

Dans le cas général, cette opération donne à chaque être associé le nom de son partenaire, de façon à ce qu'ils puissent travailler ensemble.

DISSOCIER(a,b) qui fait cesser l'association.

Finalement, la S. Machine est une boîte noire recevant en entrée des couples (Δ,a) et émettant en sortie des ordres ASSOCIER(a,b) ou DISSOCIER(a,b).

Un système hiérarchisé au sens de la TSH est un système :

A) dont la S. Machine a un algorithme particulier de la forme suivante :

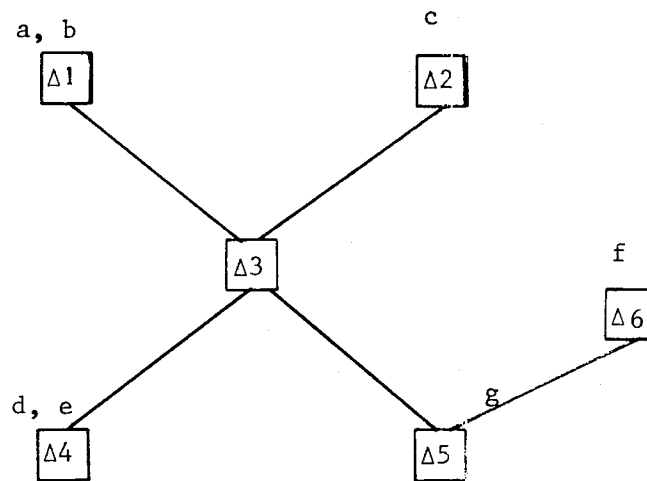
- il possède une donnée de base qui est un réseau orienté d'articulations sans circuits (encore appelé hiérarchie) dont chaque noeud est associé à une SDS Δ ,

- à chaque SDS Δ correspond un algorithme qui est exécuté chaque fois qu'une opération $S(\Delta, \text{articulation})$ est émise par la E. Machine. Cet algorithme peut à son tour émettre des ASSOCIER et DISSOCIER, mais aussi des $S(\Delta', \text{articulation})$ où Δ' est nécessairement jointive de Δ dans la hiérarchie. On appellera un tel appel émis à l'intérieur de la S. M. une opération de synchronisation induite, par opposition à celle émise depuis l'extérieur de la S. M. par la E. M.

B) dont la E. Machine a les propriétés suivantes :

- A toute articulation a de la E. Machine, on fait correspondre de manière injective, une SDS Δ de la hiérarchie.

Exemple :



- chaque articulation a de la E. M. ne peut émettre un $S(\Delta, a)$ que vers un Δ situé sur le même noeud que lui ou sur un noeud jointif.

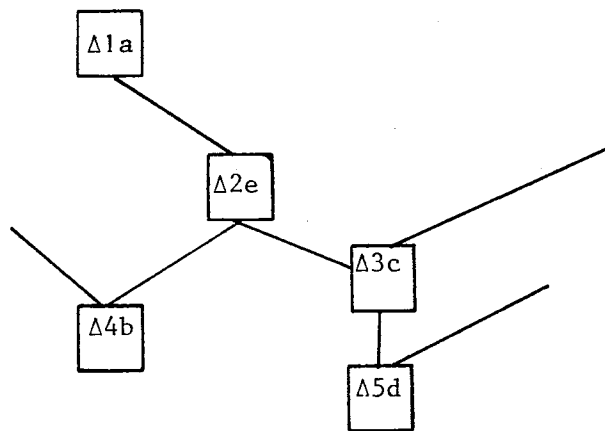
La sémantique intuitive des notions de la TSH peut alors être définie ainsi :

- les SDS Δ symbolisent des "lieux de rencontre" entre :

a) les articulations situées dans les noeuds prédécesseurs considérés comme des utilisateurs,

b) l'ensemble des articulations situés dans le même noeud et les noeuds successeurs, cet ensemble étant globalement considéré comme une seule ressource.

Exemple :



Δ2 gère la ressource constituée de (e,b,c,d).

On dit que cette ressource est représentée par e.

Pour solliciter une ressource, a exécute S(Δ2,a)

Pour solliciter un utilisateur, b exécute S(Δ2,b).

Les articulations multiples - Une articulation multiple peut être allouée simultanément à plusieurs utilisateurs. Elle représente un ensemble de ressources, vers lesquelles elle aiguille les demandes de ses utilisateurs. Il existe deux cas particuliers fréquents d'articulations multiples.

a) Les articulations multiples dupliquées : l'ensemble qu'elles représentent est constitué de ressources équivalentes.

b) Les articulations multiples dégénérées : elles représentent un ensemble de ressources vides. Elles servent à rassembler des utilisateurs devant être gérés de manière semblable ou bien à exprimer des conditions de synchronisation pure entre utilisateurs.

N.B. :

En SSH, on ne parlera pas d'articulations, mais de noeuds. La correspondance entre articulations de TSH et noeuds de SSH sera grossièrement la suivante :

- a) articulation ayant plusieurs successeurs (ressources)
→ "noeuds multiples" (définis en III).
- b) articulation multiple dégénérée → "noeuds ressource"
(définis en III.3.).
- c) autres articulations → "noeuds simples" (définis en II).

On remarquera que la notion de noeud multiple de SSH sera en quelque sorte duale de celle d'articulation multiple de TSH :
 noeud multiple de SSH \Rightarrow plusieurs successeurs (ressources)
 articulation multiple de TSH \Rightarrow plusieurs prédécesseurs (utilisateurs).

Remarque : Dans la pratique, les deux notions se recoupent fréquemment car on a à la fois plusieurs ressources et plusieurs utilisateurs.

I.2.3.2. Les inductions des primitives S dans la S. Machine

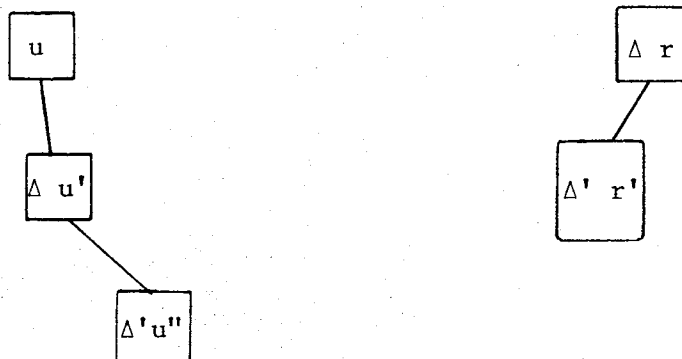
Dans certains cas particuliers très simples (qui sont souvent les plus courants), on vérifie que l'on peut regrouper les inductions en quatre classes seulement :

I.2.3.2.1. L'induction directe

A) $S(\Delta, u)$ où u est utilisateur ne peut induire que $S(\Delta', u')$ où u' est aussi utilisateur, et où Δ' est successeur de Δ :

c'est le cas où u sollicite une ressource représentée par u' gérée par Δ .

Δ décide d'associer u avec u' ; u' a elle-même besoin d'une ressource représentée par u'' gérée par Δ' , donc la S. Machine émet $S(\Delta', u')$.



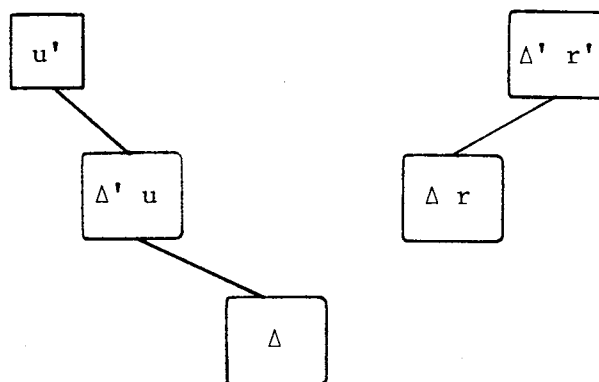
B) $S(\Delta, r)$ où r est ressource, ne peut induire que $S(\Delta', r')$ où r' est aussi ressource précédemment associée à r , et Δ' un successeur de Δ . C'est le cas où r émet $S(\Delta, r)$ pour solliciter un nouvel utilisateur ; si elle n'en trouve pas, la S. Machine émet $S(\Delta', r')$ qui signifie à Δ' , que r n'ayant plus d'utilisateur, n'a plus besoin des services de r' , donc que r' redemande un nouvel utilisateur à Δ' .

L'induction directe est ainsi appelée car elle se propage des feuilles vers la racine.

I.2.3.2.2. L'induction inverse

Elle se propage en direction des feuilles :

A) Si un utilisateur sollicitant une ressource Δ par $S(\Delta, u)$ n'obtient pas satisfaction, il convient, dans certains cas, qu'une ressource u' actuellement associée à u demande à une SDS Δ' , de lui fournir une autre ressource que u en émettant $S(\Delta', u')$.



B) Si une ressource r sollicitant un utilisateur auprès d'une SDS Δ obtient satisfaction et est associée à r' , il convient dans certains cas d'émettre $S(\Delta', r')$ pour signifier que r' devient de ce fait elle aussi candidate à un utilisateur.

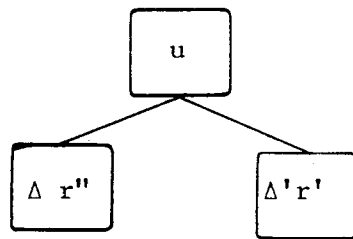
On dit que les inductions directes et inverses conservent la forme des opérations de synchronisation :

Si $S(\Delta, u)$ induit $S(\Delta, u')$ alors, u et u' agissent ou bien tous deux comme utilisateurs, ou bien tous deux comme ressources.

I.2.3.2.3. L'induction de libération

Elle ne conserve pas la forme des opérations de synchronisation :
c'est le cas où l'utilisateur u sollicitant une ressource
par $S(\Delta, u)$,

- n'obtient pas satisfaction,
- décide alors de libérer une ressource r' qu'il utilisait, celle-ci
va alors solliciter un nouvel utilisateur par $S(\Delta', r')$



I.2.3.2.4. L'induction de réquisition

C'est le cas opposé :
si la ressource r'' demande un utilisateur par $S(\Delta, r'')$,
si elle devient ainsi associée à u , celui-ci peut alors solliciter
d'autres ressources par $S(\Delta', u)$.

On voit que l'induction de libération permet de passer d'une
induction directe d'utilisateurs, à une induction inverse de ressource, et
que l'induction de réquisition fait passer d'une induction inverse de res-
source à une induction directe d'utilisateur.

I.3. LE PASSAGE DE TSH A SSH

Remarques sur les problèmes posés par le passage d'une théorie à un
langage devant respecter les concepts de la théorie.

Ces remarques sont apparues lors du passage de TSH à SSH, mais en fait,
elles ont une valeur plus générale, et s'appliquent chaque fois que l'on veut
implémenter des concepts par les moyens classiques de l'informatique :
définition d'un langage avec sa syntaxe, sa sémantique et son interpréteur.

Remarque 1 :

On doit souvent rajouter des choses prosaïques non évoquées dans la théorie. Dans notre cas, la TSH étudie surtout la synchronisation entre des activités, il nous faudra bien néanmoins introduire des moyens de décrire les activités elles-mêmes.

Remarque 2 :

L'exposé d'une théorie essaye souvent d'être minimal, et de ne pas introduire plus de concepts qu'il n'est nécessaire.

Au contraire, dans un langage, une certaine redondance est possible et même souhaitable : il y a en général plusieurs manières de programmer un algorithme, chacune répondant à des exigences particulières de l'utilisateur (efficacité, lisibilité, portabilité, etc...).

C'est pourquoi, bien que la TSH montre que seule la primitive de synchronisation S est suffisante, SSH comportera plusieurs primitives correspondant à des cas particuliers parfois redondants de S.

Remarque 3 :

Dans la définition d'un langage, on se trouve à tout moment face au choix entre "faire" et "laisser faire" :

vaut-il mieux proposer n instructions puissantes offrant chacune la solution à un problème donné, ou bien p ($p < n$) instructions de niveau plus bas avec lesquelles l'utilisateur pourra construire, grâce à un mécanisme inclus dans les p instructions, ses propres solutions (certainement plus que n).

Le second choix apporte plus de souplesse mais aussi plus de travail à l'utilisateur. D'autre part, il faut savoir où s'arrêter dans cette voie, car les p primitives peuvent à leur tour être décomposées en des primitives de plus bas niveau encore. Un compromis est donc nécessaire.

Nous pensons que le meilleur niveau de décomposition pour un ensemble d'applications donné est celui qui minimise le nombre d'instructions à écrire pour traiter ces applications. (Le critère utilisé est celui de la concision). En théorie, ceci nécessite par exemple des connaissances statistiques sur la fréquence des instructions exécutées pour cet ensemble d'applications.

Dans les faits, ces connaissances sont le plus souvent intuitives ou fondées sur l'expérience - difficilement quantifiable - du concepteur du langage. Les choix faits sont donc empreints d'un inévitable degré d'arbitraire.

Remarque 4 :

Il faut certes que la sémantique du langage soit parfaitement définie, et s'éloigne le moins possible de celle de la théorie, que des "garde-fous" existent dans le langage, en général sous forme de règles de modularité et de protection qui interdisent par exemple à n'importe quel sous-programme d'accéder à n'importe quelle donnée.

Mais il ne faut pas aller trop loin dans cette voie, car :

- cela conduit à restreindre la puissance du langage,
- on ne peut pas préjuger, en définissant un langage, des besoins futurs et de l'imagination des utilisateurs,
- un bon langage de simulation doit pouvoir simuler de mauvais modèles ("mauvais" au sens où ils ne respectent pas strictement les règles de telle ou telle école).

I.4. DEFINITION DU CAHIER DES CHARGES DE SSH

Ce chapitre passe en revue les différents postes du "cahier des charges" que nous fixons pour SSH, et pour chaque poste indique les options prises pour le réaliser.

Le cahier des charges est divisé en trois parties :

- caractéristiques sémantiques,
- caractéristiques linguistiques,
- caractéristiques opérationnelles.

I.4.1. LES CARACTERISTIQUES SEMANTIQUES DE SSH

Bien qu'il soit directement dérivé d'une théorie précise, SSH essaie de rester un outil permettant à différents utilisateurs de modéliser différents systèmes possédant des philosophies diverses concernant par exemple l'adressage, la protection, les communications, les performances, etc...

Les choix faits dans SSH ne doivent pas être considérés comme les choix d'un système, mais comme les choix d'un outil de modélisation de système.

I.4.1.1. Les objets manipulés par SSH

SSH essaie de dissocier au maximum différents aspects rencontrés dans les systèmes, et qui perdraient à être abusivement confondus.

Un modèle SSH comporte trois classes distinctes d'algorithmes :

- des algorithmes d'activité,
- des algorithmes de décision,
- des algorithmes d'archivage,

qui, grossièrement, correspondent respectivement aux aspects suivants d'un système :

- le calcul et la mémorisation des données,
- le contrôle,
- la mémorisation des événements utiles pour le contrôle.

La structure d'un modèle SSH est organisée autour de la notion de noeud, qu'on peut rapprocher de celle de SDS dans la TSH. A un noeud sont attachés des algorithmes des trois classes précédentes.

L'utilisateur décrira son système par un ensemble de noeuds - donc d'algorithmes - les interactions entre ces noeuds - donc ces algorithmes - étant exprimées par un réseau construit à partir de deux relations différentes,

- la relation fils-père,
- la relation de composition,

dont l'union est une relation analogue à celle "d'utilisateur-ressource" de la TSH.

1.4.1.2. Il faut distinguer 5 aspects principaux dans la définition sémantique de SSH

I.4.1.2.1. La structure du modèle

Elle est constituée de l'ensemble des noeuds du modèle et des relations hiérarchiques entre ces noeuds.

SSH possède des instructions pour déclarer des noeuds et pour créer des relations entre eux.

Cette structure peut évoluer dynamiquement au cours de la simulation : on peut ajouter, supprimer, recopier, modifier des noeuds, ajouter et supprimer des relations.

Ces facilités de modification dynamique imposent des conditions de modularité : on doit pouvoir substituer entre eux des noeuds, des ensembles de noeuds.

1.4.1.2.2. Les primitives de synchronisation

Comme cela a été dit plus haut, il n'y a pas une seule primitive S comme dans la TSH, mais plusieurs qui sont des cas particuliers de S, portent des noms plus parlants, et ont des paramètres implicites : par exemple, si le noeud N exécute la primitive "VIVRE" c'est équivalent à $S(\Delta, N)$ où Δ est une SDS successeur de N dans la hiérarchie de TSH.

D'autre part, la symétrie utilisateur / ressource est supprimée : ce seront des primitives différentes qui exprimeront la candidature d'un utilisateur à une ressource (par exemple, VIVRE) et la candidature d'une ressource à un utilisateur (par exemple, ASSOCIER).

1.4.1.2.3. Les mécanismes d'induction

A chaque noeud d'un modèle SSH vont être associés un ou plusieurs algorithmes de complexité arbitraire spécifiés par l'utilisateur et qui seront exécutés chaque fois qu'une primitive de synchronisation concernant le noeud est émise. Ces algorithmes pourront à leur tour émettre des primitives concernant d'autres noeuds. On retrouve ainsi les mécanismes d'induction de la TSH. Les inductions et les primitives qu'elles provoquent pourront être classées en deux catégories suivant leur signification :

- celles qui expriment des choses "positives" : poser sa candidature à une ressource, accepter de prendre en charge un utilisateur,
- celles qui expriment des choses "négatives" : cesser un travail, ne plus avoir besoin d'une ressource, abandonner un utilisateur.

1.4.1.2.4. Les mécanismes de contrôle

La base de tout système, informatique ou autre, est le fait que certaines entités du système peuvent savoir ce qui change dans certains autres endroits du système. C'est ce qu'en automatique on appelle "l'observabilité". C'est en fonction des informations ainsi reçues, que ces entités décideront de changer l'état du système, et ainsi de suite.

Dans la TSH, un changement d'état, c'est l'émission d'une primitive S, et une articulation peut observer ce changement si et seulement si une primitive (la même ou une autre) lui parvient par induction.

En principe, c'est à l'utilisateur de définir explicitement les mécanismes d'induction.

Afin d'alléger sa tâche, certaines inductions sont faites implicitement dans SSH.

Si une ressource cesse son travail pour une raison quelconque, tous les noeuds qui ont soumis un travail à cette ressource sont automatiquement avertis, libre à eux ensuite d'en tenir compte ou non.

Un noeud d'un modèle hiérarchisé met en évidence deux types de synchronisation et de contrôle, qui sont habituellement envisagés séparément ; ce sont respectivement :

- la "synchronisation haute", c'est-à-dire la gestion des demandes qu'un noeud reçoit et des offres qu'il émet en réponse. Classiquement, c'est le domaine des moniteurs, des ordonnanceurs,

- la "synchronisation basse", c'est-à-dire la gestion des demandes qu'un noeud émet et des offres qu'il reçoit en réponse. C'est le domaine des primitives sur sémaphores, des requêtes et libérations de ressources.

Si d'un côté, cela simplifie les problèmes de synchronisation en les structurant, d'un autre, cela les amplifie en les généralisant.

Ainsi, la question de savoir par exemple ce que doit être un algorithme de gestion de ressources performant et sûr - qui n'est encore qu'imparfaitement résolue dans des cas particuliers simples par l'algorithme du Banquier HABERMAN [8] - se pose avec plus de force encore dans un réseau de ressources.

Dans cet esprit, SSH introduit la notion de fonctions d'archivage associées à un noeud, qui sont destinées à faciliter et mettre en évidence la tenue à jour des structures de données utilisées par les algorithmes de synchronisation et de contrôle.

I.4.1.2.5. Les activités des noeuds et leur interprétation en temps simulé

Il ne faudrait pas oublier que toutes ces structures, inductions et synchronisations ne sont là que pour contrôler le déroulement et la coopération de certaines activités associées aux noeuds.

Il n'y a pas grand chose à en dire, si ce n'est qu'elles peuvent être quelconques, donc qu'elles doivent être implémentées avec la plus grande généralité possible.

Elles pourront soit initialiser elles-mêmes des inductions de synchronisation, soit communiquer avec les algorithmes de contrôle.

Dans l'implémentation actuellement choisie pour SSH, ces activités peuvent de plus exécuter des instructions de temporisation, demandant que soit simulé l'écoulement d'un certain temps lors de l'interprétation de ces activités.

On verra dans la suite que la simulation du temps dans une structure hiérarchisée présente des propriétés très intéressantes à exploiter.

I.4.1.2.6. Séparation claire entre les instructions de simulation du temps et les autres instructions décrivant le modèle

Cela revient à dire que l'algorithme de simulation du temps - souvent considéré comme le coeur même d'un langage de simulation - n'est en vérité qu'un interpréteur envisageable parmi beaucoup d'autres. On aurait pu par exemple, implémenter SSH non pas en temps simulé mais en temps réel sur un système multitâches.

I.4.2. LES CARACTERISTIQUES LINGUISTIQUES DE SSH

I.4.2.1. L'implémentation actuelle

Dans le chapitre "motivations" a été exposée l'une des raisons pour lesquelles SSH n'est pas un nouveau langage avec une nouvelle syntaxe et un nouveau compilateur :

- l'utilisateur préfère utiliser son langage habituel plutôt qu'un nouveau produit souvent disproportionné à ses besoins réels.

- Une autre raison est que travail d'écriture d'un compilateur qui, comme tous les compilateurs devrait traiter des expressions arithmétiques, gérer des tables de symboles, imprimer des messages d'erreur, ne nous aurait guère laissé de temps pour nous intéresser au langage lui-même.

C'est pourquoi les "instructions" SSH sont en fait des appels de sous-programmes d'un langage hôte.

Cela présente d'ailleurs au passage des avantages par rapport aux langages classiques :

- puisque générer une instruction SSH, c'est appeler un sous-programme du langage hôte, on peut utiliser des techniques de macro-génération conditionnelle très sophistiquées,

- les instructions de déclaration des composants du modèle (noeuds et relations) seront complètement dynamiques.

D'autre part, une technique particulière a dû être mise au point pour résoudre les problèmes de la simulation du temps et de l'ordonnancement des événements dans un langage de programmation quelconque. Ceci est exposé en II.3.1.10. et dans toute la partie VI.

I.4.2.2. Autres possibilités d'implémentation

Une implémentation "à part entière" et non dans un langage hôte de SSH est évidemment possible. Elle pourrait tenir compte des remarques suivantes :

Dans les pages précédentes, des aspects très divers ont été isolés :

- construction d'une structure,
- induction des primitives de synchronisation,
- mécanismes de contrôle,
- description des activités,
- instructions de simulation,
- etc....

Chacun de ces aspects possède sa sémantique particulière, et ils ont entre eux des relations parfaitement définies.

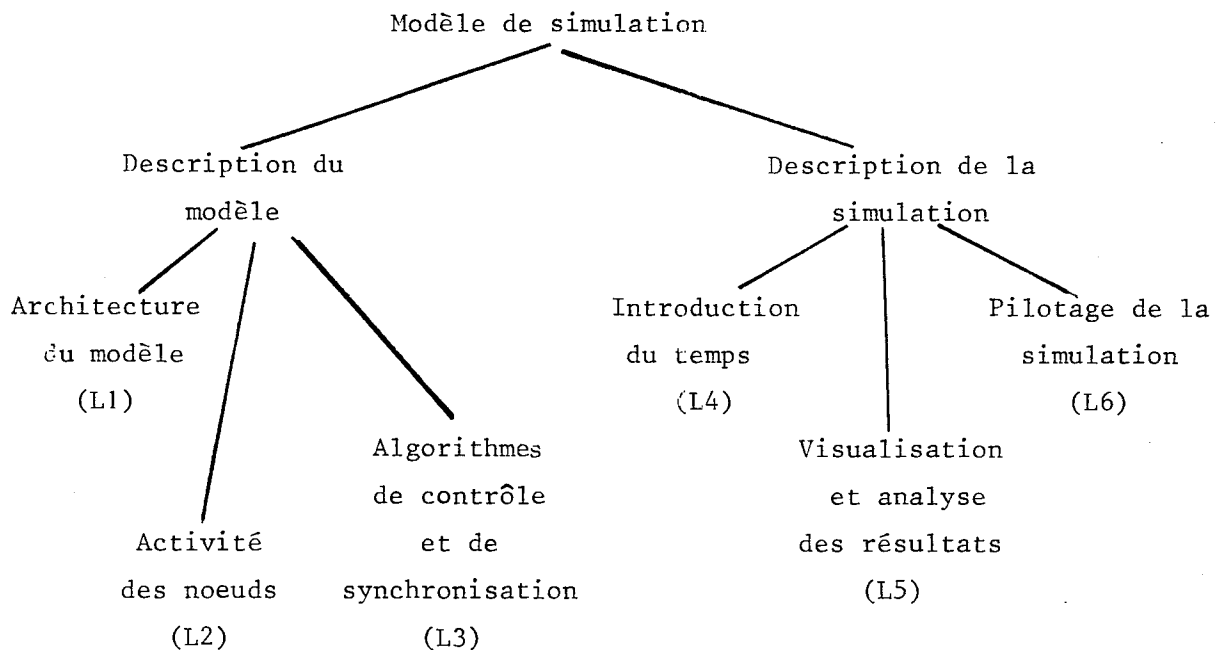
Il serait dommage de les confondre dans un seul langage universel.

Il faudrait plutôt définir autant de langages spécialisés qu'il y a d'aspects indépendants dans SSH, ainsi qu'un langage spécialisé dans l'assemblage des langages précédents.

Eclater en plusieurs langages la construction d'un modèle de simulation est d'un intérêt évident.

Cela éviterait un grand nombre de problèmes rencontrés dans l'écriture de modèles utilisant, comme cela est habituel, un langage unique, que ce soit ou non un langage de simulation.

Nous proposons le découpage suivant : (sans le prétendre canonique)



FIGURE

Bien souvent, les langages de simulation ont été construits autour de seulement un ou deux aspects représentés sur la Figure et les autres en souffrent :

Exemples :

SLMScript, SIMULA : autour de L2

GPSS : autour de L1 et L5

ALICE [9] , CASSANDRE [10] L1, L5, L6

Par opposition, adopter le point de vue d'ALGOL 68 [17] qui veut être un langage orthogonal, c'est-à-dire un langage où n'importe quelle opération doit avoir un sens sur n'importe quelle entité du langage, n'aboutit qu'au gigantisme et à la confusion, alors que l'utilisateur aspire à la clarté et à la réduction des langages à ses besoins.

I.4.2.3. Adressage modulaire de l'environnement d'un noeud

L'entité de base de SSH, le noeud, se retrouve à divers niveaux dans l'adressage :

- il y a un type de variables "référence de noeud",
- tout sous-programme écrit par l'utilisateur, lors de son appel, est nécessairement exécuté dans un contexte strictement délimité : par exemple, le nom d'un noeud ayant émis une primitive, les autres noeuds concernés par cette primitive, qui seront en général des noeuds voisins dans la hiérarchie,
- toute structure de donnée du modèle est nécessairement rattachée à un noeud, et ne peut être accédée qu'à travers le nom de ce noeud.

I.4.2.4. Conclusion

La raison première de la concision d'un modèle écrit en SSH est la "justesse sémantique" de la TSH, qui met en lumière les concepts véritablement utiles pour exprimer le fonctionnement d'un système. SSH essaie donc de ne pas perdre cette concision en la "délayant" dans un verbiage syntaxique.

I.4.3. LES CARACTERISTIQUES OPERATIONNELLES DE SSH

Il faut les distinguer de celles du langage hôte choisi, qui, en lui-même, possèdera plus ou moins de facilités de mise au point, qui pourra être conversationnel, qui sera plus ou moins "gourmand" en taille mémoire et en temps d'exécution.

A) Les performances

Il convient de dissocier les performances de l'algorithme général de celles de ses implémentations possibles.

L'algorithme tel qu'il est décrit en P.I. dans la troisième partie n'est pas d'un coût prohibitif, car :

- il n'utilise pas de procédures récursives,

- il utilise le moins possible d'appels de procédures, et presque toujours sans paramètres,
- il possède sa propre gestion de mémoire dynamique.

D'autre part, il est suffisamment simple et compréhensible pour pouvoir être à nouveau simplifié et optimisé si l'on ne désire pas implémenter l'intégralité de SSH.

B) Les interactions utilisateur-modèle

La modularité inhérente à SSH va permettre les dispositions élégantes suivantes :

- il est possible d'introduire dynamiquement des noeuds espions pouvant observer le reste du système sans le perturber ni avoir besoin de le connaître en détail,
- il suffit alors que les algorithmes de contrôle de ces noeuds espions communiquent - conversationnellement par exemple - avec l'utilisateur.

CONCLUSION

En conclusion de cette première partie, nous voulons insister sur deux aspects importants qui résultent des choix faits dans SSH.

1) SSH abandonne au langage hôte un grand nombre de problèmes purement linguistiques qui sont sans rapport avec la sémantique proprement dite de SSH. C'est le cas en particulier pour toutes les questions de désignation, de portée des variables référençant des objets SSH.

Par exemple, il existe des instructions SSH permettant de construire le réseau des noeuds. Dans ces instructions, les noeuds seront désignés par une variable de langage hôte. Tout ce qui nous importe, c'est que le langage hôte réalise bien, à l'instant où l'instruction SSH est exécutée, la liaison entre la variable et la représentation en langage hôte du noeud considéré.

Par la suite, il n'existera en général aucune bijection obligatoire entre des variables du langage hôte et les noeuds SSH. Ceux-ci seront désignés par des mécanismes propres à SSH.

2) SSH, bien que conçu dans le cadre d'une théorie concernant les systèmes informatiques, peut fort bien simuler autre chose que des problèmes informatiques. (Cette remarque est d'ailleurs valable également pour la TSH). Il nous a donc semblé légitime d'introduire dans SSH des facilités apparamment trop sophistiquées ou inutiles pour modéliser des systèmes informatiques actuels, mais qui, en fait, correspondent bien à la puissance intrinsèque de la TSH, indépendamment de son origine informatique.

Ceci est le cas en particulier pour certains concepts introduits en III à propos des noeuds multiples.

Dans les pages qui suivent, il convient donc, pour chaque notion de SSH rencontrée, de se rappeler que :

- ses conditions effectives de mise en oeuvre peuvent dépendre du langage hôte choisi, avec ses avantages et ses inconvénients,

- sa justification peut ne pas découler directement de besoins de modèles purement informatiques.

II. DEUXIEME PARTIE

LES NOEUDS SIMPLES DE SSH

Un grand nombre de primitives sont définies dans cette partie et les suivantes. Dans le but d'unifier leur présentation et de rendre ultérieurement plus aisée une consultation "aléatoire" de leurs définitions, la présentation ordonnée suivant a été adoptée :

- toute spécification d'une primitive est indiquée par deux astérisques ("**") précédant le nom de la primitive,
- ensuite, on trouve la description de la syntaxe,
- puis de la sémantique,
- enfin, des justifications comportant éventuellement des exemples d'utilisation.

RAPPEL : SSH se présente sous la forme d'un ensemble de sous-programmes écrits dans un langage hôte. Comme la syntaxe des appels de sous-programmes varie d'un langage à l'autre, nous utilisons pour les besoins de l'exposé un pseudo-langage (ou P.L.) qui va servir à décrire l'aspect externe de SSH et qui servira plus loin à décrire son implémentation.

II.1. DEFINITION DU PSEUDO-LANGAGE P.L. UTILISE COMME LANGAGE HOTE
POUR CE TRAVAIL

Le P.L. est utilisé ici

- pour programmer les algorithmes associés aux noeuds donnés en exemple,
- pour décrire l'algorithme général d'implémentation de SSH en annexe.

La plupart des caractéristiques du P.L. ont la sémantique et la syntaxe des langages du style ALGOL : SI... ALORS ... SINON

DEBUT ... FIN

FAIRE ..., TANT QUE

Nous donnons simplement ici les dispositions spéciales au P.L., qui concernent surtout des points de détail ou certaines notations plus agréables pour SSH.

a) Il existe des variables de type référence de zone mémoire, de sous-programme, de fonction, d'adresse d'instruction.

b) Un sous-programme de nom SSP possédant n paramètres s'appelle ainsi : SSP(P1, PN),

Une fonction de nom FONCT à n paramètres est un sous-programme qui retourne une valeur ; elle est utilisable par exemple dans une instruction d'affectation $A \leftarrow \text{FONCT}(X,Y)$.

c) Valeurs par défaut : si le sous-programme a N paramètres dans sa définition, si il y a p < N virgules dans son appel, cela signifie que les paramètres p + 2 à N sont pris par défaut. De même si un paramètre est vide (deux virgules consécutives), on prend sa valeur par défaut.

Exemple

SOUS-PROGRAMME P(A=5,B,C,D=7)

FIN P

L'appel P(,X,Y) est équivalent à P(5,X,Y,7).

d) Il existe un opérateur DE pour adresser un élément d'une zone de mémoire. Une zone de mémoire est une suite de mots.

Exemple : COMPTEUR DE ZONE

ZONE doit être une référence de zone de mémoire.

COMPTEUR doit avoir une valeur entière $i \geq 0$.

Alors COMPTEUR DE ZONE désigne le i ème mot de la zone mémoire.

Un modèle SSH va se présenter comme un sous-programme P.L., qui utilisera certains sous-programmes, certaines fonctions, certaines variables et constantes prédéfinies, dont l'ensemble constitue les mots-clés de SSH. Pour plus de clarté, nous soulignerons ces mots-clés afin de les distinguer du reste du programme en P.L., mais cela n'a en fait aucune valeur syntaxique.

II.2. CONVENTIONS POUR LA CONSTRUCTION DES NOEUDS SIMPLES

II.2.1. NOTION D'EVENEMENTS

Toute instruction de synchronisation de SSH génère un événement se traduisant par :

- l'appel de certains algorithmes attachés au noeud destinataire de l'événement,
- la mise à la disposition des algorithmes appelés de certaines variables prédéfinies qui vont caractériser cet événement.

II.2.2. LES DIFFERENTES CLASSES D'ALGORITHMES ATTACHEES AUX NOEUDS

Une classe d'algorithmes est caractérisée par :

- les noms des variables prédéfinies, caractéristiques de l'événement auxquelles un algorithme de cette classe a accès lorsqu'il est appelé.

Il existe trois classes d'algorithmes associés à un noeud :

- la classe de décision,
- la classe d'activité,
- la classe d'archivage.

II.2.2.1. Conventions d'appel d'un sous-programme de la classe de décision

C'est un sous-programme qui n'a pas de paramètres.

Il est automatiquement appelé chaque fois qu'un événement se produit, concernant un fils du noeud auquel il est attaché, (événement ayant son origine dans l'émission d'une primitive de synchronisation).

Il peut alors accéder en lecture aux variables prédéclarées suivantes :

MOI qui donne le nom du noeud pour lequel le sous-programme opère (car des noeuds différents peuvent utiliser le même algorithme de manière réentrante),

LUI qui donne le nom du fils concerné par l'événement,

MOTIF qui, comme son nom l'indique, explique la nature de l'événement.

Un noeud peut être alerté pour l'un des motifs suivants :

ILARRIVE qui signifie que LUI se porte candidat à MOI,

ILSENVA qui signifie que LUI s'arrête d'être candidat,

ILTOURNE qui signifie que LUI entre dans un certain état d'activité qui sera défini plus loin,

ILSTOPPE qui signifie que LUI quitte cet état d'activité,

CESTMOI qui signifie qu'il appelle lui-même ses algorithmes de décision depuis un algorithme de la classe d'activité LUI transmet alors une information de type quelconque,

ORIGINE indique comment l'événement est parvenu jusqu'à MOI.

Il peut prendre les valeurs :

DIRECTE : c'est le cas normal

SUR } ce sont des cas particuliers dont

SOUS } la sémantique est expliquée en III.4.1.

II.2.2.2. Conventions d'appel d'un sous-programme de la classe d'activité

Il n'a pas de paramètres.

Il est appelé chaque fois que le noeud simple dans la déclaration duquel il apparaît, dispose de toutes les ressources nécessaires à son exécution.

Il peut alors accéder aux variables prédéclarées suivantes :

SUITE qui est une variable du type référence d'instruction, qui lui indique à quel endroit il doit commencer son exécution.

Typiquement, la première instruction utile d'un sous-programme de la classe devra donc être de la forme :

GOTO SUITE

MOI qui donne le nom du noeud pour lequel il est présentement exécuté.

II.2.2.3. Conventions d'appel d'un sous-programme de la classe d'archivage

Il n'a pas de paramètres.

On distingue deux sous-classes :

a) La sous-classe d'archivage externe.

Les sous-programmes de cette classe sont appelés dans les mêmes conditions que ceux de la classe de décision, et peuvent accéder aux mêmes variables : (MOI, LUI, MOTIF, ORIGINE).

Le sous-programme d'archivage externe est appelé avant les algorithmes de décision.

b) La sous-classe d'archivage interne.

Elle est appelée chaque fois que le noeud lui-même auquel elle est associée émet une primitive de synchronisation (et non pas lorsqu'il en reçoit une comme dans le cas a) précédent). Le sous-programme peut alors accéder à :

MOI, LUI avec le même sens qu'auparavant

ACTION qui donne le nom de la primitive émise par MOI

LAUTRE qui donne le nom d'un autre noeud qui peut figurer en paramètre ACTION, (s'il existe, le premier étant LUI)

II.3. DECLARATION D'UN NOEUD DE TYPE SIMPLE

A ← DN(SIMPLE, PERE.DE(<liste de fils>), ALGORITHMES(<liste d'algorithmes>),
 ESPACE.LOCAL(<taille>, <fonctions de service>))

II.3.1. * * PERE.DE : SYNTAXE

<liste de fils> est de la forme :

A1, A2, ..., AN

où les Ai sont des références de noeuds simples.

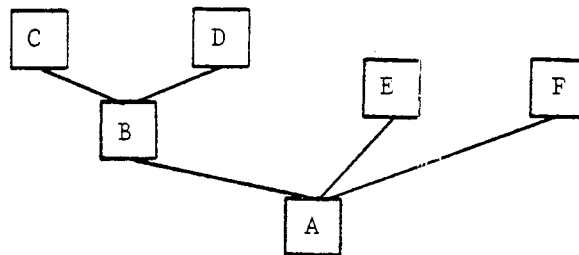
* * PERE.DE : SEMANTIQUE

Cette instruction établit une relation binaire dite relation père-fils entre le noeud déclaré et chacun des noeuds paramètres. Cette relation est analogue à la relation ressource-utilisateur de la TSH.

Cette relation doit conférer à l'ensemble des noeuds simples une structure de forêt (ensemble d'arbres)

Exemple et représentation graphique

A ← DN(SIMPLE, PERE.DE(B ← DN(SIMPLE, PERE.DE(C,D), ...), E.F), ...)



II.3.2. * * ESPACE.LOCAL : SYNTAXE

<taille> doit avoir une valeur entière non négative.

<fonctions de service> est de la forme

<initialisation>, <recopie>, <destruction>

où ces trois entités sont des références de sous-programmes à un paramètre.

* * ESPACE.LOCAL : SEMANTIQUE

Lors de la déclaration du noeud, on va allouer une zone de mémoire de (n = valeur de <taille>) emplacements consécutifs, que l'on appellera espace local du noeud.

Cet espace local pourra être accédé par l'intermédiaire du nom du noeud.

Par exemple, dans le P.L. que nous utilisons, si A référence un noeud l de A désigne le premier élément de l'espace local du noeud, COMPTEUR DE A désigne le ième élément si COMPTEUR a une valeur entière positive égale à i.

N.B. : Dans certains langages hôtes, on utilisera d'autres notations :

en A E D : COMPTEUR(A) où A est de type pointeur

en assembleur 370 :

COMPTEUR(A) où A est un registre

et COMPTEUR, un déplacement

en PL/1 : COMPTEUR seul si COMPTEUR est une sous-structure
d'une structure BASED(A)

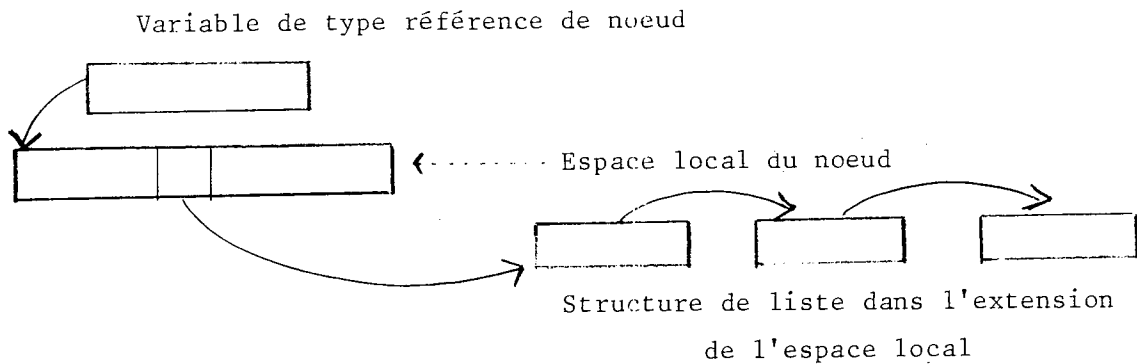
Après la réservation de cette zone locale, la déclaration du noeud provoque l'appel du sous-programme <initialisation> avec le nom du noeud comme paramètre.

Ce sous-programme pourra par exemple initialiser le contenu de l'espace local, mais éventuellement il pourra aussi acquérir de l'espace mémoire supplémentaire - si cette facilité existe dans le langage hôte - pour par exemple créer et initialiser des structures de listes, des files, ou toute autre structure de données utile au noeud, et trop complexe pour être gérée dans le seul espace local.

Dans ce cas, il est fondamental qu'une telle extension de l'espace local soit référencée par l'intermédiaire de l'espace local, et seulement par son intermédiaire, si l'on veut conserver une saine gestion des accès aux données dans le modèle.

Par exemple, tous les pointeurs sur d'éventuelles têtes de liste ou de file de l'extension de l'espace local devront être implémentés dans l'espace local lui-même.

Le schéma d'accès est le suivant :



II.3.3. SOUS-PROGRAMMES <RECOPIE> ET <DESTRUCTION>

Leur sens est alors clair :

Puisque d'une part, SSH est dynamique et permet de générer des copies d'un noeud ou de détruire des noeuds, et que, d'autre part, les extensions des espaces locaux des noeuds peuvent avoir une structure quelconque inconnue du système, c'est à ces programmes spécifiés par l'utilisateur qu'il revient d'assurer la recopie et la destruction de ces extensions.

Ces fonctions de service sont appelées par les instructions SSH suivantes :

** COPIE.DE

Syntaxe A ← COPIE.DE(B)

Sémantique :

Cette instruction crée un nouveau noeud A identique au noeud B, et en particulier appelle le sous-programme <recopie> de ESPACE.LOCAL pour dupliquer l'extension de l'espace local de B; <recopie> est appelé avec 2 paramètres : ancien noeud, nouveau noeud.

** DETRUIRE

Syntaxe : DETRUIRE(A)

Sémantique : Cette instruction détruit le noeud A. La destruction de l'extension de l'espace local est réalisée par l'appel de <destruction>. <destruction> est appelé avec un paramètre : le nom du noeud dont elle doit détruire l'espace.

II.3.4. ** ALGORITHMES - SYNTAXE

<liste d'algorithmes> est de la forme :

<décision 1>, <archives externes>, <archives internes>, <décision 2>, <activité>

<activité> doit être une variable de type référence de sous-programme ; les autres paramètres sont :

- soit de la même forme que <activité> ,
- soit de la forme :

** CONFIE.A (<noeud 1>, <noeud 2>, <activité>)

où <noeud 1> est un noeud de type SIMPLE ou MULTIPLE,

<noeud 2> est un noeud de type SIMPLE,

<activité> est une référence de sous-programme.

La sémantique de CONFIE.A est étudiée à part dans IV.2.

** ALGORITHMES: SEMANTIQUE

<activité> doit référencer un sous-programme de la classe d'activité définie plus haut.

<décision fils> et <décision père> doivent référencer un sous-programme de la classe de décision définie plus haut.

<archives externes> et <archives internes> doivent référencer un sous-programme des classes d'archivage définies plus haut.

II.3.4 BIS. ECRITURE ET SEMANTIQUE DES ALGORITHMES <DECISION-PERE>

D'UN NOEUD SIMPLE

Un sous-programme <décision-père> peut exécuter deux instructions SSH :

VIVRE et MOURIR

* * VIVRE signifie que le noeud qui l'exécute se porte candidat auprès de son père, tel qu'il est défini par l'état courant du réseau des relations père-fils. Ces relations sont initialement créées par l'instruction PERE.DE lors de la déclaration du père, mais nous verrons par la suite qu'elles peuvent varier au cours de la simulation.

* * MOURIR signifie que le noeud qui l'exécute cesse d'être candidat auprès de son père.

VIVRE exécuté par un noeud F correspond à l'événement ;

MOI : le père de F, LUI = F, MOTIF = ILARRIVE

MOURIR correspondant à la même chose pour MOI et LUI et

MOTIF = ILSENVA.

Exemple 1 :

SI MOTIF = ILARRIVE ALORS VIVRE

Exemple 2 :

SI MOTIF = ILARRIVE ALORS DEBUT

SI (COMPTEUR DE MOI ← COMPTEUR DE MOI + 1) ≥ N

ALORS VIVRE FIN

SI MOTIF = ILSENVA ALORS DEBUT

SI (COMPTEUR DE MOI ← COMPTEUR DE MOI - 1) < N

ALORS MOURIR FIN

Commentaires :

Le deuxième exemple exprime que le noeud ne se porte lui-même candidat à son père que lorsqu'il a au moins N fils candidats à lui-même. (En supposant que COMPTEUR est initialisé à zéro). Il apparaît clairement que les algorithmes de <décision-père> sont ceux qui peuvent provoquer le phénomène de l'induction directe telle qu'elle est présentée dans THS : (en I.2.3.2.1.).

Exemple :

- Un noeud fait VIVRE dans sa fonction <décision-père> ,
- L'événement MOI = Père du noeud, LUI = le noeud, MOTIF = ILARRIVE est généré,
- La fonction <décision-père> du père est alors appelée, et au vu de l'événement, peut éventuellement à son tour faire VIVRE, et ainsi de suite.

De même un MOURIR exécuté par un noeud peut induire des MOURIR chez ses ascendants.

On retrouve bien la conservation de la forme des opérations de synchronisation indiquée par la TSH.

II.3.5. ECRITURE ET SEMANTIQUE DES SOUS-PROGRAMMES <DECISION-FILS>

Un sous-programme <décision-fils> peut exécuter deux instructions SSH.

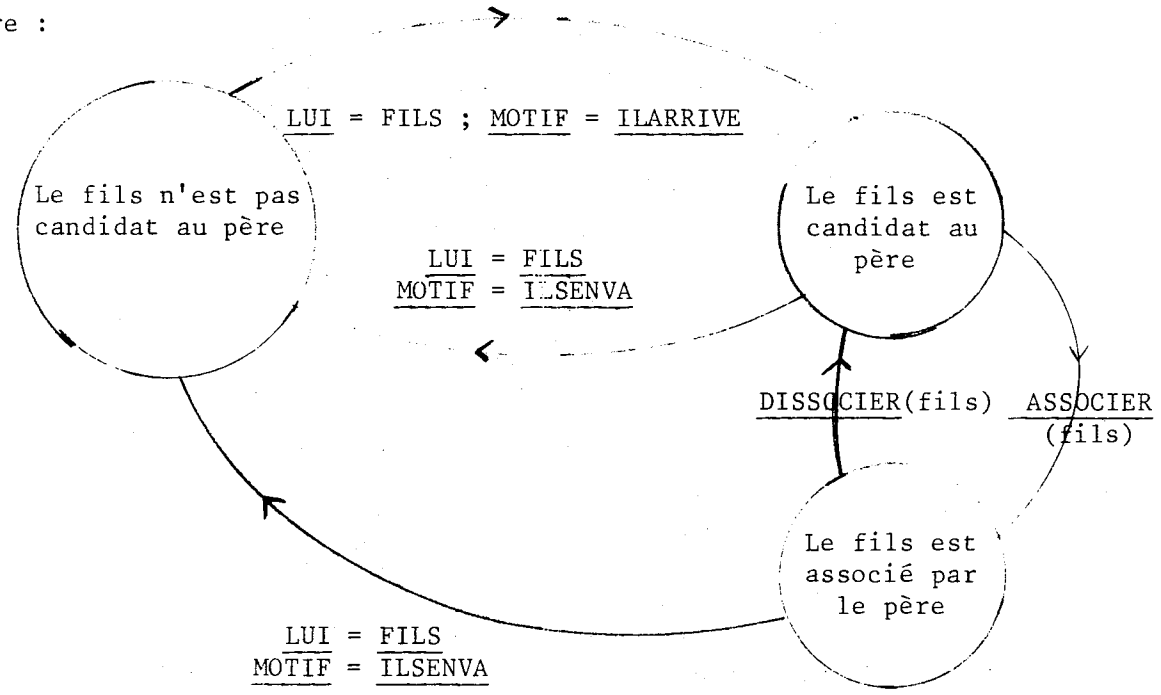
* * ASSOCIER(F)

F doit désigner un noeud fils de MOI (donc un noeud simple), tel que l'on ait déjà eu MOTIF = ILARRIVE et LUI = F.

* * DISSOCIER(F)

F doit être un fils de MOI précédemment associé.

Diagramme des situations possibles d'un noeud fils vis-à-vis de son père, comme elles sont vues par le sous-programme <décision-fils> du père :



Exemples d'instructions dans un programme décision-fils :

Exemple 1 :

```

SI MOTIF = ILARRIVE ALORS ASSOCIER(LUI)
  Le père associe le nouvel arrivant.
  
```

Exemple 2

```

SI MOTIF = ILARRIVE ET PRIORITE DE COURANT DE MOI < PRIORITE DE LUI
ALORS DEBUT
  DISSOCIER(COURANT DE MOI)
  COURANT DE MOI ← LUI
  ASSOCIER(LUI)
FIN
  
```

Commentaires

Les noeuds fils possèdent un élément désigné par PRIORITE dans leur espace local. Le père possède un élément désigné par COURANT dans son espace local qui contient le nom du fils associé à un instant donné.

Note importante

Dans le cas où MOTIF : ILSENVA et où LUI est associé, il est inutile de faire DISSOCIER(LUI) ; ceci est fait automatiquement par SSH.

II.3.6. DEFINITIONS PREALABLES

Noeud simple actif ou inactif :

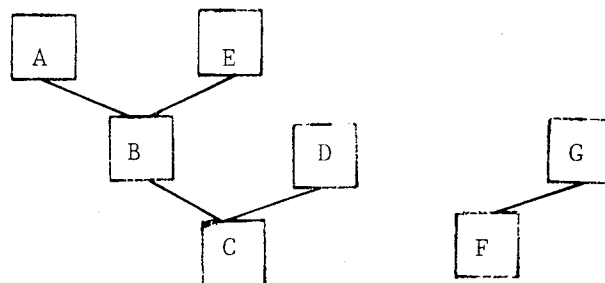
Un noeud simple est actif si et seulement si :

- il a un père actif et a été associé par son père après avoir été candidat,
- ou bien, il n'a pas de père et il a fait VIVRE.

Donc, un noeud simple est actif si et seulement si il appartient à un chemin de noeuds actifs le reliant à un noeud sans père.

Lorsqu'un noeud simple est actif, son sous-programme d'activité, s'il existe, s'exécute dans le temps simulé.

Exemple :



A est actif \Rightarrow B et C sont actifs.

A est actif \Rightarrow A est associé par B, B associé par C et C a fait VIVRE.

B est actif \Rightarrow C est actif.

B est actif $\not\Rightarrow$ A est actif.

II.3.7. ECRITURE ET SEMANTIQUE D'UN SOUS-PROGRAMME <ACTIVITE>

Il peut exécuter les instructions SSH suivantes :

* * ENVOI(P,F)

où P désigne un noeud simple ou multiple et F un noeud simple.

Elle est équivalente à la suite d'actions suivantes :

- MOURIR exécuté par F sur son père actuel,
- le père de F devient P, donc la structure hiérarchique est modifiée,
- VIVRE exécuté par F sur son nouveau père P.

* * INFO(P,F)

où P est un noeud quelconque et F une variable quelconque.

Elle génère l'événement suivant :

MOI = P ; LUI = F ; MOTIF = ILINFORME.

Cette instruction peut être utilisée par exemple pour transmettre un message au noeud P par l'intermédiaire du noeud F, sans vouloir envoyer le noeud F comme candidat, comme l'aurait fait un ENVOI(P,F).

* * DECISION(X)

où X est quelconque (pas nécessairement un noeud).

Elle génère l'événement suivant, lorsqu'elle est exécutée par l'activité du noeud N.

MOI = N ; LUI = X ; MOTIF = CESTMOI.

En conséquence, les fonctions de décision et d'archivage de N sont appelées, et peuvent accéder en particulier au paramètre X, passé dans LUI.

Remarque :

Une autre manière de procéder pour que la fonction d'activité appelle les fonctions de décision est d'exécuter :

ENVOI(MOI, noeud simple)

L'instruction DECISION est à rapprocher de certains usages des instructions d'interruption programmée dans certaines machines (SVI sur les IBM 370).

**** DUREE(D, <FONCTION DE REPRISE>)**

D doit être de type scalaire positif ou nul.

<Fonction de reprise> est facultatif et désigne une fonction à un paramètre de type scalaire, qui retourne un résultat de même type.

Cette instruction signifie qu'il faut simuler l'écoulement de D unités de temps entre l'exécution de l'instruction qui la précède et celle de l'instruction qui la suit.

Par définition, une instruction DUREE - comme toutes les instructions d'une fonction d'activité - est exécutée alors que le noeud est actif.

Un noeud peut passer de l'état actif à inactif à un instant donné dans le temps simulé. Si cela se produit alors que l'on est en train d'écouler l'intervalle spécifié par DUREE, les choses se passent ainsi :

- si le second paramètre <fonction de reprise> était absent, la portion de temps ΔT déjà écoulée est décomptée, et, lorsque le noeud redeviendra actif, il ne restera plus que $D - \Delta T$ unités de temps à simuler avant l'exécution de l'instruction suivante ;

- si au contraire, l'instruction était de la forme :

DUREE(D, f), alors on décomptera non pas ΔT unités de temps, mais $f(\Delta T)$.

II.3.8. EXEMPLES D'UTILISATION DU SECOND PARAMETRE DE L'INSTRUCTION DUREE :

LA FONCTION DE REPRISE

A) La fonction de reprise est la fonction identiquement nulle :

Cela signifie que si l'instruction DUREE est interrompue, il faut tout recommencer. C'est une manière simple de modéliser des systèmes où il n'y a pas de sauvegarde du contexte d'une tâche interrompue, qui doit donc être relancée jusqu'à pouvoir s'exécuter jusqu'au bout sans interruption.

B) Simulation économique du ralentissement dû aux changements de contextes :

Soit $FR(\Delta) = \Delta - OVH$

où OVH est une constante.

Chaque fois que le noeud est interrompu, son temps recule de OVH. Il ne reviendra à son heure normale qu'après une durée de OVH. A la fin de l'instruction DUREE(D,FR), le temps du fils aura progressé de D, ce qui est satisfaisant, et de plus le fils aura passé une durée de :

$$D + n \times \text{OVH}$$

sur son père, où n est le nombre de fois où le fils a été interrompu (donc le nombre de fois où son contexte a été sauvegardé puis restauré).

Ceci exprime bien que la charge de cette gestion de contextes incombe au père du noeud interrompu.

Cette technique est très simple et suffisante dans beaucoup de cas.

II.3.9. EXTENSION DU DOMAINE D'APPLICATION DE DUREE

DUREE est avant tout destinée à simuler le temps s'écoulant entre deux instructions de l'algorithme d'activité d'un noeud simple.

Plus généralement, une instruction DUREE(D1,F1) dans n'importe quel algorithme a le sens suivant :

a) - si le noeud concerné n'a pas d'algorithme d'activité, alors cette instruction est inopérante, sinon

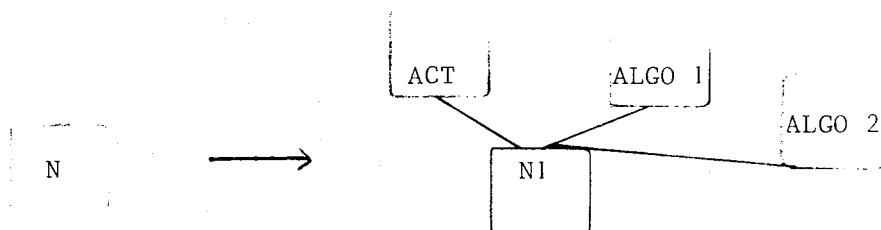
b) - ou bien le noeud est en cours d'exécution (actif ou suspendu) d'une instruction DUREE(D2,F2) : alors la nouvelle durée D1 vient s'ajouter au temps restant à écouler dans le DUREE en cours. Le second paramètre FR1 n'intervient pas, FR2 reste la fonction de reprise courante,

c) - ou bien le noeud n'est pas en cours d'exécution d'un DUREE (il est soit candidat non associé, soit non candidat). Alors le DUREE(D1,F1) vient s'insérer en tête de son programme d'activité, et sera effectivement exécuté à la première activation du noeud.

Ainsi, les algorithmes, lorsqu'ils émettent des DUREE, prolongent d'autant ceux de l'activité du noeud. Tout se passe comme si le noeud interrompait son activité normale pour exécuter ses algorithmes en mode prioritaire.

La manière la plus "architecturale" d'exprimer cela serait la suivante :

On remplace le noeud simple par un noeud simple avec plusieurs fils,
 - l'un ayant comme activité, l'activité du noeud de départ,
 - les autres ayant chacun une activité qui correspond à un algorithme d'archivage ou de décision du noeud initial :



(Les noeuds ALGO i ayant priorité par rapport à ACT).

On verra ultérieurement d'autres manières d'exprimer les algorithmes d'un noeud sous forme de l'activité d'un autre noeud.

II.3.9 BIS. L'INSTRUCTION REPRISE

** REPRISE(X)

X doit être du type référence d'instruction et doit désigner une instruction du sous-programme d'activité dans lequel la REPRISE figure.

Cette instruction indique à quel endroit l'exécution du sous-programme devra reprendre si elle vient à être suspendue lorsque le noeud redeviendra actif.

Par défaut, si aucune instruction REPRISE n'a été exécutée ou bien si la dernière exécutée a spécifié une adresse invalide, le point de reprise sera l'instruction suivant l'instruction suspendue si celle-ci n'était pas une DUREE. Si l'instruction suspendue était une DUREE, son arrêt a lieu comme indiqué plus haut.

II.3.10 NOTE SUR LA PROGRAMMATION DES ALGORITHMES D'ACTIVITE

Il faut trouver un moyen d'exprimer dans le langage hôte choisi (qui sera en général un langage usuel) des notions "système" telles que la suspension et la reprise de l'activité d'un noeud utilisateur.

Rappel : On dit qu'un sous-programme fonctionne en mode coroutine ou est une coroutine si, lorsqu'on l'appelle, son exécution commence à l'instruction suivant celle qui a servi à sortir du sous-programme lors de son exécution précédente.

Problème des sous-programmes d'activité dans SSH.

Les noeuds simples peuvent osciller entre les états actif et inactif, et en conséquence, il faut activer ou suspendre l'exécution des programmes d'activité.

La cause de la suspension est toujours l'exécution d'une instruction SSH,

- soit à l'intérieur de l'activité,
- soit à l'extérieur.

Dans le deuxième cas, il n'y a pas de problème, car puisque SSH est implémenté sur une machine séquentielle, l'exécution de l'activité est déjà interrompue à cet instant (sa dernière exécution s'est terminée par un retour de sous-programme).

Dans le premier cas, la seule solution est d'exécuter immédiatement le retour de sous-programme après l'instruction cause de la suspension.

En conclusion, il suffit que les sous-programmes d'activité soient des coroutines.

En pratique, malheureusement, il existe peu de langages de programmation où les coroutines sont implémentées (les plus courants étant certains langages machine de grosses machines (instruction BALR, sur IBM 370)).

C'est pourquoi SSH impose la solution plus lourde mais moins restrictive suivante :

- un sous-programme d'activité commence par ALLERA SUITE où SUITE est une variable prédéclarée de type référence d'instruction (variable étiquette),
- chaque fois qu'un sous-programme d'activité exécute une instruction SSH susceptible de provoquer la suspension de l'activité, il doit
 - 1) exécuter SUITE ← X où X est l'adresse où l'exécution devra reprendre. Typiquement X désignera l'instruction suivante ;
 - 2) exécuter le retour de sous-programme.

Exemple 1 :

```

      {
      SUITE ← TOTO

      DECISION

      RETURN

TOTO : {

```

Commentaires :

DECISION appelle décision-père qui peut faire MOURIR

Exemple 2 :

```

      {
      SUITE ← TOTO
      DUREE(5)
      {
      DUREE(10)

      RETURN

TOTO : {

```

Commentaires :

L'instruction TOTO sera exécutée après que le noeud ait été actif pendant 5 + 10 unités de temps.

N.B. : Si, dans une activité on exécute plusieurs DUREE avant de faire RETURN, tout se passe comme si on en exécutait un seul :

- dont le premier paramètre est la somme de tous les premiers paramètres,
- dont le second paramètre est celui du dernier.

N.B. : Souvent, le langage hôte choisi, il y a un grand nombre de manière et d' "astuces" pour alléger cette écriture :

En assembleur, la notation est très propre, il suffit par exemple de baptiser une macro du nom de l'instruction SSH.

Exemple :

DUREE D, F qui se charge de générer :
interprétation de l'instruction DUREE
BAL 14, adresse du noyau SSH

En APL, on peut procéder ainsi :

On crée une procédure DUREE

▽ Z ← DUREE X

interprétation de l'instruction DUREE

→ Z ← 0, SUITE ← (~~X~~27) [2] + 1

▽

Et, dans le corps du sous-programme d'activité, il suffit d'écrire :

→ DUREE D, Fonction de reprise.

Pour plus de clarté, dans la suite de l'exposé, nous utiliserons la notation la plus concise, où seul figure le nom de l'instruction SSH.

II.4. RESUME SUR LES NOEUDS SIMPLES

A ← DN(SIMPLE, PERE.DE(N1,...NP), ESPACE.LOCAL(TAILLE, INIT, COPIE, DESTR),
ALGORITHMES (DECIS1, ARCHI.EX, ARCH.INT, DECIS2, ACTIV))

Un noeud simple est actif si et seulement si il est sur un chemin de noeuds simples où chacun a été associé par son père.

Lorsqu'il est actif, son sous-programme d'activité s'exécute en temps simulé et peut exécuter les instructions SSH suivantes ENVOI, INFO, DECISION.

Ses algorithmes d'archivage et de décision sont appelés lors des événements dont le motif peut être :

ILARRIVE, ILSENA, ILINFORME, CESTMOI

Son algorithme décision-père peut exécuter VIVRE, MOURIR.

Son algorithme décision-fils peut exécuter ASSOCIER, DISSOCIER.

Son algorithme d'archivage interne est exécuté chaque fois que le noeud exécute une instruction SSH. Lui-même ne peut exécuter d'instructions SSH mais seulement modifier le contenu de l'espace local du noeud et son extension.

Il en est de même pour l'algorithme d'archivage externe, qui lui est exécuté chaque fois qu'un événement extérieur au noeud l'atteint.

L'instruction DN peut apparaître un peu compliquée et surchargée.

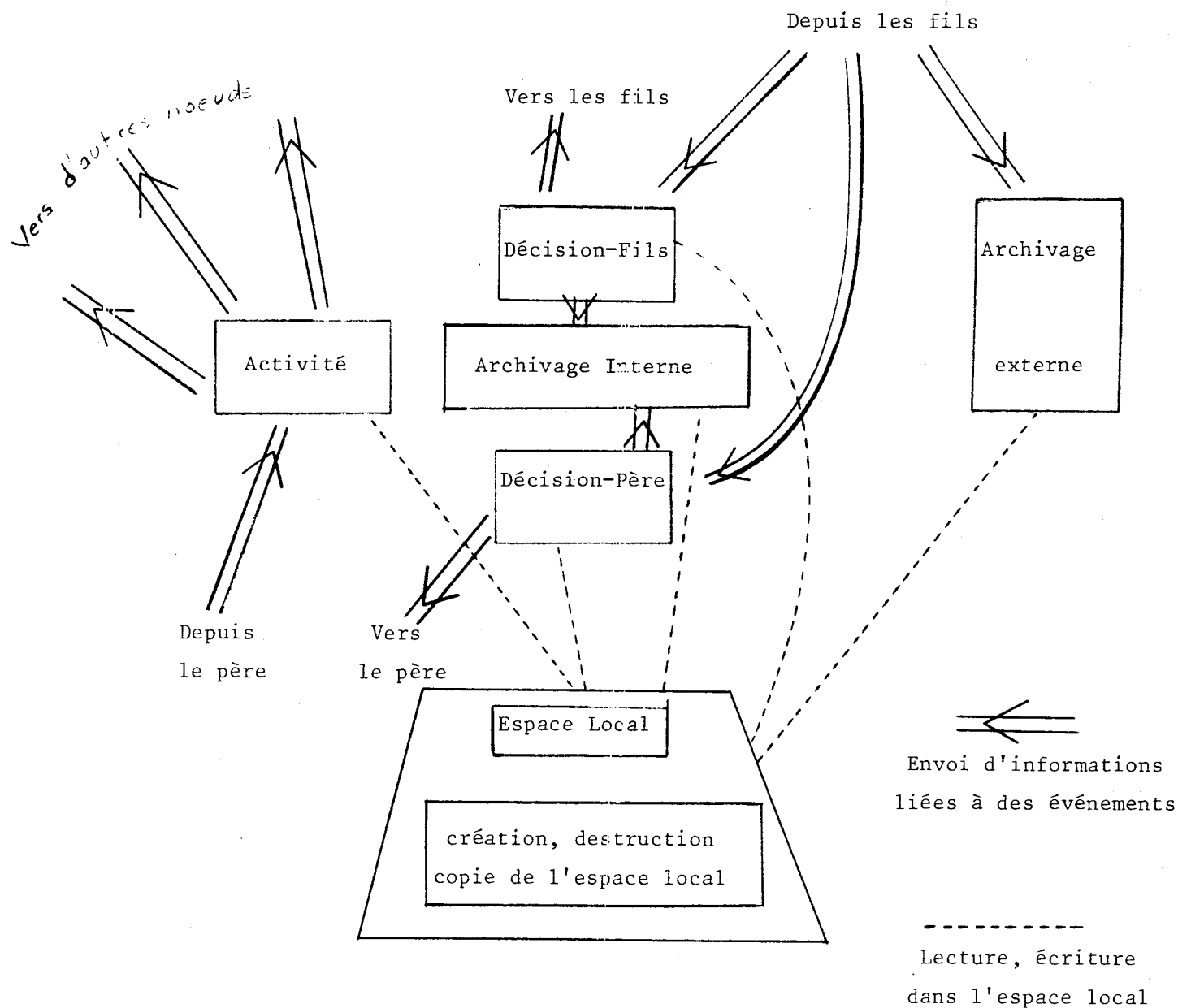
Certains dispositifs peuvent sembler faire double emploi.

Par exemple, pourquoi ne pas confondre en un seul sous-programme décision-père, décision-fils, archives internes, archives externes ?

Répondons d'abord qu'il ne serait pas difficile de faire ainsi.

Il suffit que toutes ces procédures soient inopérantes sauf une seule qui ferait tout le travail.

Mais si un noeud SSH est constitué ainsi de plusieurs "pièces détachées", c'est au contraire dans un effort pour partitionner d'une manière justifiée l'entité que constitue un noeud. C'est donc à notre sens un facteur de clarification plutôt que de complication. Ceci peut être illustré par la représentation graphique suivante :



A partir d'un tel "meccano", l'utilisateur peut concevoir ses propres types de noeuds, sous une forme plus concise et plus significative pour lui.

Les noeuds SSH ne doivent pas être considérés comme un point d'arrivée, mais comme une base de départ, un noyau autour duquel on peut construire d'autres primitives de modélisation. Nous allons illustrer ce point de vue en décrivant ce que l'on pourrait appeler la bibliothèque de noeuds standard simples de SSH.

II.5. LES NOEUDS SIMPLES STANDARD DE SSH

Ce sont des noeuds fréquemment utiles pour modéliser des systèmes informatiques.

II.5.1. LE NOEUD "LAMBDA"

Il peut être créé par un appel du sous-programme PL de nom DNL, avec les paramètres suivants :

```
DNL(PERE.DE(      ) = liste vide,ACT=RIEN,
TAILLE=0,DM=1,SA=1,PRIO=1)
```

Rappel : derrière le =, se trouvent les éventuelles valeurs par défaut. RIEN est le nom prédéclaré d'un algorithme vide.

Caractéristiques du noeud lambda :

Les trois premiers paramètres n'ont rien de particulier.

Ils désignent respectivement les fils, l'algorithme d'activité, la taille de l'espace local.

Le quatrième paramètre DM (ou degré de multi-programmation) indique le nombre maximum de fils que le noeud associe à la fois. Plus précisément, il associe tous ses fils candidats, à concurrence de DM fils associés.

Il ne dissocie jamais un fils, mais attend qu'il s'en aille de lui-même. Dans ce cas, il associe à sa place le plus prioritaire des fils en attente, s'il existe.

Le cinquième paramètre SA (ou seuil d'activité) indique le nombre minimum de fils candidats que le noeud attend avant de se porter lui-même candidat à son père (par un VIVRE).

Inversement, dès que ce nombre retombe sous le seuil SA, le noeud cesse d'être candidat (par un MOURIR)

Le sixième paramètre indique la priorité du noeud. Cette priorité n'est qu'une valeur relative à la priorité des autres fils éventuels du père du noeud.

Tous les algorithmes nécessaires à la construction d'un noeud lambda sont détaillés en annexe.

II.5.2. LE NOEUD PRIORITAIRE

Sa seule différence avec le précédent est la suivante :

Si le degré de multiprogrammation est atteint et qu'arrive un fils candidat plus prioritaire que le moins prioritaire des fils actuellement associés, alors le premier prend la place du second qui est placé en attente.

Il est déclaré exactement de la même manière en utilisant DNP à la place de DNL.

Voir ses algorithmes en annexe.

Convention commune à l'écriture des algorithmes d'activité des DNL et DNP

Il existe une valeur désignée par l'identificateur ATTENDRE qui peut être passée en paramètre de l'instruction DECISION :

DECISION(ATTENDRE) est interprétée ainsi par l'algorithme de décision-père :

décrémenter le compteur de fils candidats ;
s'il devient inférieur au seuil d'activité, alors faire MOURIR.

L'arrivée d'un nouveau fils candidat pourra ensuite provoquer un VIVRE et réveiller ainsi le noeud.

Ceci correspond à la notion de "sémaphore privé" utilisée dans BRUNET [11].

II.5.3. LE NOEUD STANDARD BARILLET

Il diffère des précédents par sa politique de décision-fils ; son seuil d'activité et son degré de multiprogrammation sont égaux à 1.

Il ne gère pas ses fils de manière prioritaire ou lambda, mais il les associe chacun à tour de rôle pendant un certain quantum de temps, jusqu'à ce qu'ils cessent d'être candidats. Chaque fils doit posséder un champ désigné par QUANTUM dans son espace local, qui indique la durée de la tranche de temps qui lui sera allouée. En d'autres termes, il fait du "temps partagé" sur ses fils.

(algorithmes en annexe).

II.6. EXEMPLES D'UTILISATION DES NOEUDS SIMPLES STANDARDS

Il s'agit d'un modèle d'architecture inspirée de celle de l'ordinateur IBM 370-115, qui comporte en particulier un processeur d'entrées-sorties travaillant en mode barillet.

Un tel système est constitué de :

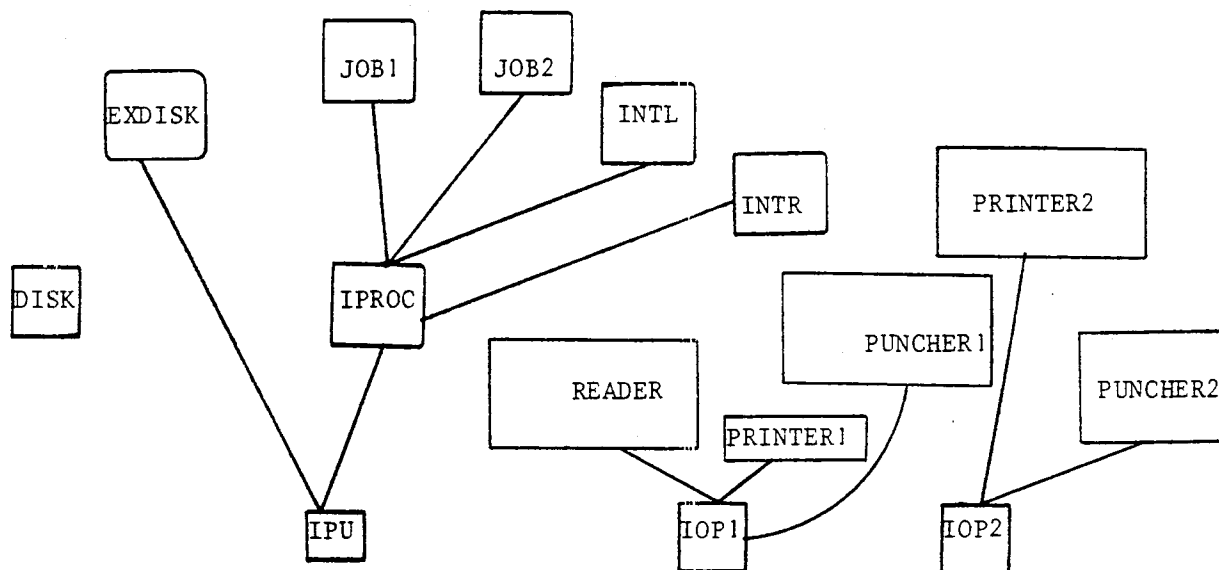
- un IPU : "Instructions Processing Unit" qui assure deux tâches :
 - . l'émulation d'un contrôleur des unités de disque,
 - . l'exécution des instructions de calcul, celles-ci se répartissant entre traitement des interruptions prioritaires et traitement des programmes utilisateurs,
- plusieurs IOP : "Input-Output Processors" qui assurent l'exécution des échanges avec les unités d'entrée-sortie lentes.

Voici la description d'une telle structure en SSH.

```

IPU ← DNP(PERE.DE(EXDISK ← DNL, IPROC ← DNP(PERE.DE(INTR ← DNL, INTL ← DNL,
      JOB1 ← DNL, JOB2 ← DNL))))
IOP1 ← DNB(PERE.DE(READER ← DNL, PRINTER1 ← DNL, PUNCHER1 ← DNL))
IOP2 ← DNB(PERE.DE(PRINTER2 ← DNL, PUNCHER2 ← DNL))
DISK ← DNL
  
```

Représentation graphique



Exemple de description de l'activité sur un tel système :

L'activité d'un noeud JOB_i peut créer des noeuds ESL, ESR, respectivement "entrée-sortie lente" et "entrée-sortie rapide" qui vont évoluer sur les feuilles de la structure qui vient d'être déclarée.

Voici les fonctions de déclaration de ces noeuds :

FONCTION ESL(L) :

LONG DE (ESL ← DNL(,ACTESL,,1))←L ;

SOUS-PROGRAMME ACTESL :

DUREE(LONG DE MOI) ; ENVOI(INTL,MOI) ; DUREE(DO) ; DETRUIRE(MOI) ,

FIN ACTESL

FONCTION ESR(N,L)

LONG DE ESR(← DNL(,ACTESR,,4))←L;ORIGINE DE ESR←MOI;

NB DE ESR←N;

FIN ESR

SOUS-PROGRAMME ACTESR :

DUREE(d1);

TANT QUE(NB DE MOI←NB DE MOI-1)≥0 FAIRE DEBUT

ENVOI(DISK,MOI);DUREE(LONG DE MOI);

ENVOI(EXDISK,MOI);DUREE(D2);FIN

ENVOI(INTR,MOI);DUREE(D3);

ENVOI(ORIGINE DE MOI,MOI);

DETRUIRE(MOI)

FIN ACTESR

Pour lancer une entrée-sortie lente, l'activité de JOBi exécutera par exemple :

ENVOI (PRINTER1,ESL(10))

Pour lancer une entrée-sortie rapide :

ENVOI(EXDISK,ESR(5,10))

Pour attendre la fin d'une ESR, JOBi utilise les conventions des noeuds simples, c'est-à-dire :

DECISION(ATTENDRE)

Commentaires :

ESL(n) va occuper pendant n unités de temps l'unité périphérique à laquelle elle a été envoyée, cette dernière étant elle-même entraînée dans le barillet d'un IOP ; puis elle occupe INTL pendant DO (traitement de l'interruption fin E/S).

ESR(n,l) va occuper les noeuds suivants :

EXDISK pendant d1

Puis n fois

DISK pendant l

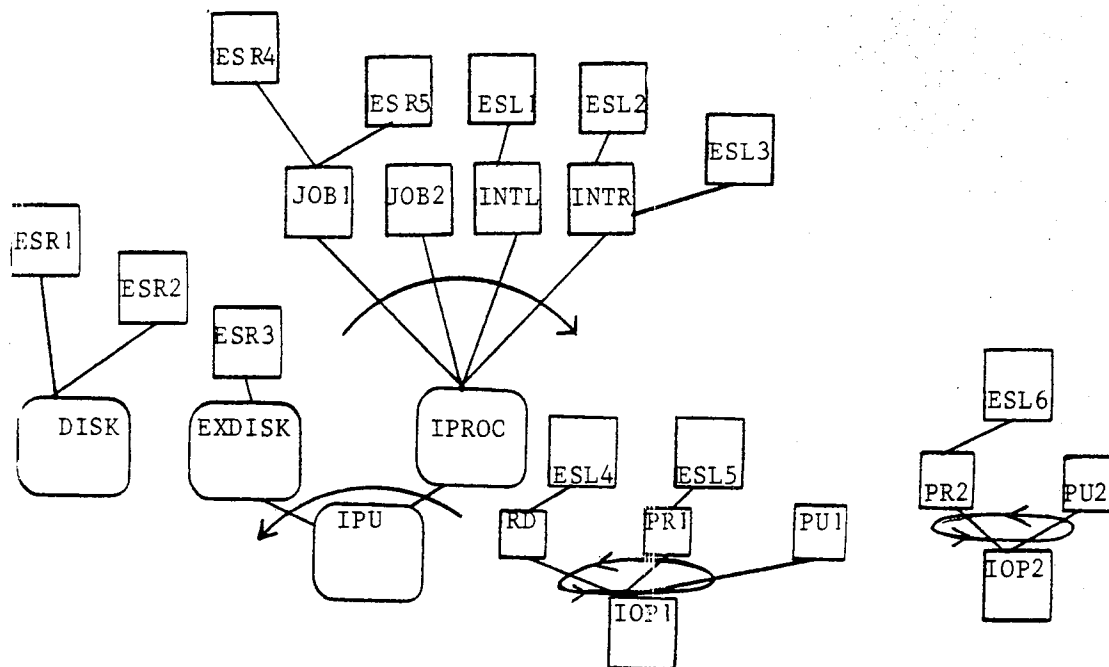
EXDISK pendant d2

} simulation de n entrées/sorties
de blocs de longueur l

Enfin,

INTR pendant D3 traitement de l'interruption fin E/S puis retour sur le JOB qui l'a initialisé.

Exemple de configuration à un instant donné



N.B. : Voir en annexe A8:conventions de représentation graphique des noeuds standard.

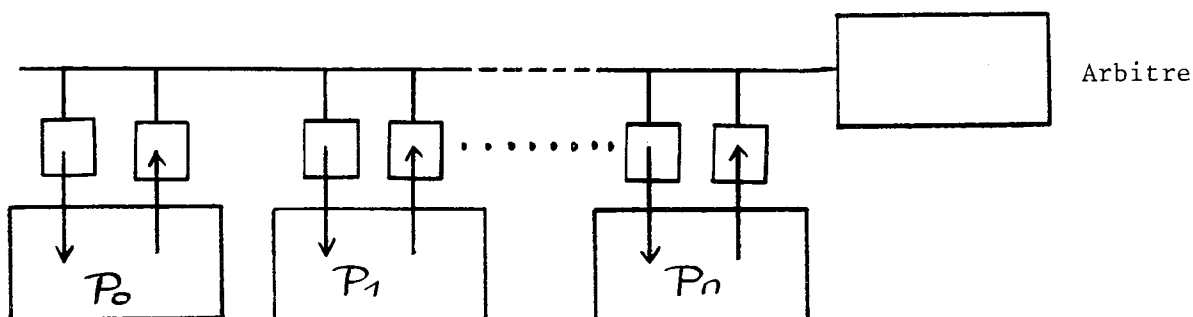
II.7. AUTRE EXEMPLE A BASE DE NOEUDS STANDARD

MODELISATION D'UN BUS DE COMMUNICATION ENTRE PROCESSEURS EN SSH

Cet exemple est intéressant dans la mesure où il montre comment un système apparemment non hiérarchisé, (contrairement au cas précédent de l'IBM 115) peut également être modélisé fidèlement et très simplement avec les noeuds simples standard SSH.

A) Description du système ELMASRY, ATTASI [12]

Des microprocesseurs sont connectés à un bus via deux buffers chacun. Un buffer d'entrée et un buffer de sortie.



Si P_i veut envoyer un message à P_j :

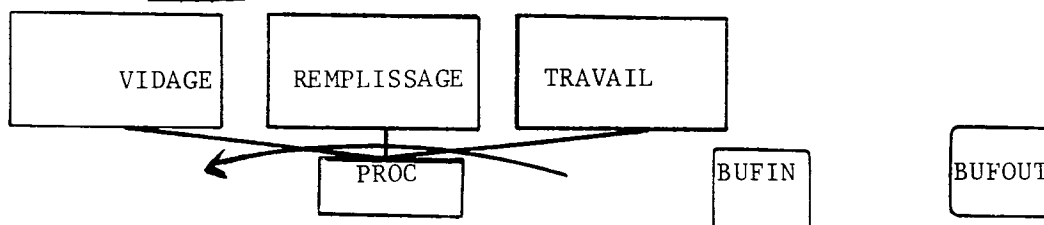
- il attend que son buffer de sortie soit vide,
- il le remplit,
- il indique l'adresse du destinataire au bus,
- il indique que le buffer est plein et prêt au transfert.

Lorsque le buffer d'entrée du destinataire sera libre, le bus assurera le transfert continu de l'un à l'autre.

Quand le buffer d'entrée du destinataire est plein, une interruption est envoyée à son processeur, qui doit alors le vider et indiquer quand il devient à nouveau libre au bus.

B) Voici comment ceci peut être modélisé en SSH avec des noeuds standard simples :

A chaque processeur, on va associer les noeuds suivants :



Soit N le nombre de processeurs,

```
DO I = 0, N-1 BEGIN
```

```
PROC[I] ← DNP( PERE.DE(VIDAGE [I] ← DNL, REPLISSAGE [I] ← DNL, TRAVAIL [I] ← DNL);
```

```
BUFIN [I] ← DNL ;
```

```
BUFOUT [I] ← DNL ; END
```

- LOGBUS va représenter la logique du bus.

BUS va représenter l'organe de transfert proprement dit.

LOGBUS ← DNL

BUS ← DNL



C'est l'activité de TRAVAIL qui va pouvoir initialiser l'envoi de messages.

Pour simplifier, on suppose qu'il y a un nombre constant P de messages circulant dans le système, et qu'un message reçu par un processeur est renvoyé après un certain temps vers un autre processeur.

A chaque message on associe un noeud simple dont l'espace de mémorisation contient les champs suivants :

ORIG est le numéro du processeur qui émet le message

DEST est le numéro de son destinataire :

Déclaration des MSG :

DO I = 0, P-1 MSG [I] ← DNL(,ACTMSG,2)

C'est l'activité des messages qui va définir leur comportement :

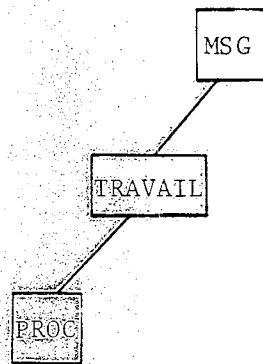
SOUS-PROGRAMME ACTMSG

1. DEBUT:DUREE(ELABORATION)
2. ENVOI(BUFOUT[ORIG DE MOI],MOI)
3. ENVOI(REPLISSAGE[ORIG DE MOI], BUFOUT[ORIG DE MOI])
4. DUREE(D.REMPLIR)
5. ENVOI(BUFIN[DEST DE MOI], BUFOUT[ORIG DE MOI])
6. DUREE(D.LOGIQUE.BUS)
7. ENVOI(BUS,BUFIN[DEST DE MOI])
8. DUREE(D.BUS)
9. ENVOI(BUFIN[DEST DE MOI],MOI)
10. ENVOI(VIDAGE[DEST DE MOI], BUFIN[DEST DE MOI])
11. DUREE(D.VIDAGE)
12. ENVOI(TRAVAIL[DEST DE MOI], MOI)
13. ENVOI(LOGBUS,BUFIN[DEST DE MOI]) ; GOTO DEBUT ; FIN

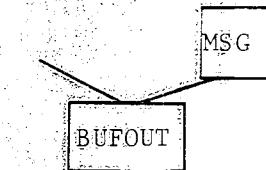
C) Commentaires :

Le fonctionnement est simple à comprendre en dessinant les modifications de la hiérarchie successivement causées par ce programme d'activité d'un message : instruction par instruction.

1. L'élaboration du message occupe TRAVAIL pendant D.ELABORATION

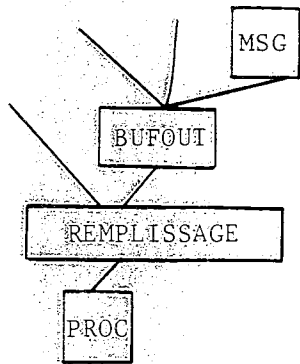


2.



Le message commence son occupation du buffer de sortie.

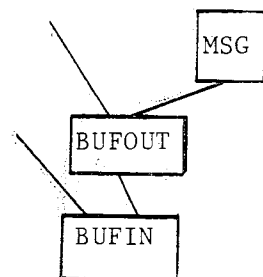
3.



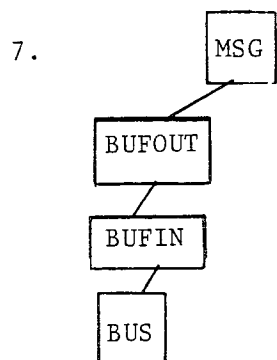
Remplissage du buffer de sortie

4. La situation précédente dure D.REMPLIR

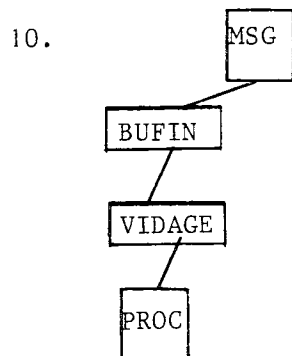
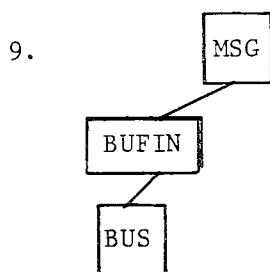
5.



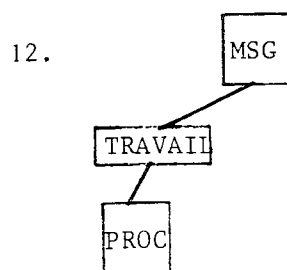
6. CECI dure D.LOGIQUE.BUS



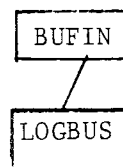
8. Durée de l'échange



11. Durée D.VIDAGE

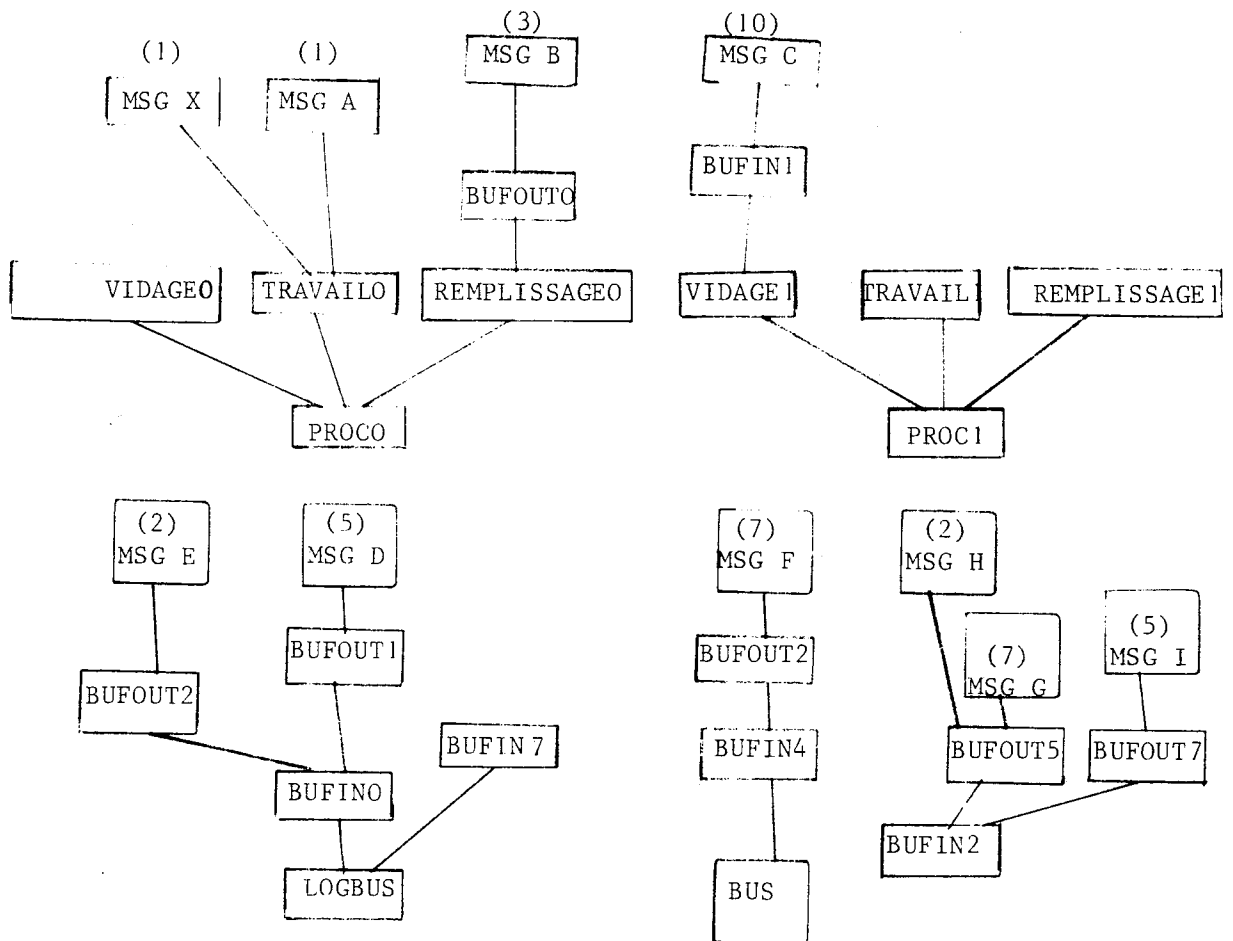


et



Puis retour à 1.

Vue d'ensemble du système à un instant donné :



Le numéro au-dessus des MSG indique la dernière instruction que leur activité a exécuté.

D) Variante de l'exemple précédent :

Chaque processeur n'a pas deux buffers, mais un seul, servant soit en entrée soit en sortie.

Un tel système comporte des risques évidents d'interblocage. Une condition suffisante pour les éviter est que les deux buffers nécessaires à un échange soient réservés dans un ordre fixé intrinsèque: non pas le buffer origine, puis le buffer destination comme précédemment, mais d'abord celui de plus petit indice par exemple.

Il suffit donc de faire la modification suivante dans ACTMSG :

BUFIN et BUFOUT sont remplacés par un seul BUF.

Après l'instruction 1, on fait le test suivant :

IF DEST DE MOI = MIN(DEST DE MOI, ORIG DE MOI)

THEN BEGIN

ENVOI(BUF [DEST DE MOI], MOI)

DUREE(D.LOGIQUE.BUS)

ENVOI(BUF [ORIG DE MOI], BUF [DEST DE MOI])

ENVOI(REPLISSAGE [ORIG DE MOI], BUF [ORIG DE MOI])

DUREE(D.REPLISSAGE)

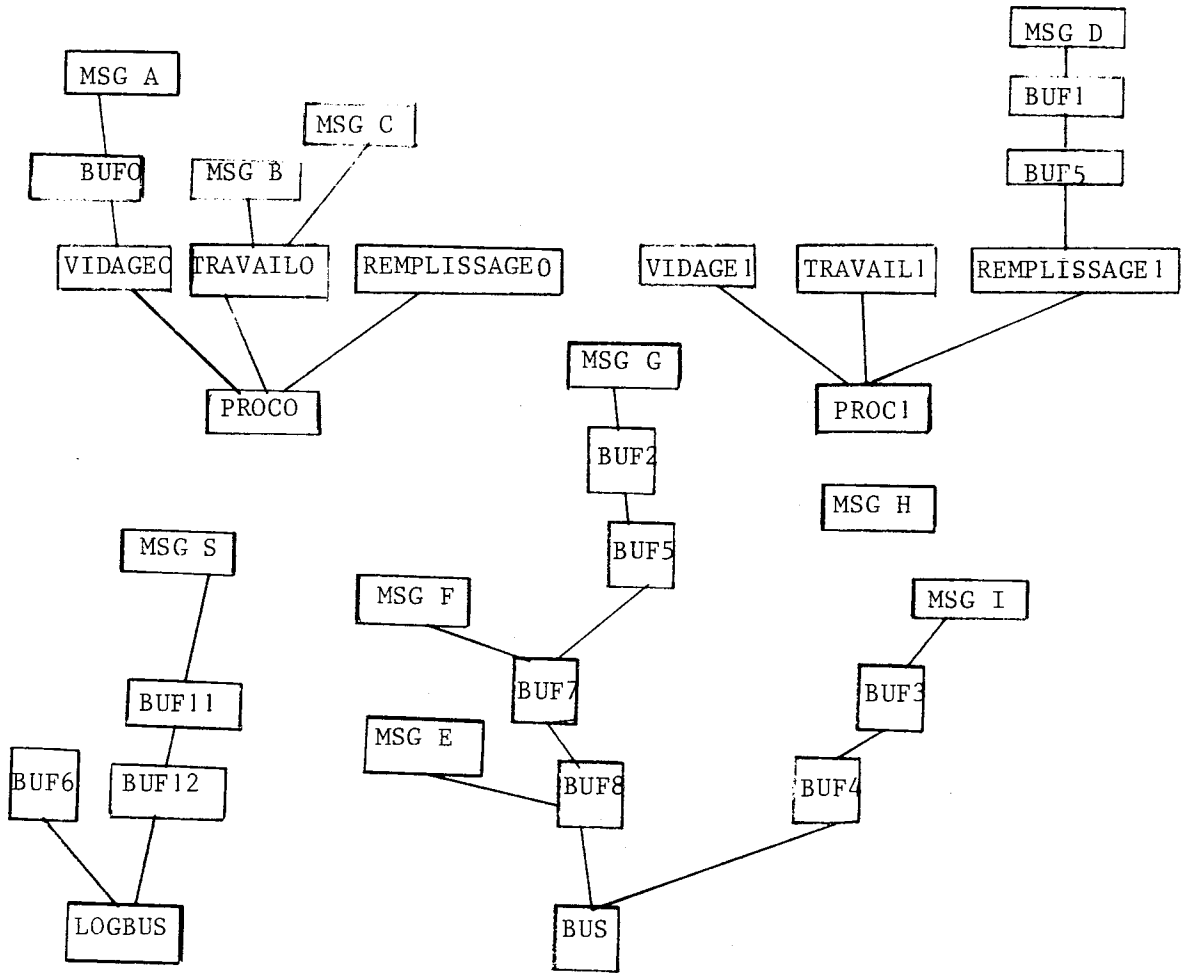
ENVOI(BUS, BUF [ORIG DE MOI])

DUREE(D.BUS)

GOTO instruction 10 du cas précédent

END.

On obtient finalement une hiérarchisation assez étonnante mais parfaitement fidèle au fonctionnement d'un tel système :



III TROISIEME PARTIE

LES NOEUDS MULTIPLES

III.1. MOTIVATIONS

Les noeuds simples, on l'a vu, permettent de couvrir un large ensemble des besoins en simulation de systèmes informatiques. Ils présentent néanmoins des insuffisances.

A) Insuffisance pour exprimer certaines synchronisations

La mutuelle exclusion entre deux noeuds peut être obtenue en les rendant fils d'un noeud simple standard de degré de multiprogrammation égal à 1. Mais il est beaucoup plus difficile ou même impossible d'exprimer avec des noeuds simples des cas plus généraux :

- un noeud peut être dans plusieurs zones de mutuelle exclusion,
- l'exclusion autorise non pas un seul, mais $n > 1$ utilisateurs au maximum.

Remarquons que ces questions sont très bien résolues par les sémaphores classiques, mais avec les inconvénients déjà cités en I.2.1. et que la TSH veut supprimer (pas de sémantique, ni de structure associée, limitation à une seule couche d'un système).

Il y a certes la solution de simuler les P et V sur sémaphores dans un modèle SSH : les fils peuvent envoyer à leur père des informations dont la sémantique est celle de P et V. Le père implémente classiquement les sémaphores dans son espace local (files d'attente, compteur) et émet éventuellement des ASSOCIER, DISSOCIER pour débloquer ou bloquer les fils.

Mais il est clair que ce n'est pas satisfaisant pour nous : nous voulons que le problème soit résolu plutôt au niveau de la structure du modèle qu'au niveau de la programmation de certains des algorithmes de cette structure (il est plus facile de dessiner une structure que d'écrire un programme).

B) Insuffisance des noeuds simples pour observer l'état du système

Ils ne sont sensibles qu'aux inductions directes :

Un noeud simple sait quand ses fils candidats arrivent ou s'en vont mais il ne sait pas lui-même s'il est associé ou non par son père, et de plus, il ignore tout de l'état de son père.

Lorsqu'un noeud simple s'arrête (par un MOURIR ou un DISSOCIER par exemple), seul l'algorithme même qui a exécuté cette instruction le sait.

Par exemple, les noeuds simples ne permettent pas de simuler simplement un système où l'on peut détecter un arrêt de fonctionnement d'un noeud, (panne, interruption par un plus prioritaire), et rendre le noeud arrêté candidat à un autre père.

Ou encore, on ne peut pas exprimer qu'un noeud actif se met en attente d'une ressource autre que son père, et que ce dernier associe alors un autre fils à sa place.

Plus grave, si un noeud simple devient inactif parce que son père lui-même est devenu inactif (par induction inverse), alors personne, c'est-à-dire aucun noeud - ne le sait. Or, nous pensons qu'un bon outil de modélisation de systèmes informatiques ou autres, doit visualiser le maximum d'informations sur l'état du système, même des informations non utilisées par le système lui-même.

C) Insuffisance des noeuds simples vis-à-vis des problèmes de désignation

Parmi les instructions exécutables par les algorithmes des noeuds simples, ASSOCIER, DISSOCIER et ENVOI (et ses formes dégénérées) doivent comporter des paramètres désignant d'autres noeuds simples.

A un instant donné, un noeud simple connaît un certain nombre de noms de noeuds, que l'on peut classiquement appeler "l'espace visible du noeud".

Ces noms peuvent avoir différentes origines : par exemple, dans le cas des paramètres d'un ASSOCIER ou DISSOCIER, ce sont des noms apparus comme valeur de LUI dans un événement avec MOTIF = ILARRIVE.

Par contre, dans le cas d'un ENVOI(P,F), P et F peuvent avoir des origines quelconques :

- connus statiquement depuis la construction du noeud,
- communiqués dynamiquement au noeud par un INFO par exemple.

Ainsi, n'importe quel noeud N peut envoyer n'importe quel autre F sur n'importe quel P, pourvu qu'il connaisse les noms P et F.

De même, il peut lui-même les changer de père.

De plus, d'après l'alinéa B), après avoir exécuté ENVOI(P,F), un noeud ne dispose d'aucun moyen de connaître ce qu'il advient de F.

En résumé, un système qui ne comporte que des noeuds simples ne dispose d'aucun moyen propre et général pour exprimer des relations entre des noeuds qui ne sont pas sur un même chemin de la relation père-fils.

Dans SSH, une solution aurait consisté à "gonfler" les noeuds simples, pour leur ajouter ces possibilités manquantes. Mais, comme les noeuds simples à eux seuls couvrent de manière satisfaisante un grand nombre de cas courants, nous avons préféré ne pas pénaliser leur programmation, en compliquant la syntaxe et la sémantique des noeuds-simples.

C'est pourquoi SSH comporte un second type de noeud, très différent du noeud simple, le noeud multiple aussi appelé noeud composé.

Ces noeuds multiples auront les propriétés suivantes :

- ils pourront en permanence observer l'état (donc, les variations d'état) de leur fils (actif/inactif),
- ils connaîtront de manière explicite, de par leur construction, un ensemble de noeuds qui seront appelés composants des noeuds multiples,
- ils pourront dynamiquement créer et détruire des relations père-fils entre leurs composants et leurs fils.

Ces trois propriétés permettent aux noeuds multiples de gérer proprement les relations "horizontales" existant entre les noeuds d'un système hiérarchisé, par opposition aux relations "verticales" utilisateur-ressource.

III.2. DEFINITION DES NOEUDS MULTIPLES

A la différence des noeuds simples :

- ils n'ont pas d'algorithmes d'activité. Néanmoins, leurs autres algorithmes peuvent engendrer indirectement des activités par l'instruction CONFIE.A (voir en IV.2.) ou par d'autres mécanismes (voir en IV.6.),

- ils n'ont pas de père dans la hiérarchie (donc pas d'algorithmes décision-père), mais, ils ont des composants, qui sont des noeuds de type quelconque, éventuellement multiples.

Voici la définition plus précise :

Syntaxe de la déclaration d'un noeud multiple :

```
** A ← DN(MULTIPLE,PERE.DE(F1,...,FN),COMPOSE.DE(C1,...,CP),
      ESPACE.LOCAL(<taille>,<fonctions de service>),
      ALGORITHMES(<décision>,<archives externes>,<archives internes>))
```

C1, CP doivent référencer des noeuds de type quelconque.

La syntaxe et la sémantique de PERE.DE, ESPACE.LOCAL, <archives internes>, <archives externes> est le même que celle des noeuds simples.

III.2.1. ** COMPOSE.DE(C1,...,CN); SYNTAXE ET SEMANTIQUE

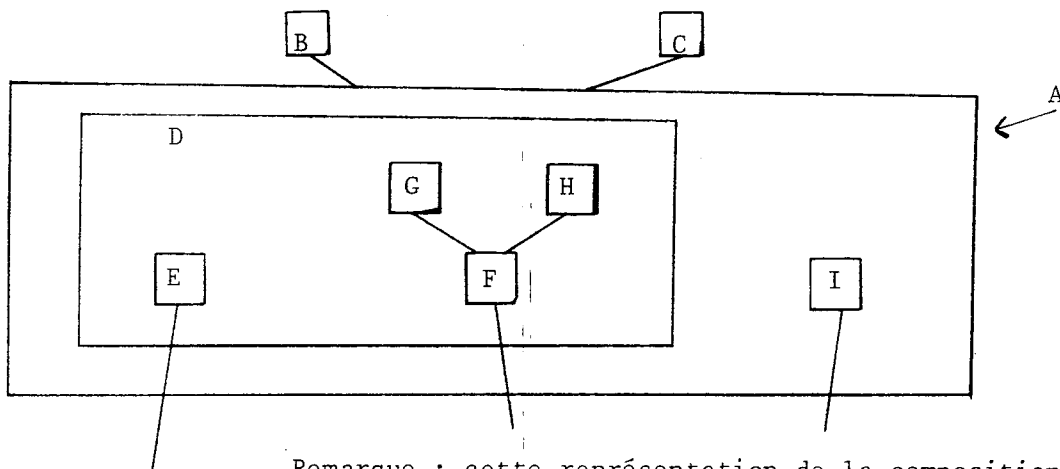
Cette instruction établit une relation binaire dite relation de composition entre le noeud déclaré et chacun des noeuds paramètres. Cette relation est à rapprocher de la relation utilisateur-ressource de la TSH dans le cas où l'utilisateur dispose de plusieurs ressources.

Cette relation doit conférer à l'ensemble des noeuds multiples une structure de hiérarchie (réseau orienté sans circuit).

De plus, l'union de cette relation avec l'inverse de la relation père-fils doit également conférer à l'ensemble de tous les noeuds du modèle une structure de hiérarchie.

Exemple et représentation graphique :

```
A ← DN(MULTIPLE,PERE.DE(B,C),COMPOSE.DE(D←DN(MULTIPLE,COMPOSE.DE
      (E,F←DN(SIMPLE,PERE.DE(G,H)......);I)...
```



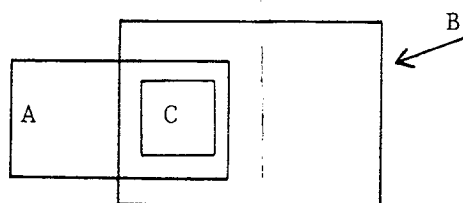
Remarque : cette représentation de la composition par imbrication de rectangles laisse supposer que la structure des noeuds composés doit être une structure arborescente.

Il n'en est rien, et un noeud peut très bien être composant commun à deux noeuds multiples.

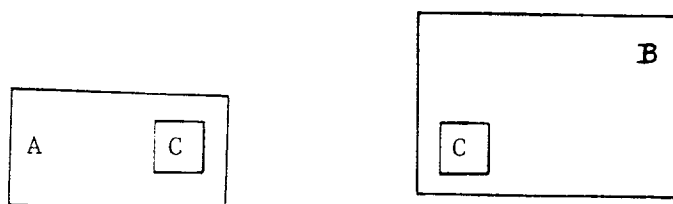
Exemple :

$A \leftarrow \text{DN}(\text{MULTIPLE}, \text{COMPOSE.DE}(C, \dots), \dots)$

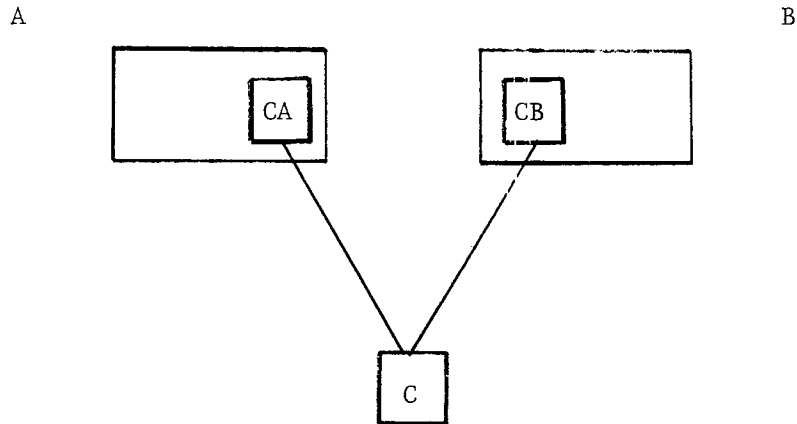
$B \leftarrow \text{DN}(\text{MULTIPLE}, \text{COMPOSE.DE}(C, \dots), \dots)$



Cette représentation n'est pas très élégante. On préférera dessiner deux fois le noeud C :



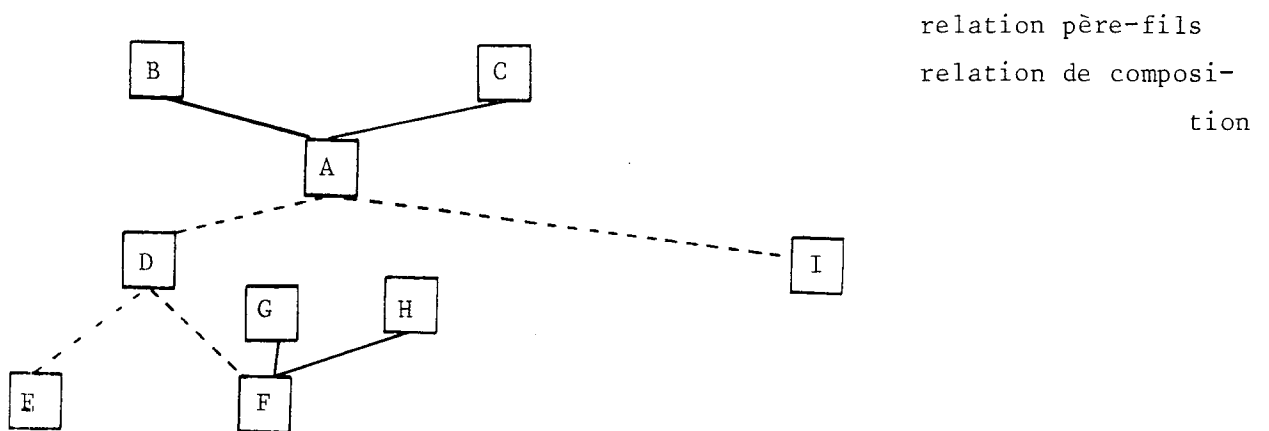
De toutes façons, il est simple de toujours se ramener à un modèle où les noeuds multiples sont structurés en arborescences par la relation de composition :



Cette solution est d'ailleurs très satisfaisante : elle permet de distinguer entre C d'une part et ce que l'on peut appeler les "mécanismes d'accès" à C, CA et CB d'autre part.

N.B. : La TSH a choisi une autre représentation, plus simple et homogène à celle des relations entre noeuds simples.

Le modèle précédent y serait représenté par :



Nous verrons que la représentation en rectangles imbriqués est plus adaptée au cas particulier de SSH.

III.2.2. L'UNIQUE ALGORITHME DE DECISION DES NOEUDS MULTIPLES

Comme ceux des noeuds simples, il est appelé d'abord par l'arrivée d'un événement concernant un fils, avec MOTIF = ILARRIVE ou ILSENA.

Il peut exécuter les nouvelles instructions SSH suivantes :

III.2.2.1. L'INSTRUCTION AFFECTER

* * AFFECTER(C,F)

où C doit désigner un composant du noeud multiple, et F un fils candidat au noeud multiple.

Cette instruction génère l'événement suivant :

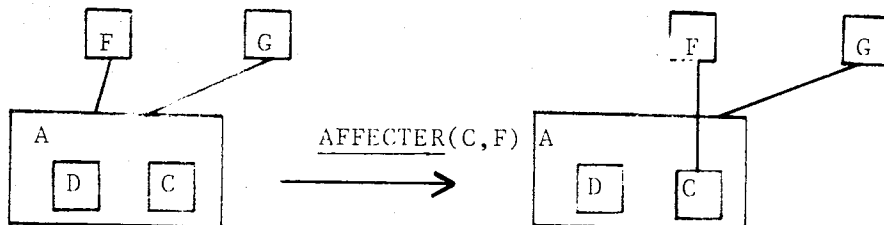
MOI = C ; LUI = F ; MOTIF = ILARRIVE

Si le récepteur C de cet événement est un noeud simple, il peut éventuellement ASSOCIER(LUI) - c'est-à-dire F - s'il est lui aussi multiple, il pourra exécuter AFFECTER(C',LUI) où C est un composant de C, et ainsi de suite.

On voit que le rôle des AFFECTER est de transmettre un fils simple candidat vers un composant terminal lui aussi simple et d'établir une relation père-fils entre les deux.

Représentation graphique :

La représentation en rectangles imbriqués traduit clairement ce phénomène :



III.2.2.2. L'instruction DESAFFECTER

** DESAFFECTER(F)

Cette instruction doit s'appliquer à un fils F qui a été précédemment affecté à un composant.

Elle consiste à enlever le fils à ce composant.

Elle provoque l'événement suivant :

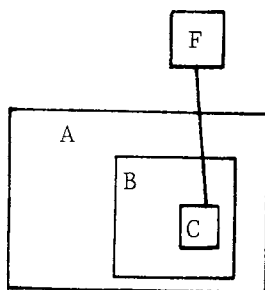
MOI = composant auquel le fils était affecté,

LUI = F ; MOTIF = ILSENVA

Note importante

De même que, dans le cas d'un noeud simple, il était inutile de DISSOCIER un fils associé pour MOTIF = ILSENVA, de même, il est inutile de DESAFFECTER un fils si il s'en va, car SSH s'en charge.

Exemple :



Supposons que F, fils statique de A, ait fait VIVRE et que A ait exécuté AFFECTER(B,F) puis B AFFECTER(C,F)

Si maintenant, F exécute MOURIR, SSH va générer dans l'ordre suivant, 3 événements :

MOI = C; LUI=F; MOTIF=ILSENVA

MOI = B; " " "

MOI = A; " " "

Ainsi, lorsqu'un noeud multiple est averti du départ d'un fils, son composant auquel ce fils était affecté a déjà été lui-même averti.

III.2.2.3. L'instruction MUTER

** MUTER(C,F)

Cette instruction est sémantiquement équivalente à DESAFFECTER(F) ; AFFECTER(C,F)

Ces trois instructions ont la propriété de modifier dynamiquement la structure du modèle, puisqu'elles créent et défont des relations père-fils.

N.B. : Les 3 instructions introduites ici ont la particularité de donner une valeur à la variable LAUTRE accessible par l'algorithme d'archivage interne :

MOI désigne le noeud multiple exécutant l'instruction,

LUI désigne le noeud simple fils du noeud multiple,

LAUTRE désigne le composant concerné.

III.3. LES POSSIBILITES DES NOEUDS MULTIPLES POUR OBSERVER L'ETAT DE LEUR FILS

Dans le cas le plus général, l'algorithme de décision fait face au problème suivant :

Il a plusieurs fils candidats, et plusieurs composants auxquels il peut transmettre la candidature des fils.

A quel composant doit-il affecter quel fils ?

C'est une question que ne se pose pas le noeud simple, qui a simplement à choisir quels fils associer à l'unique ressource qu'il représente.

Pour bien choisir, le noeud multiple doit avoir le maximum d'informations :

- 1) - sur les demandes que représentent ses fils candidats,
- 2) - sur les offres que représentent ses composants.

Une fois son choix fait, c'est-à-dire une fois l'instruction AFFECTER(C,F) exécutée, il convient également que le noeud multiple soit informé :

- 3) - sur la manière dont C va satisfaire la demande de F.

Reprenons ces 3 points :

Pour ce qui concerne 1), le noeud multiple (comme le noeud simple d'ailleurs) a accès au nom du fils, et à toute information que ce nom lui permet d'atteindre : priorité du fils par exemple.

Pour 2), la connaissance des capacités d'offre d'un composant est un problème plus complexe. Des solutions partielles sont proposées dans ANCEAU [1], ROHMER [13].

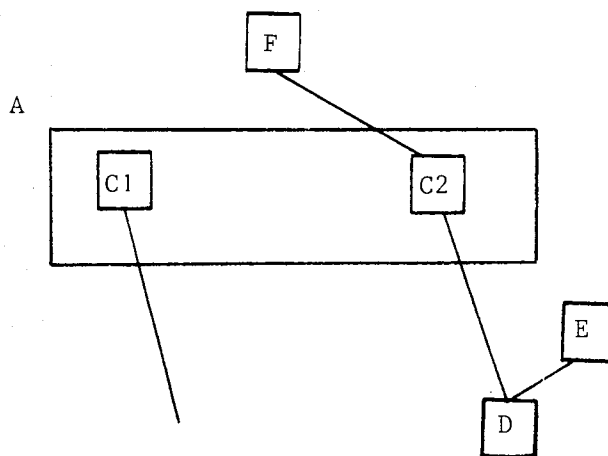
Pour résoudre 3), SSH offre la facilité suivante :

Les fonctions de décision et d'archivage d'un noeud multiple sont appelées non seulement lorsqu'un fils arrive ou s'en va, mais aussi chaque fois qu'il atteint ou quitte l'état actif, (c'est-à-dire chaque fois que le chemin de noeuds simples associés par leur père qui le relie à un noeud terminal se crée ou se rompt).

Ces événements seront indiqués par deux nouvelles valeurs possibles de MOTIF.

MOTIF = ILTOURNE et MOTIF = ILSTOPPE

Exemple :



A a exécuté AFFECTER(C2,F)

Puis, D a exécuté ASSOCIER(C2)

Alors, F est devenu actif.

E fait ensuite VIVRE ; D l'associe à la place de C2 et F devient inactif.

A en est averti : MOTIF = ILSTOPPE

Il décide alors d'essayer de le rendre actif sur C1 par MUTER(C1,F).

Un noeud multiple n'est donc pas aveugle et passif vis-à-vis de ses ressources comme l'est un noeud simple.

Alors que le noeud simple n'est sensible qu'à l'induction directe, le noeud multiple l'est aussi à l'induction inverse.

Revenons sur l'instruction MUTER(C,F).

Si elle est sémantiquement équivalente à :

DESAFFECTER(F) ; AFFECTER(C,F)

elle présente l'avantage suivant :

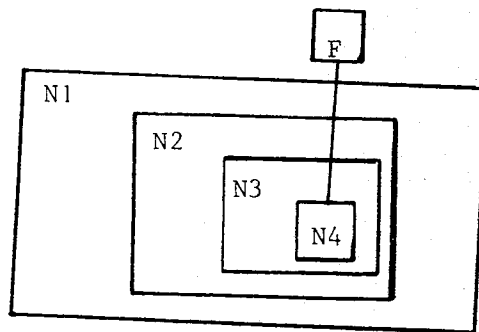
Si F était actif avant MUTER(C,F) et s'il reste actif après, sur son nouveau père C, alors aucune induction inverse n'aura été induite sur F et ses descendants.

Alors que dans les mêmes conditions, DESAFFECTER(F) aurait inutilement induit un événement avec MOTIF = ILSTOPPE sur tous les noeuds multiples ayant pour fils un descendant de F, et AFFECTER(C,F) aurait ensuite au même instant de temps simulé, induit un événement avec MOTIF = ILTOURNE sur ces mêmes noeuds multiples.

III.4. PROPAGATION DE L'INDUCTION INVERSE DANS UNE HIERARCHIE DE NOEUDS MULTIPLES IMBRIQUES

III.4.1. SOLUTION ADOPTEE

Un composant pouvant lui-même être un noeud composé, un fils peut successivement être affecté à plusieurs noeuds composés, et finalement, à un noeud simple. Il a en quelque sorte, plusieurs pères :



On réservera dans ce cas l'appellation de père au dernier noeud auquel il a été affecté. On appellera tous ces pères les "tuteurs" du fils.

La question se pose alors de savoir lequel ou lesquels doivent être avertis des primitives dont le fils peut être l'objet ou des modifications de son activité. Plusieurs solutions envisageables à ce problème sont détaillées dans le paragraphe suivant.

Voici la solution retenue :

- le noeud multiple averti en premier est celui des tuteurs possédant le père du fils comme composant,

- les algorithmes de décision d'un noeud multiple peuvent émettre les primitives suivantes chaque fois qu'elles sont appelées pour une raison quelconque :

- . SUR-TRAITER
- . SOUS-TRAITER

III.4.1.1. L'instruction SOUS-TRAITER

* * SOUS-TRAITER signifie que l'algorithme de décision :

- ne veut pas réagir lui-même à l'événement pour lequel il est invoqué,

- mais qu'il transmet cet événement au composant auquel il avait précédemment affecté le fils, objet de l'événement.

III.4.1.2. L'instruction SUR-TRAITER

* * SUR-TRAITER dans les mêmes conditions, transmet l'événement au noeud composé qui lui avait précédemment affecté le fils concerné.

Ainsi, le choix du tuteur qui finalement réagira éventuellement à l'événement peut être fait très simplement par le concepteur du modèle.

De plus, ces deux instructions retournent la valeur VRAI, sauf dans le cas où elles sont émises par le plus interne (resp. le plus externe) des tuteurs, auquel cas, n'ayant pas de sens, elles retournent la valeur FAUX.

Ces instructions permettent bien des constructions modulaires puisqu'elles n'obligent pas un noeud multiple à connaître sa position dans la liste des tuteurs.

SOUS-TRAITER et SUR-TRAITER génèrent les événements suivants :

MOI = le noeud récepteur,
LUI, MOTIF, ACTION = ceux de l'événement que l'on veut transmettre,
ORIGINE = SOUS (resp. SUR)

On voit alors apparaître ici le sens de la variable prédéclarée ORIGINE, accessible aux algorithmes recevant un événement.

Si elle vaut SOUS (resp. SUR) elle indique que l'événement a été sous-traité (resp. sur-traité).

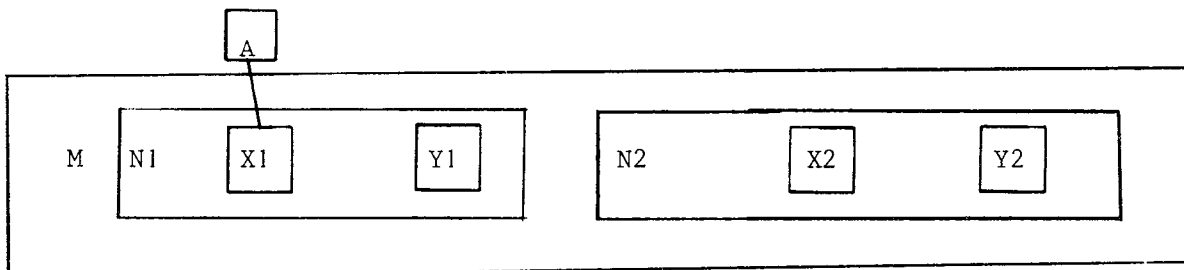
Rappel : Si elle vaut DIRECTE, elle indique que l'événement est normal.

Cette variable permet en particulier, de veiller à ce que les algorithmes de décision ne se "renvoient pas la balle" indéfiniment.

III.4.2. DISCUSSION DE DIFFERENTES SOLUTIONS POUR LA GESTION DES NOEUDS MULTIPLES IMBRIQUES, ET MOTIVATIONS DU CHOIX DE SUR-TRAITER, SOUS-TRAITER

III.4.2.1. Position du problème

Considérons la situation suivante où le noeud multiple M a pour composants des noeuds également multiples N1 et N2.



Soit A un fils de M, X1 un composant (simple) de N1.

Une séquence typique conduisant à la situation représentée sur la figure ci-dessus est :

A fait VIVRE
 M fait AFFECTER(N1,A)
 N1 fait AFFECTER(X1,A)
 X1 fait ASSOCIER(A)

Plaçons-nous dans le cas où X1 est dans l'état actif ; alors A lui-même se retrouve dans l'état actif. Supposons que, pour une raison quelconque, l'état de A passe de actif à inactif.

Selon la définition des noeuds multiples, M et N1 doivent chacun être avertis de cette transition de l'état d'un de leur fils.

Lorsque M est averti, il peut par exemple, décider :

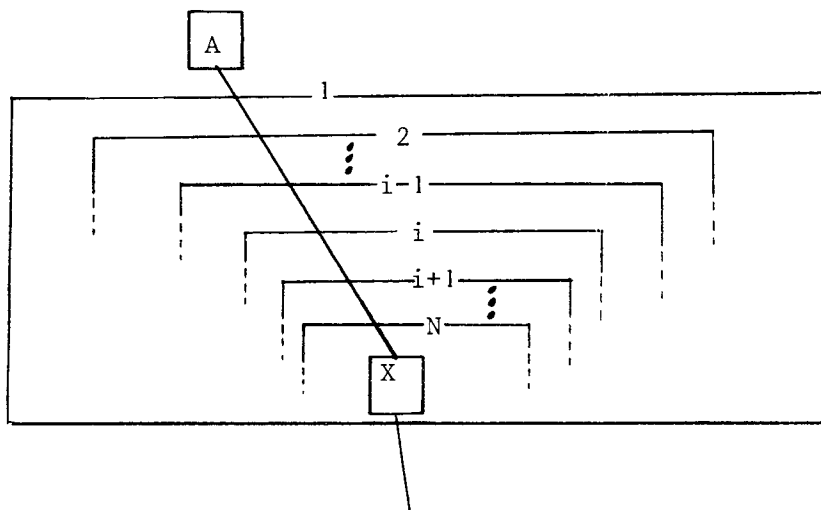
DESAFFECTER(A)
 AFFECTER(N2,A)

Lorsque N1 est averti, il peut décider :

DESAFFECTER(A)
 AFFECTER(Y1,A)

Il est clair que ces deux décisions sont incompatibles. Il faut que, par un moyen quelconque, l'un des noeuds multiples M ou N1 ait la priorité pour réagir à la transition de l'état de A.

Cette question se généralise au cas d'une imbrication de noeuds multiples à N niveaux.



La situation décrite résulte de la suite de primitives
 $[M_i \text{ exécute } \text{AFFECTER}(M_{i+1}, A)]$
 de $i=1$ à $N-1$.
 Puis, M_N exécute $\text{AFFECTER}(X, A)$.

La suite, M_1, \dots, M_N sera appelée suite des tuteurs de A.

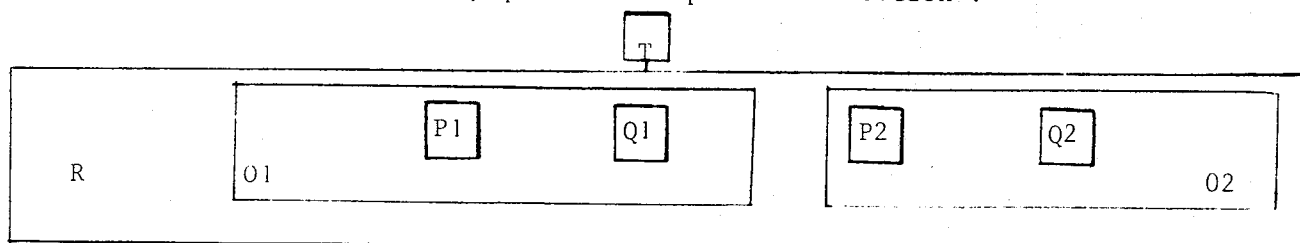
Si l'on désire que la gestion du système hiérarchisé soit strictement modulaire, il faut respecter les points suivants :

- chacun des M_i a un algorithme de décision qui prévoit de "faire quelque chose" lorsque A change d'état,
- il faut interdire que ces algorithmes soient interdépendants par exemple au vu des relations topologiques. En particulier, M_i ignore totalement son appartenance à une liste de tuteurs (il ignore en particulier l'existence éventuelle d'un M_{i-1} , ainsi que la nature de M_{i+1} : multiple ou simple). M_i ne connaît que A et ses variations d'état.

Il y a donc un conflit en puissance entre les politiques d'affectation de la suite des tuteurs d'un noeud simple. Ce problème est très important dans l'architecture d'un système hiérarchisé.

III.4.2.2. Discussion d'un exemple pratique

Un travail T est à la charge d'un réseau d'ordinateurs R, comportant deux ordinateurs O1 et O2, chacun comportant respectivement deux processeurs (P1, Q1) et (P2, Q2). Au départ, T est affecté à O1 par R, O1 l'affectant lui-même à P1, qui en entreprend l'exécution :



Pour une raison indépendante de T, l'exécution de T cesse.

La question posée globalement est :

Convient-il alors :

- solution 1 : ou bien de laisser T sur P1, c'est-à-dire de ne rien faire,
- solution 2 : ou bien de retirer A à P1 pour l'affecter à Q1,
- solution 3 : ou bien de retirer A à O1 pour l'affecter à O2.

Des arguments très divers peuvent être développés pour trancher cette question, suivant le but recherché, et suivant le degré de connaissance que l'on a de l'ensemble du système :

- on peut supposer que le temps de transfert du programme T de P1 à Q1 est plus rapide que son transfert de O1 à O2, ce dernier pouvant faire intervenir des communications de fichiers sur le réseau ; ceci penche en faveur des solutions 1 ou 2,

- si l'indisponibilité de P1 vis-à-vis de T est permanente (cas d'une panne), O1 va voir sa puissance de traitement diminuer par rapport à O2. Il peut alors être bon de rééquilibrer leurs charges respectives, par exemple en choisissant la solution 3,

- la cause de l'interruption du traitement de T par P1 peut être la soumission à P1 d'un travail T' plus prioritaire. Il faut alors distinguer deux cas selon l'origine T' :

- a) si T' a été confié à O1 par R, c'est R qui est responsable en premier lieu de l'arrêt de T, donc il est normal qu'il décide en premier de son sort, par les solutions 1 ou 3,

- b) si au contraire, T' ne provient pas de R, mais par exemple, d'un utilisateur de O1, ce dernier est plus compétent que R, qui ignore tout de T'.

Par exemple, O1 peut savoir que T' sera terminé très vite, et choisira la solution 1, ou que T' sera long et choisira la solution 2.

III.4.2.3. Quelques solutions envisageables

Notre but ici n'est pas de proposer des algorithmes ou des architectures spécialisées pour résoudre ces problèmes, mais de proposer un outil de spécification et de simulation de la solution choisie par le concepteur de système.

Cet outil doit à la fois :

- être aussi général et pratique que possible,
- ne pas limiter ou détourner la liberté de l'utilisateur, en l'induisant dans un choix faussé par des caractéristiques intrinsèques de l'outil, sans rapport avec la solution qu'il veut modéliser.

Dans le cas précis de ce chapitre, le but est de permettre à l'utilisateur de définir l'ordre dans lequel les tuteurs d'un noeud simple vont être avertis de sa variation d'état. Nous appelons cet ordre : ordre des responsabilités des tuteurs.

Voici une liste de différentes solutions possibles, et leurs critiques :

- 1) L'ordre de responsabilité est l'ordre direct d'imbrication des tuteurs : du plus extérieur au plus intérieur.
- 2) C'est l'ordre inverse du précédent.
- 3) C'est un ordre quelconque, déterminable pour chaque liste de tuteurs par une certaine fonction prédéfinie par l'utilisateur.

Commentaires :

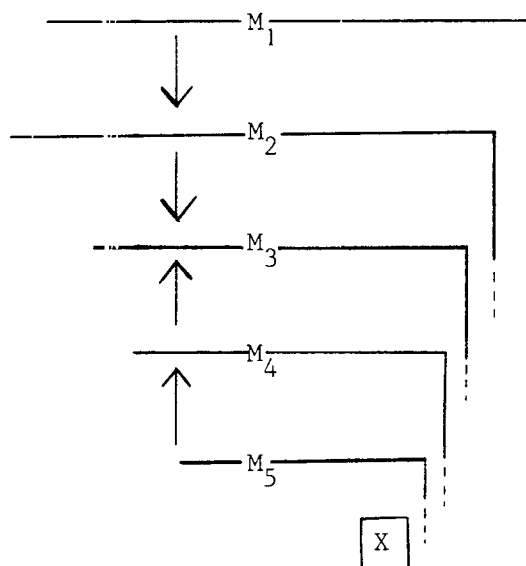
Les solutions 1 et 2 sont très rigides, mais ont l'avantage d'être applicables d'une façon modulaire. La solution 3 est plus souple, mais n'est pas modulaire : c'est une fonction qui peut être complexe, et qui doit être considérée soit comme :

- imposée de l'extérieur, dans le formalisme de description du modèle,
- soit comme décidée dynamiquement par un noeud du système, qui doit avoir connaissance des autres tuteurs de la liste.

4) Une solution naturellement modulaire se situerait au niveau de la primitive COMPOSE.DE qui permet de construire les imbrications de noeuds multiples : il suffit de spécifier l'ordre de responsabilité dans le couple objet de la relation :

Par exemple : COMPOSE.DE(B,1) dans la déclaration de A indique que A prime sur B ;
COMPOSE.DE(B,2) dans la déclaration de A indique que B prime sur A.

Le schéma obtenu est alors du type suivant :



auquel correspondent les instructions :

$M_1 \leftarrow \text{DN}(,,\text{COMPOSE.DE}(\text{$
 $M_2 \leftarrow \text{DN}(,,\text{COMPOSE.DE}(\text{$
 $M_3 \leftarrow \text{DN}(,,\text{COMPOSE.DE}(\text{$
 $M_4 \leftarrow \text{DN}(,,\text{COMPOSE.DE}(M_5,1)),1)),2)),2))$

Cet ensemble de relations, établi modulairement, ne définit malheureusement pas un ordre total de responsabilité.

Il doit de plus être complété par une indication du premier noeud à avertir, ce qui peut être fait de l'une des trois manières précédemment citées pour la suite toute entière.

(le plus interne, le plus externe, un quelconque).

L'inconvénient de ce mécanisme, est de limiter la transmission de la transition à la sous-suite - incomplète - totalement ordonnée contenant le responsable initial.

III.4.2.4. Solution préférée

La solution la plus générale qui respecte la modularité est la suivante :

L'ordre de responsabilité est défini dynamiquement par les noeuds multiples, de manière explicite, par l'intermédiaire de deux primitives :

- SOUS-TRAITER
- SUR-TRAITER

Lorsque la fonction de décision d'un noeud multiple M_k est avertie d'une transition de l'état d'un fils,

- ou bien elle réagit en émettant des AFFECTER, DESAFFECTER,
- ou bien elle émet SOUS-TRAITER, ce qui a pour effet d'avertir M_{k+1} de la transition,
- ou bien elle émet SUR-TRAITER, ce qui a pour effet d'avertir M_{k-1} de la transition.

Il reste là aussi à définir le premier responsable. Mais ce choix n'a alors que peu d'importance, puisque, par le jeu des primitives SOUS-TRAITER SUR-TRAITER, le cheminement des responsabilités peut être quelconque, et aboutir à n'importe quel ordre désiré.

N.B. : Dans un tel système, il faut veiller à ce que les tuteurs ne se renvoient pas indéfiniment la responsabilité.

Par exemple, un noeud pourra accepter p fois au maximum que son composant réponde "SUR-TRAITER" à son "SOUS-TRAITER", après quoi, ou bien il s'efforcera de prendre seul sa décision, ou bien il émettra "SUR-TRAITER" à son tour.

II.4.3. CONSTRUCTION D'UN NOEUD MULTIPLE STANDARD

Un noeud multiple a une structure potentiellement plus complexe qu'un noeud simple.

On ne peut par conséquent, espérer déterminer des configurations aussi "standard" que le sont celles des noeuds simples. C'est plutôt ici à l'utilisateur de SSH qu'il reviendra de construire ses propres noeuds multiples standard pour ses propres applications.

Nous nous limiterons donc au cas suivant :

- le noeud multiple standard a ses composants tous équivalents,
- il affecte au plus un fils à la fois à chaque composant,
- les candidats affectés sont les plus prioritaires (mêmes conventions que pour la priorité des noeuds standard simples),
- si un fils subit MOTIF = ILSTOPPE et s'il existe un autre fils de priorité inférieure qui est actif, alors on les permute.

Sa déclaration sera de la forme suivante :

$$A \leftarrow \text{DNM}(\underline{\text{PERE.DE}}(F1, \dots, FN), \underline{\text{COMPOSE.DE}}(C1, \dots, CP))$$

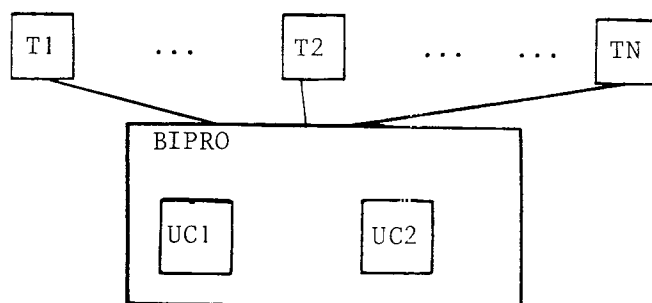
Les algorithmes détaillés sont décrits en annexe.

III.4.4. EXEMPLE D'UTILISATION DU NOEUD MULTIPLE STANDARD

Modélisation d'un bi-processeur banalisé exécutant des tâches.

$$\text{BIPRO} \leftarrow \text{DNM}(\underline{\text{PERE.DE}}(T1, \dots, TN), \underline{\text{COMPOSE.DE}}(UC1 \leftarrow \text{DNL}, UC2 \leftarrow \text{DNL}))$$

Représentation graphique :



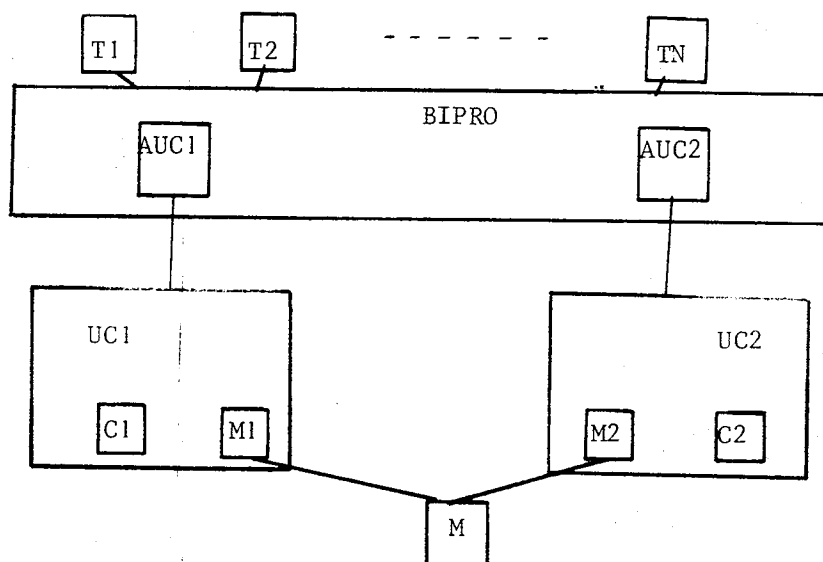
Remarque :

Dans le cas présent, où UC1 et UC2 sont identiques, (des DNL par exemple), le nouveau modèle est strictement équivalent à $BIPRO \leftarrow DNP(PERE.DE(T1, \dots, TN), , 2)$.

Il va prendre tout son intérêt si les UC sont différentes, soit en elles-mêmes, soit parce qu'elles ont des ascendants différents.

Voici un modèle de biprocesseur qui met en évidence les conflits-mémoire.

Les deux unités centrales sont supposées avoir une mémoire commune.



C1, C2, M sont des noeuds simples lambda DNL, BIPRO est un noeud multiple standard.

UC1 et UC2 affectent alternativement leur fils AUC1 (resp. AUC2) tantôt à C1 (resp. C2), tantôt à M1 (resp. M2), modélisant de cette manière le fait qu'une unité de traitement :

- ou bien accède à la mémoire : quand AUC_i est affecté à M_i,
- ou bien travaille dans ses registres locaux : quand AUC_i est affecté à C_i.

Lorsque les AUC_i sont ensemble affectés à leur M_i, M n'associe qu'un seul des M_i : il y a conflit-mémoire.

UC1 et UC2 sont en quelque sorte des noeuds multiples "barillet".

Pour modéliser le cas où M est une mémoire composée de plusieurs bancs accessibles indépendamment, il suffit de déclarer :

M ←DNM (PERE.DE(M1,M2),COMPOSE.DE(BANC1←DL,BANC2←DL))

III.5. PROBLEME DES NOEUDS AYANT PLUSIEURS PERES - NOEUDS DE TYPE RESSOURCE

Jusqu'à présent, la relation père-fils introduite dans SSH impose qu'un noeud ne peut avoir qu'un seul père alors que, dans la TSH, une articulation peut être utilisatrice de plusieurs ressources.

C'est d'ailleurs dans ce cadre que sont définies les troisièmes et quatrièmes inductions de la TSH : l'induction de libération et l'induction de réquisition. Pour traiter ces structures "multiresources" trois possibilités peuvent être envisagées.

III.5.1. LA SOLUTION TSH

Toutes les inductions résultent de programmations explicites des algorithmes associés à un noeud. Dans les cas de multiresources, ces algorithmes et les structures de données associées peuvent être très complexes et l'on connaît peu de techniques pour garantir leur optimisation ou leur sûreté. La simplification que pourrait apporter la TSH à leur écriture constitue d'ailleurs une perspective intéressante.

III.5.2. LA SOLUTION AVEC DES NOEUDS COMPOSES

La notion élémentaire du noeud simple à activité séquentielle de SSH s'accorde mal en fait avec la possibilité d'avoir plusieurs pères. En SSH, en définitive, tout se ramène à combiner les avancements des horloges de chaque noeud.

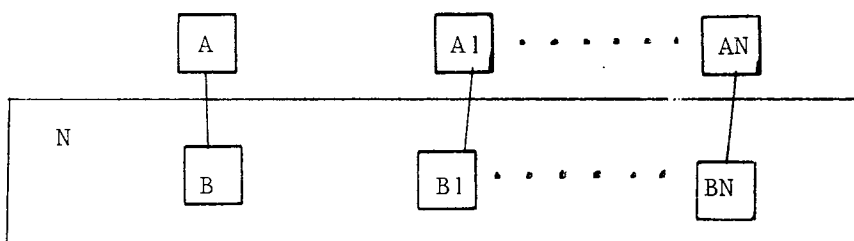
Si HF est l'horloge d'un fils et HP1,...,HPN les horloges des pères multiples de ce fils, il s'agit d'établir une relation

$$HF = f(HP1, \dots, HPN)$$

Or, ceci ne correspondrait à aucune réalité et cette fonction n'en est pas une : cela n'a pas de sens de représenter par un seul scalaire HF l'état d'avancement de plusieurs horloges en général indépendantes.

Une solution, en SSH, pour modéliser le fait qu'une activité peut en occasionner plusieurs autres est d'utiliser un noeud composé :

Exemple :



Lorsque DECISION de N affecte A à B, elle affecte aussi A1 à B1, ..., AN à BN.

DECISION est averti chaque fois que les A_i deviennent actifs ou inactifs et peut donc contrôler globalement l'activité multiple constituée par les A_i , par un algorithme de complexité arbitraire.

Exemple d'algorithme

Tous les A_i sont désaffectés dès qu'un seul cesse d'être actif : cela est un exemple d'un cas d'induction de libération de TSH.

III.5.3. SOLUTION "SIMPLISTE" : LES NOEUDS DE TYPE RESSOURCE

Dans la pratique courante, les utilisations de plusieurs ressources sont souvent simples : exemple : une unité centrale et un ou plusieurs fichiers.

Pour modéliser économiquement cela, on introduit un nouveau type de noeud : RESSOURCE tel que :

- il ne possède pas de père, ni d'activité,
- il peut être sollicité par les activités d'autres noeuds au moyen de deux primitives :

* * DEMANDER(R) où R est une ressource, et génère l'événement
MOI = R ; LUI = l'exécutant ; MOTIF = ILARRIVE

* * LIBERER(R) où R est une ressource, et génère le même événement
avec MOTIF = ILSENVA

- il possède un algorithme de décision unique qui peut

* * BLOQUER un demandeur ;
c'est équivalent à un MOURIR du demandeur ;

* * DEBLOQUER un demandeur ;
c'est équivalent à un VIVRE du demandeur ;

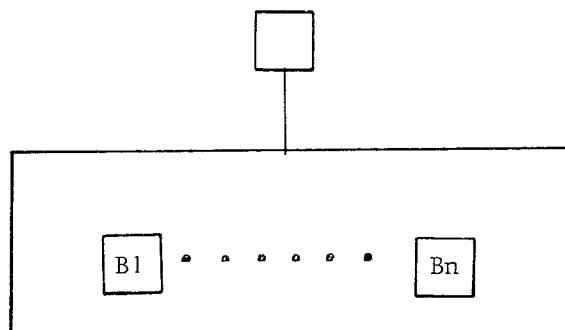
Voici l'instruction de déclaration d'un noeud ressource :

```
R ← DN(RESSOURCE,ESPACE.LOCAL(taille,<init>,<copie>,<dest.>)
      ,ALGORITHMES(<décision>,<archives externes>,
                    <archives internes>)).
```

Nous sommes conscients que ce type de noeuds RESSOURCE n'est pas homogène au reste de la TSH. Son seul intérêt est de faciliter la modélisation de systèmes simples, où de telles ressources sont "passives" : zones de mémoire centrale ou secondaires par exemple .

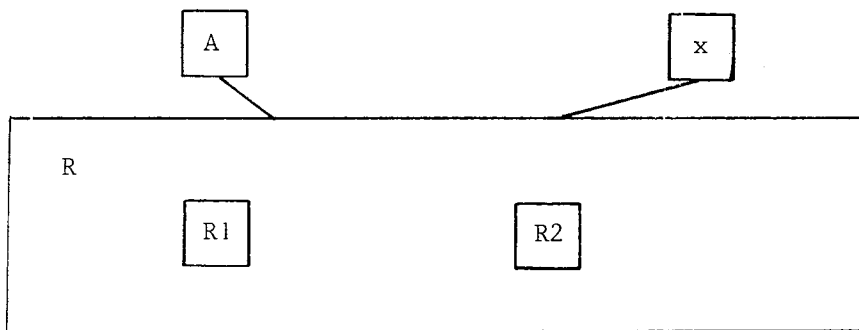
III.5.4. EXEMPLES D'UTILISATION DE NOEUDS DE TYPE RESSOURCE

Il n'existe pas de relation PERE-FILS entre les ressources ;
par contre, une ressource peut être un composant d'un noeud multiple :



On peut représenter ainsi un pool de ressources, dont la gestion est figurée par l'algorithme de décision du noeud multiple.

De cette façon également, on peut associer à l'utilisation d'une ressource le déroulement d'une certaine activité :



- R1 est une ressource
- R2 est un noeud quelconque, chaque fois que R affecte A sur R1, elle affecte également X sur R2. L'important dans cette configuration est que A ne connaît que R comme ressource, et ignore qu'une activité va être induite sur R2 lorsqu'il jouira de R.

Cet exemple montre que grâce au "relais" constitué par un noeud multiple, on peut contourner la restriction qui impose à une ressource de ne pas avoir de père.

Nous avons préféré dans SSH, ne pas introduire la possibilité de structure multiressources générales, tout en permettant de s'y ramener par l'utilisation combinée de notions plus élémentaires (noeuds simples, noeuds multiples, noeuds ressources).

IV - QUATRIEME PARTIE

Cette partie présente :

- de nouvelles instructions qui, si elles n'accroissent pas la puissance théorique du langage, augmentent sa concision et sa facilité de mise en oeuvre.

IV.1. LES INSTRUCTIONS DE LANCEMENT ET D'ARRET DE LA SIMULATION

IV.1.1. * * SIMULATION

Syntaxe :

SIMULATION(<noeud initial>,<noeud final>,<durée de simu>)

où <noeud initial> et <noeud final> sont des noeuds simples.

Sémantique :

La simulation va commencer par l'exécution par SSH d'une instruction ENVOI (père de<noeud initial>,<noeud initial>).

Après un temps absolu égal à <durée de simu>, SSH va exécuter une instruction :

ENVOI(père.de<noeud final>,<noeud final>)

IV.1.2. * * STOP

Cette instruction met fin à la simulation.

Elle peut être exécutée par n'importe quel algorithme.

IV.1.3. EXEMPLE D'UTILISATION

- Il faut que <noeud initial> devienne immédiatement actif.

Son algorithme d'activité commence alors son exécution, et par exemple, exécute une série de ENVOI qui initialisent le travail dans le reste du système.

- Il faut que <noeud final> devienne actif à la suite de son ENVOI par SSH à son père.

Suivant les cas, son algorithme d'activité peut alors :

(par exemple sur commande conversationnelle de l'utilisateur)

- soit émettre STOP,

- soit émettre une nouvelle instruction

SIMULATION(<nouveau noeud initial>,<nouveau noeud final>,
<nouvelle durée>

IV.2. L'INSTRUCTION CONFIE.A

IV.2.1. SYNTAXE

Cette instruction est de la forme :

* * CONFIE.A(<N1>,<N2>,<sous-programme>)

où <N1> doit désigner un noeud simple ou multiple

<N2> doit désigner un noeud simple.

Elle peut se trouver dans l'instruction de déclaration d'un noeud, à la rubrique ALGORITHMES, au lieu d'un nom de sous-programme, à l'exception du cinquième paramètre de ALGORITHMES d'un noeud simple, qui est l'activité de ce noeud.

Exemples

A ← DN(SIMPLE,...,ALGORITHMES(CONFIE.A(X,Y,Z),P1,P2,CONFIE.A(T,U,V),P3)
...)

A ← DN(MULTIPLE,...,ALGORITHMES(P1,CONFIE.A(B,C,D),P2)....)

IV.2.2. SEMANTIQUE DE CONFIE.A

Cette instruction permet de donner une existence temporelle au déroulement des algorithmes de décision ou d'archivage qu'elle concerne. En son absence, ces algorithmes s'exécutent, comme nous l'avons vu, sans consommation de temps, puisqu'ils ne peuvent comporter d'instruction DUREE.

Si un algorithme a été spécifié par

CONFIE.A(P,F,ACT) alors chaque fois que cet algorithme doit être appelé, va se dérouler le mécanisme suivant :

- une copie du noeud simple F est faite. Soit CF,
- l'activité ACT est attribuée à CF,
- CF est donné comme candidat à P par la génération de l'événement

MOI = P ; LUI = F ; MOTIF = ILARRIVE.

C'est l'activité ACT qui est censée exécuter temporellement l'algorithme concerné.

Elle le fera bien sûr dans la mesure où le noeud CF deviendra actif, ce qui dépend de P et de ses ascendants.

IV.2.3. EXEMPLES

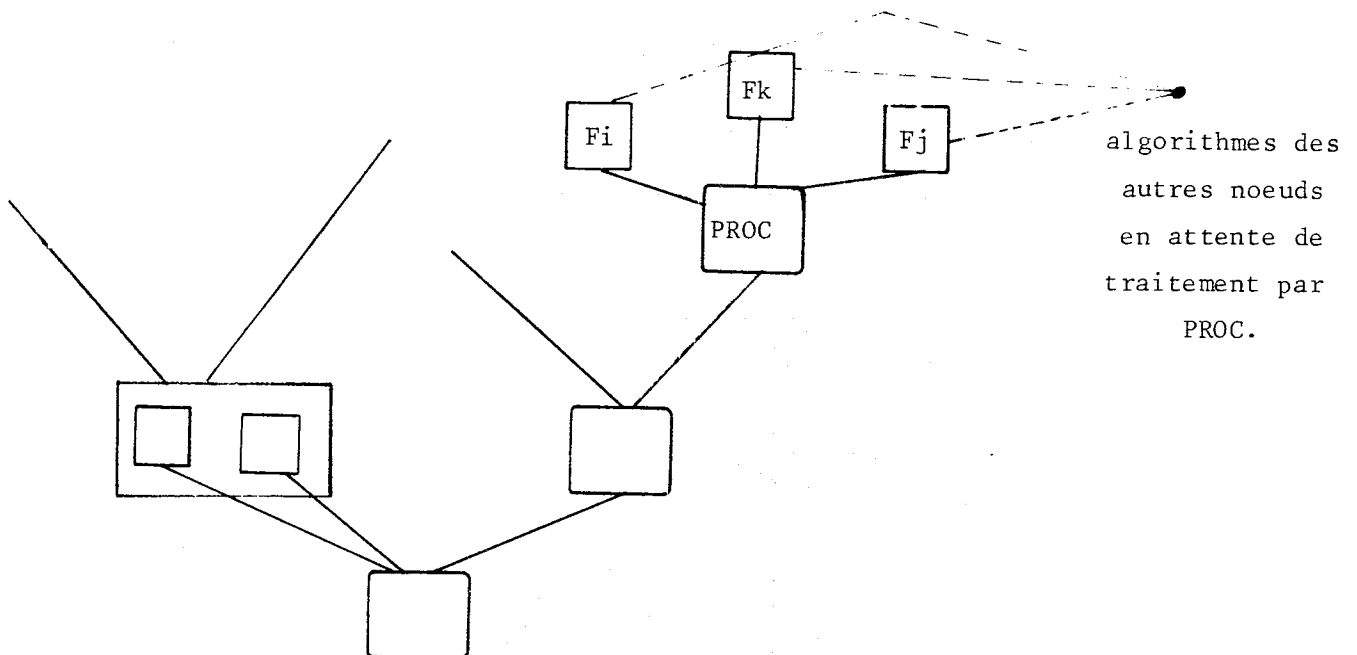
V.2.3.1. Exemple 1

ACT pourra :

- émettre des instructions du genre VIVRE, MOURIR, ASSOCIER, etc..
à des instants différents au cours d'un même appel,
- consommer du temps sur un ou plusieurs pères : ainsi, l'instruction CONFIE.A permet de simuler quantitativement ce que l'on appelle "l'overhead" dans les systèmes.

- traiter un événement un temps quelconque après qu'il soit intervenu. Par exemple, on peut imaginer un système où il existe un processeur spécialisé dans l'exécution des algorithmes de décision, et qui figure comme premier paramètre de toutes les instructions - CONFIE.A(PROC,Fi,ACTi).

Sur ce processeur va alors se constituer une file d'attente des événements arrivés, il les traitera dans l'ordre défini par son algorithme de décision.



IV.2.3.2. Exemple 2

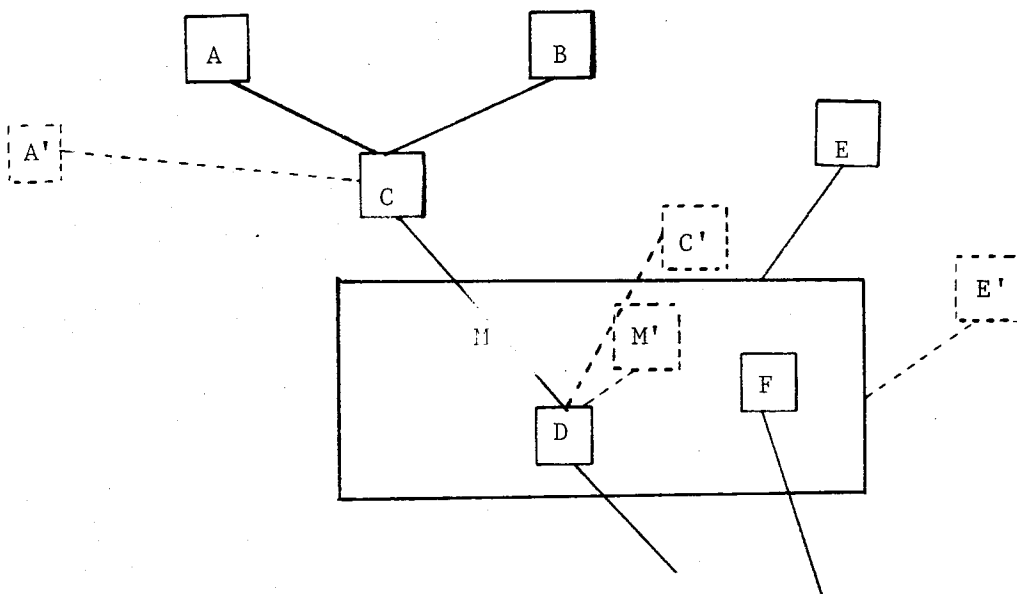
Dans les systèmes d'exploitation classiques, toutes les activités de synchronisation s'exécutent en priorité par rapport aux activités dites "en mode utilisateur", par exemple la synchronisation inter-utilisateur par messages, sémaphores, boîtes aux lettres, etc...

SSH permet :

1) de déclarer et d'écrire ces algorithmes de synchronisation de la manière la plus naturelle qui soit, c'est-à-dire en même temps que les noeuds utilisateurs, sans se soucier de l'existence d'un mystérieux "O.S." au-dessous d'eux.

2) d'exprimer que ces algorithmes sont à la charge du père des noeuds utilisateurs. C'est ce dernier qui se chargera éventuellement d'interrompre les noeuds utilisateurs pour exécuter ces algorithmes. Là encore, apparaît l'intérêt d'une structure hiérarchisée pour séparer les problèmes. Il suffit pour implémenter systématiquement cela que les algorithmes de tous les noeuds simples soient confiés aux pères des noeuds simples, et les algorithmes des noeuds multiples par exemple à un de leurs composants.

Exemple :



Un événement arrive sur A. Le traitement A' de cet événement est confié à C, ce qui provoque un événement dont le traitement C' est confié à M, celui de M étant confié à D. D exécute M', qui décide d'affecter C' à D. C' devient actif et décide que C associe A'.

Ainsi, l'événement initial est traité sur C.

Il se produit d'abord une induction directe des demandes de traitement d'événements, puis une induction inverse de leur traitement effectif.

L'instruction CONFIE.A apporte une grande finesse dans la modélisation des relations entre activités et synchronisations dans un système.

IV.3. L'INSTRUCTION CHANGER.VITESSE

IV.3.1. MOTIVATIONS

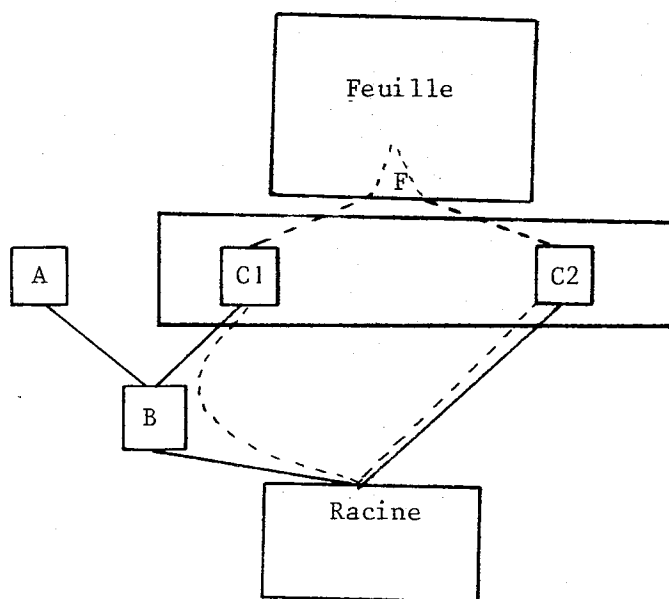
Dans le simulateur tel qu'il a été exposé jusqu'à présent, l'exécution d'un DUREE (t) dans l'activité d'une feuille se traduit nécessairement au niveau de la racine par une activité de durée t également, éventuellement en parallèle avec d'autres activités.

Cela signifie encore qu'un chemin de noeuds actifs a été établi pendant t unités de temps entre la feuille et la racine.

Comme le chemin actif peut varier au cours du temps, au niveau des noeuds multiples par l'intermédiaire des primitives AFFECTER, DESAFFECTER, cela signifie également que tous les chemins possibles doivent constituer des processeurs équivalents pour l'activité de la feuille.

La seule différence pouvant intervenir entre deux chemins de la feuille et la racine est que l'on peut être plus "occupé" que l'autre, ce qui se traduirait par des séjours plus longs de la feuille dans un état non actif :

Exemple :



L'activité de la feuille peut entrer en concurrence avec l'activité de A sur le chemin de gauche au niveau du noeud B.

Ainsi, les échéances de la feuille seront globalement traitées plus lentement sur le chemin de gauche que sur le chemin de droite.

On peut dire que du point de vue du noeud F, C1 est plus lent que C2.

La cause de cette lenteur réside dans la structure topologique du système.

Inversement, si l'on a au départ le désir de construire un système où par hypothèse tel chemin ou tel noeud est plus rapide ou plus lent que tel autre, dans un rapport donné, il n'est pas évident de trouver le système réalisant cette différence.

Par exemple, comment construire un système où il existe deux noeuds équivalents, composants d'un noeud multiple, symbolisant des unités de traitement dont l'une tournera 1,4 fois plus lentement que l'autre.

Avec notre simulateur actuel qui propage les DUREE intégralement le long des chemins, il est clair que le seul moyen serait de simuler au moins l'architecture complète de chaque unité de traitement, et peut être même le fonctionnement de leurs circuits ou de leurs composants, ce qui, en fait, ne nous intéresse pas du tout.

La solution est d'introduire plus brutalement et globalement le coefficient 1,4 sans essayer de détailler les phénomènes provoquant ce rapport de vitesse puisqu'il résulte lui-même d'une vue non détaillée du fonctionnement du système.

IV.3.2. DEFINITION DE L'INSTRUCTION

Syntaxe: CHANGER.VITESSE(N,V)

où N doit désigner un noeud de type simple et où V est un nombre rationnel positif.

Sa sémantique est la suivante :

Considérons un noeud simple N toujours actif.

Si son activité exécute DUREE(D), cela signifie jusqu'à présent que l'instruction suivante doit être exécutée dans D unités du temps simulé.

Si l'instruction CHANGER.VITESSE(N,V) a été exécutée avant DUREE(D), tout se passe comme si on exécutait DUREE(D:V).

On dira que V est la vitesse locale du noeud.

Plus V est grand, plus le moment de simulation de l'instruction suivante est rapproché. Il est donc légitime de parler de vitesse : les instructions de l'activité se déroulent plus ou moins vite selon que la vitesse locale de son noeud est plus ou moins grande.

Remarque

Pour se ramener au cas précédent, où la notion de vitesse n'existe pas, il suffit de considérer que toutes les vitesses locales des noeuds simples sont égales à 1.

IV.3.3. PREMIER EXEMPLE D'UTILISATION DE L'INSTRUCTION CHANGER.VITESSE : LA VITESSE LOCALE

Dans un modèle multiprocesseur, le conflit-mémoire entre les unités de traitement peut être représenté dans l'architecture même du système.

C'est une solution très coûteuse, puisqu'il faut pousser le détail de la simulation jusqu'au niveau des instructions et même des accès à la mémoire.

L'instruction CHANGER.VITESSE permet une solution beaucoup plus économique.

Il suffit de partir d'un modèle simple où le multiprocesseur est déclaré DNP(..., ..., Nombre de processeurs) en modifiant l'algorithme décision-père de la manière suivante :

"si le nombre des fils associés passe de n à $n+1$, alors pour chacun des deux fils associés, faire CHANGER.VITESSE(FILS, V_{n+1}), si le nombre des fils associés passe de $n+1$ à n alors CHANGER.VITESSE(FILS ASSOCIE, V_n).

Commentaires

S'il y a n fils associés à un instant donné, ils entrent en conflit pour la mémoire commune; V_n est un coefficient de ralentissement inférieur à 1, qui peut être déterminé à partir de mesures et de certaines hypothèses statistiques. Une telle détermination est exposée en détail dans BRUNET [11].

Remarque

L'introduction de la notion de vitesse fait passer de la simulation architecturale fine à une simulation basée sur des propriétés statistiques du modèle.

Mais l'instruction CHANGER.VITESSE n'est pas en contradiction avec la philosophie des systèmes hiérarchisés.

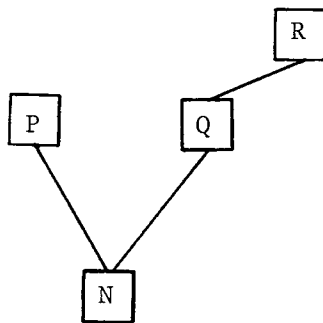
Au contraire, elle s'adapte très bien à leur structure :

a) dans un langage classique de simulation, il faudrait programmer de manière très lourde toutes les expressions figurant en paramètre d'instructions analogues à DUREE pour obtenir le même résultat.

b) ceci ne concernait que l'aspect local de la vitesse. Le paragraphe suivant va lui donner un sens global à tout le système.

IV.3.4. LA VITESSE GLOBALE D'UN NOEUD EN SSH

L'instruction CHANGER.VITESSE(N,K) n'agit pas seulement sur le déroulement de l'activité du noeud N, mais aussi sur le déroulement des activités de tous les noeuds simples descendants de N.



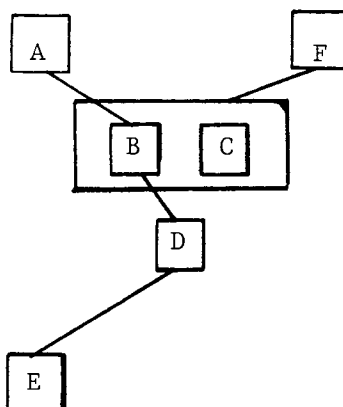
Exemple : CHANGER.VITESSE(N,K) modifie non seulement la vitesse de déroulement de l'activité de N, mais aussi, et dans la même proportion, celle des activités de P, Q et R.

Plus précisément :

On définit pour chaque noeud simple sa vitesse globale

- si un noeud simple n'est pas actif, sa vitesse globale est nulle,
- si un noeud simple est actif, sa vitesse globale est égale au produit des vitesses locales de tous le noeuds simples du chemin qui relie ce noeud à un noeud terminal, ces deux extrêmités incluses.

Exemple (Notons $VG(I)$ la vitesse globale de I et $VL(I)$ sa vitesse locale)



$$VG(F) = 0$$

$$VG(A) = VL(A) \times VL(B) \times VL(D) \times VL(E)$$

Cela signifie qu'une instruction DUREE(D) contenue dans l'activité de A du modèle avec vitesses, est équivalente à :

$$DUREE \left(\frac{D}{VL(A) \times VL(B) \times VL(D) \times VL(E)} \right)$$

dans un modèle précédent sans vitesse.

IV.3.5. MISE EN OEUVRE DE CHANGER.VITESSE

IV.3.5.1. SSH implémente cette notion de vitesse avec toute la généralité possible

En effet :

- à la déclaration d'un noeud simple, ses vitesses locales et globales sont initialisées à 1. Donc, si par la suite, aucune instruction CHANGER.VITESSE n'est exécutée, tout se passera comme auparavant,

- chaque fois que la vitesse locale d'un noeud change, toutes les vitesses globales de ses descendants sont recalculées en conséquence,

- chaque fois que la vitesse globale d'un noeud change, si son activité est en train d'exécuter un DUREE(D), on recalcule le temps restant à écouler avant l'exécution de l'instruction suivante en fonction de la nouvelle vitesse (l'algorithme exact est expliqué en VI.1),

- lorsqu'une activité est suspendue au milieu d'un DUREE(D), on décompte le temps déjà écoulé en fonction de sa vitesse globale courante.

IV.3.5.2. Induction des changements de vitesse

- De plus, puisque la notion de vitesse a été introduite pour résumer des variations de l'état d'un noeud simple et puisque les noeuds multiples sont conçus pour observer ces variations d'état et y réagir éventuellement, il est naturel que ces mêmes noeuds multiples puissent observer les variations de vitesse.

C'est pourquoi, si la vitesse globale d'un fils d'un noeud multiple varie, les fonctions de décision et d'archivage externe de ce noeud sont appelées avec l'événement suivant :

MOI = noeud multiple ; LUI = le fils

MOTIF = ILVARIE

De plus, une variable prédéfinie VITESSE donne alors la nouvelle vitesse globale de LUI.

A partir de cela, on peut imaginer toutes sortes d'algorithmes de décision dans un noeud multiple, tenant compte des variations de vitesse.

Par exemple, de même que le noeud multiple standard essaie de replacer un fils prioritaire qui subit "ILSTOPPE", sur un autre composant, de même pourrait-on affecter les fils de façon à ce que les plus prioritaires bénéficient des plus grandes vitesses globales.

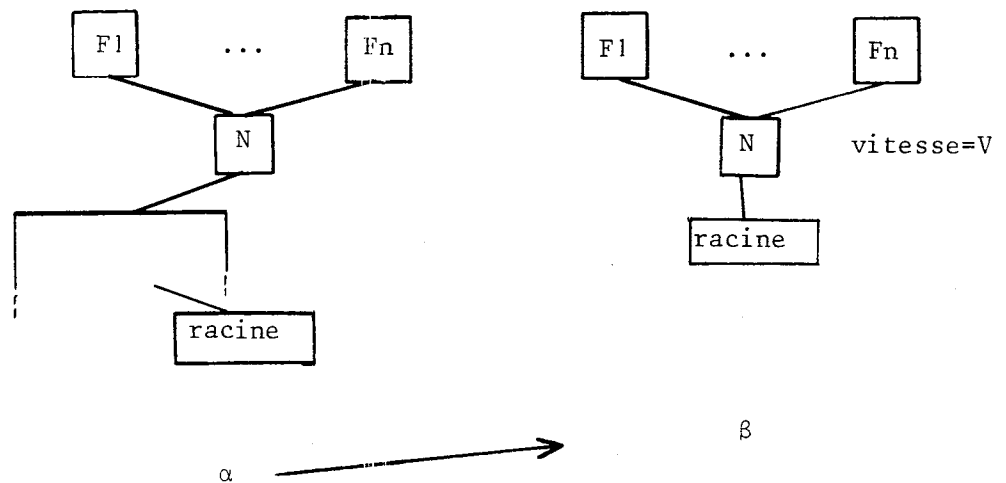
L'algorithme consiste simplement à maximiser

$$\sum_{i \in \text{fils}} (\text{PRIORITE DE } i) \times (\text{VITESSE DE } i)$$

IV.3.5.3. Autres exemples d'utilisations de la vitesse

A) La notion de vitesse est particulièrement utile pour la réduction des modèles dans le cadre d'une hiérarchie de simulations.

Exemple :



où $\frac{1}{V}$ est égal au rapport moyen (observé lors d'un nombre suffisant de simulations sur le modèle α entre la durée des périodes où N est candidat et la durée des périodes où il est actif.

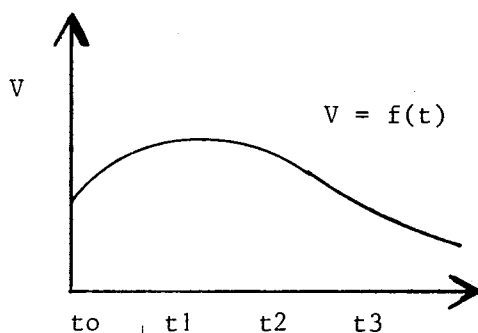
$$\frac{1}{V} = 1 + \text{moy} \frac{(\text{Durée de l'état candidat non actif})}{(\text{Durée de l'état actif})}$$

Il serait très possible d'automatiser de telles réductions, sur simple commande de l'utilisateur.

B) Utilisation "intensive" des vitesses

A la limite, on peut utiliser la possibilité de faire varier la vitesse pour rendre compte de n'importe quel phénomène continu.

Par exemple, on peut faire varier la vitesse d'un noeud suivant une fonction quelconque, avec un réajustement selon un pas de temps donné.



V_{t_i} , lorsque Heure (noeud) = t_i

Faire : CHANGER.VITESSE (noeud, $f(t_i)$).

C) Plus généralement, on peut définir certaines caractéristiques du système en imposant des relations entre les vitesses des différents noeuds. D'une manière très générale, si V est le vecteur composé des vitesses des différents noeuds et H celui composé de leurs horloges locales, il existera des instants où on calculera un nouveau V à partir des valeurs courantes de V et H .

$$\text{Nouveau } V = f(V, H).$$

Une telle formule revient à définir le calcul du temps dans le simulateur comme la solution d'un système d'équations différentielles du premier degré.

IV.6. LE MODE D'EXECUTION DIFFEREE, OU "MODE D"

Comme son nom l'indique, ce mode permet de différer l'exécution de certaines instructions SSH.

Ces instructions seront regroupées en paquets et, par la suite, ces paquets pourront être exécutés par exemple dans le cadre de l'activité d'un noeud simple quelconque du système (en mode normal ou même encore en mode "D").

IV.6.1. VOICI LES INSTRUCTIONS PERMETTANT DE METTRE EN OEUVRE LE MODE "D"

IV.6.1.1. * * PAQUET

Syntaxe : X ← PAQUET

Sémantique : On crée un paquet vide et X devient une référence à ce paquet.

IV.6.1.2. * * DIFFERER.DANS ET FIN.DIFFERER

Syntaxe : DIFFERER.DANS(X)

où X est une référence de paquet.

Sémantique : Le paquet X devient courant.

Il le restera jusqu'à ce qu'une instruction FIN.DIFFERER soit exécutée, ou qu'une autre instruction DIFFERER.DANS(Y) soit émise. Dans ce dernier cas, X est empilé et Y devient le paquet courant. Si la pile des paquets est vide, FIN.DIFFERER provoque le retour à l'état normal, sinon FIN.DIFFERER enlève le paquet du sommet de la pile des paquets et le rend courant.

Lorsqu'un paquet est courant (≡ lorsqu'on est en mode D), les instructions de synchronisation de SSH et l'instruction DUREE ne sont pas exécutées immédiatement, mais sont rangées dans le paquet courant au moment où elles auraient dû être exécutées, avec leurs paramètres passés par valeur. Les autres instructions de SSH ou de langage hôte sont exécutées normalement.

Exemple :

```

DIFFERER.DANS(P<PAQUET)
DO I=1,4
DUREE(2I-1)
IF I<2 THEN ASSOCIER(N1)
ELSE ASSOCIER(N2)
FIN.DIFFERER

```

va mettre dans le paquet P les instructions suivantes :

```

DUREE(1), ASSOCIER(N1), DUREE(3), ASSOCIER(N1),
DUREE(5), ASSOCIER(N2), DUREE(7), ASSOCIER(N2).

```

IV.6.1.3. EXEC.PAQUET

Syntaxe : EXEC.PAQUET(X)

où X doit référencer un paquet.

Cette instruction provoque l'exécution des instructions précédemment rangées dans le paquet X, dans l'ordre même où elles ont été rangées.

Si lors de l'exécution de cette instruction, on est également en mode D, alors les instructions du paquet X vont aller dans le paquet validé. Sinon, elles sont exécutées normalement, en particulier les DUREE

IV.6.2.1. Exemples d'utilisation du mode "D"

Voici un exemple de construction d'un noeud simple approprié à l'exécution de paquets, et les modalités pour l'envoi des paquets vers ces noeuds.

Pour envoyer un paquet X vers un noeud N, on utilisera l'instruction INFO(N,X).

L'algorithme d'archivage externe de N sera de la forme :

```

SI MOTIF = INFORMATION ALORS AJOUTER(LISTE.PAQUETS DE MOI,LUI)

```

L' algorithme d'activité de N sera

DEBUT :

DECISION(ATTENDRE.PAQUET)
EXEC.PAQUET(PAQUET.COURANT DE MOI)
 ALLERA DEBUT

Voici l'algorithme d'archivage interne :

SI MOTIF = CESTMOI ET LUI = ATTENDRE.PAQUET ALORS

PAQUET.COURANT DE MOI ← NOM DE NEXT DE LISTE.PAQUETS DE MOI.

L'algorithme de décision père :

SI MOTIF = CESTMOI ET LUI = ATTENDRE.PAQUET
 ET PAQUET.COURANT DE MOI = VIDE ALORS MOURIR

SI MOTIF = INFORMATION ET PAQUET.COURANT DE MOI = VIDE ALORS VIVRE

N.B. : PAQUET.COURANT doit être initialisé à VIDE

IV.6.2.2. Matérialisation de l'activité des algorithmes de décision et d'archivage

Le mode D permet de faire un peu la même chose que l'instruction CONFIE.A, avec moins de généralité, mais d'une manière plus souple et moins systématique.

Il suffit qu'un tel algorithme soit écrit ainsi :

SOUS-PROGRAMME nom :

DIFFERER.DANS(X←PAQUET)

corps de l'algorithme proprement dit, comportant éventuellement des DUREE

FIN.DIFFERER

INFO (noeud qui doit exécuter le paquet, X)

FIN nom

Un appel de ce sous-programme va générer dans le paquet X la trace temporelle des instructions SSH rencontrées lors de son exécution, et soumettre ce paquet à un noeud qui l'exécutera temporellement plus tard.

IV.6.2.3. Simulation de la projection d'un algorithme sur un ensemble de processeurs

On se trouve souvent face à la situation suivante, en particulier dans des méthodes de conception descendante d'architectures.

Etant donné un algorithme, on veut le découper en plusieurs parties, affecter un ensemble de ressources particulières (mémoires, processeurs) à chacune de ces parties et leur permettre de communiquer entre elles au moyen d'autres ressources elles aussi particularisées (bus, boîtes aux lettres).

Le mode D de SSH peut apporter une aide dans les phases de simulation d'une telle démarche. On procède ainsi :

- a) on définit $n+1$ processeurs N_0, N_1, \dots, N_N qui seront des noeuds de type exécution de paquets précédemment introduits,
- b) on partitionne le programme en segments pouvant s'exécuter indépendamment. Grossièrement, un segment sera un ensemble d'instructions compris entre deux primitives de synchronisation ou d'échange avec d'autres segments. On insère dans le programme des fonctions de délimitation de segments, de lancement de l'exécution d'un segment, d'attente de la fin de l'exécution d'un segment, ainsi que des instructions DUREE,
- c) le programme est exécuté normalement et intemporellement jusqu'à ce qu'il rencontre une fonction d'attente de fin de l'exécution d'un segment. Pendant son exécution, les fonctions insérées ont rempli un paquet pour chaque segment avec les instructions DUREE, et avec des instructions SSH exprimant les débuts et fin d'exécution des segments,
- d) chaque paquet est ensuite envoyé sur un processeur N_i affecté au segment. Là, il entrera éventuellement en concurrence avec d'autres fils du père de N_i . Le paquet sera exécuté lorsque N_i deviendra actif.

e) les fins de ces exécutions temporelles de paquets débloquent les programmes suspendus en c) et ainsi de suite.

On a ainsi dissocié l'exécution sémantique de l'algorithme initial qui s'est effectué dans le langage hôte, de son exécution temporelle qui s'est traduite par l'exécution de paquets d'instructions SSH.

Remarque :

Cette technique suppose évidemment que les calculs effectués dans un segment sont indépendants de leur instant d'exécution. Sinon, il faut découper plus finement l'algorithme en sous-ensembles qui seront exécutés de la même manière intemporelle, mais chacun à des instants différents.

V - CINQUIEME PARTIE

L'utilisation de SSH :

- Etude d'un cas concret
- Facilités de communication utilisateur-modèle
- Problèmes des données en entrée
- Analyse des résultats en sortie dans SSH

V.1. ETUDE DE CAS D'UNE UTILISATION REELLE DE SSH

La modélisation d'un multiprocesseur

Ce modèle a été construit dans le cadre d'une collaboration avec la TELEMECANIQUE ELECTRIQUE, pour l'étude des performances d'un projet de bi-processeur T1600.

Dans ce chapitre, nous présentons le programme complet de la simulation de l'exécution d'une application de contrôle de processus sur un biprocesseur.

V.1.1. LE SYSTEME A SIMULER BRUNET [11]

Il s'agit d'un calculateur T1600 auquel on a ajouté une seconde unité de calcul, le reste des composants du système restant inchangé.

Ce calculateur exécute des tâches "software" ou "hardware" (schématiquement une tâche "hardware" est un programme qui gère une facilité hardware). Ces tâches peuvent communiquer entre elles par l'intermédiaire de sémaphores qui sont soit :

- des sémaphores privés communicants (1 par tâche),
- des sémaphores généraux (par exemple pour réaliser une mutuelle exclusion).

Les seules opérations primitives utilisées dans les simulations sont les suivantes :

ARM T	appel de la tâche software T
	$\text{sema}(T) \leftarrow \text{sema}(T) + 1$
QUIT T	fin de la tâche software T
	$\text{sema}(T) \leftarrow \text{sema}(T) - 1$
RQST S	demande de la ressource S
RLSE S	libération de la ressource S

Toutes les tâches sont numérotées.

La priorité d'une tâche décroît lorsque son numéro croît.

L'algorithme d'ordonnancement des tâches est le suivant :

- Lorsqu'une tâche devient candidate pour être activée par le système biprocesseur, on compare sa priorité avec la priorité de la tâche la moins prioritaire parmi celles ayant le contrôle d'une unité de traitement. Si cette dernière tâche est plus prioritaire que la nouvelle venue, les deux tâches sont permutées.

- Lorsqu'une unité de traitement devient libre, on lui affecte la plus prioritaire des tâches en attente de traitement.

Plus simplement, disons qu'à tout instant les tâches activées sur les unités de traitement ont une priorité supérieure aux tâches en attente d'une unité de traitement.

La convention suivante, adoptée pour schématiser une opération d'entrée/sortie dans un simulateur TELEMECANIQUE, est reprise dans le modèle SSH pour préserver la compatibilité.

Une entrée/sortie est définie par :

- une durée DUR,
 - un facteur de répétition NB,
- elle sera symbolisée par ATT (NB, DUR).

Elle se déroule de la manière suivante : elle fait intervenir une tâche particularisée, la tâche T7 ou Tâche d'entrée/sortie, qui traite chacune des NB transaction élémentaire de manière identique : consommation de deux unités de temps sur le biprocesseur.

Lorsque plusieurs transactions sont en attente sur T7, celle-ci les traite suivant l'ordre de leur priorité.

(On suppose que le temps de traitement de l'E/S par l'unité est constant au cours d'une transaction, et qu'il n'y a pas de conflit au niveau des unités (autant d'unités que nécessaire) mais seulement au niveau de T7).

Ce temps de traitement est donné par DUR. On répète donc NB fois la séquence suivante :

- la tâche passe DUR unités de temps sur l'unité de sortie
- puis deux unités de temps comme fille de la tâche T7.

N.B. Ces deux unités de temps représentent un temps relatif, décompté uniquement lorsque :

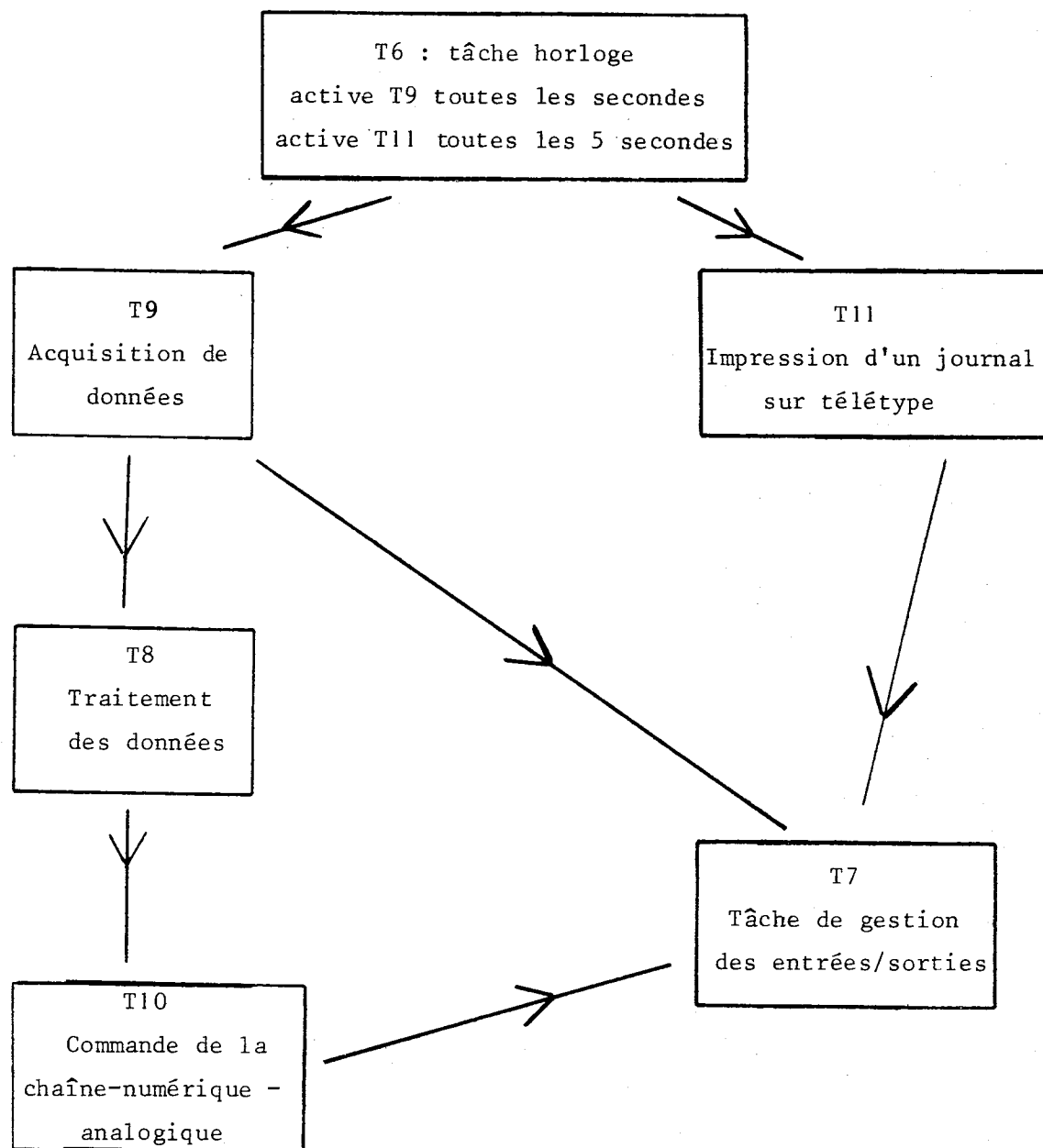
- T7 tourne,
- T7 s'occupe de cette E/S en particulier.

Remarque

T7 traite en fait une transaction en 3 unités de temps, mais libère la tâche demandeuse au bout de 2 unités de temps, l'unité de temps restante étant consacrée à son activité propre.

L'application Simulée

- a) Toutes les secondes, on lance une acquisition de données sur une chaîne analogique-numérique,
- b) Suivie de calculs sur ces données,
- c) Suivie d'une injection de commandes sur la chaîne numérique-analogique,
- d) Parallèlement, toutes les 10 secondes, on imprime un "journal" sur une console. Implémentation adoptée : 6 tâches numérotées de T6 à T11.



De plus, il existe une mutuelle exclusion entre T8 et T11, par l'intermédiaire d'un sémaphore général nommé SEMA 12.

V.1.2. LA SOLUTION SSH POUR SIMULER CE SYSTEME

La simulation des instructions de synchronisation du T1600

SOUS-PROGRAMME RQST(X):DEMANDER(X) ; FIN

SOUS-PROGRAMME RLSE(X):LIBERER(X) ; FIN

Pour les deux suivantes, on déclare un noeud utilitaire SERVICE ← DNL

SOUS-PROGRAMME ARM(T) : ENVOI(T,SERVICE) ; FIN ARM

SOUS-PROGRAMME QUIT : DECISION(ATTENDRE) ; FIN QUIT

ATT(NB,DUR) est un peu plus complexe :

WHILE NB←NB-1 ≥ 0 DO :

<u>ENVOI</u> (PERIPH, <u>MOI</u>)]	La tâche occupe le périphérique
<u>DUREE</u> (DUR)]	
<u>ENVOI</u> (T7, <u>MOI</u>); <u>DUREE</u> (2);]	La tâche occupe T7
<u>ARM</u> (FIN.T7) END;]	La tâche occupe encore T7
<u>ENVOI</u> (BIPRO, <u>MOI</u>)]	La tâche retourne sur le BIPRO

Description de la structure de la machine

```

BIPRO←DNP(PERE.DE(T6←DNL(,ACT6,1),
           T7←DNP(PERE.DE(FIN.T7←DNL(,ACTFT7))),
           T8←DNL(,ACT8),
           T9←DNL(,ACT9),
           T10←DNL(,ACT10),
           T11←DNL(,ACT11)
           ,,2)

```

Activités des tâches

```

SOUS-PROGRAMME ACT6 : E : DUREE(1);ARM(T9);QUIT;
                    IF CPT DE MOI←CPT+1 ≠ 0 MODULO 5 THEN GOTO E;
                    ARM(T11);QUIT;GOTO E;
                    FIN ACT6

```

```

SOUS-PROGRAMME ACTFT7 : E : DUREE(1);QUIT;GOTO F;FIN ACTFT7
SOUS-PROGRAMME ACT8 :
E:RQST(SEMA 12);DUREE(250);RLSE(SEMA 12); REM(T10);GOTO E;
FIN ACT8
SOUS-PROGRAMME ACT9 :
E:DUREE(2);ATT(10,25);DUREE(1);
SI CPT DE MOI←CPT DE MOI+1 ≠ 0 MODULO 30 ALORS ALLER A E;
QUIT;GOTO E;
FIN ACT9
SOUS-PROGRAMME ACT10
E:DUREE(80);ATT(2,10);QUIT;GOTO E;
FIN ACT10
SOUS-PROGRAMME ACT11 :
E:RQST(SEMA 12);DUREE(240); RLSE(SEMA 12);ATT(60,500);DUREE(1);QUIT;
GOTO E;
FIN ACT11.

```

Déclaration de la ressource de synchronisation SEMA 12

SEMA 12 ← DN(RESSOURCE,ESPACE.LOCAL(2),ALGORITHMES(DSR,AXS))

COMPTEUR DE SEMA 12←-1

SOUS-PROGRAMME DSR

SI MOTIF = ILARRIVE ALORS DEBUT

SI COMPTEUR DE MOI ← COMPTEUR DE MOI+1 > 0

ALORS BLOQUER(LUI)FIN

SI MOTIF = ILSENA ALORS DEBUT

COMPTEUR DE MOI ← COMPTEUR DE MOI-1

SI COMPTEUR DE MOI > -1 ALORS

DEBLOQUER(NOM DE LOCALISER(LISTE.FILS DE MOI,ETAT,VIDE;'='))

FIN

Commentaires

Ce sous-programme utilise les mêmes conventions de programmation que les noeuds simples standard décrits en détail en annexe.

Sous-programme AXS : c'est exactement le même que celui du noeud simple standard "lambda". Il est décrit en annexe A3.

Représentation du monde extérieur

Dans un modèle de simulation, il faut aussi représenter le monde extérieur.

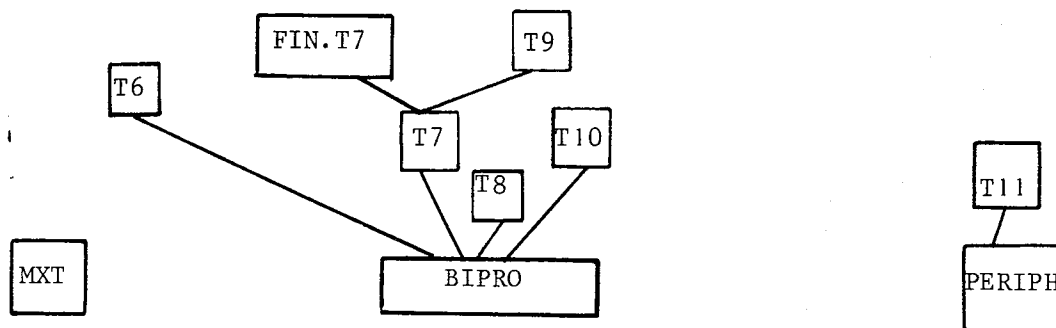
MXT←DNL(,ACT.MXT) dont l'activité consiste à démarrer le cycle de contrôle toutes les 10000 unités de temps.

SOUS-PROGRAMME ACT.MXT :

E:DUREE(10000);ARM(T6);GOTO E

FIN ACT.MXT

Représentation graphique du modèle (à un instant donné)



Ce modèle a été utilisé en parallèle avec un simulateur de la TELEMECANIQUE écrit spécialement pour le biprocesseur T1600 dont on trouvera le détail dans BRUNET [11].

Il a donné exactement les mêmes résultats.

SSH s'est révélé d'une grande concision à cette occasion puisque le modèle complet tient en une page de listing.

De plus, l'ensemble Modèle SSH + Programme d'implémentation de SSH était d'une taille comparable au simulateur écrit spécialement pour cette application par la TELEMECANIQUE.

V.2. TECHNIQUES POUR LES COMMUNICATIONS UTILISATEUR-MODELE ET POUR L'OBSERVATION DU SYSTEME

V.2.1. LES OUTILS GENERAUX D'AIDE A L'UTILISATION

Ils ne sont pas spécifiques à SSH mais dépendent :

- a) ou bien du langage hôte dans lequel SSH est implémenté,
- b) ou bien de la manière dont l'application est écrite.

Exemple pour a) :

Chaque implémentation de SSH bénéficiera des facilités du langage et du système hôte :

- commandes de mises au point, de trace,
- facilités d'enchaînement des travaux,
- modification conversationnelle des programmes, etc.

Exemple pour b) :

Si l'on simule un processeur d'instructions, on peut programmer une trace de l'exécution de ces instructions. Cela n'a rien à voir ni avec SSH ni avec son langage hôte.

V.2.2. LES OUTILS SPECIFIQUES A SSH

V.2.2.1. Insertion de noeuds espions

Il s'agit de permettre d'observer le fonctionnement d'un modèle, sans le perturber, et sans qu'il ait été nécessaire de prévoir à l'avance ces observations.

V.2.2.1.1. Noeuds espions du temps

On veut être tenu au courant de l'avancement du temps dans un noeud simple NS. Par exemple, on veut être averti chaque fois que l'horloge de NS progresse de n unités de temps.

Il suffit pour cela :

a) de créer un noeud simple `ESPION(,ACTSPY)`

dont l'activité est de la forme :
`SOUS-PROGRAMME ACTSPY`

`E:DUREE(n);appel d'un sous-programme SP;GOTO E;`
`FIN ACTSPY`

b) d'exécuter deux instructions de SSH non structurées (Cf. annexe B)

`PERE.FILS(NS,ESPION)` puis `ASSOCIER(NS,ESPION)`

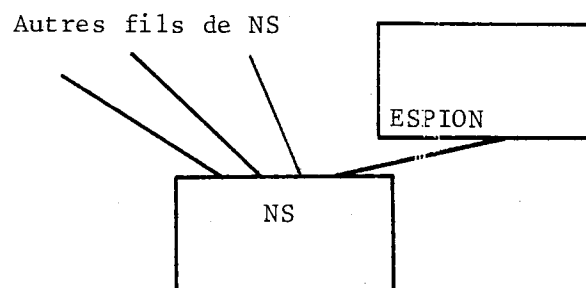
Ainsi, `ESPION` sera actif en même temps que `NS`, et chaque fois que le temps dans `NS` progressera de n, le sous-programme `SP` sera appelé, et pourra par exemple envoyer un message au terminal de l'utilisateur, ou à un autre noeud du système.

Remarque

L'important ici est que `NS` ignore totalement l'existence de `ESPION`.

- On a établi une relation `PERE.FILS` entre `NS` et `ESPION` sans rendre `ESPION` candidat à `NS` comme l'aurait fait un `ENVOI(NS,ESPION)` qui aurait alerté tous les algorithmes de `NS` (et par exemple, incrémenté des compteurs,...)

- `ASSOCIER(NS,ESPION)` a été exécuté à l'insu de `NS`, et non par son algorithme décision-fils.



V.2.2.1.2. Noeuds espions des événements

Soit un noeud simple NS. Soit PNS son père.

Créons un noeud multiple ESPION.

```
ESPION ← DN(MULTIPLE, PERE.DE(NS), COMPOSE.DE(PNS),  
ALGORITHMES(DECISION.ESPION,...),...)
```

Par définition d'un noeud multiple, DECISION.ESPION sera appelée chaque fois qu'un événement survient sur NS ; elle pourra alors transmettre les renseignements ainsi recueillis à un autre noeud ou à l'utilisateur.

De plus, pour ne pas se faire remarquer, il suffit que DECISION.ESPION simule l'ancienne relation père-fils entre PNS et NS.

Elle s'écrit donc :

SOUS-PROGRAMME DECISION.ESPION

transmettre les renseignements sur l'événement en cours.

IF MOTIF = ILARRIVE THEN BEGIN AFFECTER(PNS,NS);RETURN;END

IF MOTIF ≠ ILSENVA THEN SOUS-TRAITER

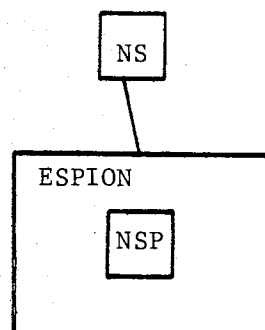
FIN DECISION.ESPION

(Rappel : si MOTIF = ILSENVA pour un fils de noeud multiple, le fils a déjà été automatiquement retiré du composant (ici NSP) par SSH).

Représentation graphique



Sans espion



Avec espion

N.B. : NSP peut être de type quelconque.

Remarque

Il est intéressant de remarquer que cette technique de pose de noeuds espions a pu être réalisée très simplement à partir des dispositions existantes de SSH, sans qu'il soit nécessaire d'inventer pour cela de nouvelles instructions.

V.2.2.2. Les outils spécifiques à SSH : l'utilisateur considéré comme noeud ou un ensemble de noeuds

C'est une solution radicale et élégante pour résoudre le problème des interactions utilisateur-modèle :

en SSH, le noeud dispose de prérogatives très étendues :
(modification dynamique des relations, des vitesses,
création / suppression de noeuds,
pose de noeuds espion).

Il est donc inutile d'inventer des prérogatives nouvelles pour l'utilisateur. Il suffit de considérer que l'utilisateur est un noeud (ou un ensemble de noeuds) du modèle. Les algorithmes d'un tel noeud sont écrits de telle façon que, lorsqu'un événement provoque leur appel :

a) l'identification du noeud, de l'algorithme et de l'événement est affichée au terminal de l'utilisateur,

b) c'est alors l'utilisateur qui exécute lui-même l'algorithme en question, en tapant sur son terminal des instructions dans un langage sémantiquement équivalent à SSH, et ce langage est interprété directement. Ceci est immédiat à implémenter dans un langage conversationnel (APL, BASIC, ALICE [9]).

V.3. APERCUS SUR LES PROBLEMES DE RECUEIL DES DONNEES EN ENTREE ET D'ANALYSE DES RESULTATS D'UNE SIMULATION

Le présent travail a surtout pour objectif de fournir un outil de description et de simulation.

Mais un outil à lui seul ne résout pas tous les problèmes, et il ne devient efficace que s'il est mis en oeuvre avec opportunité.

Il ne suffit pas de savoir programmer dans un langage de description de systèmes pour se prétendre "architecte de systèmes".

De même, il ne suffit pas de décrire et d'exécuter un modèle de simulation pour obtenir des renseignements exacts sur un système.

Tout travail de simulation

- a) doit commencer par une phase de recueil de données à fournir en entrée au simulateur,
- b) doit comporter des phases de validation du modèle,
- c) doit se terminer par une analyse critique des résultats obtenus,

De nombreuses études théoriques ou expérimentales sont faites sur ces sujets. On en trouve un exposé très complet dans BADEL [14], LEROUDIER et PARENT [15].

Dans ce chapitre, nous allons nous borner à :

- a) montrer sur un exemple la complexité du recueil des données en entrée par des mesures,
- b) proposer une technique facile à mettre en oeuvre dans SSH pour mieux interpréter certains résultats d'une simulation.

V.3.1. UN PROBLEME DE MESURES POUR LA SIMULATION

V.3.1.1. Difficulté du problème général

Examinons la question suivante :

"Quelles mesures effectuer lors de l'exécution d'un ensemble de tâches sur un monoprocesseur pour construire un modèle de cet ensemble de tâches dont l'exécution simulée sur un biprocesseur soit significative".

On se trouve face à trois inconnues :

- les mesures,
- le modèle,
- le simulateur

devant satisfaire une certaine contrainte :

- la simulation doit être significative, dans un domaine donné.

- le système réel étudié.

Ceci ressemble à un problème de programmation linéaire, malheureusement, nos inconnues ne semblent pas appartenir à un espace vectoriel normé !

Il existe, d'autre part, certaines dépendances entre les variables :

- un simulateur donné n'accepte que certains modèles,
- un type de modèle donné ne rend compte que d'un certain ensemble de mesures.

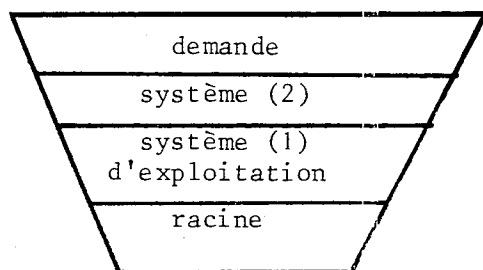
Il est réaliste de penser que l'on sera toujours limité par l'existence d'un simulateur (ou d'une famille de simulateurs), donc que nous aurons des contraintes sur les modèles, donc sur les mesures.

Ainsi, (en confondant modèle et simulateur), on se trouvera souvent dans la situation suivante :

"Approximation d'un système par un modèle donné au moyen de mesures liées au modèle".

Ainsi, dans notre exemple, on veut modéliser le fonctionnement d'un ensemble de tâches software (système 2) tournant sur un système d'exploitation multiprocesseur dont la modélisation et la simulation sont supposées ne pas poser de problèmes.

D'une manière générale, on peut schématiser le problème ainsi en employant le formalisme des systèmes hiérarchisés :



On fait une modification dans système 1, et l'on veut connaître par simulation les conséquences qui en résulteront pour le traitement de la demande, supposée connue.

La difficulté fondamentale est la suivante :

Les observations effectuées sur système 2 avec l'ancien système 1, ne permettent pas en général de se faire une connaissance exacte de ce que sera le comportement de système 2 avec le nouveau système 1 ; de nouveaux "cas de figure" vont se présenter pour système 2, que l'on n'aura jamais pu observer sur l'ancien système 1.

On peut alors essayer de suppléer aux carences des observations en pratiquant des expériences sur le système 2.

On conserve l'ancien système 1 connu, par contre, on va modifier à volonté la Demande de façon à tester au maximum système 2, en réalisant en quelque sorte son "identification" au sens des automaticiens.

De par la remarque précédente, une telle démarche reste néanmoins en général impossible.

V.3.1.2. Cas particulier

Nous étudierons maintenant un cas particulier où il est possible de conclure. Considérons une tâche pouvant être appelée dans différents contextes que nous appellerons :

p_1, p_2, \dots, p_n

(un contexte donné correspond par exemple au passage d'un paramètre donné lors de l'appel).

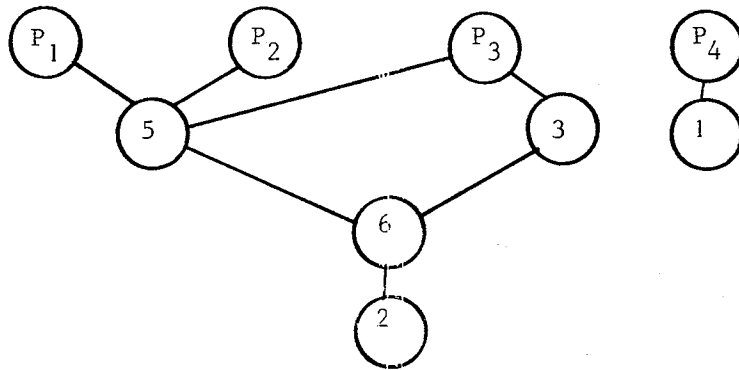
Cette tâche s'exécute strictement en monoprogrammation et l'on constate que le temps d'exécution de la tâche dépend non seulement du nom de l'appel mais aussi des appels précédents, c'est-à-dire de l'histoire de la tâche.

Le modèle qui nous sert de base est le suivant :

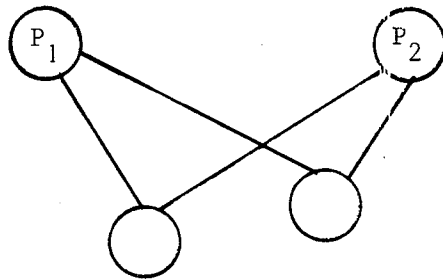
Nous considérons un graphe orienté sans circuits possédant n noeuds pères p_1, \dots, p_n non nécessairement connexe ayant les propriétés suivantes destinées à en faire un réseau de Petri :

- à chaque noeud, est associé un temps d'exécution,
- il existe une exclusion mutuelle entre tous les noeuds, (il ne peut y avoir qu'un seul noeud qui s'exécute à un instant donné).
- un noeud s'exécute si et seulement si tous les noeuds incidents ont terminé leur exécution,
- les noeuds sources ont un temps d'exécution nul,
- l'ordre d'exécution de plusieurs noeuds, tous susceptibles de s'exécuter au même instant n'est pas précisé.

Exemple :



- enfin, deux noeuds différents ont leurs ensembles de noeuds sources ascendants différents : par exemple, une telle configuration est interdite :



Un tel module peut être identifié en réalisant les expériences suivantes :

Pour chaque permutation $P_{i_2} \dots P_{i_n}$ sur les noeuds sources,
 1) on active le premier noeud,

- 2) on note le temps pendant lequel le système déroule une activité (c'est-à-dire qu'on attend que le système s'arrête de lui-même),
 3) on recommence en 1) avec le noeud suivant dans la permutation.

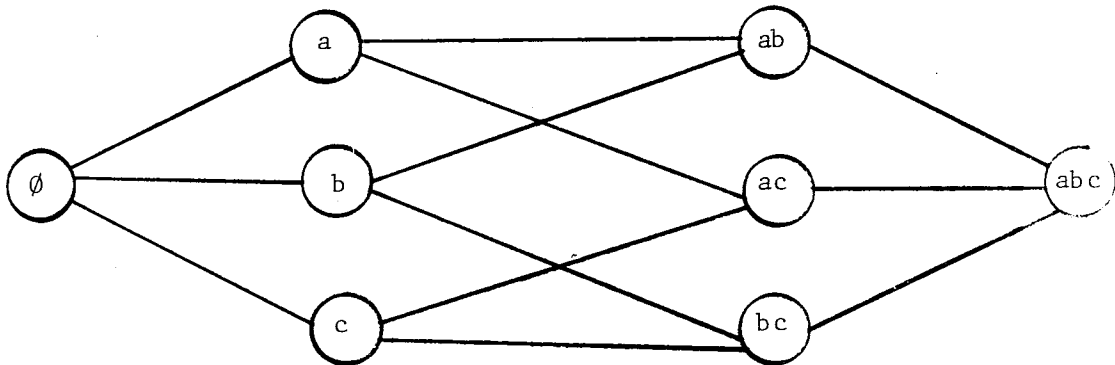
Le résultat est donc de la forme :

A toute permutation i_1, \dots, i_n on associe une suite de durées $\Delta i_1, \Delta i_n$ où chaque élément est positif ou nul.

A partir de ces résultats, on peut reconstituer le graphe du modèle de la manière suivante :

on construit le n-cube de toutes les parties de $\{P_1, \dots, P_n\}$

Exemple : pour $\{a, b, c\}$



Remarquons que si l'on enlève les 3 premières arêtes partant de l'élément \emptyset , on retrouve un graphe satisfaisant aux conditions de notre modèle.

Nous allons retrouver le système testé en le plongeant dans ce n-cube.

Nous déterminerons successivement :

- l'existence des noeuds communs au système étudié et au n-cube,
- la durée d'exécution de chacun des noeuds existants.

L'algorithme est le suivant :

Pour chaque permutation et pour tout $k \in [1, n-1]$, on marque l'arête joignant dans le n -cube le noeud $\{p_{i1}, \dots, p_{ik}\}$ au noeud $\{p_{i1}, \dots, p_{ik}, p_{ik+1}\}$ par la durée observée $\Delta\{p_{i1}, \dots, p_{ik}, p_{ik+1}\}$.

N.B. : l'ordre des p_{ij} à l'intérieur de la parenthèse n'a pas d'importance.

Propriété :

Un noeud $N = \{p_{i1}, \dots, p_{ik}\}$ existe dans le système étudié et son temps d'exécution est $T(N)$ si et seulement si :

pour chaque $ij, j \in [1, k]$

$\Delta\{p_{i1}, \dots, p_{ij}, \dots, p_{ik}\} = T(N) + \sum T$ (des noeuds prédécesseurs de N contenant p_{ij}).

On en déduit l'algorithme de construction du graphe :

1) dans le cas où l'on est sûr que le système étudié obéit strictement au modèle :

on se contentera d'utiliser une seule des égalités précédentes pour chaque noeud et l'on pourra même affirmer qu'un noeud existe si et seulement si pour un p_{ij} au choix :

$T(N) = \Delta(p_{i1}, \dots, p_{ij}, \dots, p_{ik}) - \sum T$ (des noeuds précédésseurs de N contenant p_{ij} est > 0 , alors $T(N)$ est la durée d'exécution de ce noeud.

Il suffit d'appliquer cette formule sur tous les noeuds couche par couche en commençant par les noeuds ne comprenant qu'un élément. Une fois les noeuds existants déterminés, les relations entre ces noeuds sont implicitement connues puisqu'elles s'identifient aux relations d'inclusion entre ces noeuds.

Remarque :

L'adjonction d'un noeud à temps d'exécution nul ne modifie pas le fonctionnement du modèle.

2) si le système à tester est à priori non conforme au modèle.
Pour chaque noeud N il faudra calculer le T(N) pour chaque p_{ij} .

Si sa valeur n'est pas la même pour chaque p_{ij} , alors le modèle ne rend pas compte du système réel.

L'intérêt de ce modèle est surtout de réduire considérablement l'information nécessaire à la représentation du système.

Si nous avons n points d'entrées, il faut $n \times n!$ entiers pour décrire les observations. Par contre, le modèle sera représenté par $2^n - 1$ entiers seulement.

V.3.1.3. Développements possibles

1) Il serait intéressant de généraliser le modèle en permettant l'utilisation d'une logique plus complète que le ET sur les entrées pour déclencher l'exécution d'un noeud : - OU, NON, COMPTAGE

- arrivée des paramètres dans un ordre donné. Ceci risquant évidemment d'augmenter la complexité des expériences à faire et de l'algorithme de synthèse du modèle.

2) Recherche d'heuristiques indiquant la conduite à tenir dans les cas non idéaux :

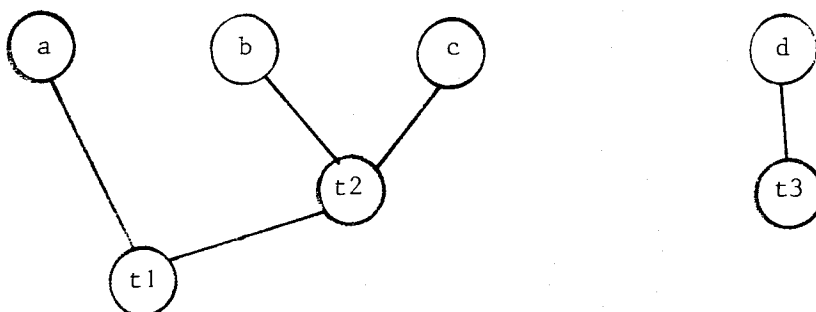
- ou bien l'on ne possède pas tous les chiffres des expériences :
quelles "interprétations" devons-nous effectuer sur le graphe ?

- ou bien les mesures n'obéissent pas au modèle, quelles règles d'approximation devons-nous choisir ?

- ou bien les deux problèmes à la fois.

V.3.1.4. Exemple

Modèle



Observations

abcd
o,o,t1+t2,t3

abdc
o,o,t3,t1+t2

acbd
o,o,t1+t2,t3

acdb
o,o,t3,t1+t2

bacd
o,o,t1+t2,t3

badc
o,o,t3,t1+t2

cabd
o,o,t1+t2,t3

cadb
o,o,t3,t1+t2

adbc
o,t3,o,t1+t2

adcb
o,t3,o,t1+t2

dabc
t3,o,o,t1+t2

dacb
t3,o,o,t1+t2

bcad
o,t2,t1,t3

bcda
o,t2,t3,t1

cbad
o,t2,t1,t3

cbda
o,t2,t3,t1

bdac
o,t3,o,t1+t2

bdca
o,t3,t2,t1

dbac
t3,o,o,t1+t2

dbca
t3,o,t2,t1

cdab
o,t3,o,t1+t2

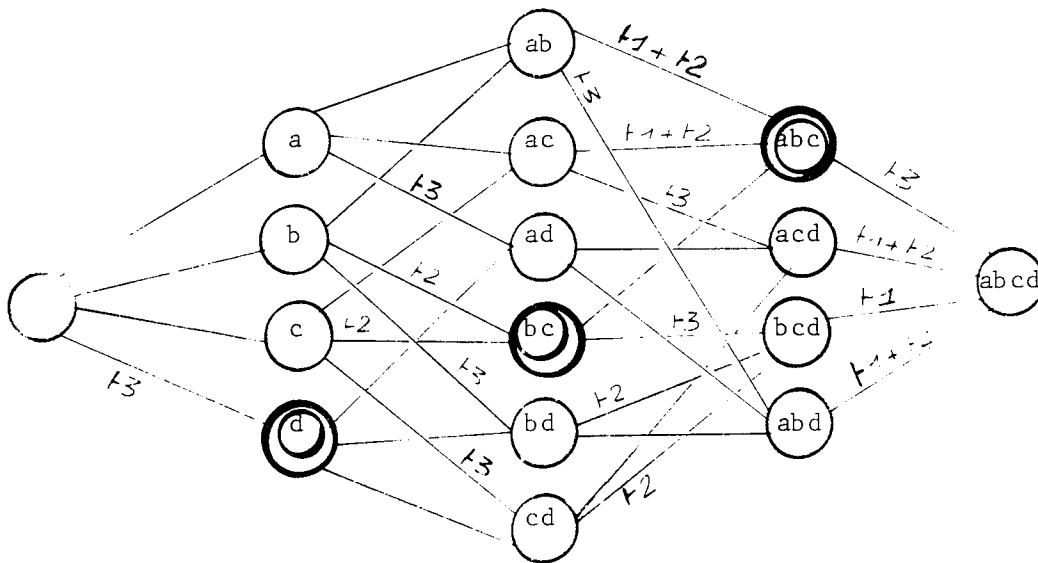
cdba
o,t3,t2,t1

dcab
t3,o,o,t1+t2

dcba
t3,o,t2,t1

en-dessous de chaque permutation est noté le temps d'activité observé après chaque libération du paramètre concerné.

Ces observations servent à remplir le 4-cube suivant, à partir duquel l'algorithme retrouve la structure du modèle ci-dessus.



L'algorithme d'existence des noeuds retrouve les noeuds doublement cerclés.

V.4. UNE TECHNIQUE POUR INTERPRETER DES RESULTATS DE SIMULATION EN SSH

Plaçons-nous dans l'hypothèse où l'instruction DUREE est toujours utilisée avec un seul paramètre, et où l'instruction CHANGER VITESSE n'existe pas, c'est-à-dire sans application d'une fonction de reprise si une activité est suspendue pendant l'exécution d'une telle instruction DUREE.

L'algorithme de simulation du temps est exposé en VI.

Son détail ne nous importe peu ici.

Par contre, si l'on examine la nature des opérations qu'il met en oeuvre, on constate que :

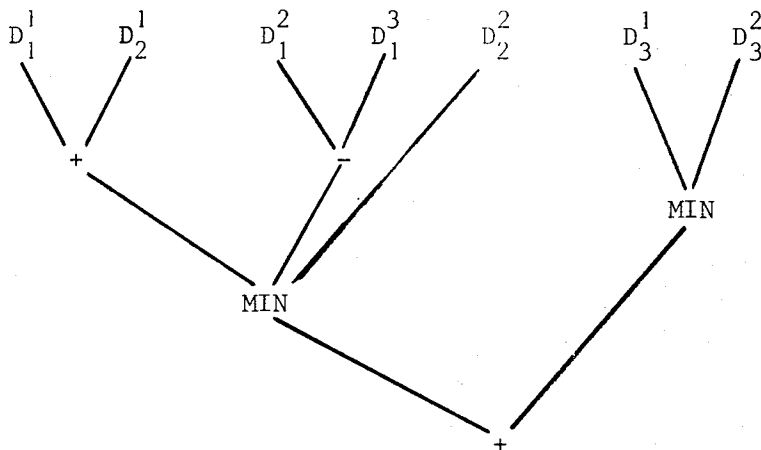
- a) elles se limitent à :
 - des affectations, des soustractions, des additions et à l'opération MINIMUM,
- b) les seules valeurs initiales intervenant explicitement dans les calculs sont les valeurs des paramètres des instructions DUREE.

Donc, à toute valeur exprimant un temps simulé, on peut associer une expression arithmétique qui a servi à son calcul et qui appartient à l'ensemble des expressions que l'on peut construire en utilisant uniquement les opérateurs +, -, MIN, les autres paramètres pouvant intervenir dans les différents algorithmes étant considérés comme figés.

Indiquons par i les instructions DUREE du modèle et soit D_i^j la valeur du paramètre de l'instruction DUREE numéro i à sa $j^{\text{ème}}$ exécution lors d'une exécution particulière du modèle (avec des bornes évidentes pour les indices j).

Alors, toute valeur représentant un temps calculée par le simulateur lors de cette exécution du modèle peut être mise sous la forme d'une expression arithmétique dont les seules variables sont les D_i^j , chacune figurant une fois et une seule, et dont les seuls opérateurs sont +, - et MIN.

Exemple



$$\text{MIN}(D_1^1 + D_2^1, D_1^2 - D_1^3, D_2^2) + \text{MIN}(D_3^1, D_3^2)$$

Faisons l'hypothèse simplificative suivante :

$$D_i^j = \text{constante } \forall j$$

(les paramètres des instructions DUREE sont des constantes).

On notera D_i le paramètre constant de la $i^{\text{ème}}$ instruction DUREE.

Alors, toute valeur temporelle est une expression du type précédent dont les variables sont les D_i , avec répétition possible.

Plaçons-nous maintenant dans l'espace vectoriel dont la base est constituée par l'ensemble des vecteurs :

$$E_i = (E_i^j) \text{ tels que}$$

$$E_i^j \neq 0 \text{ si } i \neq j \text{ et } D_i \text{ si } i = j.$$

Considérons l'expression obtenue à partir de l'initiale en remplaçant :

- D_i par E_i, \mathbf{V}_i
- les opérations +, - sur les scalaires par les +, - de l'espace vectoriel,
- le MIN sur les scalaires par un MIN sur les vecteurs tel que :

$$\text{MIN}(V1, V2) = V1 \text{ si } V1.D \leq V2.D$$

$$= V2 \text{ sinon.}$$

où $D = (D_i)$ et où $.$ désigne le produit scalaire.

Soit A le vecteur résultat de cette expression vectorielle.

Il est clair que la valeur scalaire de l'expression initiale est égale au produit scalaire de A par le vecteur D de composantes D_i

$$V = \sum a_i D_i$$

où les a_i sont des entiers relatifs.

Il est très simple d'implémenter réellement ces trois opérateurs sur vecteurs, et donc de fournir tous les résultats temporels calculés lors d'une simulation sous cette forme.

Exemple 1 :

Si l'on veut savoir pendant combien de temps un noeud a été actif, il suffit de calculer le vecteur suivant :

$$T = \sum_j T_j^2 - T_j^1$$

où T_j^1 est le vecteur représentant l'instant de début de la $j^{\text{ème}}$ période d'activité et T_j^2 l'instant de fin de cette période.

Exemple 2 :

Si l'on veut calculer la longueur moyenne d'une file d'attente, on calcule d'abord comme dans l'exemple 1 :

T_i = le temps vectoriel pendant lequel la file a eu la longueur i , la longueur moyenne est alors obtenue par :

$$L = \frac{(\sum i T_i).D}{(\sum T_i).D} \quad (\text{où } . \text{ désigne le produit scalaire}).$$

Finalement, la longueur moyenne peut être mise sous la forme :

$$L = \frac{\sum \alpha_i D_i}{\sum \beta_i D_i}$$

Concrètement, le simulateur peut donc par exemple livrer comme résultat :

- non seulement : la longueur moyenne est 3,47,
- mais encore : ce nombre a été obtenu comme le résultat de la formule :

$$\frac{3 D_1 - 2 D_2 + D_3}{4 D_1 - D_2 - D_3}$$

C'est très intéressant, puisqu'à la place d'une valeur isolée, le simulateur nous donne une formule algébrique, une fonction, pour calculer cette valeur en fonction des durées de base qui constituent les paramètres de la simulation.

C'est évidemment trop beau pour être vrai !

Les coefficients entiers des D_i dépendent des valeurs des D_i , et si l'on recommence la simulation avec d'autres valeurs, ou bien si l'on fait des simulations plus ou moins longues avec les mêmes valeurs, on trouvera certainement une formule différente à chaque fois.

Tout n'est cependant pas perdu, car l'on peut très simplement déterminer le domaine de validité d'une telle formule, c'est-à-dire l'ensemble des N-uples D_1, \dots, D_N pour lesquels elle est valable.

Cet ensemble n'est pas vide puisqu'il comporte le N-uple qui a servi lors de l'établissement de la formule.

Une condition suffisante pour qu'un N-uple satisfasse à cette formule est que, pour chaque opérateur MIN de l'expression générale qui a conduit à la formule finale, il provoque le choix (comme minimum) du même vecteur paramètre parmi les deux paramètres de MIN.

Si l'on indice par k les opérateurs $\text{MIN}(V_1, V_2)$ utilisés lors du calcul de la formule, l'ensemble des N -Uples $D(D_i)$ constituant le domaine de validité de cette formule est donc déterminé par :

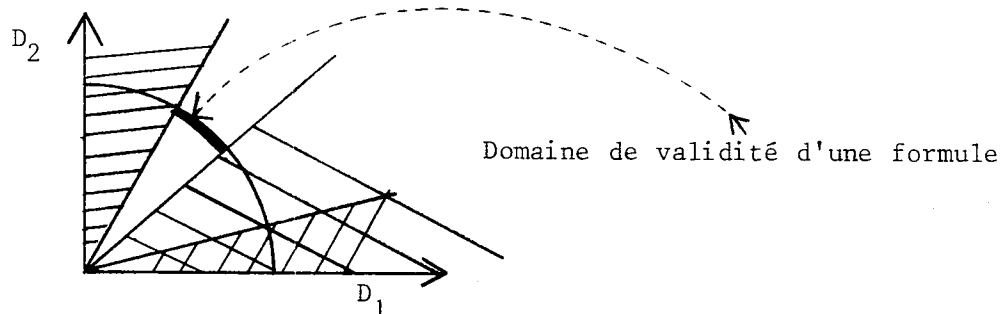
$$\mathcal{D} = \left\{ \begin{array}{l} D : S_k \times (V_1^k \cdot D) \leq S_k \times (V_2^k \cdot D) \\ \text{où } S_k = 1 \text{ si } V_1^k \cdot D_0 \leq V_2^k \cdot D \\ \text{et } S_k = -1 \text{ sinon.} \end{array} \right\} \quad v_k$$

On peut alors dire que la formule trouvée s'applique pour tout $D \in \mathcal{D}$, mais seulement pour une simulation de même durée.

Remarquons que \mathcal{D} est composé des intersections de demi-espaces délimités par des hyperplans passant par l'origine $(0, \dots, 0)$ dans l'espace à N dimensions.

On peut par conséquent le visualiser par une portion de surface de l'hypersphère de rayon 1 centrée à l'origine, et délimité par les intersections de ces hyperplans avec la sphère.

Exemple en dimension 2



Dans la pratique, tout dépend évidemment de la taille de ce domaine. Néanmoins, ceci démontre que, pour une simulation donnée, il est possible de généraliser localement le résultat ponctuel obtenu.

Autres applications possibles :

- La surface de \mathcal{D} peut être utilisée comme test d'arrêt de la simulation. On s'arrête dès que $\mathcal{D} \leq \epsilon$.

- Réciproquement, lorsqu'une simulation est terminée, la surface de \mathcal{J} donne une idée de la précision du résultat obtenu.

- Au cours d'une simulation donnée, on peut étudier la convergence du système vers un état d'équilibre non pas simplement en recherchant une éventuelle stabilisation de valeurs scalaires, mais plutôt en recherchant la stabilisation de la formule qui lui est associée, ou bien la stabilisation de \mathcal{J} .

- Si l'on dérive la formule trouvée par rapport aux D_i , on obtient une mesure de la sensibilité du modèle.

N.B. : Une analyse équivalente, mais plus compliquée, peut être faite en considérant non seulement les paramètres D des instructions DUREE(D), mais aussi les paramètres V des instructions CHANGER.VITESSE(N,V).

VI SIXIEME PARTIE

TECHNIQUE D'IMPLEMENTATION DE SSH

Cette partie expose les principaux points de l'algorithme d'implémentation de SSH, et en particulier, l'algorithme de simulation du temps, et celui de simulation des inductions.

De plus, le programme complet de SSH est donné dans un langage P.L. qui est lui-même défini en I.5.1.

Ce langage a été conçu avec le seul but de rendre sa conversion dans un langage usuel de programmation la plus simple et la plus rapide possible.

Il serait inutile de paraphraser la totalité de cet algorithme. Nous pensons que le meilleur moyen de le comprendre est de lire le programme en PL après avoir lu les chapitres qui suivent.

VI.1. L'ALGORITHME DE SIMULATION DU TEMPS

Rappelons que le temps, dans SSH, est une notion rattachée uniquement aux algorithmes d'activité des noeuds simples :

- l'activité d'un noeud simple progresse dans le temps si et seulement si le noeud est actif,
- sa progression est réglée par les instructions DUREE,
- l'écoulement du temps peut être modifié par l'instruction CHANGER.VITESSE.

Il existe donc quatre phénomènes qui concernent l'algorithme de simulation du temps :

- un noeud simple devient actif,
- un noeud simple devient inactif,
- une instruction DUREE est exécutée,
- une instruction CHANGER.VITESSE est exécutée.

VI.1.1. STRUCTURES DE DONNEES DE L'ALGORITHME DE SIMULATION DU TEMPS

A tout noeud simple, on associe 7 variables et une liste :

Les variables sont :

HEURE : elle va cumuler le temps passé dans l'état actif par le noeud. Il s'agit en quelque sorte de l' "heure relative" du noeud, par opposition à l' "heure absolue", qui est définie comme la variable HEURE attachée à un noeud qui serait toujours actif.

AHA : (Pour Ancienne Heure Absolue)
Elle mémorise la valeur qu'avait l'heure absolue la dernière fois que HEURE a été mise à jour.

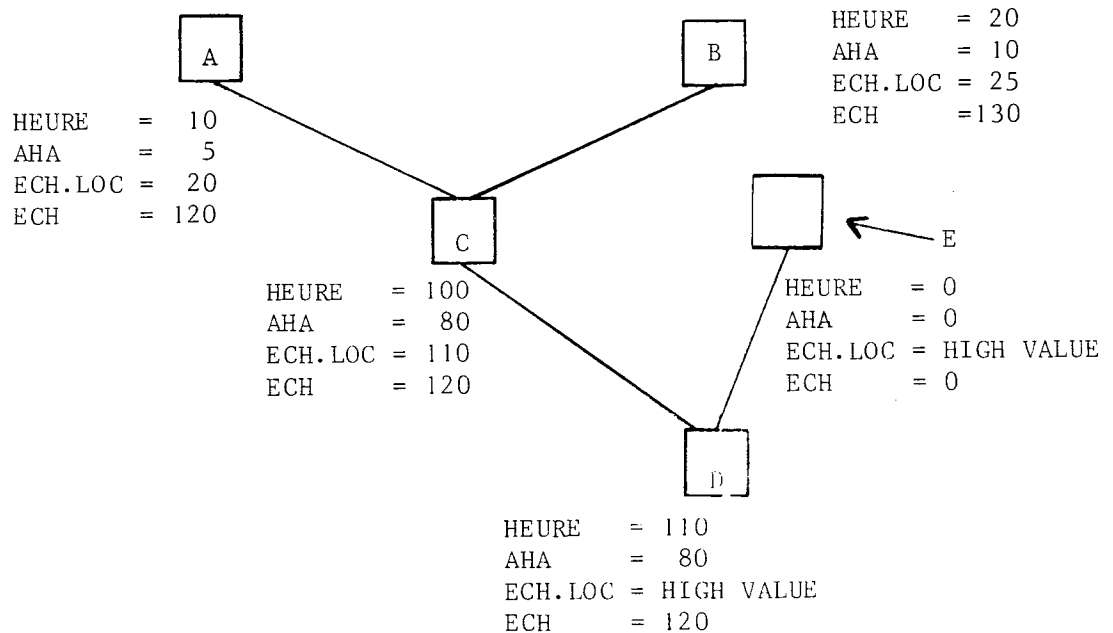
ECH.LOC : (Pour ECHéance LOcale)

Elle indique à quelle heure (exprimée en heure relative du noeud) doit se terminer la simulation de l'instruction DUREE en cours dans l'activité du noeud (si elle existe). C'est donc aussi l'heure relative du noeud à laquelle il faudra exécuter l'instruction suivant DUREE.

ECH : (Pour ECHéance)

Elle indique à quelle heure (exprimée en heure relative du père du noeud) doit se terminer la simulation d'une instruction DUREE en cours, et située soit dans l'activité du noeud lui-même, soit dans l'activité d'un de ses descendants.

Exemple :



VL : Désigne la vitesse locale du noeud telle qu'elle a été définie en IV.3.3.

VG : Désigne la vitesse globale du noeud telle qu'elle a été définie en IV.3.4.

FR : Désigne la fonction de reprise courante du noeud telle qu'elle a été définie en IV.3.1.8.

La liste est désignéé par

LISTASS : Elle contient la liste des références à tous les fils associés par le noeud, classées suivant les ECH de ces fils croissantes.

Elle n'a aucun rapport avec les listes de fils que les noeuds peuvent créer eux-mêmes dans leurs algorithmes d'archivage.

VI.1.2. L'ALGORITHME DE SIMULATION DU TEMPS EST ALORS TEL QUE : IL DOIT A TOUT INSTANT MAINTENIR VRAIES LES ASSERTIONS SUIVANTES

Assertion 1

La liste LISTASS est ordonnée suivant les ECH croissantes des noeuds qu'elle référence. Donc, il va réordonner la liste chaque fois que s'y ajoute un fils associé, (ou qu'il s'en va) et chaque fois que ECH d'un fils de la liste varie.

Assertion 2

ECH d'un noeud est définie par :

$$\begin{aligned} \text{ECH de N} &\leftarrow \text{HEURE DE PERE DE N} \\ &+ ((\text{MIN}(\text{ECH.LOC DE N}, \text{ECH DE PREMIER DE LISTASS DEN}) \\ &\quad - \text{HEURE DE N}) \\ &\quad \div \text{VG DE N}) \end{aligned}$$

Pour cela, chaque fois que l'un des constituants de cette formule variera, on la recalculera.

Si la nouvelle valeur calculée est différente de l'ancienne, cela provoquera donc récursivement le réordonnement de la liste du père de N, (s'il existe) et, si ECH du premier élément de cette liste s'en trouve affecté, le calcul de la nouvelle ECH du père de N et ainsi de suite.

Assertion 3

HEURE doit cumuler le temps passé par le noeud dans l'état actif.

On peut formaliser cela ainsi, en considérant que la vitesse globale est nulle si et seulement si le noeud est actif.:

HEURE relative de N à l'instant absolu t :

$$\sum_0^t VG(t)\Delta t \quad \text{où } t \text{ est l'heure absolue}$$

VI.1.3. D'OU L'ALGORITHME DE MISE A JOUR DE HEURE DE N

- Lorsqu'une DUREE se termine dans l'activité de N
HEURE DE N ← ECH.LOC DE N
- Lorsque le noeud devient inactif ou si sa vitesse varie
HEURE DE N ← HEURE DE N
+ ((Heure absolue - AHA DE N) x VG DE N)
AHA DE N ← Heure absolue
N.B. VG désigne l'ancienne vitesse

Remarques importantes

Cette formule de mise à jour peut en fait être exécutée n'importe quand, en dehors de l'occurrence de tout événement.

- L'heure d'un noeud doit également être mise à jour dans un cas moins évident, : chaque fois qu'un fils du noeud devient actif. En effet, à un instant donné, si un noeud commence à exécuter une DUREE, son HEURE ne sera pas mise à jour si aucun événement ne le concernant se produit. Elle ne le sera qu'à la fin de DUREE. Si pendant ce temps, on a besoin de la valeur de l'HEURE, si on ne la met pas à jour, on aura l'ancienne valeur, c'est-à-dire une mauvaise valeur.
En particulier, puisque lorsqu'un fils devient actif, il faut calculer sa nouvelle ECH, et que ce calcul fait intervenir HEURE de son père, il faut alors d'abord mettre à jour cette heure.

VI.1.4. AUTRE IMPLEMENTATION POSSIBLE

Cet algorithme constituerait un exemple très positif en faveur de la technique d'interprétation des langages, dite technique de l'appel par nécessité : KAHN [16]

- chaque variable est caractérisée par une fonction unique qui la calcule (cf. les assertions précédentes),

- chaque fois qu'on a besoin de la valeur de cette variable, on appelle cette fonction. Peut-être la valeur rendue sera-t-elle la même qu'à l'accès précédent il n'est pas besoin de s'en préoccuper, pas plus qu'il n'est nécessaire de mettre à jour la variable chaque fois qu'un des paramètres dont elle dépend change.

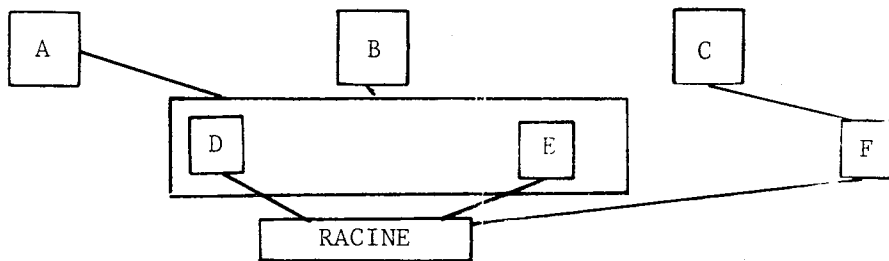
Dans notre cas précis, nous avons d'abord mis au point l'algorithme de simulation du temps intuitivement... et expérimentalement avons constaté qu'il marchait.

L'analyse ultérieure faite au moyen de cette technique, que l'on pourrait appeler Langage à Assertion Unique, a retrouvé cet algorithme.

VI.1.5. ORGANISATION GLOBALE DE LA GESTION DU TEMPS

On introduit un noeud supplémentaire, appelé RACINE, inconnu de l'utilisateur, qui est père de tous les noeuds simples terminaux, et qui a les propriétés suivantes :

- il est toujours actif,
- il associe tous ses fils candidats.



Le noeud supplémentaire RACINE.

Puisque RACINE est toujours actif, HEURE DE RACINE est l'heure absolue.

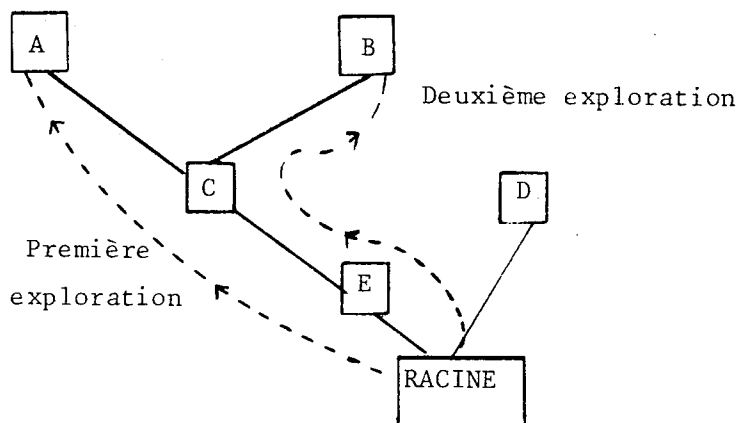
Le noeud ayant la plus proche échéance est simple à déterminer :

Partant de RACINE, on choisit le premier noeud (N1) de sa liste d'échéances. Si l'échéance locale de ce noeud est plus petite que l'échéance du premier noeud N2 de sa liste de fils, on exécute son programme d'activité. Sinon, on poursuit l'exploration en remontant dans l'arbre, N2 jouant le rôle de N1, jusqu'à trouver un noeud n'ayant pas de fils, ou ayant une échéance locale inférieure aux échéances de ses fils.

Après l'exécution du programme d'activité de ce noeud, on repart d'en-bas à partir de RACINE.

Remarque :

Cet algorithme est tel que, si deux noeuds A et B ont la même échéance, on ne les traite pas au cours de la même exploration de l'arbre, mais dans deux explorations consécutives.



Ceci peut apparaître comme peu optimisé. En fait, la solution qui permettrait, au niveau de C, d'exécuter A, puis B sans redescendre à RACINE puis remonter à C, impose la gestion d'une pile ou l'utilisation de la récursivité.

Elle ne serait avantageuse que pour des systèmes où la probabilité est grande que plusieurs événements se déroulent au même instant du temps simulé.

En général, il s'agit de systèmes synchrones qui :

- d'une part, ne sont pas très hiérarchisés, donc les mouvements inutiles dans l'arbre sont d'ampleur limitée,
- d'autre part, doivent être précisément modélisés en SSH en tenant compte de leur synchronisme : si plusieurs processeurs ont une horloge commune, il faudra traduire cela dans la hiérarchie elle-même.

N.B. : A la différence des techniques classiques de simulation à échéancier unique, où tous les événements futurs sont ordonnés dans une seule liste, cet algorithme utilise un arbre d'échéanciers. L'ordonnement des événements est donc plus rapide, et, dans le meilleur des cas, il est de l'ordre de logarithme de n, au lieu de l'ordre de n, si n est le nombre d'événements en attente de simulation à un instant donné.

VI.2. L'ALGORITHME DE SIMULATION DES INDUCTIONS

Il s'agit d'implémenter les instructions de synchronisation, et en particulier les inductions qui font que l'exécution d'une instruction peut en provoquer plusieurs autres.

Nous ne nous étendrons pas sur le fait qu'une instruction de synchronisation provoque un événement, et que cet événement entraîne l'appel des algorithmes d'archivage et de décision du noeud MOI de cet événement. Ces algorithmes pouvant à leur tour émettre des synchronisations.

Ceci constitue les inductions explicites, en ce sens qu'elles sont programmées par l'utilisateur, et leur implémentation est triviale. Notons que ce sont toutes des inductions directes (des fils vers les pères).

Intéressons-nous plutôt aux inductions implicites, c'est-à-dire à celles qui sont à la charge de SSH et non à celle de l'utilisateur.

On peut en distinguer de deux sortes :

- l'implémentation de l'instruction CONFIE.A
- les inductions dans les hiérarchies de noeuds multiples imbriqués.
- les inductions inverses, qui en SSH sont toutes implicites, et

qui comprennent :

- a) l'induction des états actif, inactif dans l'arbre des noeuds simples et sa généralisation,
- b) l'induction des variations des vitesses globales.

VI.2.1. L'IMPLEMENTATION DE L'INSTRUCTION CONFIE.A

Rappelons que si dans l'entité ALGORITHMES de la déclaration d'un noeud, un paramètre est spécifié par CONFIE.A(P,F,ACT), cela signifie que tout appel de l'algorithme correspondant à ce paramètre se traduit par la candidature auprès du noeud P d'une copie de F avec l'activité ACT.

Ceci est réalisé très simplement sans modification de l'algorithme général : dans le descripteur du noeud, à la place du nom de l'algorithme "normal" spécifié par l'utilisateur, on met le nom d'un algorithme système qui va :

- faire une copie de F,
- mémoriser dans cette copie l'événement qui a provoqué son appel,
- exécuter ENVOI(P,copie de F)

Plus tard, chaque fois que SSH appellera ACT sur la copie de F pour la faire progresser, il préparera normalement la variable SUITE, comme pour toute activité, mais :

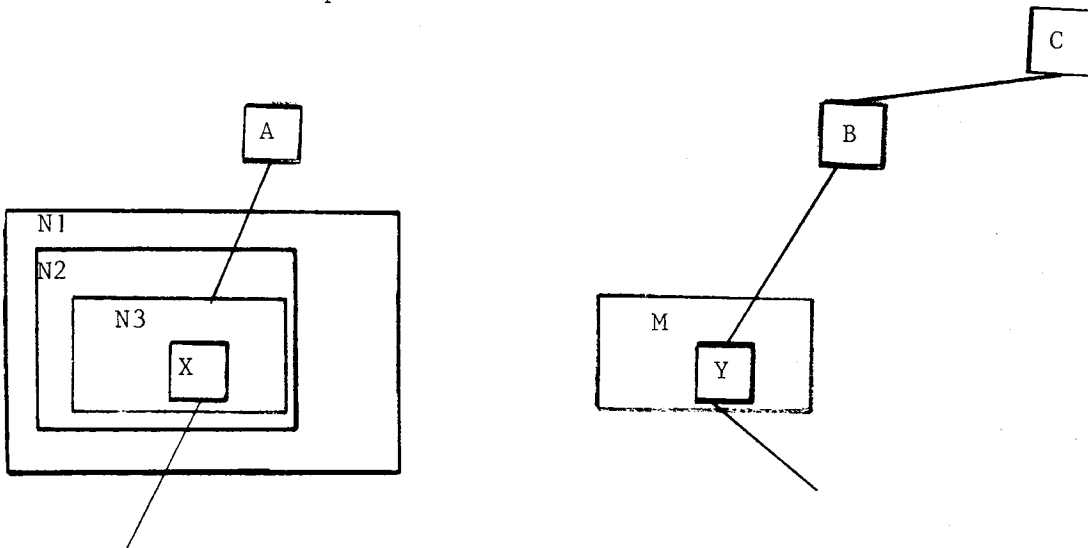
- a) il ne chargera pas MOI avec le nom de la copie de F mais avec le nom du noeud qui avait confié son algorithme,
- b) de plus, il rendra accessibles les variables LUI, MOTIF, et, suivant le cas ACTION, ORIGINE, au sous-programme d'activité, variables qui décrivent l'événement initial et qui avaient été stockées au départ dans la copie de F.

Ainsi, un sous-programme comme ACT déclaré en troisième paramètre de CONFIE.A cumule les caractéristiques des algorithmes d'archivage ou de décision et des algorithmes d'activité.

VI.2.2. LES INDUCTIONS DANS LES HIERARCHIES DE NOEUDS MULTIPLES

A tout noeud simple, on fait correspondre une liste dite "liste de tuteurs" dont les éléments contiennent les noms des noeuds multiples desquels le noeud est candidat ; cette liste est gérée en LIFO pour des raisons évidentes.

Exemple :



La liste des tuteurs de C est vide.

Celle de B contient M, celle de A contient N3, N2, N1.

D'autre part, à chaque noeud simple, on associe une variable TC, (pour tuteur-courant) qui désigne lequel des tuteurs est actuellement averti d'un événement concernant le noeud.

Implémentation des différentes inductions

VI.2.2.1. Instruction AFFECTER(C,F)

Si C est multiple, on l'ajoute à la liste des tuteurs et $TC \leftarrow C$

VI.2.2.2. Instruction DESAFFECTER(F)

Par construction, TC désigne l'exécutant de cette instruction.

Alors on envoie l'événement indiquant que F s'en va à tous les noeuds à qui F a été affecté au-delà de TC.

VI.2.2.3. Induction inverse (avec MOTIF = ILTOURNE, ILSTOPPE, ILVARIE)

Cette induction est transmise uniquement au premier élément de la liste des tuteurs, c'est-à-dire au dernier noeud multiple auquel le fils a été affecté ; et, de plus, TC prend le nom de ce noeud.

Par la suite, les noeuds de la liste des tuteurs peuvent se "repasser" l'événement au moyen des instructions :

SUR-TRAITER

si le suivant de TC existe, alors TC \leftarrow suivant de TC, et envoyer l'événement à TC ; et retourner la valeur VRAI sinon retourner la valeur FAUX.

SOUS-TRAITER

même chose avec "précédent" à la place de "suivant".

VI.2.3. L'INDUCTION INVERSE DES ETATS ACTIF/INACTIF ET DES VARIATIONS DE VITESSE DANS L'ARBRE DES NOEUDS SIMPLES

Elle est réglée par les 4 algorithmes récurrents suivants :

Si un noeud simple quitte l'état actif, tous ses fils associés (donc dans l'état actif), passent à l'état inactif : propagation vers les feuilles de la transition actif \rightarrow inactif.

Si un noeud simple passe de inactif à actif alors tous ses fils associés subissent la même transition.

Si un noeud simple change de vitesse locale, alors : on recalcule sa vitesse globale comme étant le produit de sa nouvelle vitesse locale par l'actuelle vitesse globale de son père.

Si un noeud simple change de vitesse globale, alors on recalcule la vitesse globale de tous ses fils associés comme étant le produit de leur vitesse locale actuelle par la nouvelle vitesse globale du noeud.

VI.3. LA PROGRAMMATION PRATIQUE DE L'ALGORITHME D'INDUCTION

Les pages précédentes ont décrit l'algorithme d'induction.

Voyons maintenant comment le programmer.

Ce n'est pas tout à fait trivial.

Il s'agit, dans un langage de programmation classique, de réaliser des enchaînements d'exécution de groupes d'instructions correspondant à un événement, et ce de manière dynamique, c'est-à-dire que chaque groupe calcule le ou les groupes d'instructions suivantes à exécuter.

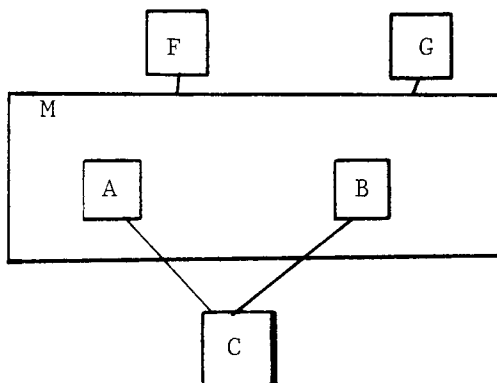
VI.3.1. PREMIERE SOLUTION : L'APPEL PROCEDURAL RECURSIF

Les instructions SSH, les algorithmes associés aux noeuds sont des procédures récursives qui s'appellent les unes les autres.

C'est une solution simple, car tout est à la charge du langage hôte, mais :

- la récursivité est coûteuse en temps et en place, et elle est peu accessible en dehors des centres universitaires,
- elle n'est pas convenable dans le cas où un algorithme d'un noeud donné peut indirectement s'appeler récursivement sur ce même noeud.

Exemple :



L'algorithme de décision de M est appelé et exécute :

AFFECTER(A,F)

ce qui peut induire entre autres :

VIVRE exécuté par A

DISSOCIER(B) exécuté par C

et enfin, l'appel de décision de M.

avec l'événement MOI=M ; LUI=G ; MOTIF = ILSTOPPE.

Ainsi, l'algorithme de décision de M s'appelle lui-même récursivement et reçoit un événement en conséquence d'une action qu'il n'a pas encore terminée.

Une telle récursivité n'a évidemment aucun sens, et il est nécessaire de l'interdire.

Pour cela, il faut introduire un mécanisme assez lourd qui met en attente les appels récursifs d'un algorithme sur un noeud donné, et qui, à la fin d'un appel, exécute les appels en attente.

C'est cette solution qui a été implémentée dans une version de SSH en AFD.

N.B. : Il est facile de voir que ces précautions sont inutiles si le modèle ne comporte pas de noeuds multiples.

VI.3.2. SECONDE SOLUTION : IMPLEMENTER UNE "RECURSIVITE FAIBLE"

L'inconvénient de la solution précédente est que si un algorithme A exécute les instructions I1, I2, les choses se déroulent ainsi :

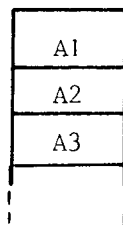
- 1) Appel de A
- 2) Appel de I1, et exécution de toutes les conséquences de I1,
- 3) Appel de I2, et exécution de toutes les conséquences de I2,
et alors que les conséquences de I1 peuvent déjà se faire sentir sur cet appel,
- 4) Fin de A.

Il serait plus naturel d'avoir l'ordre suivant :

- 1) Appel de A
- 2) Appel de I1 seul
- 3) Appel de I2 seul
- 4) Fin de A
- 5) Exécution de toutes les conséquences de I1
- 6) Exécution de toutes les conséquences de I2.

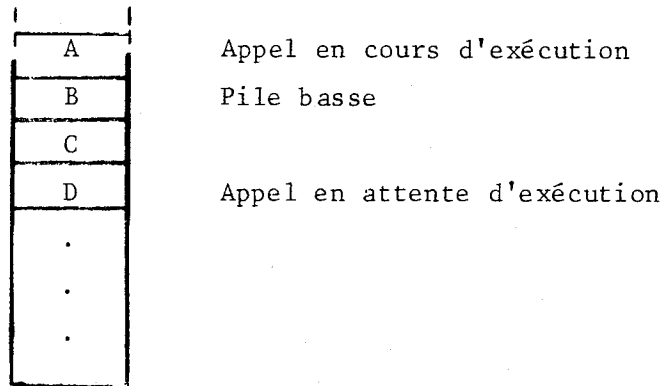
Ainsi, si une des conséquences de I1 est l'appel récursif de A, ce second appel aura lieu après la fin du premier et non au milieu comme dans la première solution.

Ceci est facile à implémenter avec deux piles opposées de mémorisation des appels :



Pile haute

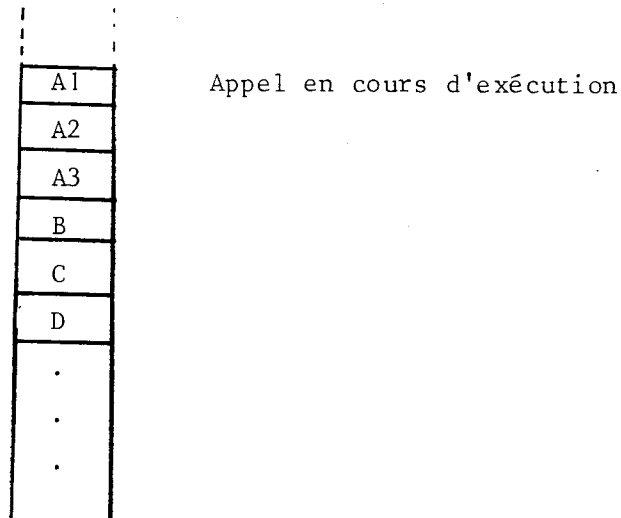
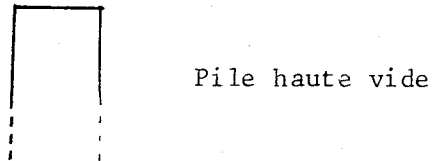
Appels émis par l'appel en cours d'exécution



Quand l'appel en cours d'exécution est terminé, on l'enlève de la pile basse, et on verse le contenu de la pile haute dans la pile basse.

Ainsi, si A a appelé successivement A1, A2, A3 au cours de son exécution, ils sont empilés dans la pile haute comme sur la figure ci-dessus.

A la fin de l'exécution de A, on se trouvera dans la situation suivante :



et l'on exécute A1, et ainsi de suite.

N.B. : Cette technique de programmation, en dehors du fait qu'elle convient pour SSH, a les avantages suivants :

- elle permet une "récursivité faible" dans des langages n'ayant pas la facilité des procédures récursives,

- les restrictions qu'elle impose, par rapport à la récursivité générale, sont en fait très "saines", dans le sens d'une meilleure modularité de la programmation :

- a) appels par valeur,

- b) dans une certaine mesure, suppression de la séquentialité habituelle :

si A appelle A1, puis A2, puis se termine, le programmeur doit raisonner comme si les appels de A, A1, A2 se déroulaient en parallèle et en toute indépendance, puisque l'exécution ne suit plus immédiatement l'appel.

En quelque sorte, cette récursivité est simplement la récursivité des appels, plutôt que celle des exécutions.

Cette solution a été implémentée dans une version de SSH en FORTRAN.

VI.3.3. TROISIEME SOLUTION : IMPLEMENTATION "AD-HOC" POUR SSH

En affinant l'algorithme tel qu'il était implémenté dans SSH-FORTRAN, il a été constaté qu'il suffisait en fait de remplacer, dans les deux piles opposées, tous les appels de procédures par les descripteurs des événements générés par l'exécution du modèle.

De plus, l'implémentation du mode D devient alors triviale, car il suffit presque de remplacer la pile haute par le paquet courant.

C'est cette solution qui est proposée dans l'algorithme en P.L. décrit à la fin de l'annexe.

VII SEPTIEME PARTIE

- RECAPITULATIF
- PERSPECTIVES
- CONCLUSION

VII.0 RECAPITULATIF

Nous avons présenté un moyen permettant à un utilisateur d'un langage usuel de programmation :

1) de déclarer une arborescence de noeuds simples, auxquels sont attachés des activités séquentielles se déroulant dans le temps simulé. L'activité d'un noeud se déroule si et seulement si le noeud est candidat auprès de son père et si ce dernier l'accepte. Durant son activité, un noeud N1 peut rendre candidat un noeud N2 vers un troisième N3.

Pour chaque noeud, l'utilisateur décrit ses algorithmes d'activité, de candidature et d'acceptation.

2) de regrouper récursivement des ensembles de noeuds simples en noeuds multiples, dont le rôle est :

- a) de transmettre les demandes de candidature qu'il reçoit de ses fils simples vers un des noeuds simples ou multiples qui le composent,
- b) de contrôler la progression des demandes ainsi transmises.

Des facilités annexes mais très utiles en pratique ont été ajoutées à ces principes de base.

1) La possibilité de matérialiser l'exécution d'un algorithme de gestion d'un noeud N1 sous la forme de l'activité - consommatrice de temps - d'un autre noeud N2,

2) L'introduction de la notion de variation de la vitesse de traitement d'un noeud - qui se répercute sur tous ses descendants - et permet de remplacer de coûteuses simulations de mécanismes détaillés par une simple simulation de leur effet moyen.

Un ensemble de noeuds standard a été défini en détail. Il permet de modéliser avec une grande concision des architectures de systèmes courants, comme le prouvent les exemples donnés dans ce travail.

Un algorithme d'implémentation est donné. Il est relativement court et efficace, et facile à implémenter dans des langages divers (Assembleur ou Langage évolué).

VII.1. DEVELOPPEMENTS ENVISAGES POUR SSH

VII.1.1. AJOUT DE FACILITES DE RECUEIL DE RESULTATS PENDANT LA SIMULATION

VII.1.1.1. Il s'agit d'introduire des instructions de cumul, moyennes, histogrammes, statistiques, de telle façon qu'elles utilisent au mieux les propriétés des objets et relations de SSH.

Exemple 1

"Cumuler le temps pendant lequel le noeud N a été actif sur les pères P1 ou P2".

Exemple 2

"Faire l'histogramme du nombre de fils associés par N, lorsque N est lui-même associé par P".

N.B. : Toutes ces mesures peuvent en particulier utiliser la représentation vectorielle du temps introduite en V.4, et donner leurs résultats sous forme de formules symboliques vérifiées localement.

Exemple 3

On peut introduire simplement des mesures conditionnelles en utilisant des mécanismes préexistants de SSH :

"longueur moyenne du nombre de fils associés de A, quand B est actif".

Il suffit d'introduire un noeud fils espion du temps de B comme cela a été défini en V.2.2.1 dont l'activité est de mesurer (par échantillonnage par exemple) le nombre demandé ;

Puisque, par construction, l'espion est actif en même temps que B, il réalisera bien la mesure conditionnelle demandée.

VII.1.1.2. Introduction de variables de type "mesurable"

Pour étendre la portée des instructions de mesure suggérées ci-dessus, et ne pas les limiter aux objets prédéfinis par SSH (noeuds, relations), il faut donner à l'utilisateur un moyen de déclarer comme devant être mesurée, n'importe quelle variable, utilisée par exemple dans l'algorithme d'un noeud.

Exemple :

"Déclarer X entier mesurable"

"Faire l'histogramme de Valeur Entière de $\text{Log}_2(X)$ "

ou bien : tenir à jour à tout instant dans le tableau T l'histogramme de X :

$T[i]$ = Temps pendant lequel $\text{Log}_2(X)$ a été égal à i.

VII.1.2. INTRODUIRE LA NOTION DE SIMULATION INTERNE A UN SYSTEME

Reprenant le dernier exemple, il est clair que le tableau T peut être utilisé avec profit par un algorithme d'archivage ou de décision d'un noeud, pour optimiser le fonctionnement du système.

Par exemple, à partir des mesures disponibles à un instant donné, on peut faire une régulation du système, comme cela est fait en automatique.

On trouve un tel exemple d'application dans BADEL et al. [18].

Ceci, comme en automatique, suppose :

- l'existence d'un modèle du système, accessible à l'algorithme qui veut réguler,
- l'existence d'un interpréteur de ce modèle, nécessaire dans le calcul des commandes de régulation.

A partir de cela, nous suggérons d'abord deux choses :

- a) le modèle peut être un modèle de simulation plutôt qu'un modèle mathématique,
- b) son interpréteur peut donc être un algorithme de simulation comme SSH.

Ensuite, on peut faire deux extensions fondamentales.

a) le modèle de simulation n'est pas monolithique et centralisé, mais chaque noeud du système réalise localement sa propre simulation, et, à un instant donné, il existe, dans un système hiérarchisé des activités "réelles" de la hiérarchie et des activités "simulées" de manière totalement décentralisée.

b) jusqu'à présent, on a considéré des simulations donnant des résultats numériques, pour améliorer des performances, en considérant le système informatique comme un système continu.

Nous conjecturons que la même démarche : simulation interne et décentralisée peut s'appliquer lorsqu'on passe du domaine continu au domaine discret, dans le sens où l'activité du système est vue comme un ensemble d'événements, (des instructions de SSH par exemple).

Alors, de même que le modèle réparti de simulation continue permet d'optimiser les performances (exprimées par des variables continues), de même le modèle réparti de simulation événementielle permet d'assurer une grande partie de la gestion du fonctionnement exact du système (exprimé par des événements discrets).

En particulier, le problème de prévention des interblocages pourrait être résolu de cette manière. D'ailleurs, si l'on examine bien l'algorithme classique du Banquier HABERMAN [8] proposé pour ce problème, il est facile de l'interpréter comme un algorithme de simulation :

On simule plusieurs futurs du système, chacun étant caractérisé par une suite d'exécutions de tâches, jusqu'à ce qu'on trouve une suite dite "saine" qui garantit une situation sans blocage.

L'utilisation générale de la simulation événementielle permettrait peut-être de résoudre des problèmes plus généraux que celui du Banquier (qui est centralisé, et n'autorise que des ressources indépendantes et non hiérarchisées).

VII.2. DEVELOPPEMENTS D'APPLICATIONS DE QUELQUES ASPECTS DE LA THEORIE DES SYSTEMES HIERARCHISES

VII.2.1. LA TSH EN TANT QUE RESEAU RENSEIGNE

Nous pensons que la TSH, d'un point de vue purement formel, a des caractéristiques analogues des systèmes comme la théorie des attributs sémantiques KNUTH [19] et la théorie des graphes fonctionnels.

Dans tous les cas, il s'agit de réseaux de relations entre des objets. A chaque objet sont associés des renseignements. Toute variation d'un renseignement dans un objet induit éventuellement des variations dans les renseignements des objets voisins dans le réseau.

Dans la théorie des attributs sémantiques, ces variations sont "statiques" si l'on peut dire, en ce sens qu'il s'agit simplement de dépendances dans un réseau invariant.

Dans les graphes fonctionnels et la TSH, elles sont dynamiques. Ce sont les inductions de la TSH.

Il y a certainement beaucoup à faire dans le domaine suivant : "Quels doivent être les renseignements associés aux objets, et quelles sont les règles de dépendance entre ces objets, pour représenter le plus grand nombre des aspects d'un système informatique".

La TSH s'est concentrée sur les problèmes de synchronisation des activités ; il serait intéressant de voir comment on peut l'enrichir en traitant de la même manière les questions liées à la désignation, aux performances, à la fiabilité, aux accès, etc...

VII.2.2. Exploitation d'un apport essentiel de la TSH : la hiérarchie de la synchronisation

La TSH permet de résoudre élégamment, en "faisant un dessin", de nombreux problèmes de synchronisation, qui, exprimés en termes de sémaphores, sont dans le pire des cas insolubles, et dans le meilleur, incompréhensibles.

On assiste actuellement à une certaine effervescence de publications sur le sujet "synchronisation structurée" qui veulent être aux P,V ce que le DO.WHILE et IF THEN ELSE sont au GOTO CAMPBELL, HABERMAN [20].

Ces nouveautés sont présentées en général sous un formalisme du genre "expressions de synchronisation" $(x+z)*(z+t)$ où + exprime le parallélisme entre deux processus et * leur exclusion mutuelle.

On peut aboutir à des notations analogues en prenant comme point de départ un modèle hiérarchisé.

Considérons un modèle SSH ne comportant que des noeuds simples :

c'est un arbre. Considérons chaque feuille comme une variable et chaque autre noeud comme un opérateur.

Appelons λ un noeud déclaré DNL

Appelons π un noeud déclaré DNP

Appelons β un noeud déclaré DNB

Appelons ρ le noeud père de tous les noeuds terminaux.

Ce sont des opérateurs de priorité décroissante : $\beta, \pi, \lambda, \rho$.

Notons l'activité d'un noeud comme une expression régulière :

$N:(abv)^*de$

elle signifie que le noeud N passe n fois sur les pères abc dans cet ordre, puis sur de.

Alors le modèle de l'IBM 115 décrit en II-7 est complètement défini par l'expression :

$DISK_{\rho}EXDISK_{\pi}INTR_{\pi}INTL_{\pi}JOB1_{\pi}JOB2_{\rho}RD1_{\beta}PR1_{\rho}RD2_{\beta}PR2_{\rho}ESR:EXDISK$
 $(DISK_{\lambda}EXDISK)^*INTR_{\rho}ESL:PR_{\lambda}INTL!$

VII.3. CONCLUSION

Cette première expérience de conception, d'implémentation et d'utilisation de SSH nous conduit à tirer quelques leçons.

1) La "justesse sémantique" des concepts de la théorie des systèmes hiérarchisés. Au-delà de considérations intuitives, nous en voulons pour preuve pratique la raison suivante :

au cours de la conception de SSH, des besoins supplémentaires se sont fait sentir, qui ont conduit aux instructions CHANGER.VITESSE, CONFIE.A, MODE "D", et aux noeuds espions. Il a toujours été très facile d'intégrer ces nouveautés sans modifier les fondements de la définition et de l'implémentation des mécanismes de base des systèmes hiérarchisés. En quelque sorte, la TSH s'est avérée "accueillante", "complète" au sens algébrique du terme ; elle recouvre bien les domaines qu'elle prétend traiter.

2) Il y a un besoin à satisfaire - en matière d'outils de simulation - entre d'une part, les langages de haut niveau généraux, coûteux et peu répandus, et d'autre part, les simulateurs "ad hoc" écrits spécialement pour une application donnée, et donc non réutilisables. SSH essaie d'être une réponse intéressante à ce besoin en permettant à l'utilisateur à la fois :

- de décrire des modèles sophistiqués (ceux de la TSH),
- d'utiliser comme outil de base son langage de programmation habituel,
- de maîtriser les coûts de temps de calcul grâce à un algorithme d'implémentation efficace et simplifiable.

3) Il reste encore beaucoup à faire pour que la notion "d'architecture d'un système" d'une part, et la nécessité de l'évaluation sérieuse d'une architecture avant sa réalisation, d'autre part, soient communément admises.

L'obstacle principal est la crainte que ces phases de formalisation et d'évaluation éloignent inconsidérément la phase de réalisation et même qu'elles finissent par s'y substituer complètement dans le pire des cas.

Nous souhaitons qu'un outil simple comme SSH - précisément parce qu'il conserve un aspect certain de "rusticité" par rapport à des langages plus généraux - puisse jouer le rôle d'un modeste "cheval de Troie" et gagner quelques utilisateurs aux méthodes nouvelles de conception des systèmes.

BIBLIOGRAPHIE

- [1] ANCEAU
"Contribution à l'étude des systèmes hiérarchisés de ressources
dans l'architecture des machines informatiques"
THESE D'ETAT - INP Grenoble - Décembre 1974.
- [2] ANCEAU - FORTIER - SCHOELLKOPF
"Conception descendante de machines informatiques : application
à une machine PASCAL" -
Rapport final ENSIMAG - Contrat SESORI
N° 73 042 - Novembre 1974.
- [3] ROHMER - TUSERA
"Experimental Top Down Specification of Specialized Microprocessors
Networks"
Euromicro Workshop on Microarchitecture of Computers
Nice - June 1975.
- [4] GPSS
GPSS 360
User's Manual - IBM Corp. H20 D326-0 - 1967.
- [5.1] SIMSCRIPT
IBM Systems Journal 3.1 - 1964.
- [5.2] SIMULA - DAHL et autres
"Simula 67 common base language"
Oslo computing center - April 1969.
- [6] GUIBOUD-RIBAUD
"Mécanismes d'adressage et de protection dans les systèmes informa-
tiques - Application au noyau GEMAU"
THESE D'ETAT - INP Grenoble - Juin 1975.

- [7] DIJKSTRA
"Cooperating Sequential Processes"
Prog. Lang. F. Genuys Ed. - Acad. Press - New York - 1968.
- [8] HABERMAN
"Prevention of System Deadlocks"
CACM Vol. 12 - 7 - July 1969.
- [9] ALICE - G.H. POUJOULAT
- [10] CASSANDRE - ANCEAU - LIDDEL - MERMET - PAYAN
"Third Symposium on Computer and Information Science"
COIN Miami dec. 69 - Acad. Press - Vol. 1 - p. 179.
- [11] BRUNET
"Etude d'une structure biprocesseur banalisée"
Thèse de Doct. Ing. - INP Grenoble - Mars 1974.
- [12] ELMASRY - ATTASI
"Communication orale"
LABORIA - Octobre 1975
- [13] ROHMER
"Etude et réalisation d'un simulateur de systèmes hiérarchisés"
Rapport Final Contrat CRI 72-21/I.41 - Septembre 1973.
- [14] BADEL
"Quelques problèmes liés à la simulation de modèles de systèmes informatiques"
Thèse de docteur-ingénieur
Université Paris VI - Mai 1975.
- [15] LEROUDIER - PARENT
"Quelques aspects de la modélisation par simulation à événements discrets"
Rapport LABORIA N° 98 - IRIA Février 1975.

- [16] KAHN
"Semantics of a simple language for parallel programming"
Procs. IFIPS 74 - p. 471-476.
- [17] ALGOL 68 - VAN WIJNGARTEN et al
"Report on the algorithmic language Algol 68"
Math. Centrum Amsterdam - MR 101
- [18] BADEL - GELEMBE - LEROUDIER - POTIER
"Adaptative Optimization fo time sharing systems performance"
Procs of IEEE - Vol. 63 - N° 6 - Juin 1975.
- [19] KNUTH
"Semantics of conctect-free languages"
Math. systems theory V.2.2. - 1968 - p. 127-145.
- [20] CAMPBELL - HABERMAN
"The specification of process synchronisation by path expressions"
Univ. of Newcastle upon Tyne Computing Lab.
Technical report series n° 55.

ANNEXE A

PROGRAMMATION DES NOEUDS STANDARD DE SSH

A.1. CONVENTIONS COMMUNES AUX NOEUDS SIMPLES, LAMBDA ET PRIORITAIRE
ESPACE LOCAL

Les quatre premiers éléments de leur espace local sont réservés ;
ils ont dans l'ordre les noms et les significations suivants :

PRIORITE : c'est un scalaire
 SEUIL.ACTIF : c'est un compteur
 DEGRE.MULTI : c'est un compteur
 LISTE.FILS : c'est un pointeur sur une liste à double chaînage dont
 chaque élément est composé de quatre zones dénommées ainsi :
 NEXT pointe sur l'élément suivant
 PREC sur le précédent
 NOM est une référence à un noeud
 ETAT est une information relative au noeud référence
 par NOM

La liste est initialisée et gérée de façon à ce que son premier
et son dernier élément soient ineffectifs et référencent le noeud VIDE,
ceci pour supprimer les cas particuliers en début et fin de liste.

En effet :

- il existe un noeud simple prédéclaré sous le nom de VIDE.
- Il a les propriétés suivantes :
- toute instruction SSH ayant VIDE comme l'un quelconque de ses
paramètres est ineffective.

A.2. ALGORITHMES D'ACTIVITE

Il existe une valeur standard notée ATTENDRE qui peut être utilisée
comme paramètre de l'instruction DECISION.

Algorithme de décision-père

Il a la forme suivante pour tous les noeuds simples standard :

On le désigne par DPS

SOUS-PROGRAMME DPS :

```

IF MOTIF = ILARRIVE THEN BEGIN
IF SEUIL.ACTIF DE MOI  $\leftarrow$  SEUIL.ACTIF DE MOI + 1 = 0 THEN VIVRE END
IF MOTIF = ILSENVA OR MOTIF = CESTMOI AND LUI = ATTENDRE THEN BEGIN
IF SEUIL.ACTIF DE MOI  $\leftarrow$  SEUIL.ACTIF DE MOI - 1 < 0 THEN MOURIR END
FIN SOUS-PROGRAMME DPS

```

Commentaires :

SEUIL.ACTIF a été initialisé à l'opposé d'un nombre positif dit "seuil d'activité" du noeud. On voit que le noeud se porte candidat à son père si et seulement si il a reçu lui-même un nombre de candidats supérieur ou égal à son seuil d'activité, que l'exécution de DECISION(ATTENDRE) décrémente ce nombre de candidats, et peut donc provoquer l'arrêt du noeud.

A.3. ALGORITHMES D'ARCHIVAGE

Ce sont eux qui vont gérer la liste référencée par LISTE.FILS dans l'espace local.

Voici l'algorithme d'archivage externe, appelé AXS :

SOUS-PROGRAMME AXS :

```

SI MOTIF = ILARRIVE ALORS ETAT DE AJOUTER(LISTE.FILS DE MOI,LUI)  $\leftarrow$  VIDE
SI MOTIF = ILSENVA ALORS ENLEVER(LISTE.FILS DE MOI,LUI)
FIN SOUS-PROGRAMME AXS

```

Commentaires :

Les fonctions AJOUTER, ENLEVER gèrent la liste selon les priorités décroissantes des noeuds référencés par le champ NOM de ses éléments.

Voici l'algorithme d'archivage interne, appelé AIS :

SOUS-PROGRAMME AIS

SI ACTION = ASSOCIER ALORS ETAT DE LOCALISER(LISTE.FILS DE MOI,NOM,LUI)←MOI
 SI ACTION = DISSOCIER, ALORS ETAT DE LOCALISER(LISTE.FILS DE MOI,NOM,LUI,='')
 ←VIDE

FIN SOUS-PROGRAMME AIS

Commentaires :

La fonction LOCALISER(L,C,V,F) retourne la référence du premier élément E de la liste tel que F(C DE E,V) = VRAI.

En conclusion, les éléments de la liste standard référencée par LISTE.FILS ont leur champ NOM qui désigne le nom d'un fils candidat, sont classés par PRIORITE DE NOM décroissante, et leur champ ETAT vaut VIDE si le fils n'est pas associé, et désigne le noeud MOI s'il est associé.

N.B. : Tous ces exemples sont des exemples d'utilisation de SSH. Nous ne prétendons pas que leur programmation est la meilleure possible. Elle cherche surtout à être facilement compréhensible.

A.4. LES ALGORITHMES DE DECISION-FILS STANDARD

Il en existe deux. Le premier ne dissocie jamais un fils (il attend qu'il s'arrête lui-même). Le second peut le faire éventuellement. Dans la littérature anglo-saxonne, on dit parfois "basic scheduling" pour caractériser le premier, et "general scheduling" pour caractériser le deuxième.

A.4.1. Premier algorithme, appelé DFL (Décision-Fils Lambda)

SOUS-PROGRAMME DFL :

SI MOTIF = ILARRIVE ALORS DEBUT
 SI DEGRE.MULTI DE MOI ← DEGRE.MULTI DE MOI + 1 ≤ 0
 ALORS ASSOCIER(LUI) FIN

```

SI MOTIF = ILSENVA ALORS DEBUT
DEGRE.MULTI DE MOI ← DEGRE.MULTI DE MOI - 1
ASSOCIER (NOM DE LOCALISER(LISTE.FILS DE MOI,ETAT,VIDE, '=')) FIN
FIN SOUS-PROGRAMME DFL

```

Commentaires :

DEGRE.MULTI a été initialisé à l'opposé d'un nombre positif dit "degré de multiprogrammation" du noeud. Un noeud associe tous ses fils candidats à concurrence d'un nombre égal à son degré de multiprogrammation. Lorsqu'un fils s'en va, on le remplace par le plus prioritaire des candidats non associés (obtenu par l'appel de LOCALISER).

A.4.2. Second algorithme appelé DFP (décision-fils prioritaire)

SOUS-PROGRAMME DFP

DFL

```

SI MOTIF = ILARRIVE ET DEGRE.MULTI DE MOI > 0
ET PRIORITE DE LUI > PRIORITE DE LP ← MINUS(LISTE.FILS DE MOI) ALORS DEBUT
DISSOCIER(LP)
ASSOCIER(LUI) FIN
FIN SOUS-PROGRAMME DFP

```

Commentaires :

Cet algorithme commence par appeler le précédent, DFL. Si le nouvel arrivant n'a pas été associé par DFL, et si sa priorité est plus grande que celle du moins prioritaire des fils associés (déterminé par la fonction MINUS), alors il prend sa place comme fils associé.

A.5. SOUS-PROGRAMMES SIMPLIFIES DE DECLARATION DE NOEUDS STANDARD

Maintenant que nous disposons d' "ingrédients" permettant de définir des noeuds standard, (à part leurs algorithmes d'initialisation, copie, et destruction de l'espace local, qui sont triviaux et que nous ne décrirons pas), nous allons construire des sous-programmes simplifiés pour leur déclaration.

A) Sous-programme DNL (Déclaration de noeud lambda)

Syntaxe :

A ← DNL(PERE.DE(B1, ..., BN), ACT, TAILLE, DM, SA, PRIO)

Sémantique :

FONCTION DNL(FILS=liste vide, ACT=RIEN, TAILLE=0, DM=1, SA=1, PRIO=1)

DNL ← DN(SIMPLE, FILS, ALGORITHMES(DPS, AXS, AIS, DFL, ACT)
, ESPACE.LOCAL(TAILLE+4, ,, ,))

SEUIL.ACTIF DE DNL ← SA

DEGRE.MULTI DE DNL ← -DM

PRIORITE DE DNL ← PRIO

LISTE.FILS DE DNL ← VIDE, VIDE

FIN FONCTION DNL

Commentaires

Les valeurs par défaut en cas d'absence d'un paramètre sont les suivantes :

TAILLE = 0, SA = DM PRIO = 1, FILS = liste vide , ACT = RIEN

On obtient alors une forme très concise (et très souvent suffisante d'après notre expérience).

Exemple :

A ← DNL(PERE.DE(B←DNL), ACT)

B) Sous-programme DNP (Déclaration de noeud prioritaire)

C'est le même que DNL, à l'exception de DFL qui est remplacée par DFP.

De plus, la priorité par défaut de ses fils sera dans l'ordre inverse dans leur apparition dans PERE.FILS.

A.6. PROGRAMMATION DU NOEUD STANDARD BARILLET

Le noeud barillet va tenir à jour dans son espace local une variable COURANT qu'il chargera avec la référence de l'élément de la liste des fils correspondant au fils qui est associé.

Son algorithme d'activité est :

SOUS-PROGRAMME ACTB :

E: DUREE(QUANTUM DE NOM DE COURANT DE MOI)
DECISION(QUANTUM.FINI); GOTO E; FIN ACT B

Son algorithme décision-fils est :

SOUS-PROGRAMME DFB :

SI MOTIF = CESTMOI ET LUI = QUANTUM.FINI ALORS
DISSOCIER NOM DE COURANT DE MOI
 SI MOTIF = CESTMOI ET LUI = QUANTUM.FINI OU MOTIF = ILSENVA
 ALORS
ASSOCIER(NOM DE (COURANT DE MOI ← SUIVANT(COURANT DE MOI)))
 FIN DFB

Commentaires :

La fonction

- SUIVANT trouve l'élément suivant COURANT dans la liste. La liste est gérée comme pour DNL et DNP. Quand on arrive en fin de liste, on repart au début.

N.B. : Cet algorithme a la particularité suivante :

Si un fils s'en va, alors qu'il est associé et que son quantum n'est pas épuisé, le reliquat de ce quantum va être pris comme premier quantum du fils associé suivant (Ceci sera une conséquence de la technique choisie dans SSH pour simuler le temps qui est exposée dans la partie VI).

Pour implémenter une solution plus exacte, où le premier quantum du fils suivant sera bien son quantum et non le reliquat du précédent, il suffit :

1) de rajouter une instruction REPRISE dans l'activité du noeud barillet,
 SOUS-PROGRAMME ACTB

E: DECISION(QUANTUM.FINI); REPRISE(E); DUREE(QUANTUM DE NOM DE COURANT DE
MOI);

GOTO E; FIN ACTB

2) que le noeud fasse MOURIR puis VIVRE avant d'associer le suivant

A.7. PROGRAMMATION DE L'ALGORITHME DE DECISION DU NOEUD MULTIPLE STANDARD

SOUS-PROGRAMME DM : (décision multiple)

```

SI MOTIF = ILSEVA ALORS AFFECTER(NOM DE LOCALISER(COMPO DE MOI,ETAT,VIDE),
    NOM DE LOCALISER(LISTE.FILS DE MOI,ETAT,VIDE))
SI MOTIF = ILSTOPPE ALORS ALLER A E SINON ALLE A F
E : SI PRIORITE DE LUI > PRIORITE DE MINUS(LISTE.FILS DE MOI)
ALORS DEBUT DESAFFECTER(MINUS(LISTE.FILS DE MOI)),
AFFECTER(NOM DE LOCALISER(COMPO DE MOI,ETAT,VIDE));RETOUR;FIN
SI MOTIF : ILARRIVE ALORS DEBUT
SI CP ← NOM DE LOCALISER(COMPO DE MOI,ETAT,VIDE) ≠ VIDE
ALORS AFFECTER(CP,LUI)
SINON ALLER A E.
FIN SOUS-PROGRAMME DM

```

Commentaires :

LOCALISER est le même que pour les noeuds simples standards.
 MINUS est différent. Il désigne ici le moins prioritaire des fils actifs,
 s'il en existe, sinon le moins prioritaire des fils affectés.

Algorithmes d'archivage et espace local du noeud multiple standard :

L'algorithme d'archivage interne, nommé AIM, gère non seulement une
 liste de fils comme AIS, mais aussi la liste des composants, référencée
 par COMPO dans l'espace local.

SOUS-PROGRAMME AIM :

```

SI ACTION = AFFECTER ALORS DEBUT
ETAT DE LOCALISER(LISTE.FILS DE MOI,NOM,LUI) ← IL EST AFFECTE
ETAT DE LOCALISER(COMPO DE MOI,NOM,LAUTRE) ← IL EST AFFECTE

```

```

SI ACTION = DESAFFECTER ALORS DEBUT
ETAT DE LOCALISER(LISTE.FILS DE MOI,NOM,LUI) ← IL EST CANDIDAT
ETAT DE LOCALISER(COMPO DE MOI,NOM,LAUTRE) ← VIDE END
FIN SOUS-PROGRAMME AIM

```

L'algorithme d'archivage externe va non seulement tenir compte des motifs ILARRIVE et ILSENA, comme le fait un noeud simple, mais aussi de ILSTOPPE et ILTOURNE

```

SOUS-PROGRAMME AXM
APPEL DE AXS
ETAT DE LOCALISER(LISTE.FILS DE MOI,NOM,LUI)←

```

```

SI MOTIF = ILARRIVE ALORS IL EST CANDIDAT SINON
SI MOTIF = ILTOURNE ALORS ILTOURNE SINON
SI MOTIF = ILSTOPPE ALORS IL EST AFFECTE

```

Voici en résumé, l'automate des transitions du champ ETAT d'un fils dans la LISTE-FILS du noeud standard multiple

MOTIF = ILARRIVE

Il n'est pas	ETAT ←	ACTION=AFFECTER
dans la liste	IL EST CANDIDAT	
		ETAT ← IL EST AFFECTE

MOTIF =
ILSENA

ACTION = DESAFF

MOTIF =
ILTOURNE

ETAT ← ILTOURNE

MOTIF =
ILSTOPPE

La fonction MINUS peut se programmer ainsi :

```

FONCTION MINUS(LISTE)
F1←F2←VIDE ; ECEM=LISTE
TANT QUE (ELEM=NEXT DE ELEM)=VIDE FAIRE

```

```

DEBUT
SI ETAT DE ELEM = ILESTAFFECTE ALORS F1←ELEM
SI ETAT DE ELEM = ILTOURNE ALORS F2←ELEM FIN
MINUS←SI F2 = VIDE ALORS F1 SINON F2

FIN MINUS

```

Maintenant, on peut donner la forme définitive de la fonction de déclaration d'un noeud multiple standard.

C'est tout simplement :

```
A←DNM(PERE.DE(F1,...,FN),COMPOSE.DE(C1,...,CP))
```

où DNM(P,C) s'écrit ainsi :

```
FONCTION DNM(P,C) :
```

```
DNM←DN(MULTIPLE,P,C,ESPACE.LOCAL(2,<Fonctions de service>),
        ALGORITHMES(DM,AXM,AIM)).
```

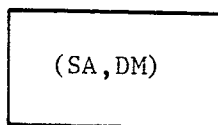
```
LISTE.FILS DE DNM←VIDE,VIDE
```

```
COMPO DE DNM←C;
```

```
FIN DNM.
```

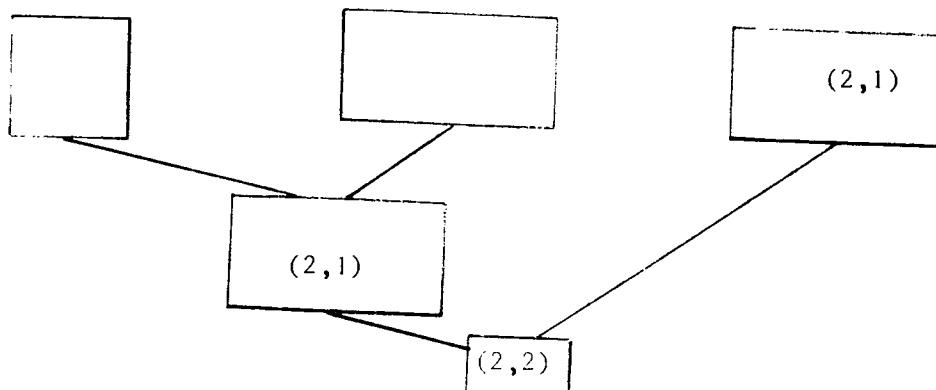
A.8. REPRESENTATION GRAPHIQUE DES NOEUDS STANDARD

Noeud simple lambda



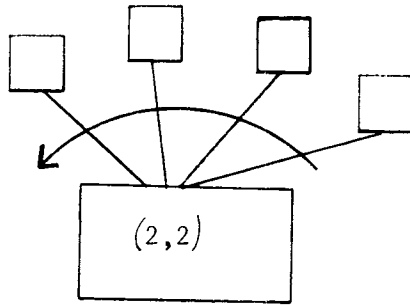
SA = seuil d'activité, 1 par défaut
DM = degré de multiprogrammation,
1 par défaut

Exemple

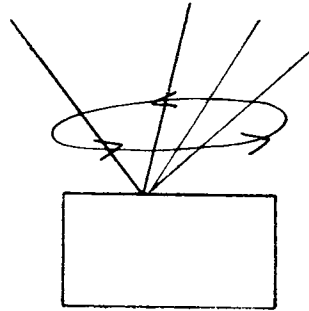


Noeud simple prioritaire :

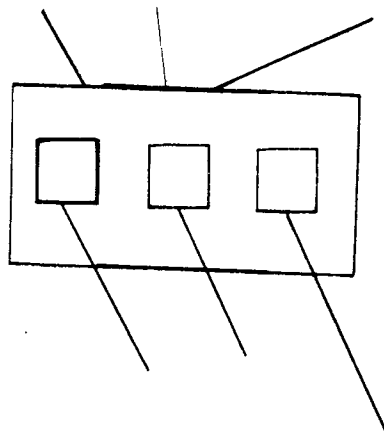
C'est la même chose, plus une flèche orientée dans le sens des priorités croissantes



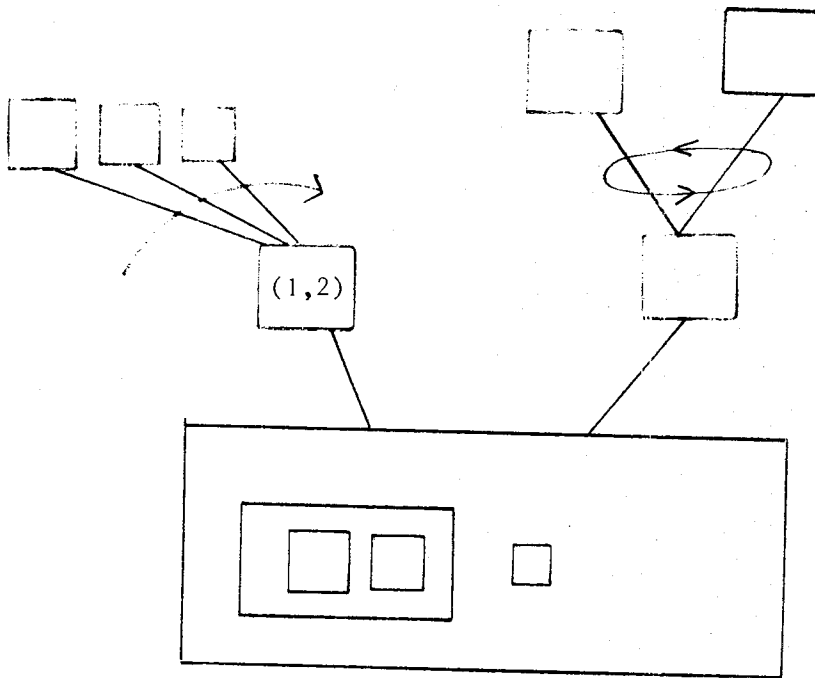
Noeud simple barillet :



Noeud multiple standard :



Exemple :



On obtient ainsi un moyen de définir complètement une architecture par un dessin.

Il ne manque que la description des activités.

N.B. : Une notation algébrique concise pour la structure et les activités est suggérée en VII.2.2.

ANNEXE B

DEFINITION DES INSTRUCTIONS DE "SSH NON STRUCTURE"

Dans certains cas, l'utilisateur peut juger excessives certaines contraintes volontaires de SSH introduites dans un but de modularité et de bonne structuration.

Il aimerait pouvoir accéder à un niveau de langage un peu plus élémentaire, où ses libertés seraient plus grandes pour construire des modèles plus généraux à ses risques et périls.

Les instructions de "SSH non structuré" répondent à ce désir.

B.1. INSTRUCTIONS DE SSHNS POUR SPECIFIER LA STRUCTURE DU MODELE
EN DEHORS DES DECLARATIONS DES NOEUDS

- * * PERE.FILS(P,F) où P est un noeud simple ou multiple et F un noeud simple. Cette instruction crée une relation père-fils entre P et F.
- * * COMPOSITION(M,C) où M est un noeud multiple et C un noeud quelconque. Cette instruction rend C composant de M.

B.2. INSTRUCTIONS EXPLICITANT LES PARAMETRES DEFINIS IMPLICITEMENT PAR
LA STRUCTURE EN SSH STRUCTURE

Exemple :

ASSOCIER(X) signifie en SSH structuré que X est associé par l'exécutant de cette instruction. En SSHNS, on ajoute un paramètre qui indique à qui le premier noeud est associé.

* * ASSOCIER(F,P)

Donc, ASSOCIER(X) est équivalent à ASSOCIER(X,MOI)

Voici les autres instructions de SSHNS

- * * DISSOCIER(F,P) : c'est P qui dissocie F
- * * VIVRE(N) : c'est N qui fait VIVRE
- * * MOURIR(N) : c'est N qui fait MOURIR
- * * DEMANDER(R,N) : c'est N qui demande la ressource R
- * * LIBERER(R,N) : c'est N qui libère la ressource R
- * * BLOQUER(N,R) : c'est R qui bloque N
- * * DEBLOQUER(N,R) : c'est R qui débloque N
- * * DUREE(D,F,N) : c'est un algorithme de N qui exécute cette instruction DUREE

ANNEXE C

C.1. COMPLEMENTS SUR LES CONVENTIONS DE PL UTILISEES DANS L'ALGORITHME
D'IMPLEMENTATION DE SSH

a) Il existe une constante HIGH.VALUE égale au plus grand nombre positif représentable dans le PL.

b) Fonctions de gestion de mémoire dynamique GET, FREE

A ← GET(X,Y,Z)

A référence une zone de mémoire de trois éléments consécutifs initialisés à X, Y et Z.

B ← FREE(A)

Libération de la zone référencée par A. B reçoit la valeur de A.

c) Un opérateur d'accès au contenu d'une zone obtenue par GET, c'est l'opérateur DE

Exemple :

NEXT = 1, PREC = 2

A ← GET(4,5)

⇒ NEXT DE A vaut 4

PREC DE A vaut 5

d) Appel des sous-programmes désignés par une variable de type référence de sous-programme :

Exemple :

SOUS-PROGRAMME SSP(X,Y)

FIN-SSP

A ← GET(5,SSP)

PROC ← 2

CALL (PROC DE A)_φ(U,V)

Cet appel est équivalent à CALL SSP(U,V)

e) Un opérateur ET pour construire des listes

A ET B ET C génèrent en mémoire dynamique :

une liste dont :

- chaque élément a la structure suivante :

NEXT : Pointeur sur l'élément suivant

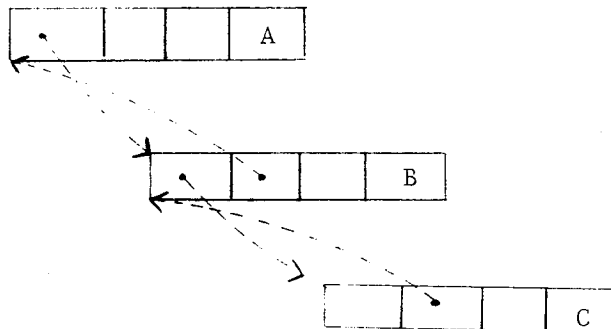
PREC : Pointeur sur l'élément précédent

ETAT : champ libre pour caractériser l'élément

NOM : nom de l'élément

Exemple :

A ET B ET C produit :




```

... ENVOI(P,F)... CALL BALLAD(P,TC DE F+SY(F),NEW)
... MUTER(C,F)... CALL BALLAD(C,F,MMU);TC DE SY(F)+P
... BLOQUER(X)... CALL PIL(PERE DE (X+SY(X)),X,ETAT DE X,I,MBL);PERE DE (TC DE X+X)+PS DE X
... DERROQUER(X)... CALL PIL(PERE DE (X+SY(X))+PS DE X,X,I,C)NDRL)
... DERMANDER(X)... CALL PIL(X,P,I,C,NDM)
... LIBERER(X)... CALL PIL(SY(X),P,C,I,VLIR)
... DECISION(X)... LUI+X;MOTIF+GESTMOI;CALL ALGO1;CALL ALGO2;CALL ALGO3;CALL ALGO4
... ALGO1...SI C01 DE P=VIDE ALORS DERUT CALL C01 DE P;RETOUR;FIN
ACTIVITE DE (Z+COPIE(C01 DE P))+C01 DE P;SMOI DE Z+MOI;SLUI DE Z+LUI;SMOTIF DE Z+MOTIF
SORG DE Z+ORIGINE;CONFIE DE Z+VRAI;CALL PIL(C01 DE P,Z,I,C)
... MEME CHOSE POUR ALGO2,3,4 EN REMPLACANT 1 PAR 2,3,4
... REPRISE(X)... PERE DE COUR+X
... ESPACE LOCAL(T=0,I=RIEN,C=RIEN,D=RIEN)...WT[1]+T;WT[2]+I;WT[3]+C;WT[4]+D;WT[5]+1
... CONFIE(A=P+RACINE,F=RACINE,A=RIEN)...WT[C][WT+1]+P;WT[C][WT+2]+A;WT+WT+3;CONFIE,A+CONF
... ALGORITHMES(A1+RIEN,A2+RIEN,A3+RIEN,A4+RIEN,A5+RIEN)...WT[5]+A5
POUR I=1,4 FAIRE DERUTS SI AI=CONF ALORS DERUT WT[5+I]+A1;RIEN;FIN
SI NON DERUT WT[5+I]+WT[6+I]+VIDE;WT[7+I]+A1;RIEN;FIN
... SIMPLE...TS DE (MN+GET(TSIM+WT[1]))+TSIN;TYPE DE MN+SIM
... MULTIPLE ET RESSOURCE... MEME CHOSE EN REMPLACANT SIMI PAR MMU, ET RESS,
... PERE,NDL)...PY)...POUR CHAQUE PI FAIRE DE PI+PERE DE PI+MN
... DN(T,P,C,A,E)...DN+MN+TS DE MN;DO I=1,17 (I DE MN)+WT[I];CALL INIT DE MN;SI TYPE DE MN+SIM ALORS RETOUR
PERE DE MN+RACINE;SY DE MN+0;TC DE MN+MN;ETAT DE MN+I;LOCASS DE MN+VIDE;VG DE MN+VL DE MN+1
SUITE DE MN+DERUTPROG;PERE DE MN+VIDE;US DE MN+VIDE;ECH;LOC DE MN+ECH DE MN+BIG.VALUE
REURE DE V+AHA DE V+0;CONFIE DE MN+FAUX;SMOI DE MN+SLUI DE MN+SMOTIF DE MN+SORG DE MN+VIDE
... SIMULATION(I,F,D)...CALL PIL(PERE DE (NI+SY(I)),MI,I,C);NF+SY(F);DUR+D;CALL PIL(PACINE,NTPS,I,C)
IR+5;IR+1;WT+1;TM[BAL,RAL]+SWI;TM[I,C]+ILSTOPPE
TM[AC,AV]+ILSTOPPE;TM[AC,C]+ILSTOPPE
... DERAC...SI MOTIF=ILARRIVE ALORS ASSOCIEE(LUI)
... ACTPS...P;DUREE(DUR);ENVOI(PERE DE NF,VF,I,C);ALTER A E
... DESCENDRE...DO I=1,IR-1 FAIRE IR-I+5]+DRI[I];IR+1;IR+IR+IR-6
... IDERE X...IDERE+X
... RIEN...RETOUR
... AI(ACT,P2)...ACTION+ACT;MOI+P;LUI+DRI[IR-4];MOTIF+TM[DRI[IR-3];DRI[IR-2]];CALL ALGO4 DE P
... COPIE(X)...COPIE+GETCOPY(X+SY(X),TS DE X+TAILLE DE X);CALL (COPY DE X)PHUS(X)
... DERROQUER(X)...CALL (PERE DE SY(X))PH;CALL PERE(SY(X))
... FONCTIONS DE GESTION DE MEMOIRE ET DE VISITE NON DETAILLEES ICI
... AJOUTER(LISTE,YOM)...
... ENLEVER(LISTE,ELEMENT)...
... SUIV(LISTE,ELEMENT)...
... PERE(LISTE,ELEMENT)...
... GET(TAILLE)...ACQUERIR UNE ZONE MEMOIRE
... PERE(ADRESSE)...LIBERER UNE ZONE MEMOIRE
... GETCOPY(ADRESSE,TAILLE)...ACQUERIR UNE ZONE MEMOIRE INITIALISEE A LA VALEUR D'UNE AUTRE
... DERMANDER(X)...POUR LE FILS DE X AVANT LA PLUS PETITE FICH

```

dernière page de thèse

AUTORISATION DE SOUTENANCE

VU les dispositions de l'article 3 de l'arrêté du 16 Avril 1974,

VU le rapport de présentation de


- . Monsieur F. ANCEAU, Maître de Conférences
- . Monsieur S. GUIBOUD-RIBAUD, Chargé de cours à l'Ecole des Mines de St ETIENNE

Monsieur R O H M E R Jean

est autorisé à présenter une thèse en soutenance pour l'obtention du diplôme de DOCTEUR-INGENIEUR, spécialité "GENIE INFORMATIQUE".-

Fait à Grenoble, le 18 Juin 1976

Le Président de l'Institut National Polytechnique,


Ph. TRAYNARD
Président
de l'Institut National Polytechnique