Le traitement des exceptions dans les programmes modulaires

Flaviu Cristian

THESE

présentée à

Université Scientifique et Médicale de Grenoble
Institut National Polytechnique de Grenoble

pour obtenir le grade de

DOCTEUR INGENIEUR

«INFORMATIQUE»

par

Flaviu CRISTIAN

LE TRAITEMENT DES EXCEPTIONS DANS LES PROGRAMMES MODULAIRES

Thèse soutenue le 12 septembre 1979 devant la commission d'examen

L. BOLLIET	Président
R. BALTER A. COSTES P. COUSOT S. KRAKOWIAK	Examinateurs

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

Monsieur Gabriel CAU : Président
Monsieur Joseph KLEIN : Vice-Président

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

PROFESSEURS TITULAIRES

MM.	AMBLARD Pierre	Clinique de dermatologie
	ARNAUD Paul	Chimie
	ARVIEU Robert	I.S.N.
	AUBERT Guy	Physique
	AYANT Yves	Physique approfondie
Mme	BARBIER Marie-Jeanne	Electrochimie
MM.	BARBIER Jean-Claude	Physique expérimentale
	BARBIER Reynold	Géologie appliquée
	BARJON Robert	Physique nucléaire
	BARNOUD Fernand	Biosynthèse de la cellulose
	BARRA Jean-René	Statistiques
	BARRIE Joseph	Clinique chirurgicale A
	BEAUDOING André	Clinique de pédiatrie et puériculture
	BELORIZKY Elie	Physique
	BARNARD Alain	Mathématiques pures
Mme	BERTRANDIAS Françoise	Mathématiques pures
MM.	BERTRANDIAS Jean-Paul	Mathématiques pures
	BEZES Henri	Clinique chirurgicale et traumatologie
	BLAMBERT Maurice	Mathématiques pures
	BOLLIET Louis	Informatique (I.U.T. B)
	BONNET Jean-Louis	Clinique ophtalmologie
	BONNET-EYMARD Joseph	Clinique hépato-gastro-entérologie
Mme	BONNIER Marie-Jeanne	Chimie générale
MM.	BOUCHERLE André	Chimie et toxicologie
	BOUCHEZ Robert	Physique nucléaire
	BOUSSARD Jean-Claude	Mathématiques appliquées
	BOUTET DE MONVEL Louis	Mathématiques pures
	BRAVARD Yves	Géographie
	CABANEL Guy	Clinique rhumatologique et hydrologique
	CALAS François	Anatomie
	CARLIER Georges	Biologie végétale
	CARRAZ Gilbert	Biologie animale et pharmacodynamie

.../...

MM.	CAU Gabriel	Médecine légale et toxicologie
	CAUQUIS Georges	Chimie organique
	CHABAUTY Claude	Mathématiques pures
	CHARACHON Robert	Clinique ot-rhino-laryngologique
	CHATEAU Robert	Clinique de neurologie
	CHIBON Pierre	Biologie animale
	COEUR André	Pharmacie chimique et chimie analytique
	COUDERC Pierre	Anatomie pathologique
	DEBELMAS Jacques	Géologie générale
	DEGRANGE Charles	Zoologie
	DELORMAS Pierre	Pneumophtisiologie
	DEPORTES Charles	Chimie minérale
	DESRE Pierre	Métallurgie
	DODU Jacques	Mécanique appliquée (I.U.T. I)
	DOLIQUE Jean-Michel	Physique des plasmas
	DREYFUS Bernard	Thermodynamique
	DUCROS Pierre	Cristallographie
	FONTAINE Jean-Marc	Mathématiques pures
	GAGNAIRE Didier	Chimie physique
	GALVANI Octave	Mathématiques pures
	GASTINEL Noël	Analyse numérique
	GAVEND Michel	Pharmacologie
	GEINDRE Michel	Electroradiologie
	GERBER Robert	Mathématiques pures
	GERMAIN Jean-Pierre	Mécanique
	GIRAUD Pierre	Géologie
	JANIN Bernard	Géographie
	KAHANE André	Physique générale
	KLEIN Joseph	Mathématiques pures
	KOSZUL Jean-Louis	Mathématiques pures
	KRAVTCHENKO Julien	Mécanique
	LACAZE Albert	Thermodynamique
	LACHARME Jean	Biologie végétale
Mme	LAJZEROWICZ Janine	Physique
MM.	LAJZEROWICZ Joseph	Physique
	LATREILLE René	Chirurgie générale
	LATURAZE Jean	Biochimie pharmaceutique
	LAURENT Pierre	Mathématiques appliquées
	LEDRU Jean	Clinique médicale B
	LE ROY Philippe	Mécanique (I.U.T. I)

.../...

MM.	LLIBOUTRY Louis	Géophysique
	LOISEAUX Jean-Marie	Sciences nucléaires
	LONGEQUEUE Jean-Pierre	Physique nucléaire
	LOUP Jean	Géographie
Mlle	LUTZ Elisabeth	Mathématiques pures
MM.	MALINAS Yves	Clinique obstétricale
	MARTIN-NOEL Pierre	Clinique cardiologique
	MAYNARD Roger	Physique du solide
	MAZARE Yves	Clinique Médicale A
	MICHEL Robert	Minéralogie et pétrographie
	MICOUD Max	Clinique maladies infectieuses
	MOURIQUAND Claude	Histologie
	MOUSSA André	Chimie nucléaire
	NEGRE Robert	Mécanique
	NOZIERES Philippe	Spectrométrie physique
	OZENDA Paul	Botanique
	PAYAN Jean-Jacques	Mathématiques pures
	PEBAY-PEYROULA Jean-Claude	Physique
	PERRET Jean	Séméiologie médicale (neurologie)
	RASSAT André	Chimie systématique
	RENARD Michel	Thermodynamique
	REVOL Michel	Urologie
	RINALDI Renaud	Physique
	DE ROUGEMONT Jacques	Neuro-Chirurgie
	SARRAZIN Roger	Clinique chirurgicale B
	SEIGNEURIN Raymond	Microbiologie et hygiène
	SENGEL Philippe	Zoologie
	SIBILLE Robert	Construction mécanique (I.U.T. I)
	SOUTIF Michel	Physique générale
	TANCHE Maurice	Physiologie
	VAILLANT François	Zoologie
	VALENTIN Jacques	Physique nucléaire
Mme	VERAIN Alice	Pharmacie galénique
MM.	VERAIN André	Physique biophysique
	VEYRET Paul	Géographie
	VIGNAIS Pierre	Biochimie médicale

.../...

PROFESSEURS ASSOCIES

MM.	CRABBE Pierre	CERMO
	SUNIER Jules	Physique

PROFESSEURS SANS CHAIRE

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

.../...

MM.	PELMONT Jean	Biochimie
	PERRIAUX Jean-Jacques	Géologie et minéralogie
	PFISTER Jean-Claude	Physique du solide
Mlle	PIERY Yvette	Physiologie animale
MM.	RAYNAUD Hervé	M.I.A.G.
	REBECQ Jacques	Biologie (CUS)
	REYMOND Jean-Charles	Chirurgie générale
	RICHARD Lucien	Biologie végétale
Mme	RINAUDO Marguerite	Chimie macromoléculaire
MM.	SARROT-REYNAULD Jean	Géologie
	SIROT Louis	Chirurgie générale
Mme	SOUTIF Jeanne	Physique générale
MM.	STIEGLITZ Paul	Anesthésiologie
	VIALON Pierre	Géologie
	VAN CUTSEM Bernard	Mathématiques appliquées

MAITRES DE CONFERENCES ET MAITRES DE CONFERENCES AGREGES

MM.	ARMAND Yves	Chimie (I.U.T. I)
	BACHELOT Yvan	Endocrinologie
	BARGE Michel	Neuro-chirurgie
	BEGUIN Claude	Chimie organique
Mme	BERIEL Hélène	Pharmacodynamie
MM.	BOST Michel	Pédiatrie
	BOUCHARLAT Jacques	Psychiatrie adultes
Mme	BOUCHE Liane	Mathématiques (CUS)
MM.	BRODEAU François	Mathématiques (I.U.T. B) (Personne étrangère habilitée à être directeur de thèse)
	BERNARD Pierre	Gynécologie
	CHAMBAZ Edmond	Biochimie médicale
	CHAMPETIER Jean	Anatomie et organogénèse
	CHARDON Michel	Géographie
	CHERADAME Hervé	Chimie papetière
	CHIAVERINA Jean	Biologie appliquée (EFP)
	COLIN DE VERDIERE Yves	Mathématiques pures
	CONTAMIN Charles	Chirurgie thoracique et cardio-vasculaire
	CORDONNER Daniel	Néphrologie
	COULOMB Max	Radiologie
	CROUZET Guy	Radiologie

.../...

MM.	CYROT Michel	Physique du solide
	DENIS Bernard	Cardiologie
	DOUCE Roland	Physiologie végétale
	DUSSAUD René	Mathématiques (CUS)
Mme	ETERRADOSSI Jacqueline	Physiologie
MM.	FAURE Jacques	Médecine légale
	FAURE Gilbert	Urologie
	GAUTIER Robert	Chirurgie générale
	GIDON Maurice	Géologie
	GROS Yves	Physique (I.U.T. I)
	GUIGNIER Michel	Thérapeutique
	GUITTON Jacques	Chimie
	HICTER Pierre	Chimie
	JALBERT Pierre	Histologie
	JUNIEN-LAVILLAVROY Claude	O.R.L.
	KOLODIE Lucien	Hématologie
	LE NOC Pierre	Bactériologie-virologie
	MACHE Régis	Physiologie végétale
	MAGNIN Robert	Hygiène et médecine préventive
	MALLION Jean-Michel	Médecine du travail
	MARECHAL Jean	Mécanique (I.U.T. I)
	MARTIN-BOUYER Michel	Chimie (CUS)
	MASSOT Christian	Médecine interne
	NEMOZ Alain	Thermodynamique
	NOUGARET Marcel	Automatique (I.U.T. I)
	PARAMELLE Bernard	Pneumologie
	PECCOUD François	Analyse (I.U.T. B) (Personnalité étrangère habilitée à être directeur de thèse)
	PEFFEN René	Métallurgie (I.U.T. I)
	PERRIER Guy	Géophysique-glaciologie
	PHELIP Xavier	Rhumatologie
	RACHALL Michel	Médecine interne
	RACINET Claude	Gynécologie et obstétrique
	RAMBAUD Pierre	Pédiatrie
	RAPHAEL Bernard	Stomatologie
Mme	RENAUDET Jacqueline	Bactériologie (pharmacie)
MM.	ROBERT Jean-Bernard	Chimie-physique
	ROMIER Guy	Mathématiques (I.U.T. B) (Personnalité étrangère habilitée à être directeur de thèse)
	SAKAROVITCH Michel	Mathématiques appliquées

.../...

MM.	SCHAERER René	Cancérologie
Mme	SEIGLE-MURANDI Françoise	Crytogamie
MM.	STOEBNER Pierre	Anatomie pathologie
	STUTZ Pierre	Mécanique
	VROUSOS Constantin	Radiologie

MAITRES DE CONFERENCES ASSOCIES

MM.	DEVINE Roderick	Spectro Physique
	KANEKO Akira	Mathématiques pures
	JOHNSON Thomas	Mathématiques appliquées
	RAY Tuhina	Physique

MAITRE DE CONFERENCES DELEGUE

M.	ROCHAT Jacques	Hygiène et hydrologie (pharmacie)

Fait à Saint Martin d'Hères, novembre 1977

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Année universitaire 1979-1980

Président : M. Philippe TRAYNARD

Vice-Présidents : M. Georges LESPINARD
M. René PAUTHENET

PROFESSEURS DES UNIVERSITES

MM.	ANCEAU François	Informatique fondamentale et appliquée
	BENOIT Jean	Radioéléctricité
	BESSON Jean	Chimie Minérale
	BLIMAN Samuel	Electronique
	BLOCH Daniel	Physique du Solide - Cristallographie
	BOIS Philippe	Mécanique
	BONNETAIN Lucien	Génie Chimique
	BONNIER Etienne	Métallurgie
	BOUVARD Maurice	Génie Mécanique
	BRISSONNEAU Pierre	Physique des Matériaux
	BUYLE-BODIN Maurice	Electronique
	CHARTIER Germain	Electronique
	CHERADAME Hervé	Chimie Physique Macromoléculaires
Mme	CHERUY Arlette	Automatique
MM.	CHIAVERINA Jean	Biologie, Biochimie, Agronomie
	COHEN Joseph	Electronique
	COUMES André	Electronique
	DURAND Francis	Métallurgie
	DURAND Jean-Louis	Physique Nucléaire et Corpusculaire
	FELICI Noël	Electrotechnique
	FOULARD Claude	Automatique
	GUYOT Pierre	Métallurgie Physique
	IVANES Marcel	Electrotechnique
	JOUBERT Jean-Claude	Physique du Solide - Cristallographie
	LACOUME Jean-Louis	Géographie - Traitement du Signal
	LANCIA Roland	Electronique - Automatique
	LESIEUR Marcel	Mécanique
	LESPINARD Georges	Mécanique
	LONGEQUEUE Jean-Pierre	Physique Nucléaire Corpusculaire
	MOREAU René	Mécanique
	MORET Roger	Physique Nucléaire Corpusculaire
	PARIAUD Jean-Charles	Chimie - Physique
	PAUTHENET René	Physique du Solide - Cristallographie
	PERRET René	Automatique

.../...

MM.	PERRET Robert	Electrotechnique
	PIAU Jean-Michel	Mécanique
	PIERRARD Jean-Marie	Mécanique
	POLOUJADOFF Michel	Electrotechnique
	POUPOT Christian	Electronique - Automatique
	RAMEAU Jean-Jacques	Chimie
	ROBERT André	Chimie Appliquée et des matériaux
	ROBERT François	Analyse numérique
	SABONNADIERE Jean-Claude	Electrotechnique
Mme	SAUCIER Gabrielle	Informatique fondamentale et appliquée
M.	SOHM Jean-Claude	Chimie - Physique
Mme	SCHLENKER Claire	Physique du Solide - Cristallographie
MM.	TRAYNARD Philippe	Chimie - Physique
	VEILLON Gérard	Informatique fondamentale et appliquée
	ZADWORNY François	Electronique

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

M.	FRUCHART Robert	Directeur de Recherche
MM.	ANSARA Ibrahïm	Maître de Recherche
	BRONOEL Guy	Maître de Recherche
	CARRE René	Maître de Recherche
	DAVID René	Maître de Recherche
	DRIOLE Jean	Maître de Recherche
	KAMARINOS Georges	Maître de Recherche
	KLEITZ Michel	Maître de Recherche
	LANDAU Ioan-Doré	Maître de Recherche
	MERMET Jean	Maître de Recherche
	MUNIER Jacques	Maître de Recherche

Personnalités habilitées à diriger des travaux de recherche (décision du Conseil Scientifique)

E.N.S.E.E.G.

MM.	ALLIBERT Michel	
	BERNARD Claude	
	CAILLET Marcel	
Mme	CHATILLON Catherine	
MM.	COULON Michel	
	HAMMOU Abdelkader	
	JOUD Jean-Charles	
	RAVAINE Denis	
	SAINFORT	C.E.N.G.

.../...

MM. SARRAZIN Pierre
SOUQUET Jean-Louis
TOUZAIN Philippe
URBAIN Georges — Laboratoire des Ultra-Réfractaires ODEILLO

E.N.S.M.E.E.

MM. BISCONDI Michel
BOOS Jean-Yves
GUILHOT Bernard
KOBILANSKI André
LALAUZE René
LANCELOT Françis
LE COZE Jean
LESBATS Pierre
SOUSTELLE Michel
THEVENOT François
THOMAS Gérard
TRAN MINH Canh
DRIVER Julian
RIEU Jean

E.N.S.E.R.G.

MM. BOREL Joseph
CHEHIKIAN Alain
VIKTOROVITCH Pierre

E.N.S.I.E.G.

MM. BORNARD Guy
DESCHIZEAUX Pierre
GLANGEAUD François
JAUSSAUD Pierre
Mme JOURDAIN Geneviève
MM. LEJEUNE Gérard
PERARD Jacques

E.N.S.H.G.

M. DELHAYE Jean-Marc

E.N.S.I.M.A.G.

MM. COURTIN Jacques
LATOMBE Jean-Claude
LUCAS Michel
VERDILLON André

Je remercie :

Monsieur L. BOLLIET de m'avoir fait l'honneur de présider le jury de cette thèse,

Messieurs R. BALTER, A. COSTES et P. COUSOT pour avoir bien voulu faire partie du jury et pour l'attention critique qu'ils ont manifestée

Monsieur S. KRAKOWIAK pour m'avoir accueilli dans son équipe lorsque j'étais encore étudiant à l'Ecole Nationale Supérieure d'Informatique et de Mathématiques Appliquées; ses avis critiques et conseils ont toujours été pour moi précieux.

Je remercie également tous les chercheurs de l'IMAG avec lesquels j'ai pu avoir des discussions et échanges de vues intéressants, et dont les avis critiques ont contribué à améliorer la présentation de cette thèse.

Je remercie enfin tous ceux et celles qui ont contribué à la réalisation matérielle de cet ouvrage.

Flaviu CRISTIAN

LE TRAITEMENT DES EXCEPTIONS DANS LES PROGRAMMES MODULAIRES

I - INTRODUCTION

Le traitement automatique des données par un programme doit préserver certaines propriétés invariantes, propres à la sémantique du traitement et des informations qui sont manipulées. Une exception est une tentative de violer à l'exécution une telle propriété invariante. Une détection d'exception constitue un signal d'alarme : si l'exception reste non traitée, alors la continuation de l'exécution risque de provoquer d'autres exceptions.

Lors des premiers stades du développement de la technologie des systèmes informatiques, les efforts ont surtout porté sur la maîtrise de la complexité inhérente au fonctionnement "normal". Les problèmes liés à la prise en compte des possibles occurrences d'exceptions ont souvent été relégués "à plus tard", ou ont fait l'objet de solutions "ad-hoc", pas toujours satisfaisantes. Des propositions de solutions et des mécanismes linguistiques pour le traitement des exceptions ont vu le jour à partir des années 70 [Parnas 72], [Horning 74], [Goodenough 75], [Kaiser 76], etc. Ainsi, on a imaginé des mécanismes pour le traitement des exceptions "prévues", et d'autres pour le traitement de celles qui étaient "imprévues". Mais entre les différentes approches il y avait assez peu de points communs, et chacun avait sa propre terminologie, ce qui ne facilitait pas la compréhension mutuelle. Par exemple, pour désigner le concept d'exception, on utilisait une variété de termes comme : erreur à l'exécution, événement anormal, événement indésirable, anomalie, panne, incident, défaillance, etc. De plus, l'utilité d'une recherche sur le traitement des exceptions n'était pas évidente pour tout le monde. Beaucoup pensaient que logiciel fiable veut dire logiciel correct, et qu'à partir du moment où un programme est correct, il n'y a plus de place pour des anomalies, des pannes ou d'autres événements "indésirables". La naissance de cette confusion est peut être due en partie aux co-notations négatives associées aux termes qui étaient utilisés pour désigner le concept d'exception. En fait, les méthodes de preuves de programmes ou d'analyse sémantique peuvent garantir que certaines exceptions ne se produiront jamais, mais ne peuvent pas les éviter toutes. Les programmeurs doivent par conséquent être préparés à traiter toutes les exceptions qui ne peuvent pas être évitées statiquement, c'est-à-dire à assurer que l'occurrence dynamique d'une telle exception n'entraînera pas derrière elle toute une cascade d'exceptions additionnelles. Un programme peut fort bien fonctionner correctement (en accord avec ses spécifications) malgré l'occurrence d'exceptions à l'exécution,

à condition que des traitements appropriés soient prévus pour ces exceptions.

Le travail que nous présentons dans cette thèse a été effectué dans le cadre du projet SESAME [Krakowiak 76], [Cheval 76]. Le but de ce projet est la construction d'un ensemble intégré d'outils d'aide à la programmation des systèmes d'exploitation. Il fournit à ses utilisateurs :

a) Un langage de programmation modulaire, conçu comme une extension de PASCAL,

b) Un langage pour la connexion des modules, distinct du premier,

c) Un système pour la gestion des textes de modules et de programmes de connexion, appelé Bibliothécaire.

Après avoir acquis de l'expérience dans le domaine de la programmation modulaire et du traitement("ad-hoc") des exceptions, lors de la conception du Bibliothécaire [Cristian 77], le rôle qui nous a été assigné au sein de l'équipe a été de concevoir un mécanisme d'exception pour le langage SESAME. Au delà de la définition d'un mécanisme particulier, ce travail a été l'occasion d'une réflexion sur ce qu'est le traitement des exceptions en général.

Nous commençons par donner des définitions précises (chapitres deux et trois) à un certain nombre de concepts fréquemment rencontrés dans la littérature (comme exception, défaillance, état cohérent, état incohérent, opération atomique). Nous introduisons également quelques nouveaux concepts pour décrire des réalités qui auparavant n'avaient pas de nom (ensemble d'incohérence, couverture d'incohérence, résidus, tolérance faible aux exceptions, tolérance forte). A l'aide de ces outils conceptuels, il nous est possible de développer une approche unifiée pour le traitement des exceptions "prévues" et "imprévues" (§3.3). Pour sa mise en oeuvre, nous avons imaginé un mécanisme d'exception, intégrable à n'importe quel langage modulaire, et en particulier à SESAME (chapitres quatre et cinq). Le but du chapitre six est de montrer que le mécanisme proposé est un outil commode pour programmer le traitement des exceptions qui peuvent être rencontrées dans la pratique. Afin d'illustrer son utilisation, nous avons choisi de l'appliquer à la programmation des traitements

exceptionnels d'un système transactionnel multi-accès. Pour la conception de ce système, nous avons appliqué les méthodes de programmation avec des types abstraits [Liskov 74], [Brinch Hansen 77]. La simplicité de la structure trouvée illustre bien le progrès que l'application de telles méthodes apporte à l'art de la programmation.

Le développement de modèles conceptuels pour la compréhension des problèmes liés au traitement des exceptions, ainsi que la conception de mécanismes pour intégrer dans les langages de programmation les concepts de base de ces modèles, sont motivés par la résolution technique des problèmes de fiabilité des programmes. Ces efforts sont complémentaires aux recherches sur les spécifications formelles, vérifications statiques ou preuves de programmes. En effet, des spécifications sont nécessaires pour la description formelles des propriétés invariantes qui doivent être respectées par les opérations d'un programme. Les méthodes de vérifications statiques peuvent réduire considérablement le nombre d'exceptions qui peuvent effectivement survenir à l'exécution. Pour programmer le traitement de ces dernières, il est nécessaire de disposer de mécanismes d'exception adéquats. Les méthodes de preuve peuvent être utilisées pour s'assurer que les traitements exceptionnels sont, comme les traitements "normaux", conformes à leurs spécifications.

II - LES EXCEPTIONS

Le concept d'exception est défini en fonction du concept de type : nous appelons exception toute tentative de violer à l'exécution un invariant défini sur un type.

Les méthodes de vérification statique ne peuvent pas garantir le respect dynamique de tous ces invariants. Le programmeur doit prévoir des tests à l'exécution pour détecter les possibles occurrences d'exceptions.

Au moment où une exception est détectée dans un programme, l'état de celui-ci est incohérent: la continuation de l'exécution peut produire des effets imprévisibles et provoquer d'autres exceptions additionnelles. Le concept d'ensemble d'incohérence d'une exception définit précisément quelle est la partie de l'état qui est incohérente au moment où l'exception survient.

II - 1. EXCEPTIONS PREDEFINIES

Une variable est une entité créée et manipulée par un programme. Toute variable possède un état, qui peut être modifié par des opérations spécifiées pour son type. Tout langage implémente un certain nombre de types prédéfinis ou "concrets". En SESAME, un type concret est soit un type primitif (entier, booléen, scalaire, caractère), soit un type structuré (enregistrement, tableau), soit un type périphérique (disque, imprimante, télétype, etc). Les types primitifs et structurés sont implémentés par le compilateur. Les types périphériques sont implémentés par des modules écrits en langage machine et conservés dans la BIBLIOTHEQUE. Soit I_0 le programme qui implémente tous les types concrets.

II - 1.1. Types concrets

Une variable de type primitif ou structuré est créée par une déclaration dans module. Une variable de type périphérique est créée par une déclaration dans un programme de connexion.
Les états externes qu'une variable v de type primitif T_p peut assumer, peuvent être définis par énumération $T_p = \{t_1, t_2, .., t_p\}$. Par exemple, une variable de type entier peut avoir les états $\{-N, -N+1, ..., N\}$ où N dépend de la machine physique utilisée.

I_0 construit l'état externe de v par l'application d'une fonction de représentation [Hoare 72] à l'état interne de la mémoire qui implémente v. Par exemple, l'état interne *X' 0000000A'* représente l'état externe *'10'* d'une variable entière.

L'état externe d'une variable de type structuré est l'agrégation des états externes de ses composants primitifs. Ainsi, l'ensemble des états externes d'une variable de type *array* $[1...N]$ *of* T_p est le produit cartésien $(T_p)^N$, et les états externes d'une variable de type *record* $A : T_A; ...; X : T_X$ *end* sont des éléments du produit $T_A \ X ... X \ T_X$.

Les types périphériques, beaucoup moins "beaux" du point de vue algébrique, sont imposés par la technologie. Un type périphérique est défini par le jeu des opérations d'entrée/sortie qui lui sont propres. L'état externe d'une variable de type périphérique peut être lu par des opérations d'entrée et changé par des opérations de sortie. Par exemple, l'état d'une variable de type *DISK* peut être lu par une opération de lecture de bloc et modifié par une opération d'écriture de bloc.

```
type BLOC = 1.. &NB; % &NB est le nombre de blocs d'un disque %
     DISKBLOC = univ* array [ 1.. 512 ] of char;

class DISK (&AD : integer); % A la création d'une variable de type
                              DISK, le programmeur indique l'adresse
                              &AD du disque physique qui mémorisera
                              son état interne %

ext procedure WRITE (B : BLOC; BUF : DISKBLOC);
ext function READ (B : BLOC) : DISKBLOC**;
```

II - 1.2. Exceptions de niveau 0

Les opérations faites sur une variable doivent préserver l'invariant qui caractérise intrinsèquement son type. Par exemple, si v est de type *1..10* alors l'invariant de v est $v \in [1,10] \cap \mathbb{N}$. La transgression des propriétés invariantes associées avec des types doit être détectée par des mécanismes de protection. Par exemple, le mécanisme de protection syntaxique permet de vérifier statiquement que les opérations faites sur une variable sont celles spécifiées pour son type. Mais l'analyse syntaxique ne suffit pas toujours, car une opération syntaxiquement légale (affectation) sur une variable (de type intervalle) peut mettre la variable dans un état qui ne vérifie pas l'invariant spécifié pour son type.

---

* La signification du mot clé *univ* est celle de CONCURRENT-PASCAL [Brinch Hansen 77]. Nous préciserons sa sémantique au §VI.3.3.

** Cette notation va être utilisée dans cette thèse pour mettre en évidence le résultat d'un appel de fonction. Dans l'implémentation actuelle de SESAME, le type du résultat d'une fonction ne peut être que primitif.

Pour empêcher l'apparition de tels états, le concepteur de I_0 définit pour chaque type concret des assertions qui doivent être vérifiées avant (préconditions) ou après (postconditions) une opération sur une variable appartenant à ce type. Par exemple, dans le cas d'une opération $v := v + 1$ on doit vérifier, avant d'effectuer l'opération, la précondition $(v < 10)$. Dans le cas d'une opération d'écriture sur *DISK* (§II.1.1)l'invariant "au bout d'un temps fini, l'état externe du bloc à écrire *B* est égal à l'état interne de *BUF*" ne peut être évalué qu'en fin d'écriture (post-condition) par un mécanisme de "time-out" et "check-sum" câblé.

Les méthodes d'analyse sémantique [Cousot 78] et de preuves [Hoare 72] de programmes, permettent dans un certain nombre de cas de garantir statiquement que certaines assertions seront vraies à l'exécution. Quand l'analyseur statique ou le démonstrateur sont incapables de décider si une assertion sera vraie ou fausse, I_0 doit exécuter un test dynamique* (run-time check). Parmi les assertions qui devront toujours être vérifiées dynamiquement figurent notamment celles qui vérifient des invariants liés aux opérations d'entrée/sortie.

Si durant une opération sur une variable v de type concret, une assertion qui vérifie le respect de l'invariant associé au type de v est trouvée fausse, nous dirons qu'il y a détection d'une exception de niveau 0.

II - 1.3. Spécification de la sémantique d'une opération

Les spécifications de I_0, décrivant la sémantique des opérations définies sur les types concrets, mentionnent les exceptions pouvant survenir pendant ces opérations. Ces exceptions portent des noms suggestifs comme *RANGE-ERROR* ou *ARITHMETIC-OVERFLOW*, etc.

---

* Les vérifications dynamiques allongent le temps d'exécution. L'idéal serait de pouvoir s'en passer par des moyens "statiques". Mais prouver statiquement qu'une certaine assertion sera vraie à l'exécution est un problème en général insoluble.

Soit *OP* une opération spécifiée pour un type concret, dont l'exécution peut produire les exceptions E_1, E_2,..., E_k.

> *L'effet d'une exécution de OP sans détection d'exception, sera appelé effet standard de OP. L'effet d'une exécution durant laquelle on détecte une exception E_i sera appelé effet exceptionnel E_i.*

Les spécifications d'une opération *OP* doivent être structurées de manière à différencier entre l'effet standard et les K effets exceptionnels possibles, correspondant à l'occurrence des K exceptions distinctes. Par exemple, les spécifications d'une opération d'écriture sur DISK peuvent être structurées ainsi :

ext procedure WRITE (B : BLOC; BUF : DISKBLOC); signals INCORRECT-BLOC, TRANSMISSION ;

exceptions : INCORRECT-BLOC si B ∉ [1,&NB]
B garde l'état d'avant l'appel
TRANSMISSION si erreur de parité ou erreur de cadence ou time-ou
B est laissé dans un état indéfini, l'opération peut être ré-essayée

standard : au bout d'un temps t ≤ T l'état externe de B devient égal à l'état interne de BUF

Une opération, pour laquelle tous les effets exceptionnels possibles sont spécifiés, sera dite totale. Par exemple, si un disque physique pouvait se trouver dans l'état "hors tension", l'opération précédemment spécifiée ne serait pas totale. Elle deviendrait totale par adjonction d'une troisième exception.

> *INTERVENTION si le disque d'adresse &AD (§II.1.1) est hors tension*
> *B garde l'état d'avant l'appel, avant de ré-essayer l'opération il faut intervenir pour mettre le disque &AD sous tension*

Soit *v* une variable et *OP* une opération dont l'effet standard consiste à modifier l'état de *v*. Si tous les effets exceptionnels E_i préservent l'état que *v* avait avant l'activation de l'opération, alors l'opération *OP* est atomique. Par exemple, si l'effet exceptionnel TRANSMISSION avait été de préserver l'état que le bloc B avait avant l'appel, l'opération WRITE aurait été atomique.

II - 1.4. Comment signaler l'occurrence d'une exception ?

Soit P un programme composé d'un ensemble de déclarations de variables v_i appartenant à des types concrets et d'une séquence d'opérations $O_1\ O_2\ \ldots O_p$ sur ces variables. Entre l'opération O_i et l'opération O_{i+1}, l'état de P est défini comme l'agrégation des états externes de toutes ses variables. La transformation d'état standard spécifiée pour P est la "somme" des effets standards des opérations O_i. Une détection d'exception E durant l'activation d'une opération O_i met en cause la possibilité de réaliser la transformation standard de P.
Face à l'occurrence de E, il y a deux attitudes possibles.

1) La progression du contrôle dans la séquence $O_1,..,O_p$ va être arrêtée sur O_i. L'identification de l'exception est imprimée sous forme d'un message d'erreur. Les variables v_i sont laissées dans l'état qu'elles avaient après l'exécution de O_i. Ceci est "ennuyeux" si l'état final, spécifié pour les v_i à la fin de l'exécution standard de $O_1\ O_2..\ O_p$, se trouve être l'état initial d'autres programmes : P_1, P_2...
   C'est le cas des variables v_i rémanentes ou partagées.

2) L'autre réaction, consiste à commuter le contrôle sur une séquence de traitement t de E, au lieu d'exécuter la prochaine instruction O_{i+1}. Le traitant t de l'exception E peut avoir pour but de placer les variables v_i dans un état préspécifié, sans danger pour l'exécution des programmes P_1, P_2,..., et éventuellement de P lui même.

L'attitude 2) est meilleure que l'attitude 1) chaque fois qu'il s'agit de variables partagées ou rémanentes. Mais comment réaliser l'activation de t quand E est détectée? Là aussi il y a deux approches possibles.

A) I_O peut indiquer à P l'occurrence de E par un code condition, valeur de retour "inhabituelle", fonction booléenne, etc. Après chaque opération O_i, dans P on doit tester explicitement l'indicateur spécifié, pour voir si une exception a été détectée. Si c'est le cas, il y a un branchement explicite à t, sinon on exécute O_{i+1}. Le rôle du "maître" est tenu par P qui utilise I_O :
   - c'est lui qui est à l'écoute des indicateurs mis à jour par I_O
   - la décision de rompre la séquence standard est explicitement prise dans P.

B) L'autre stratégie possible, confie le rôle du maître à I_0. C'est lui qui "connaît" le premier l'occurrence de E, car c'est lui qui la détecte. C'est lui qui forcera la commutation du contrôle sur le traitant t, fourni par P.
Nous dirons que I_0 signale la détection de l'exception E à P par l'activation du traitant t.

Du point de vue de P, la détection de l'occurrence de E est implicite : il y a exécution de t à la place de O_{i+1}. C'est à la fin de l'exécution de t que le problème du retour à la séquence standard de P se posera (voir §V.4). La structure de contrôle de I_0 spécialisée à signaler des exceptions est le mécanisme d'exception. Un mécanisme de déroutement d'un processeur matériel, le ON-CONDITION de PL1 [IBM 70], le SIGNAL-ENABLE de BLISS [Wulf 71] sont autant d'exemples de mécanismes d'exception.

L'approche B) a beaucoup d'avantages par rapport à A). Citons-en deux qui nous paraissent vraiment importants :

- lisibilité et modifiabilité : l'algorithme standard d'un programme P est syntaxiquement séparé des algorithmes d'exception. Ces derniers peuvent évoluer au cours de l'existence de P du plus simple au plus sophistiqué, indépendemment de l'algorithme standard. Cette séparation augmente également la facilité de lecture d'un programme.

- protection dynamique : il est en général impossible de prouver statiquement que tous les invariants associés aux variables d'un programme P seront respectés . L'utilisation d'un mécanisme d'exception évite que l'éventuelle violation dynamique de ces invariants reste non détectée parce que le programmeur de P "oublie" de tester les indicateurs d'exception de I_0, qui signalent à notre avis "trop gentiment" ces violations.

Pour décrire au §II-2, à l'aide d'un exemple, la conséquence qu'une occurrence dynamique d'exception a sur l'état d'un programme, nous supposons notre langage de programmation muni du mécanisme d'exception du § V, sans toutefois donner plus de précisions sur sa sémantique qu'il n'est nécessaire pour comprendre cet exemple. Nous faisons cette entorse à la logique de la présentation, car nous voulons étudier d'abord les conséquences des exceptions, avant de proposer des remèdes aux §IV et V.

II-1.5. Notation pour associer un traitant à une occurrence d'exception.

Soit *OP* une opération pouvant signaler les exceptions $E_1, E_2, .., E_k$. Pour désigner le traitant qui doit être activé par le mécanisme d'exception, intégré à I_0, lorsqu'une exception E_i est détectée, nous utiliserons la notation introduite dans [Goodenough 75]*. Les règles exactes permettant aux programmeurs d'associer des traitants à des occurrences d'exceptions seront énoncées au §V . Pour l'instant, nous préférons rester à un niveau"intuitif" en disant que la formule :

$$OP\ [E_1 : T_1,\ E_2 : T_2, \ldots,\ E_k : T_k]$$

signifie que si pendant une activation de *OP* l'exception E_i est détectée, alors le traitant qui doit être activé par I_0, à la place de l'instruction qui suit *OP*, est T_i. Si l'utilisateur du type concret *DISK* (§II-1.3) veut fournir des traitants pour les exceptions qui peuvent être signalées par une activation de *WRITE*, il pourra par exemple, écrire :

WRITE (B, BUF) [INCORRECT-BLOC : T_1, TRANSMISSION : T_2, INTERVENTION : T_3];

Un traitant peut être associé à une instruction simple (affectation, appel de procédure), ou peut être associé à toute une instruction composée (instruction bloc, itérative, etc..) . La formule :

while ( ) do begin
.
.
OP;
.
.
end $[E_1 : T_1, \ldots, E_k : T_k]$

signifie que si durant une activation de *OP*, l'exception E_i est détectée, alors I_0 n'exécutera plus le reste de la boucle, mais le traitant T_i, associé à la plus petite unité syntaxique contenant le point d'activation de *OP* (voir§V).

Le contexte d'un traitant T_i (l'ensemble de variables accessibles dans T_i) est égal au contexte du point d'activation de *OP* .

---

* En SESAME, les séparateurs " [" et "]" désignent aussi les fonctions d'accès aux éléments de tableau, d'où une possible confusion. Dans cette thèse, nous avons adopté cette notation pour sa concision.

II - 2. EXCEPTIONS DEFINIES PAR LE PROGRAMMEUR

Les types qui ne sont pas implémentés par I_0 (c'est-à-dire ceux qui sont "abstraits" [Liskov 74]) doivent être implémentés par des modules écrits par le programmeur. Celui-ci:

- connaît un type abstrait T uniquement à travers ses spécifications
- manipule toute variable "abstraite" appartenant à T uniquement à travers les opérations spécifiées pour T.

Dans le but d'étudier l'influence que l'occurrence dynamique des exceptions peut avoir sur l'état des modules qui implémentent des variables abstraites, nous avons choisi d'utiliser comme "matériel pédagogique" de ce chapitre deux des types abstraits qui composent le système multi-accès du §VI.Ces deux types, *RESSOURCES* et *SEGMENTS*,vont nous permettre d'introduire deux concepts fort utiles pour le traitement d'exception : les concepts d'état interne incohérent et d'ensemble d'incohérence. Pour décrire ces deux types abstraits, nous avons étendu la technique de spécification opérationnelle proposée dans [Wulf 76] de façon à pouvoir spécifier non seulement l'effet standard des opérations d'un type abstrait, mais également leurs effets exceptionnels.

II - 2.1. Spécification d'un type abstrait

La spécification opérationnelle d'un type T comprend

- la description des états externes (ou abstraits) qu'une variable de type T peut assumer,
- la description de l'état externe initial
- la description de la sémantique des opérations associées à T en termes de pré- et postconditions portant sur l'état abstrait a' d'avant l'opération et l'état a d'après.

a) Description des états externes

Un état externe a est décrit en termes d'objets mathématiques comme les ensembles, intervalles, fonctions, produits cartésiens etc. L'ensemble des états externes possibles est caractérisé par un prédicat abstrait I_a (appelé l'invariant abstrait).

b) L'état externe initial a_0, est l'état externe que toute variable de type T possède au moment de sa création.

c) La description sémantique des opérations abstraites fait intervenir la sémantique des opérations sur les objets mathématiques à l'aide desquels on représente les états abstraits a. Soit $OP(t) \rightarrow r$ une des opérations abstraites définies sur T, ayant comme paramètre d'entrée t et délivrant le résultat r. Dans [Wulf 76], l'effet de OP (que nous allons appeler par la suite standard) n'est défini que si une certaine précondition abstraite $prea(a',t)$ est vraie. Dans ce cas, l'état abstrait ultérieur a et le résultat délivré r sont décrits par une postcondition abstraite $posta(a,a',t,r)$. En utilisant la notation de [Hoare 69], cela revient à écrire

$$prea(a',t)\ \{OP(t) \rightarrow r\}\ posta(a',a,t,r)$$

Mais que se passe-t-il si la précondition n'est pas vraie ? Quel sera l'effet d'une activation de OP avec $prea = faux$?
Supposons que la précondition $prea$ peut être écrite comme un produit d'assertions plus simples

$$prea = prea_1 \wedge prea_2 \wedge \ldots \wedge prea_K$$

Nous dirons qu'une activation de OP telle que

$$\exists i \in [1,k] : \bigwedge_{j=1}^{i-1} prea_j(a',t) \wedge \neg prea_i(a',t)$$

produit l'exception E_i. L'utilisateur de OP connaît les différentes exceptions possibles par autant d'identificateurs d'exception distincts $E_1, E_2, \ldots, E_k$.

Une activation de OP qui produit l'exception E_i, aura deux sortes de conséquences :

- un effet sur l'état a et le résultat r,
- un effet sur la prochaine opération qui va être exécutée dans le programme qui a activé OP.

La première conséquence peut être décrite dans les mêmes termes que l'effet standard, en spécifiant l'état a et le résultat r après une production de E_i par une postcondition $posta\ E_i(a',a,t,r)$:

$$\exists i: \bigwedge_{j=1}^{i-1} prea_j(a',t) \wedge \neg prea_i(a',t)\ \{OP(t) \rightarrow r\}\ posta\ E_i(a,a',t,r)$$

$$\forall i: \quad prea_i(a',t)\ \{OP(t) \rightarrow r\}\ posta\ (a,a',t,r)$$

Nous ne pouvons pas décrire le deuxième type d'effet à l'aide du formalisme proposé dans [Wulf 76], qui est adapté à spécifier des données abstraites, mais inadapté à spécifier l'abstraction "contrôle". Nous le décrirons au §V en donnant la sémantique du mécanisme d'exception (dont le rôle est précisément de réaliser à l'exécution ce "deuxième effet").

Contentons nous, pour l'instant, d'adopter une notation qui va nous permettre de décrire les effets exceptionnels et standard d'une activation de *OP* (uniquement) sur a et r.

function $OP(t) \rightarrow r$

$E_1 = \neg \underline{prea}_1(a', t)$

$\underline{posta}\ E_1(a', a, t, r)$

$E_2 = \neg \underline{prea}_2(a', t)$

$\underline{posta}\ E_2(a', a, t, r)$

.

.

.

$E_K = \neg \underline{prea}_K(a', t)$

$\underline{posta}\ E_K(a', a, t, r)$

$standard = \underline{prea}_1 \wedge \underline{prea}_2 \wedge .. \wedge \underline{prea}_K(a', t)$

$\underline{posta}(a', a, t, r)$

Dans certains cas, si il y a production d'une exception E_i, aucune valeur de retour n'est calculée. Nous allons noter cela par $r = < >$.

Une opération abstraite dont l'effet est défini pour tout état abstrait initial a' et toute valeur du paramètre t sera dite <u>totale</u> *. Si* $\forall i \in [1, K]$: $\underline{posta}\ E_i(a', a, t, r) \equiv (a = a') \wedge (r = < >)$, *l'opération est dite* <u>atomique</u>.

Si toutes les opérations qu'on spécifie sont totales et atomiques, on peut simplifier la notation introduite précédemment. Par exemple, nous pouvons nous passer d'écrire la précondition standard, car elle se déduit des préconditions E_i par la formule

$$standard = \neg(E_1 \vee E_2 \vee .. \vee E_K)$$

Quand toutes les postconditions exceptionnelles sont identiques, nous n'en écrirons qu'une seule.

La Figure 1. est un exemple de spécification opérationnelle du type abstrait RESSOURCES, utilisé dans le système du §VI pour gérer un "tas" de ressources équivalentes, identifiées par des entiers entre *1* et *&N*.

Si *E* est un ensemble, nous notons *P(E)* l'ensemble des parties de *E* et *card (E)* la cardinalité de *E*.

```
type abstrait RESSOURCES (&N : integer);
début nécessite &N > 0;
      états a ∈ P({1,2,..,&N});
      invariant 0 ≤ card(a) ≤ &N;
      initial a0 = { }
      opérations function ALLOUER → i : integer;
                 DEBORD    prea a' = {1,2,..,&N}
                           postax (a = a') ∧ (i = < >)
                 standard  prea ∃r ∈ {1,2,...,&N} -a'
                           posta (a = a' ∪ {r}) ∧ (i = r)
                 procedure LIBERER (r : integer);
                 RANGE-ERROR prea r ∉ {1,2,..,&N}
                 ILLEGAL     prea r ∉ a'
                             postax   a = a'
                 standard    posta    a = a' - {r}
fin spécification ;
```

Figure 1.

II - 2.2. Implémentation

Une variable a appartenant à un type abstrait T va être implémentée par un module. Une description de module contient :

1) Des déclarations de variables rémanentes mémorisant l'état interne e. Une variable rémanente peut appartenir à un type concret implémenté par I_0 , ou peut appartenir à un autre* type abstrait et être implémentée par un autre module.

2) Des déclarations de fonctions et procédures externes qui implémentent chaque opération abstraite spécifiée pour T comme une séquence d'opérations sur les variables rémanentes du module.

3) Une séquence d'initialisation, qui place le module au moment de sa création dans un état interne e_0 .

Soit M l'ensemble des modules d'un programme exécutable, et $M, N \in M$. Nous supposons que la relation "le module M appelle (ou utilise) le module N" est une relation d'ordre (partiel) sur l'ensemble M. [Parnas 74]. Le programme I_0, qui implémente tous les types concrets prédéfinis dans SESAME (§II.1.1) est en réalité un ensemble de modules : $I_0 \subset M$. En effet, I_0 consiste en un module "code objet généré par le compilateur plus noyau d'exécution résident" [Montuelle 77], et plusieurs modules de bibliothèque qui implémentent les variables de type périphérique. Les modules de I_0 seront dits de niveau d'abstraction 0. Un module qui utilise uniquement des modules de niveau 0 sera dit de niveau d'abstraction 1 et la réunion de tous les modules de niveau 1 et de niveau 0 sera notée I_1. Par induction nous dirons qu'un module qui utilise uniquement des modules de I_k ,et parmi ces modules au moins un module de niveau k , est au niveau d'abstraction $k+1$ .La réunion de I_k avec l'ensemble des modules de niveau $k+1$ sera notée I_{k+1} .

L'état interne d'un module M de niveau $h > 0$ est défini comme l'agrégation des états externes de toutes ses variables rémanentes, implémentées par des modules de niveau $k (0 \leq k < h)$. Entre deux appels, l'état interne de M est dit stable. Les états internes par lesquels M passe durant une exécution de fonction externe sont dits intermédiaires

---

* Nous ne considérons pas dans cette thèse des types abstraits récursifs.

A l'invariant abstrait I_a, défini sur les états externes, correspond un invariant concret I_c, défini sur les états internes. L'état interne initia e_0 doit vérifier I_c, et n'importe quel état interne stable, pouvant être atteint par une séquence d'appels à M, doit également vérifier I_c.
Dans le but de préserver I_c, l'effet standard d'une activation de fonction externe A de M n'est disponible que si un certain nombre d'assertions concrètes sont vraies à l'exécution de A. (figure 2)

Comme en général il est impossible de garantir statiquement que toutes ces assertions seront vraies à l'exécution, le programmeur doit prendre en compte leur possible violation. Suivant leur position géographique, les assertions concrètes peuvent être :

a) Des assertions d'entrée. Ces assertions portent sur :
   - les paramètres effectifs de l'appel
   - l'état des variables rémanentes au moment de l'appel

b) Des assertions intermédiaires. Ces assertions concernent l'issue des activations d'opérations O_i sur des variables implémentées par les niveaux d'abstraction inférieurs. Il est en général impossible de savoir avant l'activation d'une opération O_i si cette opération produira ou non une exception (par exemple, le résultat d'une opération sur une variable périphérique ne peut être connu qu'après son exécution). Il faut donc s'attendre dans M, à ce que certaines activations d'opérations $N.B$, implémentées par des modules N de niveau k $(k = 0,1,..,h-1)$ puissent produire des exceptions EK, spécifiées pour ces modules (Fig.2)

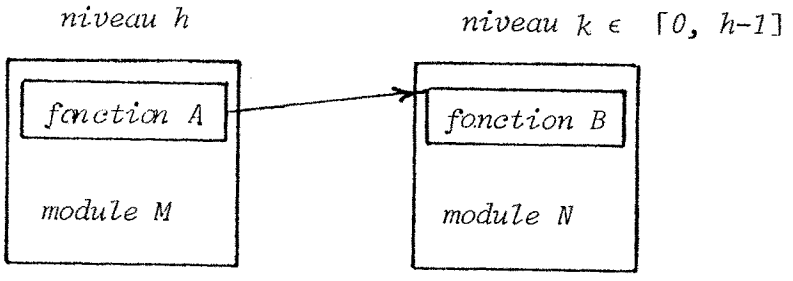


Figure 2.

Au § II.1.5 nous avons supposé que les exceptions prédéfinies, spécifiées pour les modules de niveau O, sont signalées via le mécanisme d'exception. Le même mécanisme doit également servir* à signaler les exceptions détectées dans n'importe quel module N de niveau $k = 1,2,.., h-1$ utilisé par M. L'inclusion d'un mécanisme d'exception dans le langage d'implémentation, rend la détection d'une violation d'assertion intermédiaire implicite : au lieu d'exécuter l'opération O_{i+1} suivant l'activation de $N.B$, I_o active le traitant TH associé à l'exception EK :

O_{i-1};
$N.B$ [EK:TH];
O_{i+1};

c) Des assertions de sortie portant sur
- les résultats exportés vers l'extérieur
- l'état interne du module en fin d'activation

Si durant une activation du module M, une assertion (a),(b),(c) est détectée fausse (soit par un test explicitement programmé par le concepteur de M, soit implicitement via le mécanisme d'exception) nous dirons qu'il y a une détection d'exception de niveau h dans M.**

---

* Voir § V.

** En dépit du fait qu'au moment de sa détection dans N, l'exception EK est de niveau k, l'arrivée du signal EK dans M doit être interprétée comme la détection (implicite) d'une autre exception au niveau h. En effet, si le service standard de $N.B$ n'est pas disponible, le service standard de $M.A$ ne pourra pas être rendu non plus. Dans ces circonstances, $M.A$ rendra un service différent du service standard, donc un service exceptionnel. Le concepteur de M doit identifier le nom de ce service exceptionnel par un identificateur d'exception EH de niveau h. L'arrivée du signal de EK dans M signifie donc la détection implicite de EH. Ce phénomène, dû à la propagation des exceptions via le mécanisme d'exception, sera discuté au § II.2.6 et au § III.2.1 et illustré par les exemples du § VI.

Suivant l'état de M au moment de leur détection, les exceptions peuvent être classifiées en :

1) *Exceptions d'entrée, détectées par des tests (a). L'état interne du module est égal à l'état interne stable d'avant l'appel e'.*

2) *Exceptions intermédiaires, détectées par des tests (b) ou (c). Au moment de leur détection, M se trouve dans un état intermédiaire e ≠ e'.*

Une détection (implicite ou explicite) d'exception EH dans M, va être signalée à l'utilisateur de M par une primitive du mécanisme d'exception

signal EH;

La sémantique de cette opération sera écrite en détail au § V . Limitons nous pour l'instant à dire qu'elle provoque un retour "exceptionnel" dans le module qui a appelé *M.A*. Ce retour est exceptionnel car au lieu d'exécuter (comme normalement) l'instruction qui suit l'activation de *M.A*, on exécute le traitant *T* associé à l'occurrence de *EH* :

M.A [EH : T];

Un type abstrait est décrit en SESAME par un modèle de module, c'est-à-dire par une description paramétrée d'un module compilable, et ensuite exécutable. Le modèle de module, à partir duquel on peut générer (par des déclarations dans un programme de connexion [Cheval 76]) tous les modules qui implémenteront des variables abstraites de type *RESSOURCES*, possède la métavariable *&N* = "nombre de ressources à gérer". Un identificateur de métavariable, commence par "*&*". Le mot clé *class*, indique que dans cette implémentation, il n'y aura aucune contrainte de synchronisation* sur les activations d'ALLOUER et LIBERER (voir § VI.2.4).

---

* L'adoption du moniteur [Hoare 74] comme mécanisme de synchronisation en SESAME, ne permet pas de dissocier la description d'une implémentation de type abstrait de la description des contraintes de synchronisation sur l'accès aux variables abstraites appartenant à ce type.

```
class RESSOURCES (&N : integer);
type NOM-RESSOURCE = 1.. &N;
     ETAT-RESSOURCE (LIBRE,OCCUPE);
     ETAT-INTERNE = array [NOM-RESSOURCE] of ETAT-RESSOURCE;
var  E : ETAT-INTERNE;

ext function ALLOUER : integer; signals DEBORD; %declaration d'excep-
                                                 tion connue à l'extérieur(voir §V %
var  I : integer;
begin I := 1 ;
     while (I ≤ &N) and E(I) = OCCUPE do I := I+1;
     if I > &N then signal DEBORD; %les instructions qui suivent ne
                                    sont plus exécutées, voir §V %
     E [I] := OCCUPE;
     ALLOUER := I
end;

ext procedure LIBERER (I : integer); signals RANGE-ERROR, ILLEGAL;
begin if (I < 1) or (I > &N) then signal RANGE-ERROR;
     if E [I] = LIBRE then signal ILLEGAL;
     E [I] := LIBRE
end ;

begin for I := 1 to &N do E [I] := LIBRE; %séquence d'initialisation de
                                           l'état interne%
end;
```

Figure 3.

II - 2.3. Hiérarchies

Dans ce chapitre nous allons spécifier un nouveau type abstrait : *SEGMENTS** Ensuite, nous allons montrer comment il est possible d'implémenter ce type à l'aide du type abstrait *RESSOURCES* et d'autres types de niveau *0*. Ceci nous permettra d'étudier au §II.2.4 l'influence que l'occurrence dynamique des exceptions peut avoir sur les états internes des modules qui implémentent des variables abstraites.

---

*Le concept de segment est utilisé dans un système pour créer un espace adressable contigu, composé de pages, à partir d'un ensemble de morceaux de mémoire non contigus, appelés blocs.

La fig.4 contient la spécification opérationnelle de *SEGMENTS*. Dans cette spécification, nous utilisons un certain nombre de notations que nous explicitons ci-dessous :

Notations :

Soit *&NS* (Nombre de Segments), *&NB* (Nombre de Blocs) et *&NP* (Nombre de Pages) trois entiers positifs, et $N = \lceil 1, \&NS \rceil$, $B = \lceil 1, \&NB \rceil$, $P = \lceil 1, \&NP \rceil$ des intervalles d'entiers. Notons avec $P \rightarrow B$ l'ensemble de fonctions définie sur P et à valeurs dans B.

Un segment sera défini comme une restriction de fonction $f \in P \rightarrow B$ à un domaine $\lceil 1, T \rceil \subset P$. T est la taille du segment, $\lceil 1,T \rceil$ l'ensemble des pages du segment et $\{f(i) \mid i \in \lceil 1,T \rceil\}$ l'ensemble des blocs du segment.

Soit $R = \bigcup_{T=1}^{\&NP} (\lceil 1,T \rceil \rightarrow B)$ l'ensemble de toutes les restrictions de fonctions $f \in P \rightarrow B$ à des intervalles $\lceil 1,T \rceil \subset P$ non vides. Par la suite, nous allons noter le domaine de définition d'une fonction $f \in P \rightarrow B$ par $\underline{def}\ f = P$ et l'ensemble de valeurs de f par $\underline{val}\ f = \{f(i) \mid i \in P\}$. Conformément à la définition de R :

$\forall r \in R \quad \exists T_r \in P : \underline{def}\ r = \lceil 1,T_r \rceil$ *et* $\underline{val}\ r = \{r(i) \mid i \in \lceil 1,T_r \rceil\} \subset B$

Soit $na \in P(N)$ une partie de l'ensemble N. L'ensemble na, qui n'est pas forcément un intervalle, sera appelé par la suite l'ensemble d'indexes de segments. Notons $ds = \{r_i \mid i \in na \wedge r_i \in R\}$ un ensemble indexé par na de segments et $ba = \bigcup_{r_i \in ds} \underline{val}\ r_i$ l'ensemble des blocs de tous les segments.

Un état externe s d'une variable de type *SEGMENTS*, sera représenté comme un triplet $s = (na, ds, ba)$

```
type abstrait SEGMENTS (&NS, &NB, &NP : integer);
début nécessite &NS, &NB, &NP > 0;
      états s = (na, ds, ba)
      invariant I_na ∧ I_ba ∧ I_ds où
            (I_na) card (na) = card (ds) ≤ &NS
            (I_ba) card (ba) = card (∪_{r_i ∈ ds} val r_i) ≤ &NB
            (I_ds) ∀i,j ∈ na : (i≠j) => val r_i ∩ val r_j = { }
      initial s_0 = ({ }, { }, { });
      operations function CREER (T ∈ P) → i ∈ N
                  RANGE-ERROR prea T ∉ P
                  DEBORDEMENT1 prea card (N-na') = 0
                  DEBORDEMENT2 prea card (B-ba') < T
                               postax (s = (na', ds', ba')) ∧ (i =< >)
                  standard     prea ∃r_a ∈ R : (a ∈ N-na') ∧ (def r_a = [1,T]) ∧
                                   ∧(val r_a = {b_1, b_2, .., b_T} ⊂ B-ba')
                               posta (s = (na' ∪ {a}, ds' ∪ {r_a}, ba' ∪ val r_a)).
                                    ∧(i = a)
                  procédure DETRUIRE (i ∈ N);
                  RANGE-ERROR  prea i ∉ N;
                  ILLEGAL      prea i ∉ na'
                               postax s = (na', ds', ba')
                  standard     posta  s = (na' - {i}, ds' - {r_i}, ba' - val r_i)
                  function     LIRE (i ∈ N) → r ∈ R
                  RENGE-ERROR  prea i ∉ N;
                  ILLEGAL      prea i ∉ na'
                               postax (s = (na', ds', ba')) ∧ (r =< >)
                  standard     posta (s = (na', ds', ba')) ∧ (r = r_i)
fin specifications;
```

Figure 4.

L'invariant abstrait (I_{na}) exprime le fait qu'il peut y avoir au plus *&NS* segments, (I_{ba}) dit que l'ensemble des blocs de tous les segments doit être inférieur à *&NB* et (I_{ds}) dit que deux segments ne peuvent avoir de blocs communs.

Si l'on veut implémenter *SEGMENTS*, il est nécessaire de choisir d'abord les structures de données rémanentes qui vont mémoriser les états internes, correspondant aux états externes spécifiés. Aux intervalles "abstraits" *N, B, P* correspondront les intervalles "concrets"

```
type NOMPAGE = 1.. &NP;
     NOMBLOC = 1.. &NB;
     NOMSEGMENT = 1.. &NS;
```

La famille de fonctions $P \rightarrow B$ peut être représentée concrètement [Hoare 72a], par le type tableau :

```
type MAPPING = array [NOMPAGE] of NOMBLOC;
```

Un segment n'est pas une fonction de type *MAPPING*, mais une restriction d'une telle fonction à un intervalle [*1, TAILLE*]

```
type TAILLE = 0.. &NP;
```

Toute restriction de fonction peut être décrite par la donnée de son domaine et de la correspondance pages → blocs, c'est-à-dire par un élément du produit cartésien "tailles possibles" X "correspondances possibles". Le type concret qui peut être utilisé pour implémenter des produits cartésiens abstraits est l'enregistrement [Hoare 72a] :

```
type SEGMENT = record T : TAILLE;
                      M : MAPPING;
               end;
```

Les deux variables abstraites *na* et *ba*, dont les états possibles sont *P(N)* et *P(B)* , peuvent être implémentées par deux variables *NOMS* et *BLOCS* de type *RESSOURCES* (§ II.2.1).
La variable abstraite *ds*, qui est un ensemble indexé, sera le plus "naturellement" représentée par un tableau dont l'index est de type *NOMSEGMENT*.
Le choix pour la représentation des états internes étant fait, on peut commencer à écrire le modèle de module qui implémente le type *SEGMENTS* (Figure 5.).

```
monitor SEGMENTS (&NS, &NB, 1NP : integer);
type      NOMPAGE = 1.. &NP; NOMBLOC : 1.. &NB; NOMSEGMENT : 1.. &NS;
          MAPPING = array [NOMPAGE] of NOMBLOC;
          TAILLE = 0.. &NP;
          SEGMENT = record T : TAILLE;
                           M : MAPPING
                    end;
% unique* BLOCS : RESSOURCES (&NB); %
ref function BLOCS.ALLOUER : integer; signals DEBORD;
ref procedure BLOCS.LIBERER (B : integer); signals RANGE-ERROR, ILLEGAL;
% unique NOMS : RESSOURCES (&NS); %
ref function NOMS.ALLOUER : integer; signals DEBORD;
ref procedure NOMS.LIBERER (N : integer); signals RANGE-ERROR, ILLEGAL;
var DS : array [NOMSEGMENT] of SEGMENT;
est function CREER (TA : NOMPAGE) : NOMSEGMENT; signals RANGE-ERROR,
            DEBORDEMENT1, DEBORDEMENT2;
% comme TA appartient au type concret intervalle, RANGE-ERROR sera
  détectée implicitement %
var NS : NOMSEGMENT; I : integer;
begin procedure  TDEBORD; begin % le corps de cette procédure est décrit
                                  au §II.2.6 % end;
      NS := NOMS.ALLOUER [NOMS.DEBORD : signal DEBORDEMENT1];
      with DS [NS] do
      begin T := TA; I := 1;
            while I <= TA do
            begin M [I] := BLOCS.ALLOUER;(*)
                  I := I+1
            end [BLOCS.DEBORD   : TDEBORD]
      end;
      CREER := NS
end;
```

Figure 5.(DEBUT)

* L'occurrence du mot-clé *unique* dans une déclaration de variable abstraite indique l'absence de toute synchronisation des accès à cette variable. Les variables abstraites partagées entre plusieurs processus, sont déclarées par le mot-clé *shared*. Le connecteur vérifie que les variables non partagées sont implémentées par des *classes*, et les variables partagées sont implémentées par des *moniteurs*.

```
ext procedure DETRUIRE (NS : NOMSEGMENT); signals RANGE-ERROR, ILLEGAL;
with DS [NS] do
begin if T = 0 then signal ILLEGAL;
      for I := 1 to T do BLOCS.LIBERER (M[I]); (**)
      NOMS.LIBERER (NS)
end;

ext function LIRE (NS : NOMSEGMENT) : SEGMENT; signals RANGE-ERROR, ILLEGAL;

begin if DS [NS] T = 0 then signal ILLEGAL;
      LIRE := DS [NS]
end;

begin for I := 1 to &NS do DS[I]. T := 0
      %Les variables BLOCS et NOMS sont initialisées quand elles sont crées
       par le connecteur %
end.
```

Figure 5(FIN)

II - 2.4. Etats internes incohérents

L'état interne d'une variable abstraite de type *SEGMENTS*, est l'agrégation des états externes de toutes les variables rémanentes du module qui l'implémente. Si l'on note

$$N = \{n \mid 1 \leq n \leq \&NS\}$$
$$T = \{t \mid 0 \leq t \leq \&NP\}$$
$$B = \{b \mid 1 \leq b \leq \&BB\}$$

$P(N)$, $P(B)$ l'ensemble des états externes des variables *BLOCS* et *NOMS* alors l'ensemble des états internes de *SEGMENTS* sera

$$E = (T \times (B)^{\&NP})^{\&NS} \times P(N) \times P(B)$$

L'invariant concret I_c est une fonction booléenne de l'état interne.

Soient I_{cna} : $\sum_{n=1}^{\&NS} \underline{positif}(DS[n].T) = \underline{card}(NOMS)$ où $\underline{positif}(n \in \mathbb{N}) = \begin{cases} 1 & si\ n > 0 \\ 0 & sinon \end{cases}$

I_{cba} : $\sum_{n=1}^{\&NS} DS[n].T = \underline{card}\ (BLOCS)$

I_{cds} : $\forall n,\ m \in [1, \&NS],\ \forall i \in [1, DS[n].T],\ \forall j \in [1, DS[m].T]$

$(DS[n].M[i] = DS[m].M[j]) \Rightarrow (n=m) \wedge (i=j)$

L'invariant concret de *SEGMENTS* est

$I_c = I_{cna} \wedge I_{cba} \wedge I_{cds}$

Un état interne $e \in E$ vérifiant l'invariant concret I_c sera dit <u>cohérent</u>.

Supposons maintenant que durant un appel à *SEGMENTS.CREER* , une activation (*) (fig.5) de *BLOCS.ALLOUER* produit l'exception de niveau 1 *BLOCS.DEBORD*. Le mécanisme d'exception, inclus dans le langage d'implémentation, <u>propagera</u> le signal d'exception levé en *BLOCS* , en le transformant en une détection implicite de l'exception de niveau 2 *DEBORDEMENT2*.

Au moment où *DEBORDEMENT2* sera détectée par l'exécution du traitant *TDEBORD* associée à *BLOCS.DEBORD* , le module *SEGMENTS* se trouvera dans un état intermédiaire *e*. Si la taille du segment NS en cours de création était *TA*, le nombre de blocs qui ont été effectivement alloués ne sera que $I - 1 < T$ (où I est l'entier local à *CREER*). L'état intermédiaire est tel que I_{cba} est faux.

Un état interne $e \in E$ qui ne vérifie pas l'invariant concret I_c sera dit <u>incohérent</u>.

Le concept de <u>fonction de représentation</u> (ou <u>d'abstraction</u>) a été introduit dans [Hoare 72] pour définir la correspondance entre les états internes d'une implémentation particulière, et les états externes spécifiés. Dans l'exemple simple du module SMALLINTSET, décrit dans le célèbre article, l'auteur considère (sans le dire explicitement) que la fonction d'abstraction est toujours définie. Dans cette thèse, nous allons donner un sens légèrement différent à ce concept, en disant qu'une fonction d'abstraction est définie uniquement sur les états internes cohérents et est non définie sur les états internes incohérents. La fonction d'abstraction de *SEGMENTS* est

$$\forall e \in E\ \underline{rep}(e) = \begin{cases} \underline{si}\ \ I_c(e)\ \underline{alors} \\ (\underline{rep}(NOMS), \{S\lceil n\rceil \mid (1 \leq n \leq \&NS) \wedge S\lceil n\rceil.T > 0\},\ \underline{rep}\ (BLOCS)) \\ \underline{si}\ \neg I_c(e)\ \underline{alors} \\ nondéfini \end{cases}$$

où : $\underline{rep}(NOMS) = \{n \mid (1 \leq n \leq \&NS) \wedge (E\lceil n\rceil = OCCUPE)\}$

$\underline{rep}(BLOCS) = \{b \mid (1 \leq b \leq \&NB) \wedge (E\lceil b\rceil = OCCUPE)\}$

L'implémenteur d'un type asbtrait veut que tous les états internes stables soient cohérents. Dans le cas de *SEGMENTS* , cela revient à dire que l'on ne veut pas (par exemple) qu'il y ait une non-concordance entre la taille d'un segment et le nombre de blocs dont il est effectivement composé. Dès que l'on admet qu'il est impossible, en général, d'éviter l'occurrence dynamique des exceptions (voir § II.1.2), il faut admettre la possible occurrence d'états internes incohérents*.

---

* Quelqu'un pourrait remarquer que nous avons mal choisi notre exemple pour montrer "l'inévitable" occurrence d'états internes incohérents, car dans le cas de *SEGMENTS* on aurait justement pu éviter l'occurrence dynamique de *BLOCS.DEBORD* en prévoyant dans *RESSOURCES* une "fonction de légalité" *function CARDINALITE → c : integer; standard (a = a') ∧ (c = card (a'))*

et en testant explicitement à l'entrée dans *CREER* la précondition de *DEBORDEMENT2*. [Guttag 77]
Face à cette remarque, notre justification est la suivante :

1) La solution de la Fig.5 est, dans beaucoup de cas, la seule possible. Ainsi, si un module *M*, écrit par un certain "programmeur d'application" utilise un module système *N*, il est souvent impossible de garantir que *M* va systématiquement tester une "fonction légalité" avant de faire un appel à *N*. De plus, si *N* est un module qui est (ou utilise) un module périphérique il est impossible de programmer dans *N* une fonction de légalité du genre : *Y aura-t'il une erreur de parité lors de la prochaine entrée/sortie ?*

2) Nous avons voulu que les exemples de la partie "théorique" de notre thèse puissent être utilisés dans la partie "application" § VI. Or le module *VA* qui implémente une méthode d'accès séquentiel pour des fichiers sur disque dans le § VI, et dans lequel on ne peut réellement pas éviter l'occurrence dynamique des exceptions lors des transferts avec la mémoire secondaire, est trop complexe pour être un bon "matériel pédagogique" (entre autres, ce module utilise *SEGMENTS*!).

Mais pourquoi se soucier des états internes incohérents ? Parce que les états initiaux des traitants d'exception sont des états incohérents, et l'effet "standard" de ces traitants consiste dans la "réparation" de tels états avant la sortie d'un module. Si l'on veut écrire des traitants, il est donc nécessaire d'étudier les possibles états internes incohérents.

> *Si un module est laissé dans un état incohérent stable, à la suite d'une occurrence d'exception intermédiaire "non traitée", les prochaines activations du module risquent d'avoir des effets imprévisibles, différents de ceux spécifiés. Ces activations peuvent produire des nouvelles violations d'invariants, signalées par des nouvelles exceptions, et conduisant à des nouveaux états incohérents.*

Dans le cas du module SEGMENTS, supposons que le descripteur du segment *NS* soit laissé après la fin de l'exécution de la fonction *CREER* dans l'état où *DEBORDEMENT2* a été (implicitement) détectée:

DS ⌈*NS*⌉.*T* = *TA*

DS ⌈*NS*⌉. *M* ⌈*1*⌉ ∈ *ba*

DS ⌈*NS*⌉. *M* ⌈*I-1*⌉ ∈ *ba*

DS ⌈*NS*⌉. *M* ⌈*I*⌉ = *nondéfini*

DS ⌈*NS*⌉. *M*⌈*&NP*⌉ = *nondéfini*,

Lors d'une future activation de *DETRUIRE(NS)*, il y a un risque non nul pour détecter :

1) L'exception de niveau *0 RANGE-ERROR* au point d'évaluation (**) des expressions *M*⌈*I*⌉ de type *NOMBLOC* pour *J* = *I*, *I+1*,.., *&NP*.

2) L'exception de niveau *1 ILLEGAL*. Cela peut arriver si par suite d'activations antérieures, les variables *M*⌈*I*⌉ gardent des valeurs résiduelles, qui sont des constantes de type *NOMBLOC*, mais au moins un des blocs désignés par ces *M*⌈*J*⌉, *J* := *I*, *I+1*,...,*&NP* a été entre temps alloué à un autre segment.

II - 2.5. Ensemble d'incohérence

Soit M un module de niveau $h > 0$, dont l'état interne initial e' est cohérent. Supposons que durant une activation de fonction externe A de M, une exception intermédiaire EH est détectée dans M.
Nous introduisons le concept d'ensemble d'incohérence de EH pour désigner la partie de l'état actuel e qui est "vraiment" incohérente :

L'ensemble d'incohérence $\varepsilon(EH)$ de l'exception EH est le sous-ensemble de variables d'état de M possédant les deux propriétés suivantes :

$\varepsilon 1$) Si les éléments de $\varepsilon(EH)$ sont restaurés à l'état d'avant l'entrée dans M, le module M retrouve un état interne e_r équivalent à e' : $\underline{rep}(e_r) = \underline{rep}(e')$

$\varepsilon 2$) $\varepsilon(EH)$ est inclus dans tout sous-ensemble de variables d'état de M ayant la propriété ($\varepsilon 1$)

Dans le cas de *DEBORDEMENT2*, si l'état e' d'avant l'appel à *SEGMENTS.CREER* est cohérent, les composants de $\varepsilon(DEBORDEMENT2)$ seront :

$$\varepsilon(DEBORDEMENT2) = \{DS\lceil NS\rceil.T,\ NOMS,\ BLOCS\}$$

En effet, si dans l'état interne e, on restaure les composants de $\varepsilon(DEBORDEMENT2)$ à l'état d'avant *CREER* :

$$DS\lceil NS\rceil.T \leftarrow 0$$
$$na \leftarrow na - \{NS\}$$
$$ba \leftarrow ba - \bigcup_{j=1}^{I-1} \{DS\lceil NS\rceil.M\lceil J\rceil\}$$

on replace le module *SEGMENTS* dans un nouvel état interne e_r, tel que $(IS \underline{and}\ IN \underline{and}\ IB)(e) = vrai$ et $\underline{rep}(e') = \underline{rep}(e_r)$. L'état restauré e_r est différent de l'état initial e', car les composants élémentaires $DS\lceil NS\rceil.M\lceil J\rceil$ $j = 1,2,..,\ I-1$ ont des valeurs différentes dans e' et e_r. Ceci montre que la fonction $\underline{rep}$ n'est pas en général injective.

---

* Nous voulons que l'ensemble ε soit "le plus petit" car nous voulons faire le moins de travail de restauration possible. Toutefois, dans certains cas, on sera obligé d'estimer cet ensemble par un autre ensemble plus grand, et de faire plus de travail que le strict nécessaire (§IV).

Deux états internes qui se projettent par la fonction d'abstraction rep en un même état externe seront dits équivalents.

II - 2.6. Un exemple de traitement d'exception

La possible occurrence de l'exception intermédiaire de niveau *2 DEBORDEMENT2* a été prévue dans *SEGMENTS.CREER* , où on a prévu de "traiter" l'exception de niveau *1 BLOCS.DEBORD*, en associant à tous les points d'activation (*) de *BLOCS.ALLOUER* le traitant unique *TDEBORD*. Au moment où *TDEBORD* est activé, l'état interne de *SEGMENTS* est incohérent. Son rôle va être d'implémenter l'effet exceptionnel spécifié *postax* $(s = (na', ds', ba')) \wedge (i = <>)$. Le deuxième effet, toujours sous-entendu, est de signaler l'occurrence de *DEBORDEMENT2* à l'utilisateur de *SEGMENTS*.

Pour restaurer un état interne équivalent avec l'état stable initial, *TDEBORD* va restaurer les éléments de ε*(DEBORDEMENT2)* à l'état qu'ils avaient avant l'appel à *CREER*. Comme le contexte de *TDEBORD* est égal au contexte de la fonction *CREER* (§II.1.5), cela est tout à fait faisable. L'effet $i = <>$ est assuré par le mécanisme d'exception (voir §V), qui ne réalise pas l'affectation de la valeur retournée par suite d'une activation "exceptionnelle" de fonction.

```
procedure TDEBORD;
with DS [NS] do
begin NOMS.LIBERER (NS);
      I := I-1; T := 0;
      while (I  >= 1) do
      begin BLOCS.LIBERER (M[I]);
            I := I-1
      end;
      signal DEBORDEMENT2
end;
```

III LE TRAITEMENT DES EXCEPTIONS

Il existe actuellement plusieurs propositions ou implémentations de mécanismes d'exception. Leur but est d'aider le programmeur à réaliser le "traitement"des exceptions. Mais après avoir lu toutes les descriptions de mécanismes trouvables, quand nous avons essayé de répondre à la question "au fait qu'est-ce que le traitement des exceptions ?", nous n'avons pas pu trouver une réponse satisfaisante car chaque auteur propose son propre mécanisme et à travers ce mécanisme suggère sa propre réponse, qui diffère de celle des autres.

Le but de ce chapitre est de répondre à une question un peu plus précise : "qu'est-ce que le traitement des exceptions dans un système modulaire ?"

Nous commençons par proposer une classification des divers mécanismes d'exception que nous avons pu étudier. Ceci nous permettra de répondre à une question légèrement différente : "en quoi consisterait le traitement des exceptions si on utilisait ces mécanismes ?" (§ III. 1. 1. et § III. 1 . 2).

Ensuite, nous établirons les critères auxquels, à notre avis, un mécanisme d'exception modulaire devrait satisfaire. (§ III.1.3 et § III.1. 4). Cela nous amènera à analyser comment on peut rendre systématiques certains traitements par défaut, et quelles sont les limites de ces traitements. (§III.2).

Au § III. 3 nous proposons une réponse précise à la question qui a été le fil conducteur du chapitre entier.

INTRODUCTION

Pour comprendre quel est le sens de l'expression "traitement d'exception", nous allons utiliser une représentation visuelle de quelques niveaux d'abstraction d'un système . Mais le mot "système" lui-même a un sens différent pour des personnes différentes, et donc demande à être d'abord précisé.

Un système * écrit dans un langage modulaire comme SESAME, est composé d'entités "passives" (modules) et d'entités "actives" (processus). Nous supposons que l'ensemble M des modules d'un système est structuré en une hiérarchie de niveaux d'abstraction $I_0 \subset I_1 \subset \ldots \subset I_n = M$ par la relation "le module M utilise le module N" (§II-2.2). L'état externe du plus haut niveau d'abstraction I_n est l'agrégation des états externes de tous les modules. Un processus est un ensemble de droits d'accès et un algorithme [Brinch Hansen 77]. Les droits d'accès sont une liste d'opérations O_i que le processus est autorisé à faire sur des variables implémentées par I_n . L'algorithme est une séquence d'opérations O_i sur les variables auxquelles il a le droit d'accéder. Les variables accédées par plusieurs processus sont implémentées par des moniteurs, et les variables accédées par un seul processus sont implémentées par des classes . L'exécution d'un processus a pour but de changer l'état externe de I_n.

Dans un système, les processus savent "ce qu'il y a à faire", tandis que la machine abstraite I_n sait "comment le faire".

La figure 3.1 représente un processus P ("au-dessus" de I_n) et trois modules M, N, Q de niveau h, k et l : $0 \leq l < k < h \leq n$. Les flèches symbolisent la relation "utilise" : ainsi, P apelle la fonction $M.A$ qui appelle $N.B$, qui à son tour appelle la fonction $Q.D$

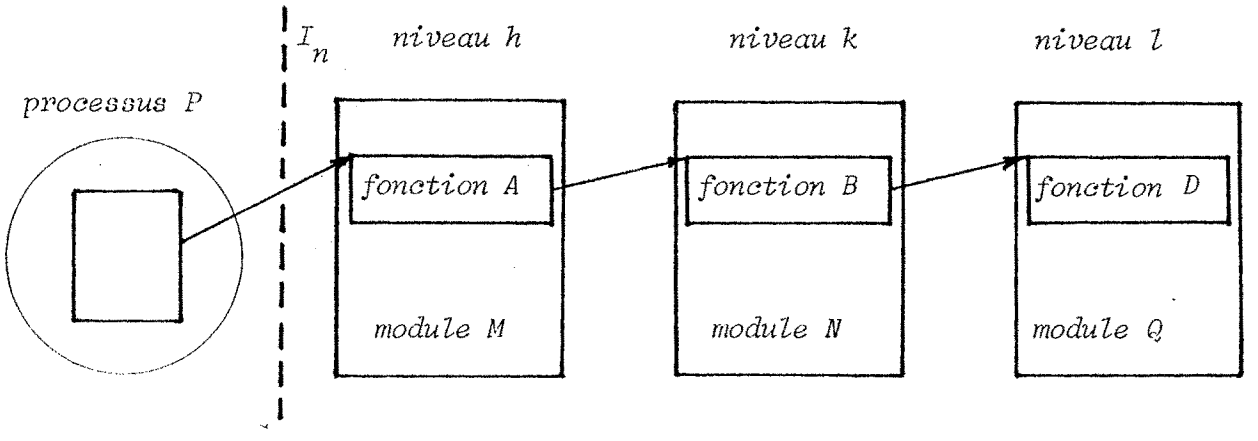


Figure 3.1

Les spécifications de $Q.D$ mentionnent une possible exception EL de niveau l, dans le cas où une assertion abstraite AA est trouvée fausse à l'exécution. Soient $d_1, d_2, \ldots, d_q$ les opérations composant le corps de la fonction D, et A l'assertion concrète correspondant à AA :

---

* Un exemple d'un tel système est donné [illegible]

```
ext function D ( <paramètres d'entrée> ) : TYPED ; signals EL,... ;
begin  d1 ;
       .
       .
       di-1 ;
       if not A then signal EL;
       di+1 ;
       .
       .
       .
       dq
end ;
```

Si pendant une activation de la chaîne d'appels *M.A*, *N.B*, *Q.D* l'assertion *A* est détectée fausse, cela signifie que :

- Il est impossible de continuer (pour l'instant) l'exécution de $d_{i+1},..,d_q$ pour rendre à *N.B* le service standard spécifié pour *Q.D* .
- Si *EL* est une exception intermédiaire (c'est-à-dire, si $d_1,..,d_{i-1}$ ont commencé à modifier l'état interne de *Q*; voir §II-2.2), alors l'état de *Q* est incohérent .

Dans ces circonstances, en quoi va consister le"traitement"de *EL*?

Suivant la réponse qu'on pourrait (indirectement) donner à cette question, en analysant le fonctionnement des mécanismes d'exception que nous avons étudiés, nous proposons de classifier* leurs auteurs en deux grandes"écoles":

1) Les premiers [Wulf 71],[Parnas 72],[Lampson 74],[Goodenough 75], [Kaiser 76], [Bron 76],[Lomet 77],[Levin 77] considèrent que l'algorithme de traitement de toute exception doit être explicitement écrit par le programme

2) La deuxième école ne contient qu'un seul représentant [Randell 75]. Cet isolement,qui a tant intrigué, est dû au fait que le mécanisme des "recovery blocs" est destiné à réaliser uniquement des traitements implicites (ou par défaut).

---

* Les écoles (1) et (2) si différentes au premier abord,s'attaquent en fait à deux aspects complémentaires d'un même problème. En accord avec [Melliar-Smit 77],nous pensons que les deux approches,au lieu d'être concurrentes sont en réalité complémentaires. Cette thèse tente de réaliser une synthèse entre ces deux écoles .

III-1. LE TRAITEMENT EXPLICITE .

Nous allons classifier à leur tour les auteurs "explicites" en deux "sous-écoles" :

1) Les premiers [Parnas 72],[Lampson 74], [Levin 77] considèrent le traitement d'une exception *EL* (figure 3.1) comme:

-L'activation de traitants explicitements déclarés dans les niveaux d'abstraction supérieurs, dont le but est de rendre l'assertion *A* vraie (masquage de *EL*)et de permettre la poursuite de la séquence standard $d_{i+1},..,d_q$ (la reprise se fait après l'instruction qui a signalé l'exception).

- Si toutefois, après l'exécution des traitants de *EL*, l'assertion *A* reste fausse, les effets standard de *Q.D*, *N.B*, *M.A* ne sont pas réalisables (puisque toutes les tentatives de masquage de *EL* ont été déjà effectuées, sans succès!) . Il est nécessaire,donc, de restaurer des états internes cohérents pour *Q*, *N* et *M* avant de dépiler les contextes d'activation *Q.D*, *N.B* et *M.A* , et notifier (ou propager)le refus du service standard de *M.A* à *P*. Nous allons appeler cette stratégie de traitement MARRE (de: MAsquage et REprise sinon Restauration Eventuelle).

2) [Wulf 71], [Bron 76], [Lomet 77] considèrent que l'occurrence d'une exception *EL* pendant l'activation de *Q.D*, entraîne (à cause de la nature même du concept d'exception), l'impossibilité de réaliser le service standard de *Q.D*dans le contexte de l'activation courante. En utilisant leurs mécanismes, le traitement d'une exception pourrait consister en :

- l'exécution (éventuelle) d'un traitement local à *Q.D* dont le rôle est de restaurer un état interne cohérent pour *Q* (cas d'une exœption *EL* intermédiaire) .

- la destruction du contexte d'activation de *Q.D*, et la propagation de l'exception *EL* dans *N.B*, se traduisant par l'activation du traitant associé à *EL* dans *N.B* . Soit *EK* l'exception de niveau k correpondant à la levée de *EL* dans *N.B* .

- au niveau de *N.B*, on peut tenter de remplacer le service standard de *Q.D* par un autre, considéré substituable (masquage de *EK* à l'utilisateur de *N.B*) . Si dans *N.B* on n'a pas prévu de tentatives de masquage de *EK* ou si l'exécution de toutes les tentatives prévues se termine par un échec (occurrence de nouvelles exceptions), il est nécessaire de restaurer un état interne cohérent pour *N*, avant de dépiler le contexte d'activation de *N.B* et propager l'exception *EK* dans *M.A*, etc...

Cette stratégie va être appelée REPROME (REstauration et PROpagation suivi d'un Masquage Eventuel) .

Il nous faut mentionner également deux propositions de mécanismes d'excep tion [Goodenough 75] et [Kaiser 76] dont le but est la mise en œuvre aussi bien de traitements de type MARRE que de traitements de type REPROME. En plus des traitements explicites, ces mécanismes doivent permettre la réalisation de traitements par défaut. Mais à la différence de [Randell 75] les traitants par défaut doivent être explicitement fournis* par le programmeur.

Le mécanisme de[Goodenough 75], oblige le signalant à indiquer explicitement si après la fin du traitement, il veut reprendre le contrôle en séquence, s'il ne le veut pas, ou s'il laisse cette décision à la discrétion du traitant .L'auteur propose des commandes distinctes pour indiquer si en fin d'exécution de traitant, on doit retourner au signalant (en séquencè), si on passe l'exception au niveau hiérarchique suivant, ou si on abandonne le traitement exceptionnnel. On propose également des exceptions particulière pour signaler des terminaisons "normales" d'exécutions de procédures, des traitements de restauration, ou des traitements par défaut, etc... Au total, 13 primitives, et autant de mots-clés nouveaux,pour construire ce mécanisme d'exception (un langage comme PASCAL possède une trentaine de mots-clés). L'inclusion d'autant de primitives dans un langage augmenterait beaucoup sa complexité [Ganon 75]: les interactions réciproques de ces primitives, ainsi que leur interaction avec les primitives du langage hôte, sont assez difficile à comprendre, ce qui augmente le risque de produire des fautes de programmati

Le mécanisme proposé dans [Kaiser 76] est destiné à être mis en œuvre dans un contexte de machines à descripteurs [England 74]. Un traitant local à un module (situé dans la zone appelée *INCIDENT*) peut se terminer par un retour en séquence après l'instruction signalante ou à un point d'entrée du module (stratégie MARRE), ou par un retour anormal au module appelant (stratégie REPROME). Dans le cas où l'exécution d'un traitant de la zone *INCIDENT* se termine par un échec (nouvelle exception) on considère qu'il y a une occurrence de *PANNE* dans le module (ce concept correspond à l'exception *DEFAILLANCE* du §III-2.). Le mécanisme fait partie du mécanisme plus général d'appel et de retour de module, et ses primitives sont destinées à être implémentées par le niveau le plus bas de la machine,celui qui est le seul à manipuler les descripteurs. Une solution intéressante est proposée pour le problème du manque d'homogénéité entre les exceptions détectées par le processeur cablé et les modules progrmmés (les filtres *AVAL* et *AMONT*).Mais comme dans le cas du mécanisme de [Goodenough 75], le nombre de primitives qui composent le mécanisme est assez élevé (12).

---

* Dans [Melliar-Smith 77] il est démontré de façon très convaincante que demander au programmeur de fournir explicitement des traitants par défaut "valables" est une tâche pratiquement impossible à réaliser en raison de sa très grande complexité. Voir également §IV-3.

III-1.1. MASQUAGE ET REPRISE SINON RESTAURATION EVENTUELLE.

Les auteurs classés dans l'école MARRE conçoivent les activations de traitants, soit comme des activations de procédures (routines) [Parnas 72], soit comme des activations de co-routines dont le nom est ignoré par le signalant [Lampson 74], [Levin 77]. Ce qui est essentiel dans cette apprche, est que le contexte d'activation de la procédure signalante *Q.D* n'est pas détruit pendant l'activation des traitants, et après la fin de l'exécution des traitants, le contrôle retourne au signalant.

Considérons à nouveau la figure 3.1. Supposons que dans *N.B* on ait déclaré un traitant *TEL* (routine ou co-routine, peu importe) de *EL*. Le rôle de *TEL* est de rendre l'assertion *A* vraie. En conformité avec le principe de la programmation modulaire (en anglais "information hiding principle") la représentation de l'état interne de *Q* est inconnue au niveau de *TEL*. IL s'ensuit que le seul moyen de rendre l'assertion *A* vraie, à partir de *TEL*, est d'appeler les fonctions externes de *Q* dont l'effet standard consiste à rendre *A* vraie .

Par exemple, si le module *Q* de la figure 3.1 était le module *BLOCS* du §II-2.3, et le module *N* le module *SEGMENTS* du §II-2.3, *EL* l'exception *DEBORD*, et *TEL* le traitant *TDEBORD* déclaré dans *SEGMENTS.CREER*, alors l'assertion concrète *A*, pouvant provoquer l'occurrence de *DEBORD* :

```
fonction A : boolean ;
var I : integer ;
begin I : =1 ;
      while (I ≤ & NB) and (E(I) = occupe) do I: =I+1 ;
      A:= (I<= &NB)
end
```

pourrait être rendue vraie à partir de *TEL* uniquement en activant la fonction externe *BLOCS.LIBERER*. En effet, l'effet standard spécifié pour *LIBERER* est de rendre l'assertion abstraite $AA = (B - ba') \geq T$ (voir §II-2.3) vraie. Si l'implémentation de *LIBERER* est correcte, et si les paramètres effectifs de l'appel de masquage sont légaux, l'assertion *A* devient vraie, et la reprise peut se faire après l'instruction qui a signalé *DEBORD* dans le module *BLOCS* .

Mais que se passe-t-il si après l'activation de *TEL*, l'assertion *A* reste toujours fausse ? :soit parce que dans *TEL* on ne disposait pas de suffisamment d'informations pour masquer *EL* (au niveau d'abstraction de *SEGMENTS* il est par exemple impossible de décider quels segments sont "périmés" et peuvent être détruits pour libérer d'autres blocs), soit parce que la tentative de masquage conduit à l'occurrence d'une nouvelle condition exceptionnelle, signalant que *A* n'a pas pu être rendue vraie.

Dans l'approche de [Parnas 72] et de [Parnas 76], le possible échec d'une tentative de masquage, ainsi que l'analyse des conséquences d'un tel échec, ne sont pas abordés. Si après l'activation des séquences de masquage, l'assertion *A* reste toujours fausse, retourne-t-on en séquence en d_{i+1} après l'appel de la routine *TEL*? Dans ce cas, il est évident que le service standard de *Q.D* doit être refusé. Comment cela va-t-il être notifié par *Q* à son utilisateur *N* ? Comment peut-on restaurer un état cohérent de *Q* avant de sortir définitivement du module ? Le mécanisme de "trap" proposé dans [Parnas 72] et développé dans [Parnas 76] nous semble incomplet, parce qu'il ne fournit pas de réponse claire à ces questions .

[Levin 77] n'aborde pas explicitement ce problème non plus. Toutefois, en analysant les exemples de traitements MARRE décrits dans sa thèse,on peut voir que l'instruction d_{i+1} qui suit *signal EL* est toujours un test explicite de *A*. Si le traitement n'a pas réussi à rendre *A* vraie, l'auteur signale une deuxième exception plus grave *ELG*, ou envisage la possibilité de restaurer un état cohérent du module *Q*,sans toutefois trop entrer dans ces détails embarrassants.

Seul le mécanisme d'exception de MPL[Lampson 74], implémenté en MESA [Mitchell 78],prend en compte explicitement un possible échec des tentatives de masquage. En utilisant ce mécanisme d'exception, à la fin d'une exécution de traitement *TEL* on a le choix entre les trois alternatives suivantes:

- ou bien on a masqué *EL*,et dans ce cas on *retourne* en séquence à l'instruction d_{i+1} et on finit par rendre le service standard de *Q.D* (masquage réussi) .

- ou bien on *re-exécute* l'instruction à laquelle *TEL*est associé (utile pour masquer des erreurs transitoires dans les entrées/sorties).

- ou bien on renonce à toute tentative future de masquage,en transférant définitivement le contrôle à partir de *TEL* dans un contexte englobant celui auquel *TEL* est associé, par un *goto*, *exit* ou *continue* [Mitchell 78]. *Goto* indique explicitement la prochaine instruction à exécuter, *exit* fait sortir d'une instruction itérative,et *continue* donne le contrôle à l'instruction qui suit, à laquelle *TEL* est attachée.

Dans MESA, un signal d'exception *EL* peut traverser plusieurs niveaux d'abstraction jusqu'à la rencontre d'un traitant. Par exemple, pour l'exception *BLOCS.DEBORD*, si au niveau de *SEGMENTS* on ne peut pas entreprendre des actions de masquage, on passe (primitive *reject*) le signal au module utilisateur (au §VI ce module est *VOIE-D'ACCES*) qui peut le masquer au niveau hiérarchique supérieur,etc...Jusqu'à ce que, ou bien on arrive au niveau d'abstraction le plus élevé (le processus), ou bien on ait trouvé un traitant qui a une vision suffisamment"globale" pour pouvoir décider de l'espace mémoire à libérer. Le module "utilisateur interactif" peut fort bien intervenir dans cette chaîne.

Si malgré toutes les tentatives de masquage,*A* reste fausse, le dernier traitant va finir par un *goto*, *exit* ou *continue*. Mais avant de détruire les contextes actifs au moment où *EL* est signalée, le mécanisme d'exception signale l'exception prédéfinie *UNWIND* . Un signal de *UNWIND* a comme effet d'activer tous les traitants pour *UNWIND* déclarés dans les contextes actifs de la pile d'exécution, en commençant par le signalant *Q.D*, et en terminant par le contexte auquel le dernier traitant exécuté se trouve attaché.

Le rôle des traitants de *UNWIND* est expliqué dans[Mitchell 78] en ces termes :

" L'activation des traitants pour *UNWIND* donne une chance à tous les contextes d'activation qui vont être détruits, d'entreprendre un nettoyage avant de mourir, en restaurant toute donnée manipulée à un état cohérent et en libérant toute mémoire allouée dynamiquement".

Les traitants de l'exception *UNWIND* doivent être fournis explicitement par les concepteurs des niveaux d'abstraction "court-circuités" par le signal premier.

Pour résumer, nous pouvons dire qu'en MESA, le traitement d'une exception *EL* peut comprendre une "remontée" du contrôle du signalant *Q.D* aux traitants *TEL*,..,et une "descente" du contrôle par un retour en séquence en d_{i+1} ou par un retour "spécial" au traitant de *UNWIND* de *Q.D* . Dans ce dernier cas, le"traitement" de *EL* se poursuit par une deuxième "remontée" du contrôle , à travers tous les traitants pour *UNWIND*, jusqu'au niveau d'abstraction du dernier traitant exécuté.

Citons trois remarques critiques de [Horning 78] à propos de ce mécanisme d'exception :

1° " La"puissance" et la "commodité" des signaux (de MESA), est due à la possibilité de passer des signaux d'exception à travers un grand nombre de niveaux intermédiaires, qui ne les traitent pas. Toutefois, plus grand est le nombre de niveaux intermédiaires traversés par un signal d'exception sans être traités, plus grande va être la distance conceptuelle

entre le signalant et le traitant. Ceci diminue la probabilité d'avoir un masquage effectif, et augmente la probabilité qu'un des niveaux traversés "oublie" de fournir un traitant pour *UNWIND*" [c'est-à-dire, augmente la probabilité d'apparition d'états incohérents].

2°"Comme toute primitive puissante,*UNWIND* doit être approchée avec prudence. Puisqu'elle existe dans le langage, le concepteur d'un module *M* ne peut jamais être sûr qu'en appelant une procédure externe implémentée dans un autre module *N*, le contrôle va retourner en *M* [parce que dans *N*, on peut signaler une exception pour laquelle *M* est "transparent"] Chaque appel de procédure externe doit être regardé comme une sortie définitive du module, et il faut nettoyer les choses avant d'appeler toute procédure externe,ou prévoir un traitant de *UNWIND* pour l'éventualité où le niveau d'abstraction de *M* risque d'être "court-circuité" par un signal d'exception généré dans *N* " .

3° " L'utilisation des signaux se révèle être génératrice de fautes de programmation, et j'ai la plus grande peine à localiser et corriger les fautes de programmation dues à l'utilisation de signaux d'exception. Chaque signal d'exception pouvant être généré par une procédure (directement ou indirectement) est une partie importante de son interface. Pourtant, les exceptions sont en général la partie la moins bien spécifiée et testée d'une interface, et il est possible de complètement "oublier" les signaux d'exception indirects, jusqu'à ce qu'ils provoquent des pannes catastrophique

III-1.2. RESTAURATION ET PROPAGATION SUIVIS D'UN MASQUAGE EVENTUEL

Les partisans de cette approche sont d'accord sur le point suivant : Un signal d'exception provoque le transfert définitif du contrôle entre le signalant et le traitant. [Lomet 77] est le seul à aborder les problèmes posés par la restauration d'états internes cohérents. Le *signal-enable* de BLISS [Wulf 71] et l' *escape* de [Bron 76] sont des mécanismes d'exception "purs", dans le sens qu'ils résolvent uniquement le problème du transfert de contrôle entre signalants et traitants, et n'abordent pas les problèmes de restauration d'état. Par exemple, dans HYDRA, système programmé en BLISS, les procédures de restauration d'états internes cohérents sont explicitement écrites par les programmeurs [Wulf 76].

Revenons à la représentation visuelle des niveaux d'abstraction de la figure 3.1 .

Si *EL* est une exception intermédiaire (§II-2.2), il est nécessaire de restaurer $\varepsilon(EL)$ avant de signaler *EL* à *N.B* .

Si l'on utilisait le mécanisme de [Lomet 77] on pourrait déclarer la séquence de restauration comme une *reset* procédure : après son exécution, le contexte de *Q.D* estdépilé, et un retour normal* a lieu en *N.B* .

L'utilisateur d'un mécanisme du style *signal-enable* de BLISS ou *escape* de [Bron 76] doit insérer l'algorithme de restauration avant l'instruction qui signale *EL* à *N.B*.

```
.
.
d(i-1);
if not A then  begin <séquence de restauration >;
                     signal (ou escape EL);
               end;
d(i+1);
.
```

Si *EL* est une exception d'entrée dans *Q.D*, il n'y aura rien à restaurer, et l'utilisation de *signal* ou *escape* suffit pour propager *EL* dans *N.B*. Mais dans *N.B* le problème de restauration se re-posera.

Soit *TEL* le traitant associé à *EL* dans *N.B*, et *EK* l'exception de niveau *k* qui sera implicitement détectée quand *TEL* sera activée à la suite d'une exécution de *signal* ou *escape* dans *Q.D*. *TEL* peut tenter de masquer l'occurrence de *EK* à son utilisateur *M.A*. Dans ce cas, la terminaison normale de *TEL* va provoquer la terminaison normale de *N.B* . Si dans *TEL* on exécute un nouveau *signal* ou *escape*, l'exception *EK* sera passée à l'utilisateur de *N.B* etc... Autant le mécanisme de BLISS, que celui proposé par [Bron 76] , permettent à un signal d'exception de "court-circuiter" plusieurs niveaux d'abstraction à la recherche d'un traitant.

Nous pouvons résumer cette brève présentation des mécanismes REPROME, en disant que le mécanisme de [Lomet77] est orienté vers la restauration, mais ne prend pas en compte les problèmes posés par la propagation des exceptions; par contre, les mécanismes de BLISS et [Bron 76] ne prennent en compte que ce dernier problème, dans un contexte de langages à structure de bloc, style ALGOL .

---

* Le retour dans l'appelant est envisagé dans [Lomet77] comme "normal". Cette approche est criticable, car elle enlève à l'utilisateur de *Q.D* la possibilité de savoir si c'est le service standard qui a été réalisé, ou si c'est un service exceptionnel.

III-1.3. CRITERES POUR UN TRAITEMENT MODULAIRE

Ce chapitre propose un ensemble de critères* auxquels un mécanisme d'exception "modulaire" devrait satisfaire.

1° Compatibilité avec les principes de la programmation modulaire (modularité
2° Capacité de prévenir la production d'états internes incohérents (sécurité)
3° Utilisabilité à tous les niveaux d'abstraction (uniformité)
4° Sémantique simple et nombre de primitives réduit (simplicité)
5° Interférences bien définies avec les autres structures de contrôle (orthogonalité)
6° Coût acceptable (économie)

III-1.3.1. MODULARITE

L'utilisateur connaît un module à travers ses spécifications et ignore sa structure interne [Parnas 72]. Les effets standard et exceptionnels des fonctions externes sont spécifiés en termes de paramètres d'appel et de l'état d'avant l'appel. Une stratégie REPROME, accordant la priorité aux restaurations d'état, permet d'implémenter tout effet exceptionnel comme une restauration d'état interne cohérent, équivalent à l'état interne d'avant l'appel. Les opérations atomiques sont simples à spécifier.
Par contre, si l'on utilise la stratégie MARRE pour traiter une exception définie dans un module *Q* (figure3.1), la transformation d'état provoquée par une activation de *Q.D* ne dépendra plus uniquement des paramètres d'appel et de l'état de *Q*, mais aussi de l'état des modules qui utilisent *Q*. Par exemple, l'effet de *BLOCS.ALLOUER* (§II-2.3) ne dépend plus uniquement de l'état de *BLOCS*, mais aussi de l'état des modules de niveau supérieur pouvant intervenir pour masquer une possible occurrence de *DEBORD*. Les techniques de spécification actuelles qui décrivent l'effet d'un appel de module uniquement en fonction de l'état des paramètres d'appel et de l'état que le module possède avant l'appel, ne peuvent pas être utilisées pour spécifier les modules dont les exceptions subissent des traitements de type MARRE

---

* La mise en évidence de ces critères ne doit pas être interprétée comme un jugement de valeur sur les mécanismes analysés au §III-1.1 et III-1.2. En effet, la plupart de ces mécanismes ont été conçus pour être utilisés dans des contextes différents, et pour satisfaire à d'autres critères.

Le respect des règles de la programmation modulaire introduit une contrainte que nous allons appeler "l'étanchéité". Dans la figure 3.1, N appelle Q mais, M n'appelle pas directement Q.
Laisser un signal d'exception EL, émis par Q, arriver en M sans être traité dans N, contredit l'abstraction implémentée par N pour M, car EL ne figure pas dans l'interface de N. L'exemple typique est le mauvais compilateur d'un langage évolué qui sort un message d'erreur incompréensible au programmeur :*"instruction inexistante à l'adresse X'000E0F1A'"* et imprime un*"dump"* de la mémoire physique.

Tous les mécanismes d'exception précédemment décrits, sauf ceux qui ont été conçus pour les programmes modulaires [Parnas 72],[Kaiser 76], [Levin 77], permettent aux signaux d'exception de "court-cicuiter" plusieurs niveaux d'abstraction à la recherche d'un traitant.

III-1.3.2. SECURITE

Plus grand est le nombre de modules pouvant être "court-circuités" par un signal d'exception, plus grande est la chance de laisser les modules traversés dans des états internes incohérents. Les mécanismes d'exception "purs" [Wulf 71], [Parnas 72], [Kaiser 76], [Bron 76], [Levin 77], laissent le problème des restaurations d'état entièrement à la charge des programmeurs. Des mécanismes comme ceux de [Lampson 74], [Goodenough 75], [Lomet 77], offrent des facilités linguistiques pour déclarer des algorithmes de restauration, mais laissent le soin d'écrire ces algorithmes aux programmeurs. Comme nous allons le voir au §IV-3, la très grande complexité de ces algorithmes contribue à augmenter,plutôt qu'à diminuer la probabilité d'apparition d'états internes incohérents.

A notre avis, un traitement d'exception "sûr" ne devrait pas laisser un signal d'exception intermédiaire sortir d'un module avant de restaurer un état interne cohérent. Pour rendre ces restaurations d'états plus"sûres", le mécanisme d'exception devrait être "épaulé" par un mécanisme de restauration fourni par le niveau de base I_0 (voir §IV).

II-1.3.3. SIMPLICITE ET UNIFORMITE

La volonté de construire un mécanisme simple, utilisable uniformément à tous les niveaux d'abstraction, résulte d'une certaine attitude envers la structuration des programmes. On préfère construire des programmes complexes à partir d'un petit nombre de concepts fondamentaux, et obtenir une grande puissance d'expression par la combinaison d'un petit nombre de primitives simples et faciles à comprendre. Les mécanismes qui permettent la mise en œuvre, autant de la stratégie MARRE que de celle de REPROME, ont le désavantage d'être composés d'un trop grand nombre de primitives. Par contre, les mécanismes destinés à mettre en œuvre une seule stratégie de traitement, sont plus simples à comprendre,et sont composés d'un nombre plus restreint de primitives.

On veut également qu'avec le même mécanisme fondamental, on puisse exprimer le traitement des exceptions de niveau 0, des exceptions détectées par les modules du "système", ainsi que les exceptions dans les programmes de l'utilisateur.

III-1.3.4. ORTHOGONALITE

Un mécanisme d'exception interagit nécessairement avec les autres structures de contrôle du langage, car lui aussi est une structure de contrôle. Le concepteur d'un langage doit veiller à ce que le mécanisme d'exception soit compatible avec les autres structures de contrôle.

Une des critiques de [Horning 78] à propos du mécanisme d'exception de MESA, (§III-1.1) porte sur le fait que ce mécanisme d'exception contredit la sémantique de mécanisme d'appel et retour de module. En effet, conformément à la définition de l'appel procédural, un module N qui appelle une procédure externe d'un autre module $Q.D$ (figure3.1) s'attend à ce qu'après l'exécution de $Q.D$, le contrôle revienne en N. Or un signal d'exception levé en Q peut traverser de façon "transparente" N. Si le concepteur de N ne prévoit pas de traitant pour *UNWIND*, il doit considérer tout appel de procédure externe au module comme une sortie définitive du module.

Un mécanisme d'exception interfère également avec le mécanisme de synchronisation. Par exemple, un mécanisme d'exception conçu pour la mise en œuvre de traitements de type MARRE, est incompatible avec un mécanisme de synchronisation comme le moniteur [Hoare 74]. En effet, supposons que dans la figure 3.1, *Q* est un moniteur,et que durant l'exécution de *Q.D*, on détecte l'exception *EL*. Le contrôle va au traitant *TEL* déclaré dans *N.B*, mais le contexte d'exécution *Q.D* est gardé en l'état. Les seules actions de masquage de *EL* possibles à partir de *TEL* sont les appels de fonctions et procédures externes de *Q*,mais comme le moniteur *Q* est déjà occupé, la première tentative de masquage conduira à un deadlock.

Dans le cas où la synchronisation est assurée par des mécanismes plus primitifs, comme le sémaphore [Dijskstra 67] , la programmation des traitements de type MARRE, devient très complexe. L'exemple de module *ALLOCATEUR-DE-MEMOIRE* qui signale une exception "niveau bas de mémoire" avant que la mémoire soit complètement épuisée, programmé par[Levin 77] à l'aide de sémaphores, illustre bien ces difficultés.*

Un autre interaction indésirable entre le mécanisme d'exception et le mécanisme de synchronisation est la suivante : Supposons que la procédure *D* implémentée par le module *Q* puisse être exécutée en parallèle par plusieurs processus à la fois. Le test de l'assertion *not A*, qui est la précondition de l'exception *EL*, et l'émission du signal *EL* sont deux opérations distinctes. Il se peut donc qu'un processus P_1 trouve *A* fausse, et avant de signaler *EL* à *N*,un autre processus P_2 trouve *A* également

---

* Les procédures externes de ce module,*ALLOCATE* et *RELEASE*,s'exécutent en exclusion mutuelle grâce à deux sémaphores *OUTER* et *INNER*. Pour entrer dans la procédure *ALLOCATE*, un processus doit exécuter successivement *P(OUTER);P(INNER)*,alors que pour entrer dans *RELEASE*,uniquement *P(INNER)*. Si pendant l'exécution d'*ALLOCATE*,l'exception *NIVEAU-BAS* est détectée,avant de la signaler aux niveaux d'abstraction qui utilisent *L'ALLOCATEUR-DE-MEMOIRE* il est nécessaire d'exécuter un *V(INNER)*pour débloquer l'accès à *RELEASE*, et rendre le masquage de l'exception possible. En même temps,l'accès à *ALLOCATE* reste bloqué. Après l'exécution des traitants pour *NIVEAU-BAS*, quand le contrôle retourne dans *ALLOCATE* (stratégie MARRE),on exécute à nouveau un *P(INNER)* pour empêcher le reste de la procédure *ALLOCATE* de s'exécuter en parallèle avec *RELEASE*.Si le masquage n'a pas réussi (la précondition de *NIVEAU-BAS* persiste à être vraie), avant de signaler l'exception "fatale": *MEMOIRE-EPUISEE*, on exécute à nouveau *V(INNER);V(OUTER)*etc.. La programmation se complique d'avantage dès qu'il y a plusieurs exceptions qui peuvent être signalées par une même procédure, et on envisage plusieurs tentatives de masquages possibles,incluses dans des traitants disposés à des niveaux d'abstraction différents et qui doivent s'exécuter en parallèle (cas du mécanisme de [Levin 77]).Dans ce dernier cas, l'auteur recommande l'introduction d'autres sémaphores et d'autres "flags" etc...

fausse, et signale EL à son tour. Dans ces conditions, faut-il imposer l'exclusion mutuelle de l'accès au traitant TEL, ou faut-il l'admettre ? Ne vaut-il pas mieux "jeter à la poubelle" le deuxième signal considéré comme superflu ? Ou bien faut-il exécuter le test de A et la signalisation de EL en exclusion mutuelle ? Une telle solution revient à synchroniser l'exécution des processus P_1 et P_2, c'est-à-dire à faire exactement le contraire de ce qu'on voulait faire initialement : laisser P_1 et P_2 s'exécuter chacun à sa propre vitesse dans $Q.D$!

L'utilisation des moniteurs comme outil de synchronisation en SESAME, exclut la possibilité de telles interférences, et nous évite (heureusement) de penser aux conséquences que ces interférences peuvent avoir sur la cohérence des états internes des modules impliqués .

III-1.3.5 ECONOMIE

Si l'on voulait évaluer la rentabilité de l'introduction d'un mécanisme d'exception dans un langage, il faudrait étudier l'influence de la présence du mécanisme sur le temps nécessaire à l'implémentation d'un programme, à son exécution et à sa maintenance .

En séparant l'algorithme standard des traitements d'exception, l'utilisation d'un mécanisme d'exception rend les programmes plus lisibles et modifiables. Mais en contre-partie, le temps d'exécution se trouve augmenté. Dans le monde de la "programmation pratique", les exemples de sous-utilisation des outils de structuration du logiciel sont fréquents, à cause du coût prohibitif à l'exécution. Par exemple, on minimise les appels de procédure, en construisant des procédures "géantes", avec des algorithmes nécessairement compliqués, qui coûtent cher en temps d'implémentation, et qui sont surtout illisibles et incompréhensibles, donc coûtent encore plus cher en temps de maintenance. De plus, la fiabilité de tels programmes est rarement élevée, et il faut ajouter au coût total, le coût des défaillances.

En ce qui concerne le coût à l'exécution, il est nécessaire de distinguer entre un coût distribué, et un coût propre [Levin 77]. Le premier est le coût de l'utilisation effective de la primitive. Par exemple, un *goto* non local en ALGOL 60, représente un coût propre assez élevé, car il faut rechercher dans la pile des contextes l'adresse de l'étiquette à atteindre, et il faut dépiler tous les contextes intermédiaires. Le coût distribué d'une primitive est le coût réparti dans tous le programme, même si la primitive n'est pas utilisée du tout. Par exemple, le *goto* non local entraîne un coût supplémentaire à chaque changement de contexte, car il est nécessaire de mettre à jour les informations qui permettent d'interpréter un éventuel *goto* non local.

Un mécanisme d'exception doit bien sûr avoir un coût distribué aussi bas que possible. Le coût distribué résulte de l'interaction du mécanisme avec les autres structures de contrôle du langage; il faut par conséquent, surveiller de près ces possibles interactions.

III-1.4 CHOIX D'UNE STRATEGIE DE TRAITEMENT D'EXCEPTION

Récapitulons les principaux désavantages de la stratégie MARRE :

-impossibilité d'utiliser les techniques de spécification actuelles pour spécifier les modules dont les exceptions subissent des traitements MARRE.

- Le niveau d'abstraction ayant une vue suffisamment"globale"pour pouvoir entreprendre une action de masquage effectif, peut être fort éloigné du niveau signalant, d'où la nécessité de permettre aux signaux d'exception de " court-circuiter" plusieurs niveaux intermédiaires. Ceci accroît les risques de production d'états incohérents et peut contredire la sémantique d'autres structures de contrôle du langage (par exemple l'appel procédural).

- Forte interférence avec le mécanisme de synchronisation. Risque de production de deadlocks.Incompatibilité avec un outil de synchronisation comme le moniteur de [Hoare 74] , adopté en SESAME .

Une stratégie REPROME"étanche"n'a aucun de ces désavantages. Son application systématique permet l'implémentation d'opérations atomiques (voir §III-3).

- Les opérations atomiques sont facilement spécifiables si l'on utilise l'extension au langage de spécification suggérée au §II-2.1 . La partie "effets exceptionnels" d'une spécification d'opération ne doit contenir que les identificateurs des diverses exceptions qui peuvent être détectées, et les assertions qui sont leurs préconditions.

- Si la propagation d'un signal d'exception à travers une frontière de module est systématiquement précédée d'un traitement local qui restaure un état interne cohérent, les risques d'apparition d'états incohérents se trouvent diminués. L'impossibilité pour un signal d'exception de "court-circuiter" plusieurs niveaux d'abstraction, élimine le conflit avec une structure de contrôle comme l'appel/retour de module.

- L'émission d'un signal d'exception pendant une exécution de procédure termine l'exécution de celle-ci. Comme dans un environnement modulaire, "l'unité" de synchronisation est souvent une opération abstraite tout entière, l'interférence avec le mécanisme de synchronisation se trouve réduite. Si l'outil de synchronisation est le moniteur, l'interférence est minime (voir §V). Sur l'exemple du système multi-accès du §VI, on peut voir que la stratégie REPROME est bien adaptée au traitement des exceptions dues aux conflits de partage des ressources,car elle élimine les risques de deadlock.

Un des avantages de la stratégie MARRE est, selon[Mitchell 78], la commodité de mise au point en mode interactif. Citons le manuel de MESA :

" MESA garantit que tout signal d'exception qui ne rencontre aucun traitant dans son chemin est récupéré au plus haut niveau d'abstraction (qui est le système) et reporté par le module de mise au point ("Debugger") à l'utilisateur. D'où l'utilité pour la mise au point,car tous les contextes actifs,au moment de la génération du signal, sont conservés et peuvent être inspectés en vue d'un diagnostic" .

Dans le cas de REPROME, au moment où un signal d'exception arrive au plus haut niveau d'abstraction, tous les contextes d'activation sont détruits Ceci ne nous semble pas être un obstacle à la mise au point interactive. Il suffit de faire intervenir le "Debugger" au moment de la génération du signal d'exception, et non à son arrivée éventuelle au plus haut niveau d'abstraction. Dans une phase d'exploitation normale, la place du "Debugger" interactif pourrait être prise par un module système de "journalisation" des exceptions,qui mesure la fréquence de détection d'exceptions, et prend des "instantanés" sur bande magnétique de l'état des contextes actifs à la génération des signaux d'exception, en vue d'une inspection "off-line".

Les considérations précédentes nous amènent a opter pour une "pure" stratégie REPROME, malgré notre intention initiale de construire un mécanisme "mixte" qui puisse permettre la mise en oeuvre des deux stratégies à la fois . Le mécanisme décrit au § V prévoit des traitements par défaut à chaque niveau d'abstraction . Le but est d'empêcher les signaux d'exception de passer des frontières de modules sans être "traités".

Nous allons analyser les problèmes posés par ces traitements dans le chapitre qui suit.

III-2; TRAITEMENT PAR DEFAUT DES EXCEPTIONS INTERMEDIAIRES NON PREVUES.

Considérons à nouveau les niveaux d'abstraction de la figure 3.1. Supposons que le processus *P* ait activé *M.A* qui active *N.B*, et que pendant le test des assertions d'entrée en *N.B* aucune exception d'entrée n'est détectée. La fonction *N.B* réalise une certaine opération abstraite, en la décomposant en une séquence d'opérations $b_1, b_2, .., b_n$ sur les variables déclarées dans *N* . Ces variables sont implémentées par des modules *Q* de niveau *l* inférieur à $k, l=0,1,..,k-1$. Supposons qu'une exception *EL* soit levée dans *N.B* à la suite d'une activation d'opérations intermédiaires *Q.D* .

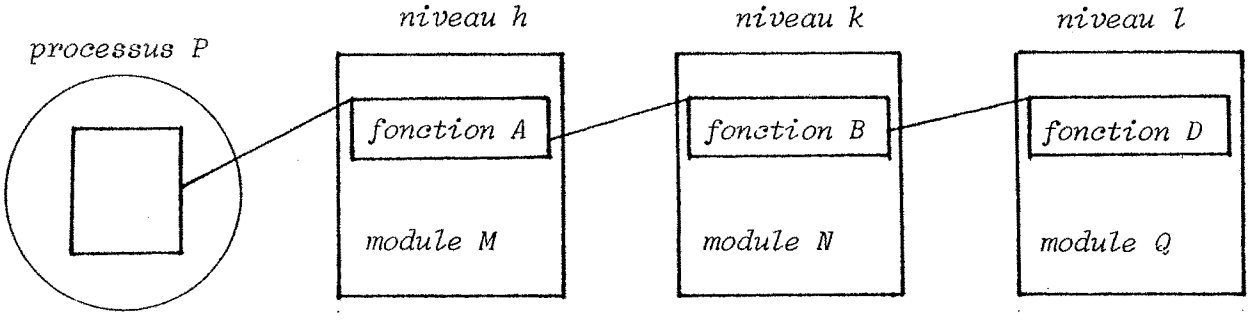


Figure 3.1

Dans le §III-1.3, nous avons argumenté qu'il est nécessaire de rendre les frontières de modules "étanches" à la propagation des signaux d'exception. Ceci nécessite le respect de la règle suivante : "pour toute exception spécifiée dans l'interface d'un module *Q* utilisé par *N*, il faut prévoir dans *N* un traitant ". Cette contrainte peut être vérifiée statiquement par le compilateur et le connecteur. Toutefois, son respect strict rendrait la tâche de la programmation très lourde, sinon impossible. En effet :

1) Parfois le programmeur peut prouver qu'une exécution d'opération ne finira pas par un retour exceptionnel (par exémple *+(1,1)* ne provoquera aucun débordement arithmétique). Il n'y a donc aucune raison d'attacher systématiquement un traitant d'exception à chaque appel d'opération.

2) Chaque traitant d'exception est une séquence d'opérations pouvant produire de nouvelles exceptions. Les traitants de ces nouvelles exceptions sont à leur tour des séquences d'opérations, etc...

Le respect strict de la règle "un traitant par exception possible", conduirait à la situation absurde où le niveau d'imbrication des traitements tendrait à devenir infini .

Par exemple, un programmeur aurait à se demander :"que faut-il que je fasse quand je reçois une exception *ERREUR-PARITE*,quand je veux libérer une page virtuelle sur disque,quand je traite une exception *FAUTE-DE-PAGE*, pendant que j'essaye d'amener en mémoire centrale le traitant non résident de l'exception *MEMOIRE-VIRTUELLE-SATUREE* etc.."[Horning 78].

Pour les raisons 1) et 2), il n'est pas réaliste d'imposer la règle "un traitant par opération pouvant signaler une exception". Ceci nous conduit à admettre qu'il est possible qu'une opération *Q.D* puisse signaler une exception *EL* pour laquelle aucun traitant explicite ne soit prévu dans l'appelant *N.B*. La propagation, via le mécanisme d'exception, du signal *EL* dans *N.B*, signifie (d'un point de vue conceptuel, voir §II-2.2) qu'une exception intermédiaire de niveau k est implicitement détectée dans *N*. Ce phénomène est indépendant du fait que le programmeur de *N.B* a, ou n'a pas prévu de traitant pour *EL* . Ce qui est essentiel, c'est que le service standard de *N.B* n'est pas réalisable, car le service standard d'une opération intermédiaire n'est pas disponible. Tout ce qu'on peut faire au niveau de *N.B*, est de rendre à la place du service standard, un service exceptionnel. Mais par quel nom d'exception identifier à l'extérieur ce service exceptionnel, qui n'a pas été explicitement prévu, ni spécifié par le programmeur de *N.B* ? Nous proposons de résoudre ce problème de la manière suivante :

Si une exception intermédiaire non prévue, est signalée dans un module N via le mécanisme d'exception, N va fournir à son appelant un service exceptionnel prédéfini, identifié par un nom d'exception prédéfinie : DEFAILLANCE. Le nom DEFAILLANCE permet de désigner (en fait), toute la classe des exceptions intermédiaires pouvant être (implicitement) détectées dans N, et qui n'ont pas de traitant explicite .

L'introduction du concept de *DEFAILLANCE* permet de résoudre le problème de l'étanchéité des frontières de modules à la propagation des exceptions. A chaque corps de fonction externe d'un module, on attachera un traitant par défaut pour toutes les exceptions intermédiaires non prévues :

```
functionB ( < paramètres d'entrée >): TYPED;
var  RD:TYPED;
begin  b1 ;
       :
       RD:=Q.D; % l'exception EL peut être signalée
       :              mais aucun traitant explicite n'est prévu pour EL %
       :
       bn;
end  [DEFAILLANCE : <traitant par défaut >];
```

C'est le mécanisme d'exception quidoit transformer tout retour exceptionnel pour lequel aucun traitant explicite n'est prévu dans *N.B*, en une activation du traitant par défaut (l'arrivée d'un signal *DEFAILLANCE* dans *N.B* doit être également traduite par le mécanisme d'exception en une activation du traitant par défaut) . Le but d'un traitant par défaut pourrait être :

- de restaurer un état interne cohérent pour *N* (restauration)
- de tenter de réaliser le service standard de *N.B* malgré l'indisponibilité de l'effet standard de *Q.D* (masquage)
- de signaler l'indisponibilité du service standard de *N.B* en *M.A* si le masquage n'est pas prévu ou ne réussit pas (propagation de *DEFAILLANCE)*

III-2.1. RESTAURATION D'UN ETAT INTERNE COHERENT

Dans le cas d'une occurrence d'exception intermédiaire prévue *EK*, au niveau k, le concepteur de *N* possède une connaissance a priori de l'ensemble d'incohérence $\varepsilon(EK)$: il sait quelles variables rémanentes ont été modifiées depuis l'opération b_1 jusqu'à l'opération b_{i-1}, et il sait quelle partie de ces variables est vraiment "significative" par rapport à la fonction *rep* associée au module *N*. Il peut donc utiliser cette connaissance (comme au §II-2.6) pour écrire un traitant qui restaure uniquement les éléments de $\varepsilon(EK)$.

Dans le cas d'une exception imprévue *EKI*, appartenant à la classe *DEFAILLANCE* , cette connaissance a priori manque. La définition du concept d'ensemble d'incohérence, ne permet pas d'imaginer un algorithme simple pour calculer à l'exécution $\varepsilon(EKI)$ pour $\forall$ *EKI* $\in$ *DEFAILLANCE*.

Ce problème est abordé en détail au §IV . Activé à la suite d'une détection de *DEFAILLANCE* dans *N*, le mécanisme de restauration du §IV peut replacer *N* dans un état interne cohérent e_r équivalent à l'état e' d'avant l'entrée en *N*, à condition que $I_c(e')$=vrai.

III-2.2. LE PROBLEME DU MASQUAGE DE L'EXCEPTION *DEFAILLANCE*

Une proposition de technique pour masquer des exceptions *DEFAILLANCE* a été faite dans [Horning 74]. L'idée est de rendre les actions de masquage d'une *DEFAILLANCE* indépendantes de sa précondition. Cette technique (qui a été et reste encore beaucoup contreversée) préconise l'utilisation d'algorithmes redondants (appelés "alternants") pour l'implémentation d'une même opération. Pour sa mise en œuvre, les auteurs ont imaginé le mécanisme des"recovery blocs" (MRB) qui ressemble peu aux autres mécanismes d'exception, car il est

destiné à exécuter exclusivement des traitements par défaut.

Nous allons présenter brièvement le fonctionnement d'un MRB, en nous aidant de la figure 3.1. Sopposons que durant l'exécution de l'opération *N.B* une exception *DEFAILLANCE* soit détectée dans *N*. Le MRB restaure pour *N* un état interne équivalent à l'état interne d'avant l'activation de *N.B*, et donne le contrôle à un alternant *B'* de l'algorithme défaillant *N.B*. Si l'exécution de *B'* ne provoque pas de nouvelle *DEFAILLANCE*, le retour dans l'appelant *M.A* sera normal, comme si la première *DEFAILLANCE* n'avait jamais été détectée (masquage réussi). Si durant l'exécution de *B'* une nouvelle *DEFAILLANCE* est détectée, le MRB reprend le contrôle, restaure à nouveau l'état interne de *N* ,et active un deuxième alternant *B"* etc...Si le dernier alternant provoque toujours une détection de *DEFAILLANCE*, alors le MRB restaure une dernière fois l'état interne de *N* et propage un signal de *DEFAILLANCE* dans l'appelant *M.A* (masquage non réussi). L'existence de plusieurs alternants pour une certaine opération *N.B* est ignorée par l'utilisateur *M* , car il ne peut pas les appeler explicitement. Le service qu'un alternant *B'* doit réaliser peut être , ou bien identique à celui de *N.B*, ou bien différent "moins souhaitable,mais acceptable par l'utilisateur quand même" [Randell 75]

Un des points faibles de cette approche est que les différents alternants *B',B'',...* d'une opération *B* sont contraints de travailler sur la même* structure de données . Le succès de son application dépendra donc de la capacité des programmeurs à trouver des algorithmes différents pour implément une même opération abstraite, à l'aide des mêmes données concrètes. Par exemp si la *DEFAILLANCE* de *N.B* est due à une activation d'opération *D* sur une variable *Q* (figure 3.1) provoquant une exception *EL* non prévue dans *N*, alors le succès du masquage dépendra du fait que dans l'alternant *B'* il y a, ou il n'y a pas la même activation "fatidique" de *Q.D*.

Malgré son coût d'application élevé, la technique proposée dans [Horning 74] ne permet pas à ses utilisateus potentiels de prédire les chances qu'ils auraient de résoudre effectivement le problème du masquage des *DEFAILLANCES*, s'ils l'utilisaient . En effet, elle ne permet pas de garantir (par exemple) que la probabilité de masquer avec succès les possibles *DEFAILLANCES* d'un système, va être supérieure à un certain seuil imposé par le cahier des charges.

---

* Le mécanisme des "recovery blocs" duplique les algorithmes , mais ne duplique pas les données sur lesquelles ces algorithmes travaillent. En effet, quand un alternant *B'* est activé, il voit les mêmes données que *B* et dans le même état qu'au début de l'exécution de *N.B*.

Pour ces raisons, ainsi que pour des raisons de simplicité, nous n'envisagerons pas l'inclusion dans le mécanisme d'exception d'une primitive linguistique spéciale, permettant de déclarer plusieurs alternants pour une même opération*. Le seul traitement par défaut que nous envisageons, consite en une restauration d'état interne cohérent, suivie d'une émission de signal de *DEFAILLANCE*.

III-2.3. PROPAGATION DE L'EXCEPTION *DEFAILLANCE*.

Comme aucune tentative de masquage de *DEFAILLANCE* n'est envisagée, il s'ensuit qu'une occurrence de *DEFAILLANCE* à un certain niveau d'abstraction, entraînera sa propragation jusqu'au plus haut niveau d'abstraction (le processus).

Le programmeur d'une certaine opération ne devra pas spécifier la possible émission d'un signal de *DEFAILLANCE*. pendant son activation, car toute opération peut, par convention, produire un signal d'exception *DEFAILLANCE*

III-2.4. LES CAUSES DE *DEFAILLANCE*

Dans l'hypothèse où le compilateur du langage d'implémentation d'une machine modulaire I_n est correct., et que la machine d'exécution est correcte ainsi que fiable (voir pour plus de détails les hypothèses (H1 ,H2) du §IV-1), alors une détection de *DEFAILLANCE* dans un module N de niveau $k, 0 < k \leq n$, est causée par le non-respect de la règle suivante : "Si dans N on utilise une variable v, alors le concepteur de N doit être préparé à traiter toute possible violation dynamique d'un invariant caractérisant le type de v". Le respect de cette règle nécessite:

1° De connaître et spécifier les propriétés invariantes de tous les types de variables d'un système.

2° De spécifier pour chaque type, toutes les exceptions possibles; de prévoir les tests dynamiques qui détectent l'occurrence de ces exceptions.

3° De prévoir pour chaque activation d'opération autant de traitants explicites que d'exceptions possibles.

---

* Il nous semble que si quelqu'un souhaitait ajouter au mécanisme d'exception décrit au §V une telle primitive, cela ne devrait pas poser de problèmes majeurs. Une description détaillée de l'implémentation des "recovery blocs" dans le langage CONCURRENT PASCAL peut être trouvée dans [Shrivastava 78c]

Le respect strict des règles 1°,2° et 3° est une tâche monumentale. Toute personne qui a essayé d'écrire un programme robuste*, dont la taille dépasse quelques milliers de lignes**, peut en apprécier la difficulté.

Le problème 1° reste encore à résoudre à l'heure actuelle, malgré des progrès récents fort encourageants [Guttag 75], [Wulf 76], [London 76], [Burstal 77], [Bert 79]. Par exemple, très peu de travail a été fait pour la formalisation des propriétés invariantes des types périphériques. Des futures recherches sur les spécifications formelles sont donc nécessaires pour permettre aux concepteurs de programmes de résoudre ce problème. Une fois les propriétés invariantes connues, écrire toutes les assertions qui doivent être vérifiées à l'exécution pour garantir leur respect, et identifier par des noms d'exceptions distincts toute possible violation d'une telle assertion, pose un autre problème. Nous avons remarqué que la spécification des effets exceptionnels devient plus simple si l'on se restreint à construire uniquement des types ayant des opérations atomiques (§II-2.1). La règle 3°entraîne un accroissement non négligeable de la complexité des programmes. Nous avons analysé ce problème au §III-2, et nous avons montré que son respect strict est impossible en pratique.

Quand les règles 1°, 2° et 3° ne sont pas respectées, on parle d'une faute .

Les fautes sont la cause de l'occurrence des exceptions
DEFAILLANCE.

Il y a des :

- Fautes de codage , quand un certain algorithme abstrait est traduit en un algorithme concret non équivalent, soit par inattention, soit parce que l'on utilise partiellement ou d'une façon erronée des spécifications correctes en elles-mêmes.

- Fautes de spécification, quand les propriétés invariantes spécifiées formellement, ne coïncident pas avec celles qui existent "intuitivement" dans la tête du concepteur. C'est le cas des spécifications ambiguës, non complètes ou contradictoires [Guttag 75] .

---

* qui traite toutes les exceptions pouvant survenir pendant son exécutic

** C'est le cas typique des systèmes d'exploitation ou des systèmes de gestion de bases de données.

III-2.4.1. FAUTES DE CODAGE

Prenons le code de la fonction externe *ALLOUER* du module *RESSOURCES* du § II-2.2. Supposons que le programmeur ait, à la place de la ligne 8, écrit la ligne :

8' while E[I] =OCCUPE) and (I≤&N) do I+1;

Une activation de *ALLOUER* quand le module se trouve dans l'état interne $e'=(OCCUPE)^{\&N}$ provoquerait l'exception de niveau *0 RANGE-ERROR* à cause de l'accès au tableau *E* avec l'index *I=&N+1*. Comme aucun traitement explicite n'est prévu dans *RESSOURCES* pour cette exception, son occurrence signifiera une détection de *DEFAILLANCE*. L'instruction de la ligne 8'est un exemple de faute de codage. La terminologie qui suit est inspirée de [Kaiser 74], et de [Melliar-Smith 77].

Quand le programmeur écrit la ligne 8', il y a production de la faute. Quand l'utilisateur de RESSOURCES appelle ALLOUER avec comme état d'entrée $e'=(OCCUPE)^{\&N}$, la violation de l'invariant $I\in[1,\&N]$ caractérisant la variable index I est une manifestation de la faute.

Il existe des invariants qui ne sont pas vérifiés dynamiquement par des tests d'assertions. Par exemple, si le niveau I_0 n'avait pas généré implicitement le test de l'assertion $I \leq \&N$ pour garantir le respect de la propriété invariante $I\in[1,\&N]$, la manifestation de cette faute serait passée inaperçue. Une manifestation non détectée de faute provoque l'apparition d'états incohérents qui peuvent causer de futures détections d'exceptions additionnelles (§II-2.4). Par exemple, si en réponse à la demande d'allocation, la variable *BLOCS* de type *RESSOURCES* avait alloué pour *SEGMENTS* le bloc fictif *&N+1*, lors d'une future tentative de lecture ou écriture de ce bloc sur *DISK*, on aurait détecté l'exception *INCORRECT-BLOC* (§II-1.3).

Entre sa production et sa manifestation, une faute est potentielle. Entre sa production et sa détection, par un test d'assertion qui signale une exception, la faute est latente.

La faute de la ligne 8' a une période de latence nulle, car sa manifestation coïncide avec sa détection.

La conception de langages qui évitent la production des fautes, ou empêchent les fautes de rester latentes, est un art mal maîtrisé. Il est cependant admis [Ganon 75], [Horning 78], que de tels langages doivent être composés d'un nombre restreint de primitives dont la sémantique doit pouvoir être spécifiée formellement. Ces langages doivent être à types, sans goto, sans variables globales, sans pointeurs, sans conversions de types implicites etc...

III-2.4.2. FAUTES DE SPECIFICATION

Supposons que le type abstrait *RESSOURCES* du §II-2.1 ait été "spécifié" en français, comme suit :

ext function ALLOUER : 1 .&N;
si il y a encore une ressource libre, on la marque occupée et on retourne son nom,
sinon on renvoie le signal d'exception DEBORDEMENT

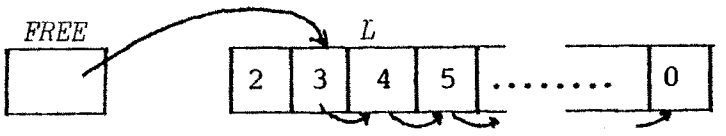
ext function LIBERER (n : 1...&N) ;
si (n < 1) ou (n > &N) on renvoie le signal RANGE ERROR,
sinon on libère la ressource n

Supposons que le programmeur qui écrit le module *RESSOURCES* , chaîne les ressources libres en une liste *FREE* . Quand une demande d'allocation est faite, il cherche dans la liste libre un élément,et il retourne son nom; en cas de demande de libération, il inclut la ressource dans la liste libre. Initialement, toutes les ressources sont dans la liste libre. Le programme de *RESSOURCES* pourrait être alors :

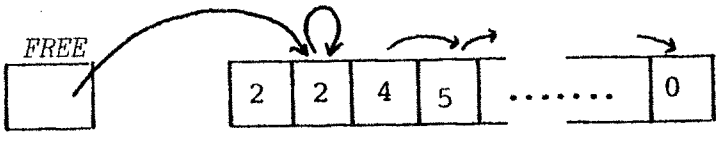
```
1   class  RESSOURCES (&N: integer);
2   type  LIBRE=0..&N; %0= nil %
3   var  L: array [1..&N]of LIBRE;
4        FREE:LIBRE;
5   ext function ALLOUER: integer; signals DEBORD;
6   begin if FREE=0 then signal DEBORD; % liste libre vide %
7         ALLOUER:=FREE; % la première ressource de la liste libre est allouée
8         FREE:=L[FREE]; %on déchaîne la ressource allouée de la liste libre%
9   end;

10  ext procedure LIBERER(N: integer); signals RANGE-ERROR;
11  begin if (N<1) or (N>&N) then signal RANGE-ERROR;
12        L[N]:=FREE; FREE:=N; %on chaîne la ressource à libérer en tête de
                                la liste libre %
13  end;
14  begin %initialement toutes les ressources sont dans la liste libre%
15        for I:=1 to &N-1 do L [I]:=I+1;
16        L[&N]:=0; %fin de la liste libre %
17        FREE:=1; %tête de la liste libre %
18  end RESSOURCES.
```

Supposons qu'un utilisateur active *ALLOUER* après l'initialisation de *RESSOURCES*. Cette activation retournera le résultat *1* et mettra le module dans l'état interne :



Supposons maintenant que la prochaine activation est *LIBERER(2)*. L'état interne de *RESSOURCES* après l'exécution de *LIBERER(2)* est incohérent, car il y a un cycle dans la liste *FREE* :



A partir de cet état (incohérent), à toutes les demandes d'allocation, le module va fournir invariablement la ressource *2* . Ce qui signifie que le module qui"gère" les ressources peut fort bien allouer la même ressource plusieurs fois, et éventuellement à des utilisateurs différents !

Ceci est en contradiction avec le concept abstrait d'allocateur de ressources. Le"mauvais" fonctionnement du module *RESSOURCES* est dû à une faute de spécification . En effet, les spécifications considérées sont "ambiguës," car elles ne décrivent pas uniquement le concept abstrait d'allocateur de ressources,mais un concept beaucoup plus vague. Ne pas avoir spécifié explicitement ce qui doit se passer quand il y a une demande de libération de ressource déjà libre, conduit à la faute concrète de la ligne 12.

Cette faute ne peut plus être qualifiée de faute de codage, car il n'y a aucune discordance entre le programme concret et les spécifications. La manifestation de la faute consiste en une violation de l'invariant qui carctérise le type utilisé "liste linéaire". Mais cette manifestation passe inaperçue, et la faute va rester latente jusqu'à ce que l'un des utilisateurs "exclusifs" de la ressource *2*, détecte que d'autres utilisateurs "exclusifs" l'ont utilisée aussi .

III-2.5. LES LIMITES DU TRAITEMENT PAR DEFAUT

L'idée de [Horning 74] , d'utiliser les techniques de"Backtracking" connues en algorithmique [Floyd 67] , [Prener 72] pour construire du logiciel tolérant aux fautes, se heurte à quelques limites d'application. En effet, l'utilisation de traitements par défaut comme ceux décrits au §III-2·3 et au §V peut être utile uniquement pour traiter des *DEFAILLANCES* résultant

des fautes de codage dont la période de latence ne dépasse pas une activation de module .

Cela suppose l'existence de spécifications complètes et non contradictoires, décrivant explicitement les invariants qui caractérisent les types utilisés. La connaissance de ces invariants permet à l'implémenteur de définir les assertions qui doivent être vérifiées durant une activation de module, pour garantir que si aucune exception n'est détectée, l'état du module après l'activation sera cohérent.

Si l'on ne possède pas de spécifications formelles, il n'y a aucun moyen de détecter à l'exécution les états incohérents.* La présence d'un tel état dans un module, par suite d'une manifestation de faute non détectée avant la fin de l'activation du module, finira par produire une *DEFAILLANCE*, peut-être dans un autre module. Le mécanisme de restauration du §V, qui intervient lors des traitements par défaut, restaure des états équivalents à ceux qui existaient avant l'activation qui a produit la *DEFAILLANCE*.

Or, si l'état interne d'avant la *DEFAILLANCE* était déjà incohérent, l'état restauré va être incohérent et le danger de résultats imprévisibles et de nouvelles *DEFAILLANCES* (voir §II-2.4) va persister.

III-3. LA TOLERANCE AUX EXCEPTIONS

III-3. TOLERANCE FAIBLE ET TOLERANCE FORTE

Considérons encore une fois les niveaux d'abstractions de la figure 3.1 et supposons qu'une exception *EL* de niveau l soit détectée en Q.

Le module Q sera dit tolérant à l'exception EL s'il contient une séquence d'opérations de traitement de EL, capable de :
- *restaurer les éléments de l'ensemble d'incohérence $\varepsilon(EL)$ et de replacer Q dans un état interne cohérent (si EL est une exception d'entrée, la séquence de restauration est vide) .*
- *signaler l'exception EL à son utilisateur N .*

---

* Le mot incohérent implique l'existence d'un terme de comparaison. En effet, quelque chose ne peut pas être incohérent en soi, mais uniquement par rapport à un certain critère. Dans cette thèse, le mot incohérent signifie toujours "incohérent par rapport aux spécifications". La cohérence peut être définie uniquement par rapport à des spécifications qui ont une existence formelle, et non pas par rapport à la représentation abstraite des objets, qui existe dans notre esprit.

Soit *EK* l'exception de niveau k qui sera (implicitement) détectée dans *N* lorsque le signal d'exception *EL* sera levé en *N* . (*EK* peut être spécifié par le concepteur de *N*, ou peut être une exception *DEFAILLANCE)*

N masque EK s'il est capable de fournir le service standard de N.B malgré la détection de EK .

Si dans *N* le concepteur n'a pas prévu de tentatives de masquage, ou si toutes les tentatives prévues produisent de nouvelles exceptions, le problème de la tolérance de *N* à *EK* se trouve à nouveau posé.

Ainsi, la propagation d'un signal d'exception à travers les niveaux d'abstraction *l*, *k*,..entraîne l'existence d'autant d'ensembles d'incohérence $\varepsilon(EL)$, $\varepsilon(EK)$,...que de niveaux traversés.

Une opération O_i implémentée par le niveau I_n, sera dite faiblement tolérante à une exception EL si tous les modules traversés durant sa propagation sont capables de la tolérer : cela signifie la restauration successive des ensembles d'incohérence $\varepsilon(EL)$, $\varepsilon(EK)$....et l'envoi d'un signal d'exception au processus P qui a activé O_i.

O_i sera dite fortement tolérante à EL si un des modules rencontrés durant la propagation de EL, exécute un masquage réussi, et arrête la propagation.

O_i est atomique et totale (§ II-1.3 et §II-2.1) *si elle est ,soit faiblement, soit fortement tolérante à toutes les exceptions pouvant être détectées durant son exécution .*

Comme le titre de notre thèse l'indique, l'objet de notre étude est le traitement des exceptions. Notre hypothèse de travail est que le concepteur d'une machine modulaire I_n connaît les propriétés invariantes caractérisant le type des variables qu'il utilise, et nous supposons que les problèmes posés par la détection des exceptions sont résolus : toute tentative de violer un invariant est signalée par une exception. Cela revient à dire que toutes les opérations définies sur les types de I_n sont totales. Par la suite,nous nous intéresserons uniquement à la propriété d'atomicité. Ainsi,quand nous parlerons d'une opération atomique, nous sous-entendrons en fait une opération totale et atomique.

III-3.2. OPERATIONS ATOMIQUES.

Au § III-1 . nous avons vu qu'un système est constitué d'un ensemble fini de processus P_1, P_2..., partageant une machine modulaire I_n, structurée en niveaux d'abstraction.

Le but du traitement des exceptions détectées dans I_n est d'assurer que les opérations implémentées par I_n sont atomiques. Le traitement d'une exception à un certain niveau d'abstraction de I_n signifie plus précisément la résolution des problèmes posés par :

a) La restauration de l'ensemble d'incohérence de l'exception.
b) Son masquage (éventuel) aux niveaux d'abstraction supérieurs.
c) Si le masquage réussit, la reprise du traitement standard, sinon la propagation du signal d'exception vers les niveaux d'abstraction supérieurs.

Pour résoudre le problème(a) pour les exceptions prévues, ainsi que pour *DEFAILLANCE*, nous proposons au §IV un mécanisme de restauration, comme une extension de la structure d'exécution modulaire de I_o .

Pour résoudre les problèmes (c) liés à la reprise, ou à la propagation des signaux d'exception entre modules, nous proposons au § V un mécanisme d'exception satisfaisant aux critères du §III-1.3.

En ce qui concerne (b), nous avons déjà exprimé notre scepticisme sur la possibilité de masquer des exceptions *DEFAILLANCE*. Le masquage des exceptions prévues dépend de la connaissance de leurs préconditions, ainsi que de la disponibilité de services substituables, et on ne peut pas donner de solution générale. Le mécanisme du § V aide le concepteur à programmer des masquages explicites. Nous illustrons cette possibilité par des exemples au § VI.

L'utilisation systématique des mécanismes du § IV et du § V doit permettre la construction de machines abstraites I_n, ayant les deux propriétés suivante

1) Intégrité. I_n possède la propriété d'intégrité si elle ne peut pas se trouver dans des états internes incohérents résultant du traitement incorrect ou de l'absence de traitement des exceptions internes à I_n.
2) Abstraction. I_n possède la propriété d'abstraction si une détection d'exception (prévue ou non prévue) pendant l'exécution d'une opération O_i sera imputé uniquement au processus P_j qui exécutait O_i (même si ce processus n'a aucune "responsabilité" dans la production de l'exception). L'occurrence d'une exception interne doit rester invisible pour les autres processus P_e exécutant indépendamment de P_j des opérations O_i sur I_n.

Après avoir reçu un signal d'exception à la suite d'une activation d'opération O_i, un processus doit également faire un traitement d'exception à son niveau, dans le but de retrouver un état de progression cohérent, vu l'absence du service standard demandé. Dans cette thèse, nous n'abordons pas de façon générale* le problème du traitement d'exception dans les processus "au-dessus" de I_n . En nous situant au niveau de I_n , nous avons ignoré la sémantique de ces processus et des relations qui les lient.

Les processus "au-dessus" de I_n peuvent être indépendants (par exemple, dans un système *MULTIACCES* comme celui du § VI, des processus indépendants sont associés à des utilisateurs indépendants). Au § VI, les processus sont cycliques, et le but du traitement d'exception dans un processus, est de le replacer dans l'état initial cohérent du début de son cycle.

Les processus peuvent être coopérants. Le problème de la restauration d'un état de progression réciproque cohérent pour ces processus est connu sous le nom de"problème de re-synchronisation"[Russel 75].Ce problème peut être résolu si le premier problème de l'atomicité des opérations du mécanisme de communication implémenté par I_n est résolu. Dans [Russel 75], où une solution à ce problème est proposée, l'auteur suppose explicitement que le mécanisme de communication "marche", c'est-à-dire, est fortement tolérant à toutes ses exceptions internes . La recherche de solutions au problème de "re-synchronisation" est un domaine qui reste ouvert .

---

* Un exemple d'un tel traitement est donné au § VI-3.5.

IV. PROPOSITION POUR UN MECANISME DE RESTAURATION .

Considérons un module N au niveau d'abstraction k , et EK une exception pouvant être détectée dans N (figure 4.1) . Le rôle d'un mécanisme de restauration est d'aider le concepteur de N à restaurer l'ensemble d'incohérence $\varepsilon(EK)$.

La mise à jour des informations redondantes nécessaires pour la restauration de $\varepsilon(EK)$, peut être laissée à la charge du niveau I_0 [Horning 74]. Dans cette approche (que nous appelons implicite, car elle décharge l'implémenteur du souci de programmer lui-même les algorithmes de restauration) , I_0 calcule à l'exécution une estimation C_k de $\varepsilon(EK)$, et utilise cette estimation pour restaurer $\varepsilon(EK)$. Au §IV-1, sous certaines hypothèses de validité, nous proposons une telle estimation . Au § IV-2, nous montrons comment la stratégie implicite, (initialement imaginée pour un langage à structure de bloc, style ALGOL) peut être intégrée dans une structure d'exécution modulaire, et nous mettons en évidence deux de ses inconvénients majeurs .

La stratégie explicite, présentée au IV-3, peut résoudre les problèmes que l'approche implicite est inapte à résoudre, mais possède à son tour, ses propres inconvénients .

Enfin, au § IV-4, nous proposons une stratégie de restauration mixte, comme un compromis entre les premières deux stratégies *.

---

* Dans [Randell 78], les stratégies explicite et implicite sont appelées "en avant" et "en arrière" (en anglais,"forward et backward") . Nous utilisons les termes implicite et explicite dans le but de mieux souligner la distribution des rôles que doivent jouer, dans la restauration, le concepteur de système d'une part, et le concepteur du langage d'autre part .

IV-1. COUVERTURE D'INCOHERENCE.

Quand le concepteur d'un module *N* (figure 4.1)écrit un traitant exlicite pour une exception *EK* , il utilise sa connaissance de la fonction d'abstraction de *N* (comme au § II-2.6), pour calculer exactement $\varepsilon(EK)$.

Mais, la définition donnée à $\varepsilon(EK)$ ne permet pas d'imaginer un algorithme efficace, pouvant être inclus dans I_0 pour calculer automatiquement $\varepsilon(EK)$ quand le concepteur de *N* n'a pas prévu de le faire lui-même .

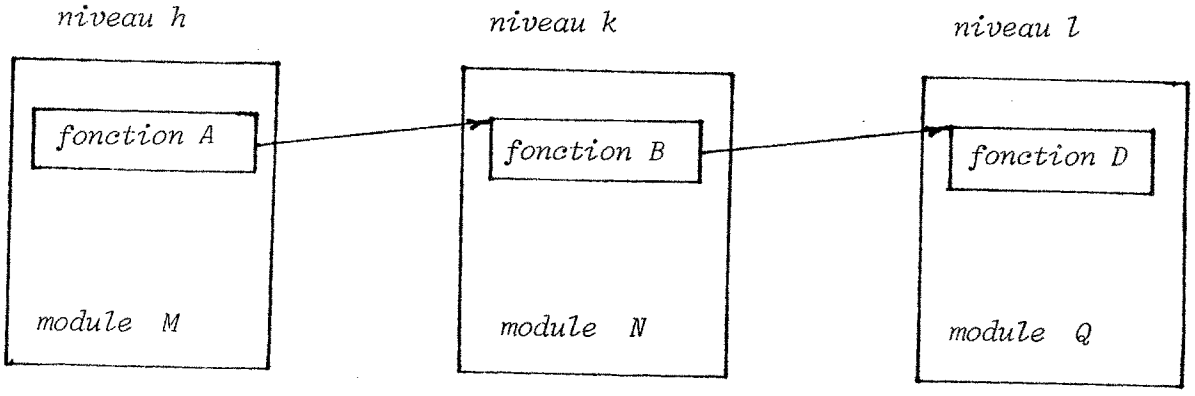


Figure 4.1

Un des avantages des techniques d'encapsulation modulaire est de fournir une structure à l'exécution rigide et bien définie . Le concepteur de I_0 peut alors se baser sur l'existence, à l'exécution, d'une telle structure pour donner une estimation a priori des ensembles d'incohérence associés à certaines classes d'exceptions (où il est sous-entendu que l'occurrence des exceptions appartenant aux classes considérées, ne doit pas invalider cette structure même) .

L'estimation de $\varepsilon(EK)$ que nous proposons, est basée sur les hypothèses "structurales" suivantes :

H1) Les modules "mémoire centrale" et "processeur" de I_0 sont corrects et fortement tolérants à toute défaillance physique transitoire ou permanente . Nous admettons par contre que des défaillances transitoires puissent survenir lors des entrées / sorties physiques, mais nous supposons que le niveau I_0 garantit que les opérations définies sur les types périphériques de I_0 sont atomiques * .

---

* Au § VI-3.2, nous décrivons comment il est possible de construire de tels types périphériques "atomiques", à partir des périphériques physiques existants,qui peuvent avorter certains transferts dans des états intermédiaires incohérents .

Les seules classes d'exceptions que nous considérons sont:

-Les exceptions prédéfinies dans le langage et signalées par le niveau de base I_0 .

-Les exceptions définies par le programmeur, ou *DEFAILLANCE*, détectées dans des modules de niveau $0 < k \leq n$.

H2) Les mécanismes de protection statiques , inclus dans le Compilateur et le Connecteur (que nous supposons corrects) , garantissent le respect, à l'exécution, de la structure modulaire spécifiée :

1). Pendant l'exécution d'une fonction *N.B*, les seules variables directement accessibles sont :

- Les variables rémanentes de *N* appartenant à des types primitifs ou structurés (accessibles en lecture / écriture)
- Les paramètres d'entrée de *N.B* (accessibles en lecture; nous supposons qu'il n'y a pas de paramètres par référence ou nom).

-Les variables locales à *N.B* (accessibles en lecture / écriture).

-Le résultat que *N.B* va rendre à son appelant (nous supposons que ce résultat peut être d'un type structuré).

Nous supposons également que I_0 ne possède pas le type pointeur .

2) . Les seules manières possibles de sortir d'un module *N* sont :

-Soit par un appel de procédure externe, implémentée par un module de niveau inférieur *Q* .(figure 4.1).

-Soit par un retour dans le module appelant *M* . Si le retour est exceptionnel, le mécanisme d'exception, intégré au langage, garantit que le contrôle sera donné soit à un traitant explicite, soit au traitant de *DEFAILLANCE* , attaché implicitement par le compilateur à la procédure appelante *M.A* .

H3) Le concepteur de la machine I_n connaît les propriétés invariantes associées avec les types utilisés dans I_n . Des tests d'assertions sont prévus dans le but de détecter toute exception qui révèle la possible violation dynamique de ces invariants .

Nous supposons que les états internes initiaux de tous les modules de I_n sont cohérents . Alors, si pendant une exécution de fonction externe, aucune exception n'est détectée, l'état interne stable d'après la sortie du module est cohérent . Dans le but de garantir qu'à l'entrée d'un moniteur son état est cohérent [Shrivastava 78], il est nécessaire d'imposer une structure particulière pour les fonctions externes des moniteurs :

ext function F(<paramètres d'entrée>):<résultats>;
[<déclarations locales>;] %les opérations entre crochets sont optionnelles %
begin [S1;]% les instructions de S1 ne modifient pas l'état du moniteur%
[wait;]
S2; % ces instructions modifient l'état interne %
[signal;]
end;

Cette structure garantit que toute séquence d'opérationsqui provoque une transition d'état interne est "non interruptible" (fin de H3) .
Notons avec C_k l'ensemble des variables d'état de N qui ont été modifiées entre l'entrée dans N et la détection de l'exception EK . Si les hypothèses H1 , H2,H3 sont valides, alors C_k est un surensemble de $\varepsilon(EK)$.

ESQUISSE DE PREUVE.

L'hypothèse H1 dit explicitement que l'on considère uniquement les classes d'exceptions qui n'invalident pas H2 et H3 . L'hypothèse H3 garantit qu'à l'entrée dans tout module N , son état interne e'est cohérent. L'hypothèse H2 permet de confiner l'ensemble des variables C_k modifiables entre l'entrée dans N et la détection de EK . Si les variables de C_k retrouvent l'état qu'elles avaient avant l'entrée dans N , le module N retrouvera un état e_r identique à e' ,donc équivalent à e' . En d'autres termes, C_k vérifie la propriété $(\varepsilon 1)$ du § II-2.5 . Comme $\varepsilon(EK)$ est par définition inclus dans tout sous-ensemble de variables d'état de N ayant la propriété $(\varepsilon 1)$, il s'ensuit que $\varepsilon(EK) \subseteq C_k$. c.q.f.d.

L'ensemble C_k peut être déterminé dynamiquement par I_0 durant l'exécution d'un module quelconque N à un niveau d'abstraction quelconque k(k=1 ,2,..,n) . Si une exception EK est alors détectée dans un tel module, I_0 peut utiliser C_k comme une estimation * de $\varepsilon(EK)$ dans le but de restaurer pour N un état interne e_r équivalent à l'état e'd'avant l'entrée dans N. Nous allons appeler C_k la couverture d'incohérence de EK .

* Nous n'aborderons pas dans cette thèse le problème de l'efficacité de notre estimation . Pour une machine abstraite donnée I_n ,si EP est l'ensemble d'exceptions qui ne peuvent pas être évitées par des moyens statiques, l'efficacité pourrait être mesurée par le taux de couverture moyen:

$$\eta = \frac{1}{card(EP)} \sum_{EK \in EP} \frac{card\,\varepsilon(EK)}{card\ C_k} \leq 1$$

Dans le cas de l'exception *DEBORDEMENT2* spécifiée pour *SEGMENTS*(§II-2.5.) ce taux est par exemple :

$$\frac{card\ \varepsilon\ (DEBORDEMENT2)}{card\ C_2} = \frac{1+1+(I-1)}{1+1+2(I-1)} = \frac{I+1}{2I} > \frac{1}{2}$$

Les variables concrètes modifiées depuis l'entrée dans une fonction externe de niveau k *(NB)* et une detection d'exception *EK* peuvent être classifiées en :

a) variables rémanentes de *N* appartenant à des types primitifs ou structurés, directement modifiables par les instructions de *N.B* . La valeur qu'une telle variable possède, au moment où *EK* est détectée, est erronée . En accord avec la terminologie introduite dans [Melliar-Smith 77], nous désignons par la suite une valeur erronée par le terme plus court d'erreur.

b) variables locales ou valeur résultat de *N.B*, ou d'autres fonctions internes à *N* appelées par *N.B*. Ces variables, se trouvant dans la pile d'exécution du processus actif au moment où *EK* survient [Montuelle 77], seront rendues inaccessibles par le retour exceptionnel signalant l'occurrence de *EK* dans *M* . N'ayant aucune influence sur le futur comportement de *N* , ces variables n'appartiennent pas à la couverture d'incohérence C_k .

c) variables rémanentes déclarées dans des modules de niveau inférieur *Q*, activés depuis l'entrée dans *N.B* jusqu'à la détection de *EK*. La valeur qu'une telle variable possède au moment de l'occurrence de *EK* sera appelée valeur résiduelle , ou plus simplement : résidu .

La production dynamique de résidus est illustrée par l'exemple suivant (figure 4.2). *FD* est un module qui implémente une variable abstraite appartenant à un type abstrait appelé *FICHIER-SUR-DISQUE* . Un processus *P* appelle *FD.CREER-FICHIER*, qui demande un buffer en mémoire centrale, en appelant *MBLOC.ALLOUER*, et un premier bloc sur disque, en appelant *DBLOC.ALLOUER*. Supposons qu'au moment où *FD* appelle *DISQUE.ECRIRE* (dans le but de transférer sur disque les informations fournies par *P*), le module *DISQUE* répond par une notification d'exception . Cela correspond à une détection d'exception intermédiaire dans *FD* . Les blocs mémoire et disque alloués depuis le début de l'exécution de *FD.CREER-FICHIER* deviennent

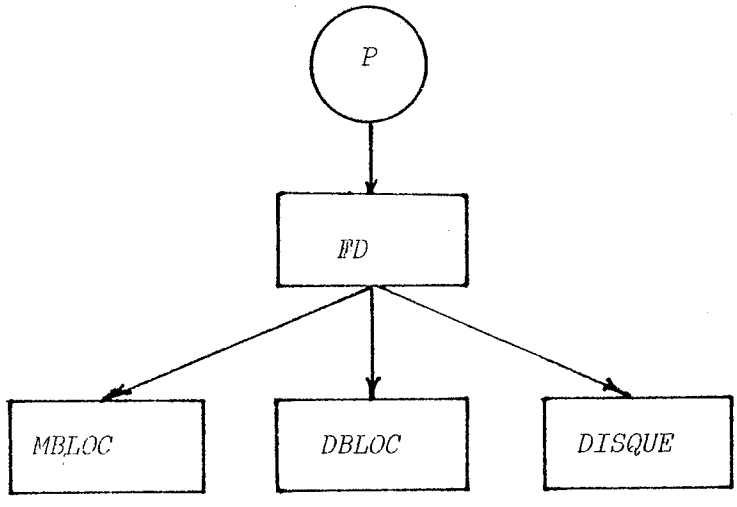


Figure 4.2

des résidus . L'existence de résidus dans les modules *MBLOC* et *DBLOC* peut provoquer, durant les futurs appels,des futures exceptions additionnelles, dues à cette première exception, et non à un épuisement réel des blocs mémoire et disque initialement disponibles .
L'état interne d'un module Q (figure 4.1) contenant des résidus, est cohérent ; en effet, cet état a été atteint par l'exécution standard d'une fonction externe de Q . La présence de résidus dans un module diminue seulement sa capacité de réponse standard aux futures demandes de service . Nous appellerons un état interne contenant des résidus, un état dégradé.

IV-2. STRATEGIE IMPLICITE.

Le fonctionnement d'un mécanisme de restauration implicite est basé sur l'estimation de l'ensemble d'incohérence donnée au § IV-1 : Les erreurs et les résidus se trouvent dans les variables d'état de N qui ont été modifiées depuis l'entrée dans le module et l'occurrence de l'exception . A chaque processus P au dessus de I_n , on doit associer un espace mémoire privé, appelé cache [Horning 74]. Le niveau I_0 a en principe deux possibilités différentes pour gérer les données redondantes nécessaires à la restauration de C_k :

A) I_0 mémorise dans le cache les valeurs que les variables concrètes d'état, modifiées par P,avaient avant leur modification. Si une exception intermédiaire de niveau k est détectée, les erreurs et les résidus peuvent être restaurés en affectant aux variables dans C_k leur valeur antérieure .

B) I_0 empêche P de modifier directement toute variable d'état, en gardant dans le cache une copie " à jour ". Si une exception survient dans N , son état interne ne doit plus être restauré . Par contre, si une opération O_i se termine normalement, la transformation d'état induite en I_n doit être "actualisée" en copiant les valeurs "actuelles" à partir du cache dans les variables d'état .

Nous décrivons par la suite, un A-cache, conçu pour une structure d'exécution modulaire . Dans [Cristian 79], nous décrivons également un B-cache ,et nous comparons le coût des A-cache et B-cache . La conclusion est que, si les exceptions sont des événements rares, le A-cache coûte moins cher que le B-cache .

A l'introduction d'un mécanisme de restauration dans I_0 , doit correspondre l'adjonction au langage de programmation d'une nouvelle primitive que nous allons appeler *reset* . Cette primitive peut être utilisée, soit dans un traitant d'exception explicite, soit dans un traitant par défaut . L'effet d'une exécution de *reset* pendant l'activation d'une fonction externe *N.B* (figure 4.1) est la restauration pour *N* d'un état interne équivalent à l'état interne d'avant l'entrée dans *N.B* .

L'adjonction d'un mécanisme de restauration implicite à SESAME (ou à n'importe quel autre langage modulaire, comme par exemple CONCURRENT PASCAL) nécessite la modification du compilateur et du noyau d'exécution *NEX*, inclus dans I_0 [Montuelle 77]:

1) Le *NEX* de SESAME a lui même une structure hiérarchique modulaire, comme le noyau de CONCURRENT PASCAL [Brinch Hansen 77]. Une nouvelle classe *ACACHE* (programmée comme les autres classes du noyau d'exécution en langage d'assemblage), doit être ajoutée au *NEX* . Chaque processus possède une instance propre,de cette classe . Dans la figure 4.3 nous décrivons le *ACACHE* dans un langage de haut niveau, style PASCAL .

2) Le code produit par le compilateur, doit être modifié comme suit:

a1) A l'entrée dans une procédure externe de niveau k *N.B* le compilateur doit insérer un appel au noyau d'exécution *ACACHE.ENTRER-NIVEAU(K)* .

a2) Pendant l'exécution d'un module *N* au niveau K, le compilateur doit insérer avant chaque première modification d'une variable d'état concrète *v* déclarée dans *N* , un appel à *ACACHE.SAUVER (adresse de v, valeur de v)* .

a3) Une primitive *reset*, pouvant être appelée avant un retour exceptionnel vers l'appelant *M* de *N* (fig-4. 1), sera traduite par le compilateur en un appel à *ACACHE.RESTAURER(K)* .

a4) Si l'exécution de *N* se termine sans aucun appel à *reset*, alors, avant le retour dans l'appelant *M* , le compilateur doit insérer un appel à *ACACHE.SORTIR-NIVEAU(K)* .

```
class ACACHE;
type  ENTREE= record AV: adresse; %adresse de variable d'état concrète%
                     VALANT: valeur;* % valeur antérieure%
              end;
      NIVEAU= sequence of ENTREE;
      CACHE= stack of ZONE;
var   C: CACHE;

ext procedure ENTRER-NIVEAU(k: integer); % k est le niveau d'abstraction
                                           entrée %
C. push (Z_k=∅); %la zone Z_k, initialement vide, va contenir les adresses
                  et les valeurs antérieures des variables de C_k %

ext procedure SAUVER(E: ENTREE);
Z_k:=Z_k ∪ E;

ext procedure RESTAURER(k: integer);
begin for each ENTREE in Z_k do [AV]:= VALANT;
      % nous notons avec [AV] l'état interne de la variable concrète
        située à l'adresse AV%
      C. pop(Z_k)
end;

ext procedure SORTIR-NIVEAU (k: integer);
% soit M le module de niveau h qui a appelé N,Fig.4.1 %
begin if k<n then Z_h := Z_h ∪ Z_k;
      %les variables concrètes modifiées dans N contiennent des résidus
      potentiels,jusqu'à ce que toute l'opération O_i se termine normalement
      C.pop (Z_k)
end;
begin % initialement, le cache est vide %
        C. empty;
end  ACACHE;
```

Figure 4.3

---

* Pour des raisons de simplicité,nous ne prenons pas en compte dans cette description,les problèmes liés à la représentation plus"longue" ou plus "courte"en mémoire,des diverses variables concrètes du langage. Dans l'implémentation du cache réalisée par [Shrivastava 78b]pour PASCAL,la granularité de la sauvegarde est le mot mémoire.

Avantages de la stratégie implicite .

1) Fiabilité . Le cache est intégré au noyau d'exécution, et l'appel aux opérations qu'il implémente est généré automatiquement par le compilateur. Si ces deux derniers programmes sont fiables, le mécanisme de restauration est fiable .

2) Simplicité . L'adoption d'une stratégie de restauration implicite libère le programmeur du souci de restaurer les ensembles d'incohérence associés aux exceptions de son programme .

Inconvénients de la stratégie implicite .

1) La restauration implicite des résidus laissés dans des moniteurs, produit " l'effet domino", décrit dans [Randell 75].

Ajoutons à la figure4.2 un nouveau module *FB*, implémentant une variable appartenant à un certain type abstrait *FICHIER-SUR-BANDE* . *FB* utilise lui aussi, des buffers gérés par *MBLOC* (figure 4.4). *MBLOC* est dans ce cas un moniteur, activé par des processus indépendants, qui mettent à jour des fichiers sur bande ou sur disque..

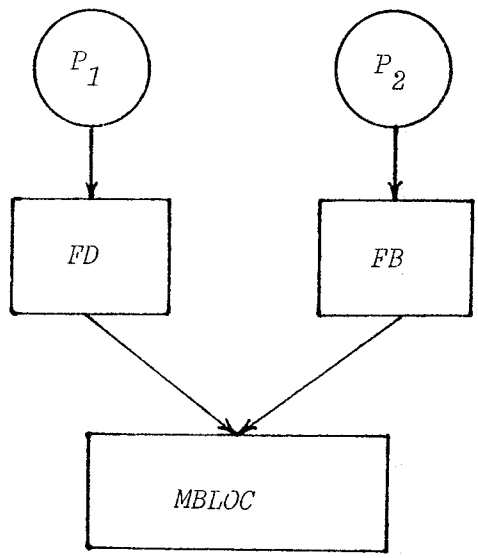


Figure 4.4

Supposons qu'à l'intérieur de *MBLOC*, les blocs mémoire libres et occupés soient représentés chaînés en deux listes linéaires *LIBRE* et *OCCUPE* , et que l'état interne initial de *MBLOC* soit celui de la figure 4.5a. Le processus P_1 appelle *FD.CREER-FICHIER* qui appelle *MBLOC.ALLOUER* .

L'état interne de *MBLOC* après ce premier appel,est représenté dans la figure 4.5b.Supposons qu'un autre processus P_2 appelle *FB.MODIFIER-FICHIER*, qui à son tour appelle *MBLOC.ALLOUER*. Le nouvel état interne de *MBLOC* est représenté en 4.5c.Supposons maintenant qu'une exception intermédiaire détectée dans *FD*,transforme le bloc *A* en un résidu. La restauration implicite des variables d'état modifiées par P_1(à l'aide d'un *ACACHE* par exemple), place *MBLOC* dans l'état interne de la figure 4.5d.Cet état est incohérent, car il ne satisfait pas l'invariant "*LIBRE et OCCUPE sont des listes linéaires* " . L'occurrence de l'état de

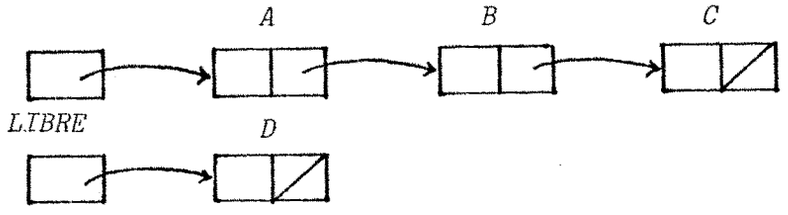


fig-4.5a

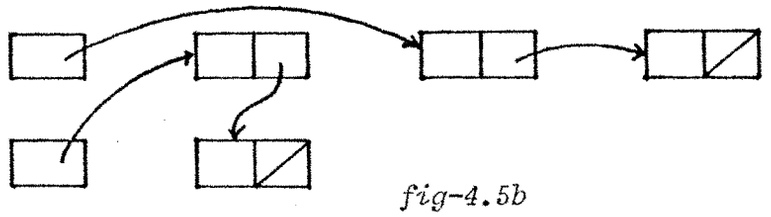


fig-4.5b

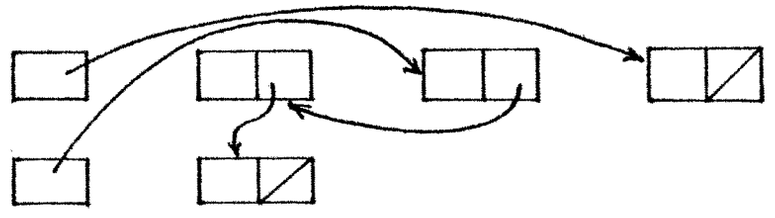


fig-4.5c

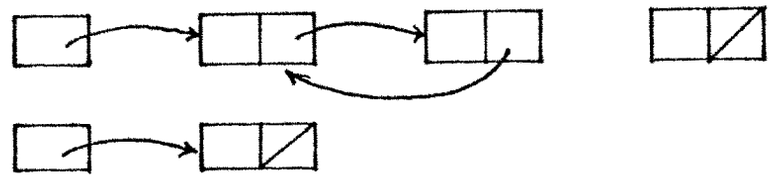


fig-4.5d

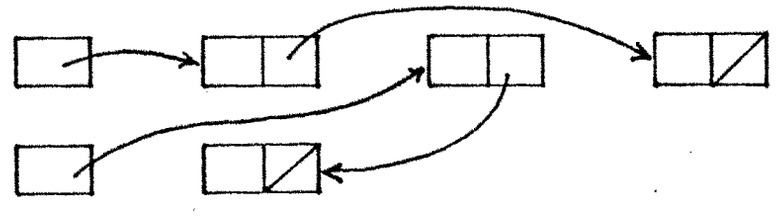


fig-4.5 e

la figure 4.5d peut être évitée si I_0 annule également les modifications induites par P_2 dans l'état de *MBLOC* . Ce module retrouve alors l'état de la figure 4.5a, qui est cohérent . Mais, le retour en arrière (en anglais "roll-back") de P_2 peut provoquer le retour en arrière d'autres processus, etc..[Randell 75]. La cause du "domino effect" est l'ignorance par le niveau I_0 de l'invariant concret caractérisant la représentation de l'état interne de *MBLOC* . Quand I_0 restaure les variables de niveau 0 modifiées par P_1 (et qui ont joué par la suite un rôle dans la transition d'état interne induite par P_2), il peut donc fort bien violer cet invariant. Une autre possibilité, pour annuler la transition d'état externe correspondant à la transition d'état interne 4.5a ⟶ 4.5b , sans que cela ne cause de"dérangement" pour P_2 , est de contraindre P_1 à exécuter la fonction d'annulation *MBLOC.LIBERER(A)* , explicitement prévue par l'implémenteur de *MBLOC*. Le programmeur de *MBLOC* peut ainsi garantir qu'après une telle action de restauration, *MBLOC* va atteindre un état interne cohérent (figure 4.5e), équivalent à un état interne qui aurait pu être atteint si uniquement P_2 (et non P_1 et P_2) avait appelé *MBLOC.ALLOUER* dans l'état 4.5a . L'implémenteur de *MBLOC* est le seul à pouvoir fournir une opération de restauration qui respecte l'invariant interne du moniteur, car il est la seule personne à connaître cet invariant .

2) La stratégie implicite ne prend pas en compte l'existence des modules périphériques .

Les algorithmes décrits précédemment supposent (implicitement) que les états internes de tous les modules d'un système sont représentés en mémoire centrale . Ceci n'est pas vrai pour les modules qui implémentent des variables périphériques . Par exemple, l'état d'un module *IMPRIMANTE* est, en partie défini par les octets de contrôle en mémoire directement accessible, en partie par l'état du papier qui a déjà été imprimé . De la même façon, l'état d'un *DISQUE* est, en partie représenté par les octets de contrôle du tournedisque, en partie par les bits du disque physique .
Les modules qui implémentent des variables périphériques appartiennent à I_0, et nous avons supposé (§ IV-1) que les opérations définies sur les types périphériques de I_0 sont atomiques . Donc, un état interne de périphérique ne peut jamais contenir des erreurs, mais peut cependant contenir des résidus . En effet, une exception intermédiaire détectée à un niveau d'abstraction supérieur, peut transformer quelques-unes des informations précédemment mémorisées par un périphérique en des résidus .

Si l'on veut pouvoir restaurer ces résidus, il est nécessaire de prévoir, pour les modules périphériques, des fonctions d'annulation . Celles-ci peuvent être des fonctions inverses : par exemple, à
DISQUE.ECRIRE(NO-BLOC,NOUVEAU-CONTENU):(NUMERO-DU-BLOC-ECRIT,ANCIEN-CONTENU)
le concepteur peut associer la fonction d'annulation :
DISQUE.ANNULER(NUMERO-DU-BLOC-ECRIT,ANCIEN-CONTENU) .
Une fonction d'annulation peut être une fonction de compensation : à
IMPRIMANTE.ECRIRE(LIGNE):NUMERO-DE-LIGNE-ECRITE ,
le concepteur de I_0 peut associer :
IMPRIMANTE.ANNULER(NUMERO-DE-LIGNE-ECRITE) , qui va par exemple imprimer le message *("ignorez S.V.P. la ligne erronée",NUMERO-DE-LIGNE-ECRITE)* [Shrivastava 78]. Dans tous les cas, le paramètre effectif d'un appel de fonction d'annulation est le résultat retourné par un appel précédent de la fonction "directe" .

IV-3 STRATEGIE EXPLICITE .

Les algorithmes de restauration ne sont plus inclus dans un interpréteur central . Le concepteur de chaque module *N*, à n'importe quel niveau d'abstraction k >0, doit prévoir explicitement les variables qui vont contenir les données de restauration, et doit programmer les traitants qui restaurent les ensembles d'incohérence associés à toutes les exceptions qui peuvent survenir dans *N* . En d'autres termes,le concepteur doit fournir des traitants pour toutes les exceptions prévues, et un traitant par défaut qui doit être activé en cas de détection d'une exception *DEFAILLANCE* (§III-2). Dans le cas des exceptions *EK* prévues, le concepteur peut utiliser sa connaissance de la fonction d'abstraction pour déterminer exactement les éléments de $\varepsilon(EK)$. Les algorithmes de restauration explicites peuvent par conséquent, être beaucoup plus performants que ceux du § IV-2 . Dans le cas d'une *DEFAILLANCE* , le concepteur de *N* peut utiliser une estimation de $\varepsilon(EK)$, et les algorithmes du traitant par défaut peuvent être analogues à ceux présentés au § IV-2 .

1) Restauration des erreurs.

Une fois $\varepsilon(EK)$ (ou au moins une estimation de $\varepsilon(EK)$) connu, les variables concrètes d'état déclarées dans *N*, peuvent être restaurées si le programmeur a prévu de sauvegarder leurs valeurs antérieures .

2) Restauration des résidus .

Le concepteur de *N* a une vue "abstraite" des résidus induits dans des modules de niveau inférieur *Q* . Il connaît uniquement les fonctions *Q.D* appelées depuis l'entrée dans *N.B*, et leur effet sur l'état externe de *Q* . Par exemple,le concepteur de *FD* (figure 4.2) sait que dans le cas où *DISQUE* signale une exception, *MBLOC* et *DBLOC* contiennent des résidus. Le résultat *A* retourné par l'appel précédent à *MBLOC.ALLOUER* (figure 5.4) renseigne *FD* sur la transformation d'état induite dans *MBLOC*.

Le concepteur de *N* ignore la structure interne de *Q*, et doit annuler toute transition d'état précédemment induite , par un appel à une fonction d'annulation *Q.AD*, connue à l'extérieur de *Q* . Par exemple, si *MBLOC.LIBERER* est la fonction d'annulation de *MBLOC.ALLOUER*, alors pour restaurer le résidu *A*, *FD* doit appeler *MBLOC.LIBERER(A)*.

Si l'exécution de *N.B* se termine sans aucune détection d'exception,mais qu'une exception est détectée ultérieurement dans un module *M* à un niveau d'abstraction supérieur (figure 4.1), alors *M* à son tour doit restaurer les éventuels résidus induits dans *N* en appelant une fonction d'annulation *N.AB*, associée par le concepteur de *N* à *N.B* . Par exemple, si *FD.CREER-FICHIER* se termine bien, mais qu'un module qui utilise *FD* doit annuler la transition d'état correspondant à une création précédente, alors la fonction d'annulation *FD.DETRUIRE-FICHIER* doit être appelée.

La restauration explicite des résidus est possible uniquement si le concepteur d'un module spécifie pour toute fonction externe provoquant un changement d'état, une fonction d'annulation .

Le lecteur peut se demander si cela est toujours possible. Nous ne pouvons pas l'affirmer. Basé sur notre expérience concrète de programmation, nous pouvons seulement affirmer qu'en pratique, nous avons pu toujours nous restreindre à ne programmer que des modules qui fournissent à leurs utilisateurs des fonctions d'annulation raisonnables. En [Cristian 76] par exemple, nous décrivons le BIBLIOTHECAIRE de SESAME,où toutes les restaurations sont explicitement programmées. Les traitants d'exception intégrés au BIBLIOTHECAIRE retaurent tous les résidus induits dans des modules de niveau inférieur par des appels explicites à des fonctions d'annulation. Au §VI, nous décrivons un système hiérarchique *MULTIACCES*, où les résidus laissés dans des moniteurs ou des périphériques, sont restaurés à l'aide de fonctionsd'annulation explicitement fournies .

La construction des fonctions d'annulation dépend de la sémantique particulière de chaque module. Souvent, le besoin de prévoir des fonctions d'annulation influence fortement la construction des fonctions directes. Ainsi, la nécessité de prévoir une action d'annulation pour une opération d"écriture de bloc contraint le concepteur du module de niveau *0 DISQUE* à transformer la

procédure "habituelle" *DISQUE.ECRIRE (NO-BLOC, CONTENU)* en une procédure un peu"moins habituelle":
DISQUE.ECRIRE (NO-BLOC, NOUVEAU-CONTENU) : (NO-DU-BLOC-ECRIT, ANCIEN-CONTENU) .

Avantages de la stratégie explicite .

1) Cette stratégie s'accomode bien de l'existence des périphérique car elle ne fait aucune hypothèse sur la représentation de l'état interne de ces modules .

2) La restauration explicite des résidus laissés dans des moniteurs n'engendre pas "l'effet domino" .

En effet, comme les fonctions d'annulation sont fournies par les implémenteurs des moniteurs, il est possible de garantir que la restauration des résidus ne viole pas les invariants caractérisant les états internes de ces moniteurs .

Inconvénients de la stratégie explicite.

1) Complexité . La nécessité de prévoir des fonctions d'annulation accroît la complexité des modules . Le programmeur doit prévoir, non seulemen des traitants pour les exceptions explicitement prises en compte, mais aussi un traitant par défaut pour une détection de *DEFAILLANCE* .

2) Non-fiabilité . C'est le corollaire de la complexité .

IV-4. STRATEGIE MIXTE.

Le but de ce chapitre est de proposer une stratégie de restauration mixte . L'idée est de limiter l'utilisation de la stratégie explicite aux situations où elle est strictement nécessaire : la restauration des résidus laissés dans les modules périphériques de I_0 et dans les moniteurs de niveau $k > 0$. Dans tous les autres cas: restauration des erreurs dans l'état de n'importe quel module de niveau $\tilde{k}>0$ et restauration des résidus laissés dans les classes de niveau $k>0$, la stratégie implicite sera utilisée . I_0 peut gérer la restauration implicite des résidus laissés dans des classes et , en même temps, aider à réaliser la restauration des résidus laissés dans des périphériques ou moniteurs . Pour cela, I_0

doit sauvegarder dans le cache:

- à la sortie d'une classe (comme au § IV.2): l'ensemble des doublets (adresse de variable concrète contenant un résidu potentiel, valeur précédente)
- à la sortie d'un module périphérique ou d'un moniteur : un seul doublet (adresse de fonction d'annulation, paramètre d'annulation).

Ceci nécessite d'imposer aux concepteurs de modules périphériques ou moniteurs, le respect des conventions de programmation suivantes :

a) pour chaque fonction externe *F* qui provoque un changement d'état, il est nécessaire de spécifier une fonction d'annulation, désignée par l'identificateur conventionnel *$F* (cet identificateur est obtenu en préfixant l'identificateur de la fonction "directe" par le caractère réservé *$*) .

b) pour permettre l'appel automatique de *$F* lorsqu'une activation précédente de *F* doit être annulée, il est nécessaire que le paramètre effectif de *$F* (que nous appelons paramètre d'annulation de *F*), calculé lors de l'appel à *F*, puisse être reconnaissable par le niveau I_0 . Ceci peut être atteint si tous les paramètres d'annulation sont préfixés par le même caractère réservé *$*.

c) si *$PAF* est le paramètre d'annulation de *F*, alors le seul paramètre formel de *$F* doit être du même type que *$PAF* .

Par exemple, le paramètre d'annulation *$PA* et la fonction d'annulation *$ALLOUER* de *MBLOC.ALLOUER* (fig-4.2) doivent être déclarés comme suit :

```
monitor MBLOC(&NB: integer); %&NB= nombre de blocs %
type  $PA =1..&NB;
  :
ext function  ALLOUER : integer, $PA;
begin .
      :
      $PA := % calcul du paramètre d'annulation de l'appel courant%
      :
end;
 :
can function $ALLOUER(P:$PA); % cancel function%
begin
      :
end;
 :
end  MBLOC;
```

D'autres exemples de paramètres et fonctions d'annulation seront donnés au § VI . Le compilateur peut vérifier statiquement le respect des règles a), b)etc) . La classe *MCACHE*, qui doit être ajoutée au noyau d'exécution, est décrite dans la figure 4.6 . Le code généré par le compilateur doit être modifié lui aussi :

M1) A l'entrée dans une procédure externe de niveau k *N.B* (fig4.1), le compilateur doit insérer un appel à *MCACHE.ENTRER-NIVEAU(k)* .

M2) Pendant l'exécution de *N* , le compilateur doit insérer avant chaque première modification d'une variable concrète d'état *v* , un appel à *MCACHE.SAUVER (implicite, adresse de v, valeur de v)* .

M3) La primitive *reset* est traduite dans un appel à *MCACHE.RESTAURER(k)* .

M4) Si l'exécution de *N* se termine sans aucune activation de *reset*, alors, si *N* est une classe, le compilateur doit insérer un appel à *MCACHE.SORTIR-CLASSE(k)*, et si *N* est un moniteur ou un module périphérique, le compilateur doit insérer un appel à *MCACHE.SORTIR-MONITEUR-OU-PERIPHERIQUE(k)* .

```
class  MCACHE;
type   ENTREE = record case TAG:(implicite, explicite) of
       implicite : (AV: adresse, VALANT: valeur);
       explicite : (AFA: adresse, PA: valeur); %Adresse de Fonction
                   d'Annulation, Paramètre d'Annulation %
       end;
       ZONE = sequence of ENTREE;
       CACHE = stack of  ZONE;
var    C : CACHE;

ext procedure ENTRER-NIVEAU(k: integer);
C. push (Zk = ∅ );


ext procedure  SAUVER (E:ENTREE);
Zk := Zk∪E ;

ext  procedure  RESTAURER (k: integer);
begin for each  ENTREE in Zk do
       case  TAG of
       implicite: [AV] := VALANT;
       explicite: appeler la fonction dont le point d'entrée est AFA avec
                  comme paramètre effectif PA[DEFAILLANCE:<voir remarque>];
       end;
       C. pop (Zk);
end;

ext procedure SORTIR-CLASSE (k: integer);
begin if k<n then Zh := Zh∪ Zk ;
      C.pop (Zk);
end;

ext procedure SORTIR-MONITEUR-OU-PERIPHERIQUE (k: integer);
begin if k<n then% $B et $PAB sont la fonction et le paramètre d'annulation de NB,
         Zh := Zh ∪ ( explicite, adresse de $B, valeur de $PAB );fig 4.1 %
         C. pop (Zk)
end;

begin  C.empty;
end  MCACHE;
```

Figure 4.6

REMARQUE : Pour une fonction d'annulation, soit *N.$B*, on demande uniquement la disponibilité de l'effet standard . Celui-ci, consistant en une restauration d'état interne équivalant à un état interne qui existait précédemment, est en principe toujours réalisable . Mais comme les fonctions d'annulation sont fournies par le programmeur, elles peuvent contenir des fautes . Ces fautes peuvent provoquer des *DEFAILLANCES* . Dans le cas où une *DEFAILLANCE* est détectée durant l'activation d'une fonction d'annulation, la continuation de l'activité du système dépendra des exigences contenues dans le cahier des charges .

Si la dégradation de l'état interne est inacceptable pour des raisons de sécurité,une détection de *DEFAILLANCE* pendant l'activation d'une fonction d'annulation arrêtera le système et avortera les processus "au dessus" de I_n.

Pour certaines applications, l'ensemble *M* des modules d'un système peut être partitionné en un sous-ensemble *EM* $\subset$ *M* de modules dont les fonctions sont considérées comme"essentielles" pour l'accomplissement de la mission du système, et en un sous-ensemble *SM*$\subset$*M* de modules dont les fonctions sont considérées comme "secondaires" par rapport à cette mission. L'occurrence éventuelle d'états internes incohérents dans les modules de *SM* est acceptable : un risque positif de détecter des futures *DEFAILLANCES* additionnelles dans ces modules n'est pas considéré comme"catastrophique". Par contre, ce risque doit rester toujours nul pour les modules de *EM* . Soit *N* (figure 4.1) un module dans *EM*, et son appelant *M* un module dans *SM* . Supposons,qu'à la suite d'une détection d'exception intermédiaire dans *M* ,la fonction d'annulation *N.$B* soit activée, et que cette activation produise une *DEFAILLANCE* dans *N.$B* . Le mécanisme d'exception peut donner le contrôle au traitant pour *DEFAILLANCE* (§ III-2) attaché à *N.$B* qui contient un appel à *reset*. Si l'exécution de *reset* ne produit aucune autre exception, *N* retrouvera un état interne cohérent, mais dégradé (voir § IV-1), équivalent à l'état d'avant l'activation de *N.$B* . Mais l'indisponibilité de l'effet standard de *N.$B* produira un état interne incohérent stable dans *M* . Comme *M* est dans *SM* , cela est acceptable, et le système peut continuer son exécution.

Donc,si le risque d'apparition d'états internes incohérents est jugé acceptable pour certains modules"secondaires" d'un système, alors il est possible de continuer à exécuter le système en dépit de certaines détections de *DEFAILLANCE* dans des fonctions d'annulation.

IV-5. RESUME .

Si les mécanismes de protection (statiques et dynamiques) peuvent garantir la validité d'une structure modulaire à l'exécution, alors il est possible d'estimer tout ensemble d'incohérence, associé à une exception détectée dans un système, par ce que nous avons appelé la couverture d'incohérence de l'exception. Les données d'état endommagées, incluses dans la couverture d'incohérence, ont été partitionnées en erreurs et résidus .

Basé sur une telle estimation, nous avons analysé deux stratégies de restauration :

- La stratégie implicite, proposée dans [Horning 74],est facilement adaptable à une structure d'exécution modulaire, et les algorithmes de gestion du cache sont plus simples que ceux qui ont été proposés dans [Horning 74],à cause de l'absence des variables globales. Mais cette stratégie ne prend pas en compte l'existence des périphériques, et provoque le "domino effect" .

- La stratégie explicite apporte une solution au problème des périphériques, et évite la production du "domino effect", mais elle accroît la complexité des programmes et, par conséquent, diminue leur fiabilité .

- La stratégie mixte que nous avons proposée, est un mélange des deux stratégies précédentes : la restauration des erreurs et des résidus laissés dans des classes est implicite, mais la restauration des résidus laissés dans des moniteurs ou des périphériques se fait à l'aide de fonctions d'annulation .

Si l'on veut imposer une utilisation systématique de la stratégie mixte pour restaurer les ensembles d'incohérence associés aux *DEFAILLANCES* d'un système, il est nécessaire d'intégrer le mécanisme de restauration du §IV-4 au langage d'implémentation . Rien n'empêche, bien sûr, le programmeur d'utiliser la primitive *reset* lors du traitement des exceptions prévues .

Au § VI nous montrons qu'un langage de programmation qui possède un mécanisme de restauration intégré, est un outil commode pour la construction de machines abstraites possédant les propriétés d'intégrité et d'abstraction définies au § III-3 .

V - PROPOSITION POUR UN MECANISME D'EXCEPTION

Un mécanisme d'exception permet
- à l'implémenteur d'une opération de signaler une détection d'exception;
- à l'utilisateur de l'opération de programmer le traitement de l'exception.

Nous décrivons le mécanisme d'exception que nous proposons, en présentant les services qu'il fournit aux programmeurs (§V.1 - §V.6). Au §V.7 nous analysons dans quelle mesure ce mécanisme satisfait aux critères énoncés au §III.1.3.

V - 1. DECLARATION STATIQUE DES EXCEPTIONS

Une déclaration d'exception doit être intégrée à une déclaration d'opération
Pour chaque opération définie sur un type concret (§II.1.1) le concepteur du langage doit spécifier les exceptions qui peuvent être détectées (§II.1.2 et §II.1.3). Ces exceptions ont des identificateurs prédéfinis et sont implicitement déclarées dans tout programme. Le programmeur connaît les exceptions de niveau 0 à partir des spécifications du langage.
Par exemple :

- L'utilisateur du type concret *integer* doit savoir que l'opération de division entière *div* peut détecter l'exception prédéfinie *DIVISION - BY - ZERO.*
- L'utilisateur du type concret *DISK* (§II.1.3) doit savoir que l'opération *WRITE (B : BLOC; BUF : DISKBLOC)* peut détecter les exceptions de niveau 0 : *INCORRECT-BLOC, TRANSMISSION, INTERVENTION.*

Pour résoudre un problème il est souvent nécessaire de construire des nouveaux types et opérations, qui ne sont pas disponibles dans le langage. Les nouvelles opérations doivent être implémentées par des procédures (ou fonctions)*. Le programmeur peut distinguer l'effet standard d'une activation de procédure des i = 0,1,2,.. effets exceptionnels possibles. Pour cela, il doit déclarer explicitement dans l'en-tête de la procédure les identificateurs des i exceptions qui peuvent être détectées. Ces identificateurs, créés par le programmeur, doivent être introduits par la clause déclarative *signals* et sont exportés vers l'utilisateur de la procédure. Par exemple :

- L'implémenteur de la fonction externe *ALLOUER* de *RESSOURCES* (II.2.2) doit déclarer l'exception *DEBORD* comme suit :

```
ext function ALLOUER : integer; signals DEBORD;
% corps de la fonction ALLOUER %
```

Une déclaration d'exception intégrée à une déclaration de procédure externe de module, rend le nom de l'exception connu à l'extérieur du module. Par contre, un identificateur d'exception déclaré dans l'en-tête d'une procédure interne reste inconnu aux utilisateurs du module.

---

* Des concepts comme co-routine, itérateur ne sont pas implémentés en SESAME

Toutes les exceptions pouvant être détectées dans des modules de niveau $k > 0$ doivent être déclarées explicitement par les concepteurs de ces modules, sauf l'exception prédéfinie *DEFAILLANCE* . L'identificateur *DEFAILLANCE* ne doit donc pas figurer dans une liste de noms d'exceptions qui suit une clause *signals*. Chaque activation de procédure (fonction) peut en effet produire par convention une exception *DEFAILLANCE*.

V - 2. ASSOCIATION STATIQUE DES TRAITANTS EXPLICITES A DES POINTS D'ACTIVATION ET A DES EXCEPTIONS

Soit *OP* une opération pour laquelle l'implémenteur à déclaré une liste d'exceptions $EOP = \{.., E, ...\}$
Une activation de *OP* est une tentative de réaliser l'effet standard de *OP*. Un point d'activation est une instruction simple (affectation, appel de procédure) dans un programme dont l'exécution entraîne l'activation de *OP*. Par exemple, si l'opération *OP* est implémentée par une procédure, alors le programme

```
while ( ) do
begin .
      .
      .
      OP;
      .
      .
      .
end;
```

contient plusieurs activations possibles de *OP*, mais un seul point d'activation.
Le programmeur peut associer un traitant explicite *T* à un point d'activation de *OP* et à une exception particulière $E \in EOP$. Pour cela, il doit utiliser une directive d'association de la forme

I [*E* : *T*] ;

où avec *I* on a noté une instruction englobant syntaxiquement un point d'activation de *OP* . Une juxtaposition d'instruction *I* et de directive d'association [E : T] sera appelée instruction protégée. La figure 5.1 contient une description des règles syntaxiques permettant la construction d'instructions protégées.
I peut être une instruction simple dont l'exécution entraîne au moins une activation de *OP* . Par exemple si *OP* est une procédure, alors un traitant pour *E* peut être associé à l'instruction d'appel de *OP* :

OP [*E* : *T*] ;

Le traitant *T* peut contenir à son tour un point d'activation d'une autre opération *OP1*, pouvant détecter une autre exception *E1*. Le programmeur peut alors prévoir un traitant *T1* pour *E1* (imbrication de traitants) :

OP [*E* : *T* [*E1* : *T1*]] ;

L'instruction *I* peut être une instruction composée (instruction bloc, instruction itérative, etc, voir Fig.5.1) contenant au moins un point d'activation de *OP* :

```
while (  ) do
begin .
      .
      .
      OP;
      .
      .
      .
end [E : T] ;
```

Si une instruction composée contient plusieurs activations de l'opération *OP* sur des variables (concrètes ou abstraites) distinctes *V1*, *V2* alors les ambiguités peuvent être évitées si le programmeur préfixe le nom de l'exception *E* par le nom des variables sur lesquelles l'opération *OP* est effectuée :

```
while (  ) do
begin .
      .
      .
      V1.OP;
      .
      .
      V2.OP;
      .
      .
end [V1.E : T1, V2.E : T2];
```

T1 est le traitant associé à l'exception *E* détectée par l'activation *V1.OP*.

```
<instruction protégée> ::= <instruction> <directive d'association>
<directive d'association> ::= [<liste d'associations>]
<liste d'associations> ::= <nom d'exception> : <traitant>|
                           <nom d'exception> : <traitant>, <liste
                                                           d'associations>
<nom d'exception> ::= <identificateur>|
                      <identificateur> . <identificateur>
<traitant> ::= <instruction>|
               <instruction protégée>|
               <instruction signal ou reset> % voir § V.4%
<instruction> ::= <instruction simple>|
                  <instruction composée>
<instruction simple> ::= <affectation>|
                         <appel de procédure>|
                         <instruction vide>
<instruction composée> ::= <instruction bloc>|
                           <instruction case>|
                           <instruction if>|
                           <instruction while>|
                           <instruction repeat>|
                           <instruction for>|
                           <instruction with>|
```

Figure 5.1

V - 3. LES TRAITANTS PAR DEFAUT

Supposons que le programmeur d'un module *N*, au niveau d'abstraction k, ait besoin d'utiliser une opération *OP* (implémentée par un niveau d'abstraction < k) dans une procédure externe *N.B* (Fig.5.2).
Soit *EOP* l'ensemble des exceptions déclarées pour *OP*, et $E \in EOP$.

```
module N (      );
:
ext procedure B (      ); signals...;
begin .
      :
      begin
(I3)  (I2)  :
      begin (I1) OP;
            :
      end [E : T1];
      :
end  [E : T2, DEFAILLANCE : <traitant par défaut> % voir §V.4%];
:
end N;
```

Figure 5.2

Le traitant explicite activable si l'exception *E* est détectée au point d'activation *I1* est, par définition, celui qui est attaché à la plus petite instruction protégée englobant *I1*. Dans notre exemple, le traitant activable est *T1*, et non *T2*, car *T2* est attaché à l'instruction *I3* qui englobe l'instruction *I2*.
Pour chaque instruction simple *I* d'un programme, dont l'exécution entraîne l'activation d'une opération *OP* pouvant détecter une exception $E \in EOP$, le compilateur peut déterminer statiquement s'il existe un traitant explicite activable, et si oui, quelle est l'adresse d'implantation de ce traitant*. En effet, tous les traitants explicites doivent être associés statiquement aux instructions du programme et aux exceptions pouvant être détectées par ces instructions (§V.2). Soit $EOPTE \subset EOP$ le sous-ensemble des exceptions pour lesquelles il existe au moins un traitant explicite activable.

* Pour simplifier notre exposé, nous avons supposé que l'exécution d'une instruction simple entraîne l'activation d'une seule opération. Si *I* est une affectation ayant comme partie droite une expression dont l'évaluation fait intervenir plusieurs opérations *OP1,OP2,..., OPI*, alors le compilateur peut faire le même travail pour toute exception $E \in EOP1, \ldots, EOPI$ pouvant être détectée durant l'exécution de *I*.

Au §III.2 nous avons admis qu'il est possible d'avoir *EOP-EOPTE* ≠ { }. Comme les ensembles *EOP*, *EOPTE* peuvent être déterminés statiquement, l'ensemble *EOP-EOPTE* des exceptions pour lesquelles il n'existe aucun traitant explicite activable, peut être également déterminé statiquement. La détection d'une exception $D \in$ *EOP-EOPTE* pendant l'exécution de *OP* correspond à la détection d'une exception prédéfinie *DEFAILLANCE* dans le programme qui a activé *OP*.

1) Si l'instruction *I* qui a signalé l'exception *D* fait partie d'un module *N*, nous avons argumenté aux §III.1.3.1 et §III.1.3.2 qu'il est nécessaire d'imposer un traitement par défaut de *D* dans *N*, pour des raisons de sécurité et de modularité. Le traitant par défaut sera, par définition, le traitant pour *DEFAILLANCE* attaché implicitement à la procédure externe *N.B* qui englobe *I* (Fig.5.2). Au §V.4 nous verrons que tous les traitants per défaut sont identiques, et nous détaillerons leur contenu. Si le compilateur protège systématiquement toute procédure externe de module *N.B* par un traitant pour *DEFAILLANCE*, il est possible de garantir qu'à l'exécution, pour toute exception *E* pouvant être levée dans *N*, il existe un traitant activable. Si $E \in$ *EOPTE* , ce traitant est explicite, et si $E \in$ *EOP-EOPTE*, alors ce traitant est le traitant par défaut. La propriété d'étanchéité des frontières de modules face à la propagation des exceptions est ainsi assurée.

2) Si l'instruction *I* qui signale l'exception $D \in$ *EOP-EOPTE* fait partie d'un programme de processus *P* (Fig.5.3), alors le seul traitement par défaut qui est envisagé dans cette thèse est l'arrêt de l'exécution de *P* (dans une phase de mise au point, ce traitant peut être un appel à un module de mise au point, une tentative de diagnostic, une impression d'état, etc). En effet, au §III.3.2, nous avons vu que les traitements par défaut considérés pour les modules de I_n, ne résolvent pas les problèmes de "re-synchronisation" qui se posent "au dessus" de I_n .

```
process P (    );
%déclaration des droits d'accès de P%
:
begin % l'algorithme de P %
      :
      :
      I; % il n'existe aucun traitant explicite activable pour D ∈ EOP %
      :
      :
end [DEFAILLANCE :  <arrêter l'exécution>].
```

Figure 5.3

V - 4. SIGNALISATION D'UNE DETECTION D'EXCEPTION

Soit *U* un programme qui utilise une opération *OP* (soit prédéfinie au niveau I_0, soit construite par un programmeur) pour laquelle on a déclaré les exceptions *EOP* = {..., *E*,...} . Supposons que pendant l'exécution de *OP*, l'exception $E \in EOP$ soit détectée. L'implémenteur de l'opération *OP* doit signaler au programme *U*, qu'à la place du service standard on lui délivre le service exceptionnel *E*. Au II.1.4 nous avons argumenté que la meilleure façon de signaler la détection de *E* à *U*, est de provoquer une rupture de séquence dans *U* : au lieu de continuer à exécuter l'instruction qui suit l'activation de *OP*, on exécutera le traitant activable pour *E* dans *U*. Si l'on se place maintenant au niveau d'observation de *U*, nous dirons que l'activation de *OP* lève (en anglais "raises" [Goodnough 75]) l'exception *E*, quand à la place de l'instruction qui suit *OP* on exécute le traitant de *E*.

L'implémenteur de I_0 doit prévoir de signaler toute détection d'exception de niveau 0 au programme qui est en train d'exécuter l'opération concrète qui a causé l'exception.

Pour signaler une détection (explicite ou implicite, voir §II.2.2) d'exception *EK* de niveau $k>0$ pendant une exécution de procédure *B*, l'implémenteur de la procédure doit utiliser une instruction *signal* ou *reset*.

Sémantique de signal

Le mécanisme d'exception interprète l'exécution d'une instruction *signal EK* dans une procédure *B* de la façon suivante :

```
procédure B (      ); signals EK,..;
var LOCALE : type;
begin  b1;
       .
       .
       bi-1;
       if (  ) then signal EK;
       bi+1;
       .
       .
       bn;
end;
```

- on n'exécute plus les instructions $b_{i+1}, .., b_n$ qui suivent le *signal*,
- le contexte d'activation de *B* (contenant entre autres la variable *LOCALE*) est dépilé, l'activation courante est considérée terminée,
- si *B* est une fonction, alors aucune valeur n'est assignée au résultat de *B*,
- l'exception *EK* est levée (voir plus loin §V.5) dans le programme qui a activé *B*.

Sémantique de *reset*

L'instruction *reset* ne peut être utilisée que dans une procédure externe de module *N.B*, ou dans un traitant attaché à toute une procédure externe* (par exemple un traitant pour *DEFAILLANCE)*.

Le mécanisme d'exception interprète une exécution de *reset EK* comme suit :

- Il exécute d'abord une opération *RESET(k)* sur la variable abstraite de type *MCACHE* qui est propre au processus en train d'exécuter *N.B* (§IV.4). L'activation de cette opération implémentée par le noyau d'exécution, a pour effet de placer le module *N* dans un état interne équivalent à l'état interne d'avant l'activation de *N.B*.
- Il fait ensuite exactement le même travail que pour une instruction *signal EK* : termine l'activation courante de *N.B* et lève l'exception *EK* dans le programme qui a appelé *N.B*.

---

* Cette restriction est imposée par le fait que le mécanisme de restauration du §IV, a été conçu pour résoudre le problème de restauration d'état dans un module quelconque *N* de I_n, à la suite d'une détection d'exception intermédiaire dans *N* et avant la sortie définitive de *N*. Ce mécanisme ne résoud pas le problème de re-synchronisation qui se pose au niveau des processus. Ce problème doit être explicitement résolu par le concepteur de chaque processus, car il est le seul à connaître la sémantique des relations qui le lient aux autres processus du système. Pour programmer le traitement d'une exception, l'implémenteur d'un processus peut bien sûr utiliser tous les autres services du mécanisme d'exception.
Il nous faut également remarquer que pour des raisons de prévention de deadlock, ainsi que pour des raisons de restauration, une procédure externe de module *N.B* ne doit pas être récursive. En effet, si *N* est un moniteur, et si *N.B* est récursive, la première activation de *N.B* conduit à un deadlock [Brinch Hansen 77]. De même, si une instruction *reset EK* est exécutée dans une activation imbriquée récursivement de *N.B*, alors la levée de *EK* dans les activations de niveau d'imbrication inférieur risque de conduire à de nouvelles exécutions de *reset*. Or le mécanisme du §IV.4 a été conçu dans l'hypothèse : toute activation de procédure externe au niveau d'abstraction k peut contenir au plus une activation de *RESET(k)*.

Dans une procédure, on ne peut signaler que les exceptions qui ont été explicitement déclarées dans l'en-tête de la procédure, ou l'exception *DEFAILLANCE*. Cette règle peut être vérifiée à la compilation.
Le concept de *DEFAILLANCE* a été introduit au §III.2 pour identifier le service exceptionel qu'une procédure externe *N.B* rend à son utilisateur dans le cas où une exception non traitée explicitement est levée à l'exécution dans *N.B*. Le rôle du traitant par défaut attaché à *N.B*, est de restaurer l'état de *N* et de signaler la détection de l'exception *DEFAILLANCE* à l'appelant de *N.B*. Tout cela peut être fait par une seule instruction : *reset DEFAILLANCE*. Tout traitant par défaut ne contiendra donc que cette seule instruction :

<traitant par défaut> :: = reset DEFAILLANCE

Pour les opérations *signal* et *reset*, seul l'effet standard est défini. Une activation d'instruction de signalisation ne peut lever aucune excepti

Le lecteur peut fort bien se demander : et que se passe-t-il si durant une exécution de *signal* ou de *reset*, le niveau I_0 détecte une nouvelle exception ? Peut-on utiliser un mécanisme d'exception pour traiter à l'exécution ses propres exceptions ?

Au §IV.4 nous avons remarqué que durant une exécution de *reset*, il est possible de détecter une *DEFAILLANCE* dans une fonction d'annulation. Mais la réponse à la question "va-t-on arrêter l'exécution?" doit être connue à l'avance, et la décision à prendre doit être intégrée dans l'algorithme du mécanisme d'exception. Ainsi, si aucune dégradation d'état interne n'est permise, l'exécution doit s'arrêter.
Dans le cas d'un système à dégradation progressive des performances dont les modules sont partitionnées en "principales" et "secondaires", le mécanisme d'exception doit tenir compte du fait qu'un reset est exécuté dans un module *N* principal ou secondaire : si durant une exécution de *reset*, le niveau I_0 détecte une *DEFAILLANCE*, alors, si *N* est un module principal, l'exécution doit s'arrêter, et si *N* est un module secondaire, l'exécution doit continuer. Mais, dans ce dernier cas, le *reset* en question ne doit lever en aucun cas une *DEFAILLANCE* dans *N*. Le mécanisme d'exception doit masquer cette *DEFAILLANCE*. Il s'ensuit que dans un tel système, les modules secondaires ne "sauront" jamais si leur état interne est cohérent ou non, car ils ne peuvent pas "savoir" si l'effet standard

d'un *reset* est, ou n'est pas, réalisé. Ils continueront de toute façon à répondre aux futures demandes de service, comme si de rien n'était. Bien sûr, la probabilité de détection de *DEFAILLANCES* additionnelles peut devenir positive dès la première exécution d'une instruction *reset* qui produit (et masque) une *DEFAILLANCE*. Mais ce risque est acceptable par hypothèse.

Si l'on admet que les opérations *signal* et *reset* puissent lever des exceptions, alors le programme qui les active aurait de fortes chances de boucler à l'infini, en essayant de signaler une exception,qui signale une nouvelle exception, qui signale une nouvelle exception,etc. La nécessité de rompre de tels cercles vicieux nous conduit à considérerle très philosophique problème du "hardcore", c'est-à-dire à admettre l'existence d'un noyau minimal de primitives auxquelles on peut faire confiance. Le lecteur qui connaît la complexité des compilateurs peut sourire quand on lui parle de "hardcores", car il sait que, souvent, un compilateur qui lui est présenté comme un "hardcore" s'avère lors de son utilisation n'être qu'un "softcore". Faut-il se résigner à admettre que le mécanisme d'exception doit être un "hardcore", ou faut-il poursuivre les recherches, et imaginer des mécanismes capables de traiter leurs propres défaillances ? Face à cette -admettons-le- très embarassante question, notre réponse est optimiste : si un mécanisme d'exception est suffisamment simple et sa sémantique suffisamment bien définie, il doit être possible de l'implémenter correctement, de façon à garantir que l'activation de ses primitives ne lève pas d'autres exceptions.

V - 5. LEVEE D'UNE EXCEPTION

Considérons de nouveau le programme U qui utilise l'opération OP pour laquelle l'implémenteur a déclaré les exceptions $EOP = \{..., E, ...\}$. Si une exception E est signalée pendant une activation de OP, alors la tâche du niveau I_0 (et plus précisément du mécanisme d'exception intégré dans I_0) sera de lever E dans U, c'est-à-dire d'exécuter le traitant activable pour E dans U, à la place de l'instruction qui suit le point d'activation de OP. L'insertion systématique par le compilateur de traitants par défaut, garantit qu'un tel traitant,soit T, existe toujours.

L'adresse d'implantation de *T* est connue à la compilation de *U*.

La levée de *E* n'entraîne aucun changement de contexte dans *U* : le traitant *T* peut accéder toutes les variables qui étaient accessibles au moment où l'opération *OP* a été activée.
Par exemple, la construction :

```
    procédure U (<paramètres d'entrée>);
    var A : type de A;
    begin .
          .
(b)       OP;
          .
          .
    end [E : A := A+1];
```

est légale, malgré le fait que la référence à la variable locale *A* figure en dehors du bloc *b* qui définit classiquement la portée de *A*. L'adjonction d'une directive d'association à l'instruction bloc *b* étend la portée de *A* à toute l'instruction protégée *b*.
Au niveau d'abstraction de *U*, on ne doit pas savoir si *E* est signalée dans *OP* à l'aide d'une instruction *signal*, ou d'une instruction *reset*. L'effet exceptionnel *E* de *OP* ne doit être connu par le concepteur de *U*, qu'à travers les spécifications de *OP*. Si au niveau d'abstraction qui implémente *OP*, à l'exception *E* correspond un $\varepsilon(E)$ non vide, alors le fait de restaurer $\varepsilon(E)$ implicitement ou explicitement est une décision qui concerne exclusivement l'implémenteur de *OP*. Le mécanisme d'exception est construit de façon à assurer que cette décision reste invisible aux utilisateurs de *OP*.

V - 6. EXECUTION D'UN TRAITANT EXPLICITE

La levée d'une exception *E* dans un programme *U* au niveau d'abstraction $k > 0$, coïncide avec une détection implicite d'exception de niveau *k* dans *U* (§II.2.2).
Si dans *U* il n'existe aucun traitant explicite pour *E*, alors l'exception de niveau *k* qui sera détectée est une exception *DEFAILLANCE*. Nous avons présenté la sémantique des traitants pour *DEFAILLANCE*, générés automatiquement par le compilateur pour les procédures externes des modules et pour les programmes des processus, aux §V.3 et §V.4.

Supposons que le concepteur de E ait prévu un traitant explicite T pour E et soit EK l'exception de niveau k dont la détection (implicite) coïncide avec l'activation de T. Le programmeur de T doit résoudre les problèmes suivants :

a) Masquage éventuel de EK. Si le masquage réussit, il est nécessaire de reprendre l'exécution de la séquence standard d'instructions de U.

b) Si aucune tentative de masquage n'est prévue, ou si toutes les tentatives lèvent de nouvelles exceptions, alors, si U est une procédure de module N, il est nécessaire de

- restaurer $\varepsilon(EK)$ dans le cas où EK est une exception intermédiaire,
- signaler EK au programme qui a appelé N.

Dans le cas où U est un processus, les conséquences d'un masquage non réussi n'ont pas été abordées de façon systématique dans cette thèse (§III.3.2). Des exemples de traitements explicites d'exceptions sont donnés au §VI pour les processus d'un système transactionnel *MULTIACCES*. Dans le cas particulier de ce système, chaque processus est cyclique et indépendant des autres, et l'impossibilité de masquer une exception levée par I_n entraîne tout simplement le repositionnement du processus en début de cycle, en attente d'une nouvelle transaction.

Masquage

Une tentative de masquage d'une exception EK consiste en l'activation d'une opération OP' dont l'effet standard peut être substitué à celui de OP pour réaliser la transformation d'état standard spécifiée pour U. Quand OP est une opération de transfert avec un périphérique, et que E est une exception considérée comme transitoire, alors OP' est la même opération que OP et les tentatives de masquage consistent dans la répétition d'un nombre fini $(tm > 1)$ de fois de l'activation de OP . Au §VI.3.2 nous donnons des exemples de masquage par activation d'opérations substituables, et aux §VI.3.1, §VI.3.4 et VI.3.5 nous donnons plusieurs exemples de masquages d'exceptions transitoires par activation d'une même opération. La programmation des masquages successives de EK peut être faite en imbriquant des traitants :

```
procédure U (      ); signals EK;
begin
        .
        .
        .
        OP [E : OP' [E' : OP" [E" : signal EK]]]; % deux tentatives
        .                                          de masquage %
        .                          T" de E"
end;                    T' de E'
        T de E
```

Toutefois, dans le cas des exceptions transitoires, le nombre de tentatives de masquage peut être assez élevé. (Exemple : si un utilisateur interactif de *MULTIACCES* tape à sa console une commande erronée, le module qui gère la communication avec cette console détecte l'exception transitoire *IC* (Incorrect Command). Une tentative de masquage consiste dans une lecture de nouvelle commande. *MULTIACCES* exécute 12 tentatives successives de masquage). Si le nombre de tentatives de masquage est élevé, l'utilisation de traitants imbriqués alourdit beaucoup l'écriture,à cause de l'imbrication des crochets. Pour résoudre ce problème, nous avons adopté une primitive *retry*, analogue à celle qui existe dans MESA [Mitchell 78]. L'exécution d'une instruction *retry* dans un traitant d'exception, provoque la ré-exécution de l'instruction à laquelle le traitant est attaché. La programmation d'un nombre quelconque *tm* de tentatives de masquage pour une exception transitoire *E* peut se faire ainsi :

```
procédure U (     ); signals EK;
var essai : 1.. tm+1;
begin
      .
      .
      essai := 1;
      if essai ≤ tm then
      OP [E : begin essai := essai +1; retry end] ; % tm tentatives
                                                      de masquage %
      if essai = tm+1 then signal EK;
      % si on arrive ici, cela signifie que le service standard de OP
        est disponible %
      .
      .
end;
```

Reprise

Si l'exécution des instructions qui composent le traitant *T* de *E* se termine sans aucune nouvelle levée d'exception, et si la dernière instruction de *T* n'est ni un *signal*, ni un *reset*, ni un *retry*, alors le mécanisme d'exception redonne le contrôle à l'instruction qui suit celle à laquelle *T* est attaché, en assurant ainsi la reprise de la séquence d'instructions standard de *U*.

Restauration éventuelle et propagation

Si durant l'exécution du traitant *T* de *E*, le mécanisme d'exception rencontre une instruction *signal EK* ou *reset EK* (§V.4), alors les éventuelles instructions de *T* qui suivent cette instruction ne sont plus exécutées, l'activation de *T*, ainsi que l'activation de la procédure *U* qui englobe syntaxiquement *T* est terminée, et l'exception *EK* est levée dans le programme qui a appelé *U* (dans le cas d'un *reset*, *U* doit être forcément une procédure externe de module, voir §V.4).

Si au niveau d'abstraction de *U*, à l'exception *EK* correspond un ensemble d'incohérence $\varepsilon(EK)$ non vide, alors le concepteur de *T* doit restaurer $\varepsilon(EK)$ avant la sortie définitive du module *N* qui contient la procédure *U* . Il peut faire cela, soit implicitement (en utilisant la primitive *reset EK*, qui restaure la couverture d'incohérence $C_k \supseteq \varepsilon(EK)$) , soit de manière explicite (s'il veut restaurer seulement les variables de $\varepsilon(EK)$ avant de propager *EK* par une instruction *signal EK*).

V - 7. DISCUSSION

Nous allons analyser maintenant dans quelle mesure le mécanisme d'exception proposé répond aux critères établis au §III.1.3.

a) Modularité. Le traitement d'une exception *EK* dans un module *N* (Fig.3.1) dépend uniquement de l'état interne du module et de l'état des paramètres d'appel. Les modules "au dessus" de *N* n'interviennent jamais lors d'un tel traitement. Tous les effets exceptionnels peuvent donc être décrits en fonction de l'état de *N* et de l'état des paramètres d'appel. Ceci est en accord avec les idées actuelles sur la spécification formelle des types.

L'insertion automatique de traitements par défaut pour les procédures externes des modules permet de garantir que les frontières des modules de I_n sont "étanches" à la propagation des exceptions.

b) Sécurité. La sécurité de fonctionnement d'une machine modulaire I_n est conditionnée par :

- la validité des hypothèses *H1*, *H2*, *H3* du §IV.1 sur lesquelles le fonctionnement du mécanisme de restauration est basé,
- la correction avec laquelle les programmeurs de I_n réalisent l'implémentation des fonctions d'annulation et des traitants d'exception qui restaurent explicitement les ensembles

d'incohérence associés aux exceptions internes à I_n.
Si ces deux conditions sont remplies, alors l'activation automatique des traitants par défaut en cas de détection de *DEFAILLANCE*, garantit que des états internes incohérents stables ne peuvent apparaître dans I_n par suite de détections d'exceptions non traitées.

c) Simplicité et uniformité. L'intégration de ce mécanisme d'exception dans un langage modulaire, nécessite l'enrichissement du langage par les primitives *signals*, [,], *signal*, *reset* plus le "sucre" *retry*. Toutes ces primitives peuvent être utilisées pour traiter autant les exceptions prédéfinies dans le langage I_o que les exceptions définies par les programmeurs de I_n.
Elles peuvent être utilisées pour programmer le traitement des exceptions dans les modules de I_n, ou dans les processus au dessus de I_n (sauf *reset* qui ne peut être utilisée que dans les procédures externes des modules de I_n).

d) Orthogonalité. La composante "contrôle" du mécanisme ne contredit pas la sémantique de l'appel procédural, mais plutôt l'enrichit, permettant de distinguer entre l'effet standard d'une activation de procédure et plusieurs effets exceptionnels possibles. L'interférence des primitives de signalisation d'exceptions *signal* et *reset* avec un mécanisme de synchronisation comme le moniteur [Hoare 74] adopté en SESAME est très réduite. I_o Interprète toute signalisation d'exception dans une procédure externe de moniteur comme une sortie définitive du moniteur : les risques de deadlock dus aux actions de masquage ultérieurs sont éliminés (voir §III.1.3.4). Puisque les procédures externes des moniteurs s'exécutent en exclusion mutuelle, il n'y a aucun danger de voir signaler deux fois la même exception (voir également §III.1.3.4).
La composante "restauration" du mécanisme interfère avec les instructions qui provoquent des changements d'état (affectation, appel de procédure). Mais, là aussi, nous pouvons constater que la présence d'un mécanisme de restauration, au lieu de contredire la sémantique "habituelle" de ces opérations, la complète plutôt. Si l'exécution, à un certain niveau d'abstraction *N*, d'une opération *OP* , provoque une transition d'état interne légale (le nouvel état interne de *N* sera cohérent, voir §II.2.4) alors l'effet de l'activation de *OP* est celui qui est "habituel", mais si

l'exécution de *OP* risque de produire une transition d'état interne illégale (le nouvel état interne sera incohérent, voir §II.2.4) alors l'effet de l'activation de l'opération sera "nul", dans le sens qu'après l'exécution de *OP*, l'état interne de *N* restera équivalent avec l'état interne d'avant l'activation de *OP*.

e) Economie. La décision d'intégrer un mécanisme comme celui que nous avons proposé dans un langage de programmation à des conséquences sur :

1) Le temps nécessaire à l'implémentation, à la mise au point et à la maintenance des programmes.
2) Le temps nécessaire à l'exécution des programmes.

Jusqu'à maintenant, aucune implémentation du mécanisme n'a été faite. Il nous est donc impossible de donner dans cette thèse une évaluation quantitative des conséquences économiques que l'intégration du mécanisme dans un langage de programmation peut avoir. Tout au plus pouvons nous donner quelques appréciations a priori :

E1) La disponibilité du mécanisme dans le langage de programmation réduit le temps (1), surtout pour les programmes qui ont une taille assez importante*. En effet, dès que le nombre de variables d'un programme croît, le nombre d'exceptions qui peuvent être détectées lors des opérations sur ces variables croît rapidement. L'utilisation du mécanisme d'exception contribue à simplifier la programmation du traitement de ces exceptions. La disponibilité d'une primitive comme *reset* décharge le programmeur de la nécessité de programmer explicitement la restauration des ensembles d'incohérence qui peuvent être associés aux exceptions du programme, et la présence de l'exception prédéfinie *DEFAILLANCE* garantit que toutes les exceptions vont être traitées. L'utilisation du mécanisme rend les programmes plus lisibles, à cause de la séparation des traitements standard et des traitements d'exceptions. Comme le mécanisme proposé respecte les principes de la programmation modulaire, la modifiabilité des programmes produits

---

* Systèmes d'exploitation, systèmes de bases de données, traducteurs, systèmes de traitement de texte, systèmes de simulation d'environnements complexes, etc.

se trouve améliorée. Nous espérons que la simplicité des programmes de traitement d'exceptions, inclus dans un système assez complexe comme le système MULTIACCES du §VI, convaincront le lecteur habitué à ce genre de systèmes de l'avantage de disposer d'un mécanisme d'exception. L'influence que la présence du mécanisme d'exception a sur le temps (1), pourrait être estimée quantitativement en comparant le temps nécessaire à l'implémentation, mise au point et à la maintenance de deux programmes ayant des spécifications identiques, dont l'un utilise les primitives du mécanisme et l'autre ne les utilise pas. La description d'un expériment analogue, effectué pour d'autres primitives de structuration de programmes est donnée dans [Gannon 75].

E2) La présence du mécanisme d'exception accroît le coût (2) en alongeant le temps nécessaire à l'exécution des programmes. Le coût distribué (voir §III.1.3.5) résulte de l'interférence du mécanisme avec les autres primitives du langage. Le coût propre dépend beaucoup de l'implémentation qui est choisie. La composante la plus "lourde" du coût distribué, est le résultat de l'interférence du mécanisme de restauration avec les primitives qui changent l'état des variables d'un programme*.
Si la sécurité de fonctionnement imposée par le cahier des charges doit être élevée et le temps de réponse ne constitue pas un facteur critique, la présence du mécanisme de restauration peut être acceptable. Mais si le niveau de sécurité que l'on veut atteindre est moins élevé, ou le temps de réponse est une contrainte critique, il est possible de n'utiliser que la composante "structure de contrôle" du mécanisme. En effet, le mécanisme d'exception est lui-même "modulaire", dans le sens qu'il est composé de deux parties relativement indépendantes : la partie restauration (*reset* + *MCACHE*) et la partie contrôle (*signals*, [,], *signal*, *retry*). Cette dernière paut fort bien fonctionner toute seule. Il est donc possible de concevoir une implémentation du mécanisme, de façon à ce qu'il puisse fournir des services "à la carte", suivant les directives de compilation fournies par le programmeur : service complet (restauration + contrôle) ou service "économique" (uniquement contrôle).

---

*Des mesures ont été effectuées à Newcastle pour le "recovery cache" [Shrivastava 78d]. L'overhead introduit par le cache était, pour les programmes mesurés, inférieur à 11%. Des mesures analogues sont nécessaires pour le mécanisme du §IV.

VI- UN EXEMPLE.

Le but de ce chapitre est de montrer l'adéquation du mécanisme d'exception du chapitre V à traiter les exceptions d'un système .

Les idées contenues dans ce mécanisme sont le fruit d'une expérience pratique de programmation modulaire [Cristian 77]. Pour tester ces idées, on aurait pu re-programmer les traitements d'exception du BIBLIOTHECAIRE, en utilisant le mécanisme proposé .

Mais avant de décrire le traitement des exceptions dans un programme, il est d'abord nécessaire de présenter sa structure*. Le BIBLIOTHECAIRE est un programme assez complexe : entrer dans les détails de sa structure allongerait trop cet exposé . Il nous faut donc choisir un autre système-exemple plus simple . Sa structure ne doit pas être trop éloignée de celle d'un système " réel " pour ne pas être taxée d'académique ; et en même temps, elle doit être aussi pédagogique que possible, pour pouvoir être comprise facilement .

Face à ces exigences contradictoires, nous avons opté pour une solution de compromis :

-La"macro-structure" du système va être suffisamment proche d'une "macro-structure" de système "réel" . Le terme "macro-structure" se réfère à l'ensemble des relations existant entre les objets de la programmation "in the large [De Remer 75]:modules et processus .

-La "structure fine" va être aussi simple que possible pour ne pas encombrer inutilement l'attention du lecteur . Le terme "structure fine" englobe les relations qui existent entre les objets de la programmation "in the small [De Remer 75]: constantes, variables, blocs, procédures etc... servant à construire les macro-composants.

Les § 1, 2 et 3 du chapitre VI contiennent la description de notre système-exemple,qui est un système transactionnel *MULTIACCES*.

Le traitement des exceptions prévues et non prévues est fait en utilisant le mécanisme d'exception du chapitre V . Au § VI-4 , nous montrons que l'utilisation systématique de ce mécanisme permet de garantir que les transactions implémentées par *MULTIACCES*, sont atomiques .

---

* On ne peut pas parler d'exceptions, quand il n'y a pas de structure . Le concept d'exception lui-même n'est qu'une conséquence de la volonté de structurer les traitements " à la limite " de cette première structure .

VI-1. DESCRIPTION DE *MULTIACCES* .

La macro-structure du système sera décrite à l'aide du langage de connexion présenté en [Cheval 76], ou à l'aide de figures illustrant la relation "utilise" qui s'établit entre des macro-composants . La structure fine est décrite en SESAME [Cheval 76]. Nous éviterons pourtant de trop entrer dans les détails de structure fine, chaque fois que cela n'est pas indispensable à la compréhension de l'exemple .

VI-1.1. LES OBJETS GERES .

La configuration physique qui supporte le système est celle de la figure 6.1 :

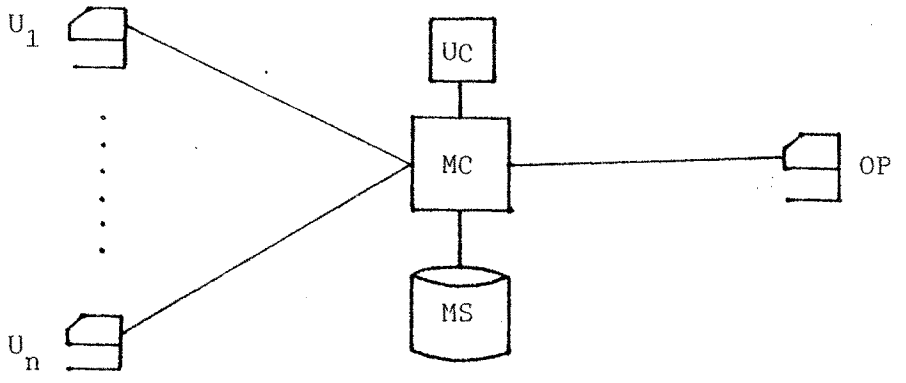


Figure 6.1

n télétypes pour les utilisateurs interactifs, une unité centrale UC , mémoire centrale MC et mémoire secondaire MS . Le système peut communiquer avec un utilisateur spécial, l"opérateur" , à l'aide d'une console réservée OP . Pour permettre au lecteur de concentrer son attention sur le traitement des exceptions,nous prenons déjà une première décision de simplification : le système va implémenter pour la communauté des utilisateurs interactifs un seul * type d'objet "abstrait" : le type "fichier séquentiel" . Un objet de ce type va être simplement appelé "fichier" .

---

* Les systèmes réels implémentent beaucoup plus de types d'objets.Par exemple, un système d'exploitation implémentera des types"fichier source","fichier binaire translatable","fichier partitionné",etc...Ces types peuvent fort bien être implémentés en termes d'autres types plus primitifs,comme le type "fichier séquentiel".C'est le cas,par exemple,du BIBLIOTHECAIRE,qui implémente des types d'objets plus complexes:objets de référence,versions,objets binaires [Cristian 77]en utilisant le type "fichier séquentiel".Dans ce sens,notre système peut être considéré,si l'on veut,comme un "sous-système" d'un véritable système multi-accès "réel".

Les opérations définies sur le type "fichier séquentiel" sont :

-création d'un fichier

-destruction d'un fichier

-ouverture d'un fichier en lecture,ou lecture / écriture

-fermeture d'un fichier

Pour chaque fichier ouvert le système mémorise son article"courant".
L'utilisateur peut :

-rendre courant un certain article repéré par son numéro d'ordre

-lire l'article courant

-écrire l'article courant.

VI.1 2. PROBLEMES DE PARTAGE .

Le but de *MULTIACCES* est le partage de tous les fichiers entre tous les utilisateurs *. Il se peut donc que, à certains moments, plusieurs utilisateurs accèdent simultanément au même fichier. Les utilisateurs communiquent au moyen d'informations mémorisées dans les fichiers. Cette communication ne va pas sans poser quelques problèmes [Gray 76] :

a) Mises -à- jour perdues. Supposons que l'utilisateur U_1 met à jour l'article *3* du fichier *N* , en changeant sa valeur de *u* en *uu* . Supposons qu'un autre utilisateur U_2 change après la valeur de cet article en *uuuu* . Si une exception,détectée durant l'activité ultérieure de U_1, oblige le système à restaurer toutes les transformations d'état partielles induites par U_1 , alors l'article *3* de *N* sera restauré à *u* et la mise à jour de U_2 sera perdue.

b) Lectures "sales" . Si U_1 met à jour l'article *3* de *N* en changeant sa valeur de *u* en *uu* ,et que U_2 lit cet article , alors, si U_1 est retourné en arrière à cause d'une exception, U_2 aurait lu une valeur qui n'aurait jamais existée !

c) Lectures non répétables . Supposons que U_1 change la valeur de l'article *3* de *N*,de *u* en *uu*,et que U_2 lit cette valeur . Supposons qu'ensuite, U_1 change encore une fois la valeur de l'article en *uuuu* . Si U_2 lit à nouveau l'article, il va obtenir une valeur différente de la lecture précédente !.

Dans [Gray 76],on distingue quatre "degrés de cohérence" pour les communi-cations à travers des fichiers partagés,suivant le nombre d'utilisateurs

---

* Les problèmes de confidentialité,liés à l'accès sélectif des utilisateurs aux fichiers,ne vont pas être pris en compte .

qui peuvent ouvrir simultanément un même fichier en lecture ou lecture / écri-ture . Ces degrés quantifient la confiance qu'un utilisateur peut avoir dans une information échangée avec un autre, et donc concernent la sémantique des échanges entre utilisateurs . Au niveau de *MULTIACCES* , la sémantique de ces échanges sera ignorée : *MULTIACCES* implémente seulement le mécanisme de communication . Une analyse détaillée des conséquences que l'occurrence des conflits (a),(b) ou (c) peut avoir sur les actions ultérieures des utilisateurs [Gray 76] ,[Russel 75],[Davies 78] dépasse la portée de notre étude . Si le mécanisme de communication assure un degré ou un autre de cohérence, cela est un problème de cahier des charges . Mais si l'occurrence des conflits (c),(b) ou (a) est admise , ce cahier des charges obligera probablement le système à en avertir les utilisateurs, et éventuellement à assurer leur "roll-back" automatique . Pour des raisons de simplicité nous avons décidé d'implémenter le degré de cohérence le plus élevé* : A ce degré, le système garantit l'accès exclusif d'un utilisateur à tout fichier ouvert (créé) en lecture / écriture [Gray 78] . Ceci évite la production des conflits (a),(b) et (c) .

compatibilité		mode d'accès actuel	
		lecture	lecture / écriture
demande d'ouverture	lecture	compatible	incompatible
	lecture/écriture	incompatible	incompatible

Pour éviter l'occurrence des deadlocks, tout conflit détecté par le système produit une exception (les utilisateurs ne sont pas mis en attente) . L'occurrence d'une telle exception est notifiée par un retour exceptionnel .

---

* Il nous semble que le mécanisme d'exception du § V peut être également utilisé pour implémenter un degré de cohérence "bas" , mais notre exemple se trouverait compliqué du fait qu'il faudrait assurer le "roll-back" d'un processus qui reçoit une notification de conflit (a),(b) ou (c) .

VI.1.3. LES TRANSACTIONS .

Initialement,tous les fichiers sont fermés . Du point de vue du système, un fichier fermé est dans un état cohérent . Une fois assis devant leurs télétypes , les utilisateurs ouvrent les fichiers . Durant l'accès à un fichier *N* ,l'assertion " *N* est dans un état fermé" devient fausse .

Du point de vue du système , *N* est dans un état intermédiaire, incohérent . Du point de vue de l'utilisateur, la cohérence a une tout autre signification. Pour lui , *N* est dans un état cohérent si l'assertion "l'état externe actuel de *N* reflète l'historique des opérations faites sur *N* " est vraie . Par exemple, un fichier *N* de 26 articles, créé quelques jours en arrière pour contenir les 26 lettres de l'alphabet français, va être dans un état cohérent (du point de vue de l'utilisateur) si à sa prochaine ouverture on peut lire les 26 articles et constater qu'ils contiennent effectivement les 26 lettres de l'alphabet français .
L'utilisateur peut amener les fichiers gérés par le système dans des états qui sont incohérents du point de vue du système . Par exemple :

U_1) L'utilisateur peut ouvrir un fichier *N* pour y accéder en lecture / écriture (du point de vue du système cela demande une surveillance car *N* est rendu inaccessible aux autres utilisateurs; or, son rôle est le partage de tous les fichiers entre tous les utilisateurs) .

U_2) L'utilisateur peut être"défaillant," c'est-à-dire qu'il peut violer (volontairement ou involontairement) les règles d'utilisation des fichiers : par exemple, il peut "oublier" de fermer *N* . De façon symétrique, le système peut mettre les fichiers gérés par l'utilisateur dans des états qui sont incohérents de son point de vue .

S_1) Durant la création du fichier de 26 articles, parce qu'il détecte une occurrence de l'exception "débordement de l'espace disque disponible", le système peut répondre:"il n'y a de place que pour 25 articles".

S_2) Des *DEFAILLANCES* (§III-2) peuvent survenir dans le système à cause des fautes résiduelles . Les fichiers de l'utilisateur peuvent ainsi être mis dans des états tout à fait imprévisibles.

Le concept de transaction [Gray 76] institutionnalise la méfiance mutuelle qui existe entre l'utilisateur et le système.

En obligeant l'utilisateur à déclarer des points de cohérence (les débuts et fins de transactions), le système peut contrôler que les assertions temporairement rendues fausses par l'utilisateur ,redeviennent vraies . De cette façon, le système se protège contre les "oublis" ou les "malices" de l'utilisateur . Symétriquement,l'utilisateur se protège contre les exceptions prévues ou imprévues du système . Par exemple, si une défaillance est détectée pendant une mise à jour de fichier, le système lui restitue le fichier dans l'état d'avant la séquence de mise a jour .

Les transactions sont les "unités de cohérence" [Gray 76], dans le sens que "en partant d'un état cohérent de fichier, si pendant l'exécution d'une transaction,aucune exception n'est détectée ,ni par le système, ni par l'utilisateur, alors l'état du fichier aprés la transaction est cohérent " . De même, on peut affirmer que les transactions sont des " unités de restauration ",dans le sens que,"si durant l'excécution d'une transaction sur un fichier, une exception est détectée par le système ou par l'utilisateur, alors , l'état que le fichier va avoir en fin de transaction va être l'état d'avant la transaction ".

Les transactions sont les opérations atomiques qu'un système implémente pour l'utilisateur .

Pour des raisons de simplicité, nous ne considérerons que des transactions qui consistent en des séquences d'opérations sur un seul fichier . Permettre l'accès simultané à plusieurs fichiers pendant une transaction ne pose aucun problème spécial , mais complique la structure du système (voir § VI.2.4) et augmente les difficultés de compréhension d'un exemple . Ainsi , nous allons considérer uniquement les quatre types de transactions suivants :

- -création d'un nouveau fichier
- -mise à jour d'un fichier ancien
- -lecture d'un fichier ancien
- -destruction d'un fichier ancien .

L'utilisateur annonce au système le début d'une transaction par le mot - clé *dt* .

S'il s'agit d'une transaction de mise à jour de fichier préexistant, l'utilisateur doit ouvrir d'abord le fichier et ensuite faire les opérations de mise à jour sur les articles du fichier . Durant la transaction, le système assure l'accès exclusif de l'utilisateur au fichier (§ VI.1.2). La mise à jour terminée , l'utilisateur annonce la fin de la transaction par le mot-clé *ft* . Une commande de fin de transaction (en anglais "commitement") confirme au système que l'utilisateur rend "publique"

la mise à jour qu'il vient de faire. Une transaction en cours, peut être annulée par une commande *at* .

VI-2 MACRO-STRUCTURE .

Pour trouver la macro-structure (§ VI.1) de notre système , nous avons essayé de répondre successivement aux quatre questions suivantes :

1) Quelles sont les activités indépendantes, qui doivent être assurées par des processus distincts ?
2) Quelles sont les opérations abstraites qu'un processus doit pouvoir exécuter ? Quels sont les types abstraits autour desquels ces opérations peuvent être regroupées ?
3) Comment implémenter les types abstraits mis en évidence au point (2) à l'aide de types de plus en plus concrets ?
4) Quelles sont les variables abstraites partagées , et comment éviter les erreurs de synchronisation pouvant résulter de leur accès simultané par plusieurs processus ?

VI-2.1. ACTIVITES PARALLELES.

Chaque utilisateur travaille à sa vitesse et à ses heures . Il y aura, par conséquent, autant d'entités actives(*PROCESSUS*) que d'activités parallèles indépendantes * possibles (figure 6.2) . Les processus P_1 ,P_2 ...P_n auront tous des algorithmes identiques :

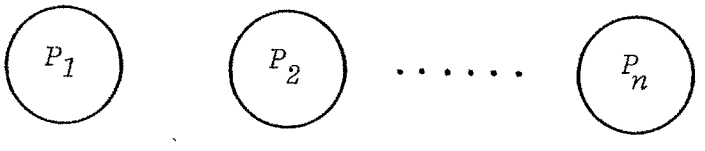
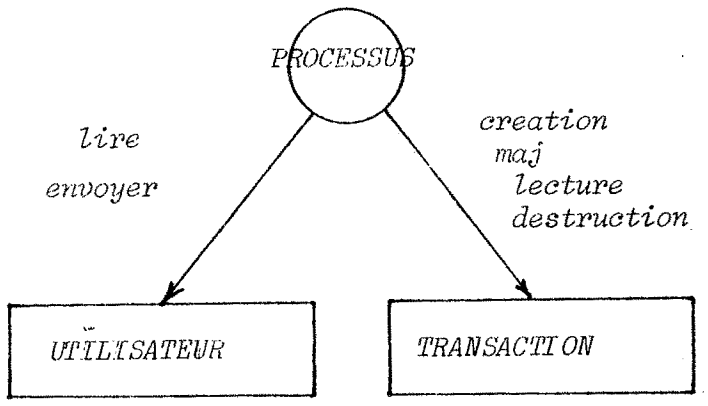


Figure 6.2

* Pour des raisons de simplicité , nous ne prenons en compte que partiel-lement , au § VI-3.1. les problèmes liés à l'existence d'une autre activité indépendante " opérateur " .

VI-2.2. OPERATIONS ABSTRAITES AU NIVEAU D'UN *PROCESSUS* .

Tout *PROCESSUS* communique avec l'utilisateur à travers une console . Le *PROCESSUS* doit lire les commandes de l'utilisateur et envoyer des réponses . Soit *UTILISATEUR*. le type abstrait dont les opérations seront : lire la commande courante , envoyer la réponse courante .
Un *PROCESSUS* doit également pouvoir exécuter les transactions décrites en § VI.1.3. Soit *TRANSACTION* le type abstrait dont les opérations seront : création d'un nouveau fichier , mise à jour d'un ancien fichier , lecture d'un ancien fichier , destruction d'un ancien fichier. La figure 6.3 contient la représentation visuelle de la relation "utilise" [Parnas 74] qui s'établit entre un *PROCESSUS* et les types *UTILISATEUR* et *TRANSACTION* .



Passé ce stade de la décomposition " en processus et modules ", il est nécessaire d'entrer un peu plus dans le détail des opérations abstraites implémentées par *TRANSACTION* et *UTILISATEUR* .
Un fichier , créé par un utilisateur , sera identifié par un nom externe . Nous supposerons que tout nom de fichier est composé de moins de 8 caractères , qu ' un fichier ne doit jamais contenir plus de 128 articles, et que tous les articles d'un fichier ont une longueur de 32 caractères :

```
const  LNOMX = 8 , % longueur d'un nom externe de fichier %
       LMAXF = 128 , % longueur maximale d'un fichier %
       LA    = 32 ; % longueur d'un article %

type   NOMX   = array [ 1...LNOMX ] of char ;
       TAILLE = 1 .. LMAXF ;
       ARTICLE = array [1 .. LA] of char ;
```

Les opérations définies sur le type *TRANSACTION* seront

```
ext   procedure  CREATION  ( N : NOMX ;T : TAILLE ) ;
ext   procedure  DESTRUCTION ( N : NOMX ) ;
ext   procedure  MAJ ( N : NOMX ) ;
ext   procedure  LECTURE ( N : NOMX ) ;
```

Occupons-nous maintenant des communications avec l'utilisateur . L'utilisateur identifie chaque commande adressée au système par un code mnémonique :

```
type  COD = (DT , % début transaction %
             FT , % fin de transaction %
             AT , % annulation de transaction %
             C  , % création d'un nouveau fichier %
             D  , % destrution d'un fichier existant %
             A  , % accès à un fichier %
             AC , % article courant %
             L  , % lecture de l'article courant %
             E  ); % ecriture de l'article courant %
```

Une commande consiste en un code et un ensemble de paramètres qui corresponden à ce code .

```
type  COMMANDE  = record  case CODE :COD of
                          DT , FT , AT : (  );
                          C : (NOM : NOMX ; T : TAILLE ) ;
                          D : ( NOM : NOMX ) ;
                          A : ( NOM : NOMX ; ACCES : MODE-ACCES ) ;
                          AC :(NAC : TAILLE ) ;%numéro de l'article couran
                          L : (  ) ;
                          E : ( ART : ARTICLE )
                    end;
```

La fonction externe d' *UTILISATEUR* qui lit la commande courante sur la console sera :

```
ext  function  LIRE : COMMANDE ;
```

Pour formater et envoyer les réponses du système vers l' *UTILISATEUR*, nous supposons l'existence d'une procédure " intelligente " pouvant envoyer vers la console des *MESSAGES* de longueur variable, consistant en des chiffres, lettres ou texte :

ext procedure ENVOYER (M : MESSAGE) ;

VI-2.3. IMPLEMENTATION DES TYPES *TRANSACTION* et *UTILISATEUR* .

La question à laquelle il nous faut répondre dans ce chapitre est : "comment implémenter les types abstraits mis en évidence dans § VI-2.2. à l'aide de types de plus en plus concrets ? " .
Prenons d'abord le type *TRANSACTION* . Une opération de ce type consiste, soit en une création ou destruction de fichier, soit en un accès à un fichier existant . En d'autres termes, des opérations classiques sur des fichiers * . Il est bien de rendre les opérations de création ou de destruction aussi indépendantes que possible des accès aux articles, si l'on veut, par la suite , pouvoir accéder à un même fichier physique avec plusieurs méthodes d'accès compatibles [Cheval 77] . Nous allons donc résoudre les problèmes posés par l'allocation de mémoire à la création et destruction des fichiers par le type abstrait *GES* (Gestionnaire de l'Espace), et les problèmes posés par l'accès séquentiel aux articles d'un fichier, par le type abstrait *VAS* (Voie d'Accès Séquentiel) .

Une opération d'allocation d'espace pour un fichier va avoir comme paramètres la taille de l'espace à allouer et le nom externe qui va identifier cet espace . Une opération de libération de l'espace occupé par un fichier qui est à détruire ,va avoir comme seul paramètre le nom externe du fichier . Les fonctions externes de *GES* seront :

ext procedure CREER (N : NOMX ; T : TAILLE) ;

ext procedure DETRUIRE (N : NOMX) ;

GES "voit"tout fichier comme une association entre son nom externe et la séquence d'articles qui le composent. Les articles sont mémorisés sur la mémoire secondaire de type *DISK* (figure 6.1) . Comme nous l'avons vu

---

* Un exemple de réalisation modulaire d'un système de gestion de fichiers offrant à ses utilisateurs de telles opérations, est donné dans [Cheval 77]. La décomposition modulaire que nous donnons ici suit, avec quelques modifications , celle suggérée dans cet article .

au § II-1.3, l'unité de transfert avec un périphérique de type *DISK* est le bloc de 512 caractères . Un bloc peut donc contenir 512 div 32 = 16 articles (chaque article a 32 caractères) . Mais un fichier doit pouvoir contenir plus ou moins d'articles qu'un seul bloc peut en contenir. La solution classique est "d'étaler" les articles d'un fichier sur plusieurs blocs qui peuvent éventuellement ne pas être contigus . Pour cacher les détails d'allocation de mémoire secondaire, nous utilisons le type *SEGMENTS* du § II-2.4 . Ce type implémente un espace contigu de pages à partir de blocs non contigus. Comme la longueur maximale d'un fichier est 128 articles, 8 blocs physiques suffiront pour mémoriser n'importe quel fichier . La variable abstraite *SEG* de type *SEGMENTS* va avoir le paramètre de génération *&NP %nombre de pages % = 8*.Nous avons vu dans le §II-2.1, que *SEGMENTS* utilise deux variables *NOMS* et *BLOCS* de type *RESSOURCES* (II-2.1) et des variables appartenant à des types concrets, donc la décomposition s'arrête là . L'association entre le nom externe d'un fichier et le nom interne du segment contenant les pages du fichier sera implémentéepar une variable appartenant à un type abstrait très connu* : *CATALOGUE* . La figure 6.4 illustre les relations "utilise" qui s'établissent entre les types *TRANSACTION, GES, CATALOGUE, SEGMENT, RESSOURCES.*

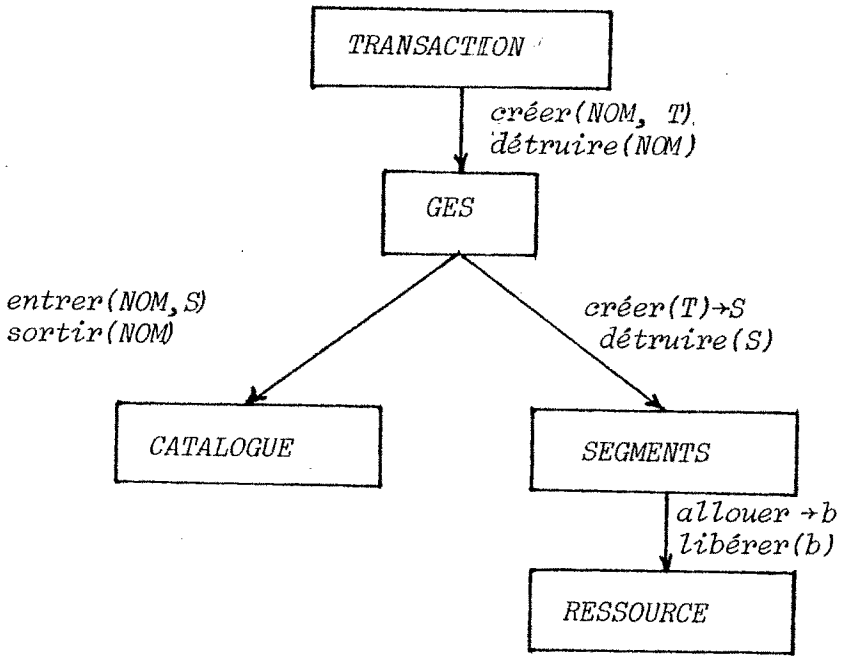


Figure 6.4

* Ce type sera développé en § VI-3.3.

Voyons maintenant la réalisation du type abstrait *VAS* *. Ses fonctions externes sont :

```
ext procedure  OUVRIR ( N : NOMX ; A : MODEACCES );où type MODEACCES=(L,LE
ext procedure  FERMER ;
ext procedure  COURANT (NAC : TAILLE) ;% NAC= numéro d'article courant %
ext function   LIRE : ARTICLE ;
ext procedure  ECRIRE ( A : ARTICLE ) ;
```

Pour retrouver le nom interne i du segment contenant les pages d'un fichier *N* , *VAS* consulte le *CATALOGUE* . Lors d'une opération d'accès à un certain article a,dans *VAS* on doit calculer le bloc b qui le contient, et le déplacement d de a à l'intérieur de ce bloc . Ce calcul peut être fait en quatre étapes "classiques" :

- -la page p qui contient l'article a est $p = a$ *div* 16 (16 articles par page)
- -le bloc b correspondant à la page p,peut être trouvé en lisant la valeur de la restriction de fonction $DS[i]$ (voir § II.2.4) pour l'argument p , soit $b = DS[i].M[p]$.
- -le bloc b peut être lu en utilisant un module périphérique de type *DISK* (§ II.1.1) .
- -Soit *BUF* le buffer déclaré dans *VAS* , contenant l'image du bloc b ; le déplacement de l'article a dans le buffer *BUF* va être $d = a$ *mod* 16.

La figure 6.5 illustre comment le graphe d'accès de la figure 6.4 s'enrichit par l'introduction des nouveaux composants abstraits *VAS* et *DISQUE* . Revenons à nouveau au type *TRANSACTION* . Toute opération du type *TRANSACTION* se décompose en une séquence d'opérations sur *GES* et *VAS* ; par exemple, une transaction *MAJ* (mise à jour de fichier)consiste en une ouverture de voie d'accès , suivie d'une séquence de *LIRE* , *ECRIRE* , *COURANT* ,et enfin, la fermeture de la voie d'accès . Pour lire les commandes d'accès aux niveau des articles , une opération de type *TRANSACTION* utilise (comme le *PROCESSUS* dont le rôle est de démarrer les transactions) les opérations implémentées par *UTILISATEUR* .

---

* Ce type abstrait est connu dans les systèmes"classiques" sous le nom de "méthode d'accès" . En fait, dans ces systèmes , on appelle "méthode d'accès" uniquement l'ensemble des opérations spécifiées sur un tel type , et *DCB* (Data Control Bloc), la représentation de l'état interne d'une variable abstraite appartenant à ce type . Pour des raisons de simplicité, nous avons décidé d'implémenter une seule méthode d'accès : l'accès séquentiel . Si l'on avait voulu implémenter n méthodes d'accès (Direct , Virtuel Direct , Séquentiel Indexé , etc...)on aurait dû prévoir autant de types de voies d'accès distinctes : *VAD* , *VAVD* , *VASI* ,etc...

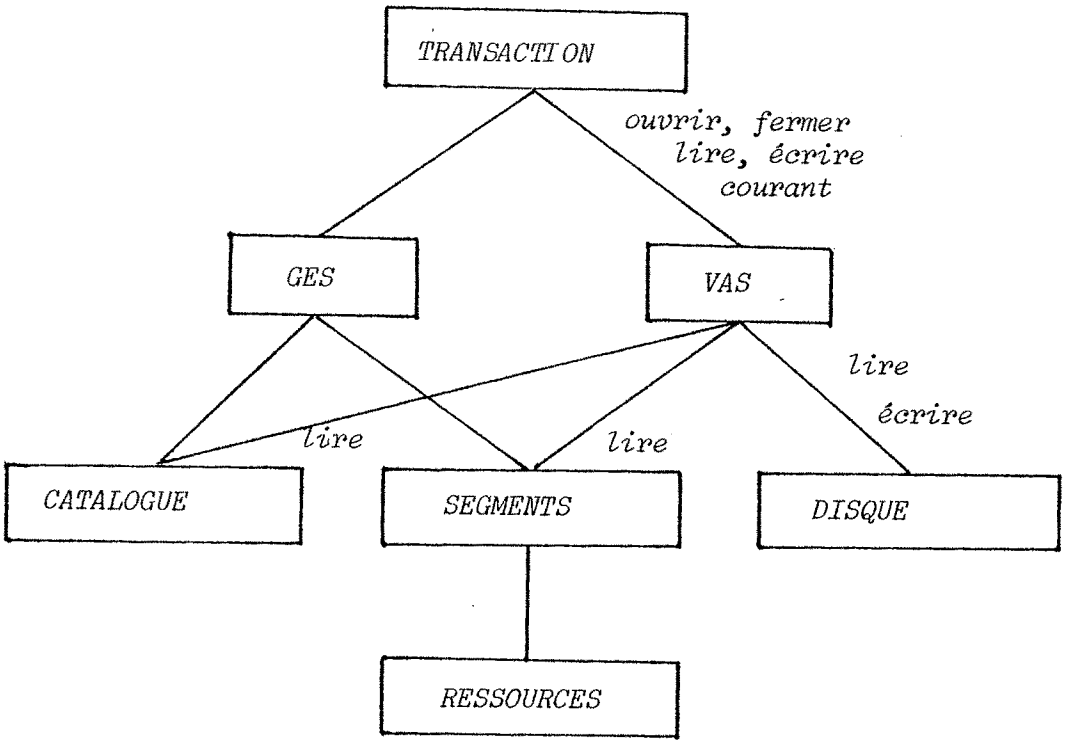


Figure 6.5

Comme nous ne voulons pas trop allonger cette présentation, nous n'allons pas entrer dans les détails de réalisation d' *UTILISATEUR* . Disons simplement que dans la bibliothèque de modules périphériques de I_o, il existe un modèle de module qui implémente un type concret *TTY* , permettant de faire des entrées / sorties sur une console . Ce type possède deux opérations pré--définies : *READ-LINE* et *WRITE-LINE* . *UTILISATEUR* appelle *TTY.READ-LINE* pour *LIRE* la commande courante . Ensuite , dans *UTILISATEUR* , on fait l'analyse lexicographique et syntaxique de la ligne entrée , et l'on calcule le résultat de *LIRE* , de type *COMMANDE* . *UTILISATEUR* peut formater un message reçu comme paramètre d' *ENVOYER* et le faire imprimer en appelant la procédure *TTY.WRITE-LINE* . Le noyau de gestion du parallélisme [Montuelle inclus dans le niveau I_0 , met en attente les processus P_i qui font des demandes d'entrée / sortie . Ainsi, quand devant une console il n'y a pas d'utilisateur , le processus qui"gère" cette console est en attente passive , provoquée par une instruction *TTY. WRITE-LINE* .

La figure 6.6 illustre les relations d'accès entre tous les types de macro-composants que nous avons mis en évidence jusqu'à maintenant .

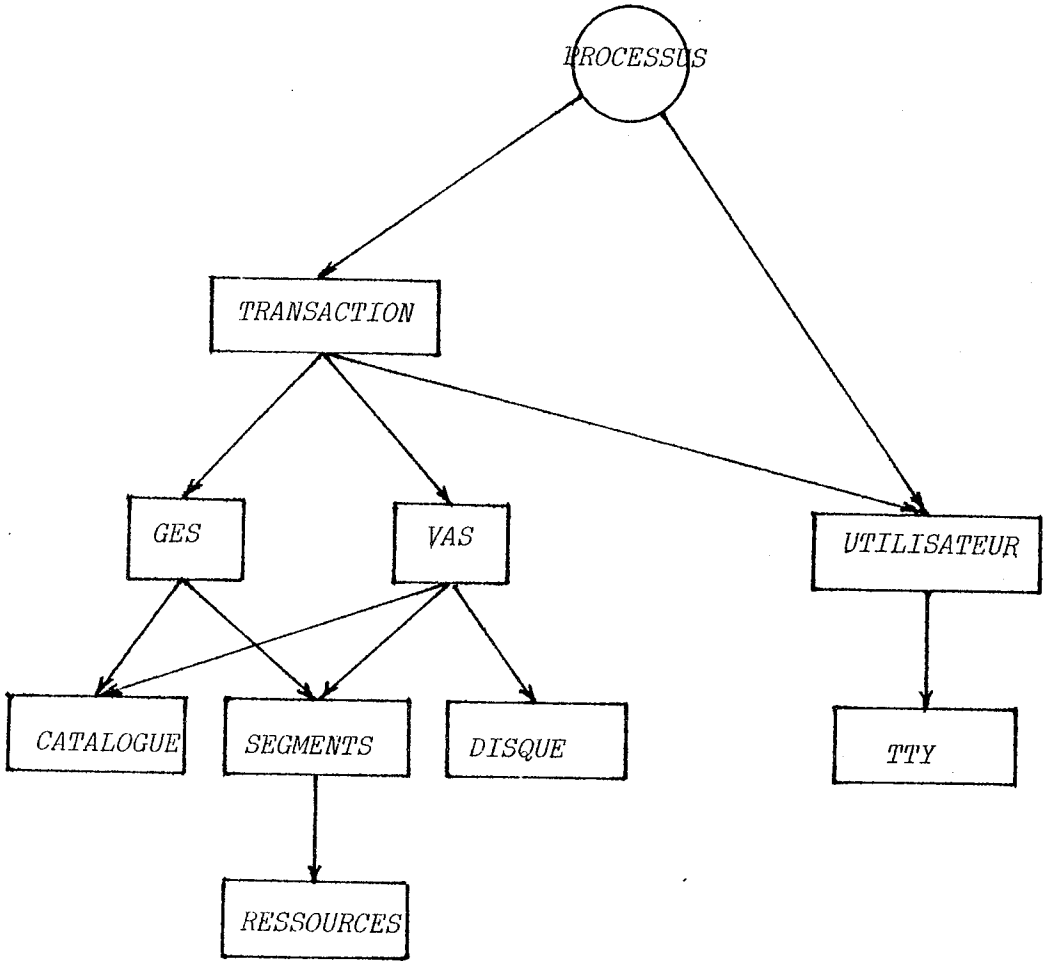


Figure 6.6

VI-2.4. VARIABLES ABSTRAITES PROPRES ET VARIABLES ABSTRAITES PARTAGEES .

Un processus exécute une seule transaction à la fois et chaque transaction manipule un seul fichier à la fois . Comme il est nécessaire d'avoir une voie d'accès par fichier ouvert, chaque transaction utilisera une seule * variable abstraite de type *VAS* . Chaque processus P_i possède également sa propre variable de type *UTILISATEUR* qui mémorise l'état de la communication avec la console de son utilisateur . Mais tous les processus partagent une seule variable *CAT* de type *CATALOGUE* ,une seule variable *SEG* de type

---

* Si on avait voulu accéder à plusieurs fichiers pendant une transaction, il aurait fallu prévoir autant de variables de type *VAS* par transaction. L'allocation de mémoire pour les différentes voies d'accès aurait pu être faite statiquement, en attribuant à chaque processus un nombre maximum , fixé à l'avance , de variables de type *VAS* . Si on avait voulu une solution "plus souple", on aurait pu imaginer une politique d'allocation dynamique des noms de voies d'accès existant dans un "tas" partagé par tous les processus et géré par un moniteur de type *RESSOURCES* (§ II-2.1). Ce moniteur aurait pu allouer à chaque processus autant de voies d'accès qu'il réclame à un certain moment , dans la mesure où il n'y a pas d'occurrence de l'exception *DEBORD* .

SEGMENT, et une seule variable *D* de type *DISQUE* . La figure 6.7 illustre la relation "utilise" [Parnas 74] qui s'établit entre deux processus distincts P_i et P_j , et les variables appartenant aux types de la figure 6.6 . Une variable propre au processus P_i est indexée par i .

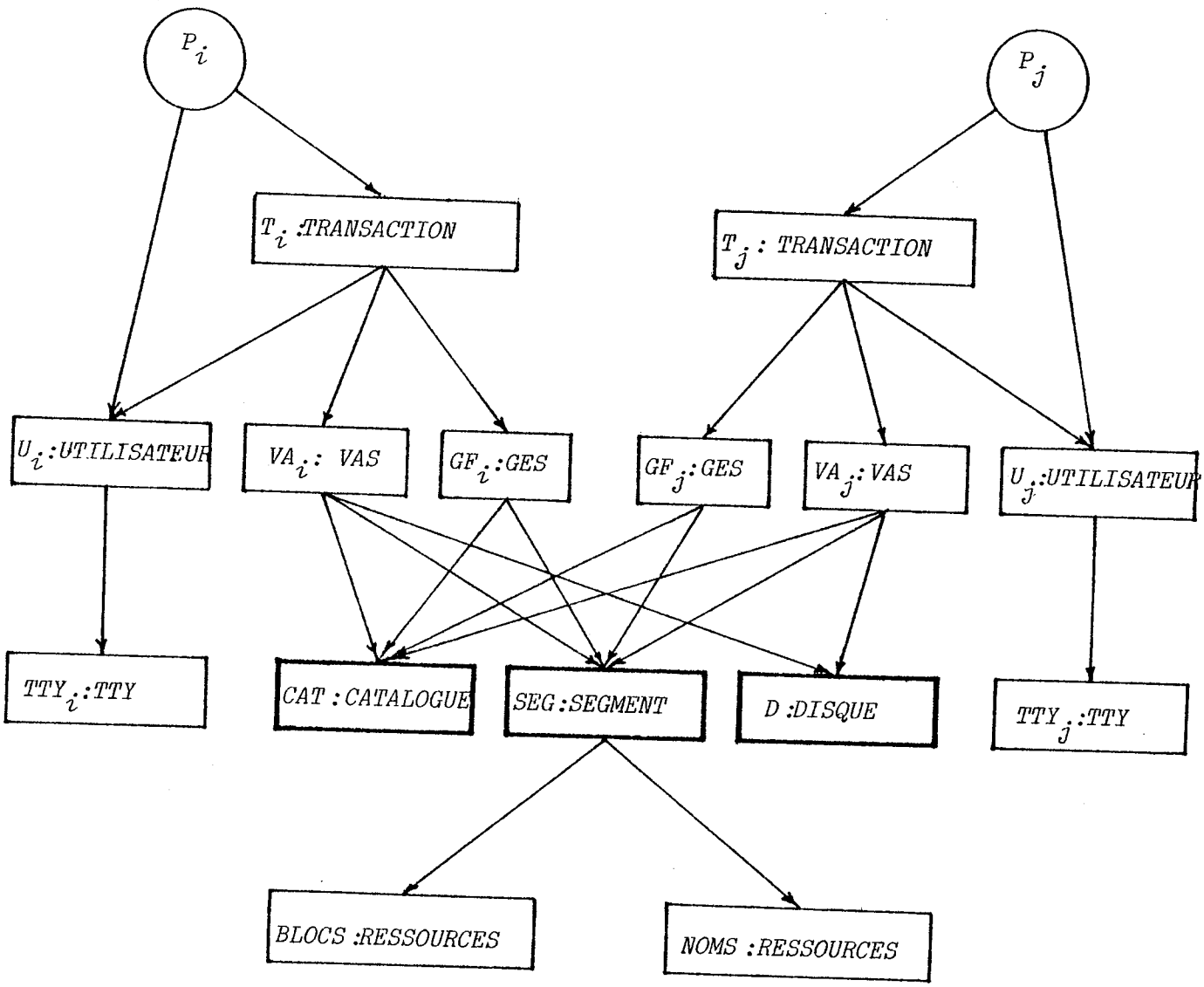


Figure 6.7

Pour éviter les erreurs de synchronisation [Brinch Hansen 73] pouvant résulter des accès simultanés aux variables abstraites *CAT*, *SEG*, *D* , ces variables seront implémentées par des moniteurs [Hoare 74] .

Nous supposerons que la représentation des états internes de toutes les variables abstraites de la figure 6.7(sauf les variables de types *TTY* et *DISQUE*) réside en mémoire centrale*.Le code des procédures qui implémentent les opérations spécifiées pour un type abstrait décrit par une classe ,figure en un seul exemplaire. Mais,chaque variable abstraite appartenant à un tel type possède ses propres données d'état interne. C'est le mécanisme de gestion à l'exécution [Montuelle 78] qui assure que des processus distincts partagent le code de ces procédures, tout en travaillant sur des variables propres distinctes (réentrance) .

Implémenter *CAT* et *SEG* par des moniteurs ne semble pas être une solution trop mauvaise,car ces variables sont consultées uniquement au début et éventuellement à la fin d'une transaction. Par contre, cette solution est assez mauvaise pour la variable *D* de type *DISQUE* ,car elle conduit à servir les demandes d'accès au disque dans l'ordre de leur arrivée (FIFO) . Cette solution produit,à l'exécution, des mouvements désordonnés du bras du disque,et conduit à un temps moyen de réponse assez long. Une solution meilleure,bien connue [Hoare 74],est de faire attendre les processus qui demandent des accès au disque dans une file d'attente gérée de façon à servir les requêtes dans un ordre qui par exemple, induit un minimum de changements de direction du bras du disque. Un tel algorithme de gestion de file d'attente est représenté dans [Hoare 74] sous le nom "d'algorithme de l'ascenseur". Si *ACCES-DISQUE* était le type abstrait gérant la file d'attente des demandes d'accès, on pourrait multiplexer le *DISQUE* réel entre tous les P_i , en donnant à chacun l'impression de posséder son propre *DISQUE-VIRTUEL*. Il suffirait de respecter les deux règles suivantes :

a) chaque accès de *DISQUE-VIRTUEL* à *DISQUE* doit être précédé par un appel à *ACCES-DISQUE.DEMANDER* .

b) quand l'accès est terminé, *DISQUE-VIRTUEL* doit appeler *ACCES-DISQUE.LIBERER*

[Brinch Hansen 75]

La figure 6.8 montre les relations "utilise" qui s'établiraient entre le disque réel, son multiplexeur et le disque virtuel . Pour des raisons de simplicité, encore une fois, nous nous contenterons de la solution grossière mais simple : " la variable *D* de type *DISQUE* sera implémentée par un moniteur " .

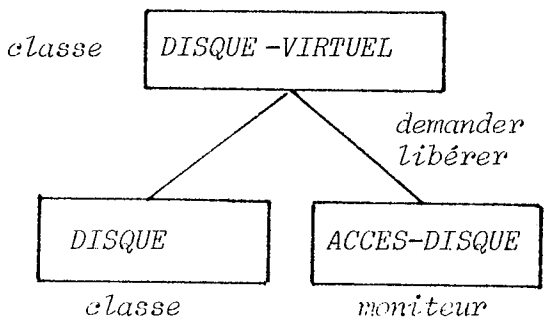


Figure 6.8

* Nous n'aborderons pas ici le problème des sauvegardes d'état sur mémoire secondaire,entre deux sessions de fonctionnement du système. Dans le cas du BIBLIOTHECAIRE, une description de la solution donnée à ce problème est décrite dans [Cristian 76]

VI-3. STRUCTURE FINE.

Nous ne décrirons pas la structure fine de tous les composants de la figure 6.7 du § VI-2.4 . Nous nous limiterons à décrire dans ce chapitre uniquement la structure fine des modules qui interviennent lors d'une exécution de la transaction *MAJ* (Mise A Jour), dans le but de montrer, au § VI-4, que *MAJ*est atomique . *MAJ* est une opération du type *TRANSACTION*. Cette procédure fait appel à toutes les fonctions externes spécifiées pour les types abstraits *VAS* et *UTILISATEUR* . *VAS* utilise certaines fonctions externes de *CATALOGUE* et de *SEGMENTS* pour obtenir la description du segment qui mémorise les articles du fichier *N* à mettre à jour, et utilise les fonctions externes *LIRE* et *ECRIRE* de *DISQUE* pour changer l'état de *N* . Nous étudierons la structure fine de *DISQUE* au § VI-3.2, de *VAS* au § VI-3.3, et de *TRANSACTION* au § VI-3.4 . La procédure de connexion *NIVEAU4* du § VI-3.4 , décrit comment il faut connecter entre eux les composants "passifs" du système, pour offrir à chaque *PROCESSUS* P_i les opérations définies comme des fonctions externes de *TRANSACTION* et de *UTILISATEUR* . Ensuite, nous allons décrire la structure fine d'un *PROCESSUS* associé à un utilisateur quelconque . Enfin, au § VI-3.5, nous décrirons notre système par un programme de connexion entre n instances de *PROCESSUS*, et n instances de variables abstraites de type *NIVEAU4* .

VI-3.1. TYPES DE PERIPHERIQUES PHYSIQUES.

L'utilisateur de SESAME trouve dans la bibliothèque un certain nombre de modules prédéfinis, écrits en langage machine, qui implémentent les types de périphériques disponibles *.

Notre système utilise deux types de périphériques : *TTY* est utilisé pour assurer la communication avec les utilisateurs et l'opérateur, et *DISK*, pour des échanges entre la mémoire centrale et la mémoire sécondaire .

---

* Le choix de ne pas intégrer directement dans le langage SESAME des opérations d'entrée / sortie , a été fait dans un but d'adaptabilité . En effet , chaque configuration-cible particulière , est caractérisée par son propre jeu de périphériques physiques . Si la configuration-cible change : il suffit de modifier la bibliothèque de modules périphériques , et il n'est pas nécessaire de "toucher" au compilateur .

L'interface externe de *TTY* est :

```
class TTY(&ATTY:integer);
% env ENVTTY; %
type LINE = univ array[1..80]of char;
ext function READ-LINE:LINE; signals TRANSMISSION;
% standard : le résultat de READ-LINE est la ligne écrite par l'utilisateur
           de la console d'adresse physique &ATTY %

ext procedure WRITE-LINE (L:LINE); ; signals TRANSMISSION,INCORRECT-LINE;
%standard : sort sur le télétype d'adresse physique &ATTY la ligne L%
```

Certaines occurrences d'exceptions (par exemple l'échec d'une opération d'échange avec la mémoire secondaire , voir § VI-3.2) doivent être signalées à l'opérateur , pour déclancher une maintenance " on-line " . Pour se prémunir contre la rupture des communications avec l'opérateur , la console de l'opérateur dessinée dans la figure 6.1 , est en réalité " doublée " . Une disponibilité de 100 % peut être obtenue dans l'hypothèse suivante :

- -Les deux consoles de l'opérateur , d'adresses *&OP1* et *&OP2* ne tombent jamais simultanément en panne ;
- -Si une des consoles *&OP1* ou *&OP2* tombe en panne , l'opérateur le remarque immédiatement et déclanche la maintenance on-line . Nous supposons que la console défaillante est réparée avant que la deuxième ne tombe en panne .

La figure 6.9 décrit le moniteur *CONSOP* qui assure l'exclusion mutuelle des processus qui appellent l'opérateur et gère en parallèle les deux consoles *&OP1* et *&OP2* . L'occurrence de l'exception *INCORRECT-LINE* peut être évitée statiquement , et nous ne l'avons plus mentionnée dans notre programme .

```
monitor CONSOP;
% env ENVTTY %;
type LINE = univ array [1..80]of char;
dummy procedure PRINT1(L:LINE); signals TRANSMISSION;
dummy procedure PRINT2(L:LINE); signals TRANSMISSION;
ext procedure ALARM(L:LINE);
begin PRINT1(L) [TRANSMISSION: PRINT2(L)
                              [TRANSMISSION: signal DEFAILLANCE] ];
      PRINT2(L) [TRANSMISSION: ]; % si la première écriture a marché,
                                    mais pas la deuxième,l'opérateur le détecte lui-même%
```

```
end;
begin % pas d'initialisation %
end  CONSOP.
```

Figure 6.9

La figure 6.10 décrit la connexion des deux consoles de l'opérateur d'adresses *ADR1* , *ADR2* , avec le moniteur *CONSOP* .

```
connection  OPERATOR;
env  ENVTTY;
unique  OP1:TTY(ADR1);
        OP2:TTY(ADR2);
shared  OP:CONSOP;
OP.PRINT1:= OP1.WRITE-LINE;% La procédure réelle OP1.WRITE-LINE correspond %
OP.PRINT2:  = OP2.WRITE-LINE. % à  la procédure  dummy OP.PRINT1 %
ext procedure  OP.ALARM(L:LINE);
% la procédure ALARM de OP pourra être appelée par la suite de l'extérieu
 d'OPERATOR %
init   ; % pas d'initialisation des modules OP1,OP2 et OP %
end  OPERATOR.
```

Figure 6.10

Les spécifications de l'autre type de périphérique utilisé , *DISK* , sont celles du § II-1.3 . L'occurrence de *INCORRECT-BLOC* peut être évitée par des vérifications en amont (dans le module *DISQUEA* du § VI-3.2) et *INTERVENTI* ne peut jamais survenir , car la procédure d'initialisation du système assure que les disques sont toujours montés et sous tension . Par contre , l'occurrence de *TRANSMISSION* ne peut pas être évitée (voir § VI-4.1.) et dans ce cas , le bloc échangé est laissé dans un état non défini . Le module *DDDISK* ,de la figure 6.11, tente de masquer *TRANSMISSION* en ré-essa un nombre (12) fini de fois , un échange interrompu par erreur de transmiss Si toutes les tentatives d'échange avec le disque se soldent par un échec, un message est envoyé à l'opérateur pour signaler la défaillance permanent d'un disque et demander la maintenance.

```
1   class  DDDISK(&D: integer); % &D: adresse physique du disque %
2   % env  ENVD%
3   const  NB=4096;%nombre de blocs d'un disque %
4   type  NOMBLOC=1 ..NB;
5         DISKBLOC=univ array[1..512]of char;
6   % env  ENVTTY%
7         LINE=univ array[1..80]of char;
8   % shared OP:OPERATOR; %
9   ref  procedure OP.ALARM(L:LINE);
10  % unique  D:DISK(&D);%
11  ref  procedure D.WRITE(B:NOMBLOC;BUF:DISKBLOC); signals  TRANSMISSION;
12  ref  function  D.READ(B:NOMBLOC):DISKBLOC; signals TRANSMISSION;
13  ext  function    READ(B:NOMBLOC):DISKBLOC; signals PCF,%Permanent
                                                Communication Failure %
14  % si PCF survient,le résultat retourné est non défini,
15    standard: retourne l'état externe du bloc B du disque &D %
16  var  essai : integer;
17  begin  essai:=1;
18        if essai<=12 then
19        READ:=D.READ(B)[TRANSMISSION: begin essai:=essai+1;
20                                          retry
21                                      end];
22        if essai=13 then
23        begin  OP.ALARM("le disque",&D,"est défaillant");
24              signal  PCF
25        end
26  end;
27  ext  procedure  WRITE(B:NOMBLOC; BUF:DISKBLOC); signals  PCF;
28  var  essai:integer;
29  begin  essai:=1;
30        if essai<=12 then
31        D.WRITE(B,BUF)[TRANSMISSION:begin essai:=essai+1;
32                                          retry
33                                    end];
34        if essai=13 then
35        begin OP.ALARM("le disque",&D,"est défaillant");
36              signal  PCF
37        end
38  end;
39  begin % pas d'initialisation de ce module %
40  end  DDDISK.
```

Figure 6.11

VI-3.2. LE TYPE DISQUE

Dans le § V-1, nous avons supposé que les opérations définies sur les types périphériques sont atomiques. Or, les opérations implémentées par *DDDISK* au §VI-3.1, ne le sont pas. *DISQUEA* simule un disque "abstrait" atomique, à partir de deux disques physiques *D1* et *D2* non atomiques (fig-6.12 Les hypothèses sur lesquelles est basée la construction de *DISQUEA* sont:

-H1) *D1* et *D2* ne sont jamais simultanément défaillants .

-H2) Toute défaillance permanente de *D1* ou *D2* est signalée avec une sécurité de 100%(grâce à OPERATOR du §VI-3.1) à l'opérateur qui déclanche un processus de maintenance on-line *. Nous supposons que le disque défaillant est répar avant que le deuxième ne tombe en panne .

Le programme de *DISQUEA* est celui de la figure 6.13 .Toute écriture abstraite est traduite en deux écritures physiques sur *D1* et *D2* .Si les deux finissent bien, l'écriture abstraite est bonne.Si l'écriture sur *D1* signale *PCF*,le bloc mal écrit est verrouillé,l'écritur sur *D2* n'a plus lieu,et l'écriture abstraite es refusée par l'envoi du signal *BV* (Bloc Verrouill Si la première écriture finit bien,mais la deu -ième mal,on verrouille simplement le bloc mal écrit sur *D2*,et l'écriture abstraite est accept Si un des blocs physiques à écrire est verroui depuis une tentative d'écriture antérieure, aucune écriture physique n'a lieu,et l'écritur abstraite est refusée par le renvoi de signal *B*

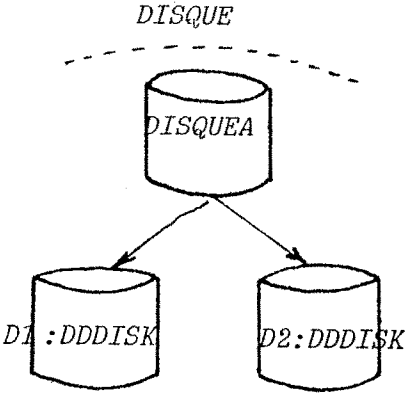


Figure 6.12

Ainsi,un appel à *DISQUEA.WRITE (B,BUF)* ne peut avoir que deux issues :

rien : si un des blocs physiques d'adresse *B* était verrouillé, ou si l'écriture sur *D1* se termine mal, l'état du disque"abstrait" ne change pas, et l'appelant reçoit un signal *BV*.

tout : si au moins l'écriture physique sur *D1* réussit,on n'envoie aucun signal d'exception, et l'état du disque"abstrait"change comme voulu.

---

* La possibilité de maintenance on-line influence l'architecture du systèm Ainsi, dans *DISQUEA*, on devrait prévoir une procédure *DEVERROUILLAGE* accessibl par un processus spécial *MAINTENANCE*, qui doit être appeléepar ce processus après avoir par exemple, réparé *D1* défaillant pour:

-actualiser l'état du disque réparé,*D1*, à partir de l'état de *D2* resté val

-déverrouiller les blocs verrouillés depuis le début de la défaillance permanente de *D1* (voirfigure 6.13) .

Pour des raisons de simplicité,cette procédure n'est pas incluse dans la fig-1 et nous n'aborderons pas non plus, le problème de la conception du processus *MAINTENANCE*. L'étude des conséquences que le choix d'une certaine stratégie de maintenance a sur la structure d'un système dépasse le cadre de cette thèse.

```
1   monitor DISQUEA;
2   %env  ENVD%
3   type  NOMBLOC=1..4096;
4         DISKBLOC= univ array [1..512] of char;
5         $CPARAM = record X:BLOC; Y:DISKBLOC end; %type du paramètre
                                                    d'annulation d'écriture%
6   %unique D1:DDDISK(&D1);%
7   ref function D1.READ(B:NOMBLOC):DISKBLOC; signals PCF;
8   ref procedure D1.WRITE(B:NOMBLOC; BUF:DISKBLOC); signals PCF;
9   %unique D2:DDDISK(&D2); %
10  ref function D2.READ(B:NOMBLOC):DISKBLOC; signals PCF;
11  ref procedure D2.WRITE (B:NOMBLOC; BUF:DISKBLOC);signals PCF;
12  var VERROU : array [NOMBLOC,0..1] of boolean;
13  ext function READ(B:NOMBLOC) :DISKBLOC;
14  begin if not VERROU(B,1) then begin READ:=D1.READ(B)
15        [D1.PCF: if not VERROU(B,2) then READ:=D2.READ(B)
              [D2.PCF: signal DEFAILLANCE] ] end
16        else if not VERROU(B,2) then READ:=D2.READ(B)[D2.PCF:signal DEFAILLANCE
17        else signal  DEFAILLANCE;
18  end;
19  ext procedure WRITE (B:NOMBLOC; NEW:DISKBLOC) :$CPARAM; signals BV; %Bloc
                                                          Verrouillé %
20  %standard : $CPARAM.X :=B; $CPARAM.Y : = DISQUE.READ (B),
21              l'état externe du bloc B du disque,devient égal à NEW %
22  begin $CPARAM.X:=B; $CPARAM.Y:=READ(B)[DEFAILLANCE: signal DEFAILLANCE];
23        if VERROU(B,1) or VERROU(B,2) then signal BV;
24        D1.WRITE (B,NEW) [D1.PCF: begin VERROU(B,1):=true; signal BV end];
25        D2.WRITE(B,NEW)[D2.PCF: VERROU(B,2):=true]
26  end;
27  can function $WRITE(P:$CPARAM);
28  % cette fonction d'annulation,inconnue à l'exterieur de DISQUEA,ne peut
      être appelée que par le mécanisme de restauration %
29  with P do
30  begin if VERROU(X,1) and VERROU(X,2)then signal DEFAILLANCE;
31        if not VERROU(X,1) then D1.WRITE(X,Y)[D1.PCF: begin VERROU(X,1):=true;
32                                                    if VERROU(X,2) then signa
                                                      DEFAILLANCE end];
33        if not VERROU(X,2) then D2.WRITE(X,Y)[D2.PCF:begin VERROU(X,2):=true;
34                                                    if VERROU(X,1)then signal
                                                      DEFAILLANCE end];
35  end;
```

Figure 6.13 (DEBUT)

```
36   begin for  I:=1 to 4096 do
37         for  J:=1 to 2 do  VERROU(I,J):=false;
38   end  DISQUEA.
```

Figure 6.13 (fin)

La figure 6;14 contient le programme de connexion des deux disques physiques *D1* et *D2* avec le module qui implémente le disque "abstrait" *DISQUEA* ; *ADRD1* et *ADRD2* sont les adresses physiques de *D1* et *D2* et *ENVD* , *ENVTTY* les environ--nements * [Mossière 78] communs aux modules connectés par *DISQUE* .

```
connection DISQUE;
env  ENVD,ENVTTY;
unique  D1:DDDISK(ADRD1);
        D2:DDDISK(ADRD2);
shared  D:DISQUEA;
        OP:OPERATOR;
ext function  D.READ(B:NOMBLOC):DISKBLOC;
ext function  D.WRITE(B:NOMBLOC; BUF:DISKBLOC); signals BV;
init  D;
end  DISQUE.
```

Figure 6.14

* Les déclarations de constantes et types qui sont communes à plusieurs module sont écrites en un seul exemplaire dans un environnement conservé par le BIBLIOTHECAIRE . En tête des déclarations d'un module , le programmeur peut spécifier des environnements . Ces environnements seront insérés dans le module par le compilateur, au même niveau que les déclarations globales au module .

VI-3.3. LE TYPE *VAS* .

VAS ,construit à l'aide de *CATALOGUE* , *SEGMENTS*, et *DISQUE* (fig.6.5 du § VI.2.5) importe de ces modules un certain nombre de constantes et types concrets que nous supposons déclarés dans les environnements *ENVCAT* , *ENVSEG* et *ENVD* , et exporte à son tour,vers *TRANSACTION*, un autre ensemble de constantes et types concrets , contenus dans l'environnement *ENVAS* . Dans le modèle de module de la figure 6.15 , nous avons donné le résultat de l'expansion des déclarations de ces environnements par le compilateur .

Pour obtenir et résilier le droit d'accéder aux articles d'un fichier , *VAS* utilise deux fonctions externes *DEMANDE-ACCES* et *FIN-ACCES* du moniteur *CAT* de type *CATALOGUE* . Ainsi, dans notre système , *CAT* mémorise non seulement la correspondance " noms externes de fichiers "⟶" noms internes de segments ", mais aussi l'état de partage des fichiers ouverts dans le système . *CAT* notifie les conflits de partage (§ VI-1.2) par l'émission du signal d'exception *CONFLICT* . L'implémentation de *CAT* (que nous ne décrivons pas ici, pour des raisons d'espace) prévoit pour la procédure de *DEMANDE-ACCES* qui induit un changement d'état , une procédure d'annulation de ce changement *$DEMANDE-ACCES* , dont le corps et les paramètres sont identiques à ceux de *FIN-ACCES* . Cette procédure d'annulation ne peut être activée qu'implicitement par le mécanisme de restauration (§ VI-4), si dans un module qui utilise *CAT* (*GES* , *VAS* , *TRANSACTION*), on active la primitive *reset* . C'est pour cette raison que *$DEMANDE-ACCES* ne figure pas parmi les fonctions externes appelables explicitement à partir de *VAS* (voir figure 6.15).

Pour obtenir le descripteur du segment qui mémorise les articles d'un fichier ouvert , *VAS* appelle la fonction *LIRE* de *SEGMENTS* (§II.2.3) .

Pour sauvegarder ou lire l'état des pages d'un segment sur la mémoire secondaire, *VAS* utilise les opérations *READ* et *WRITE* du type *DISQUE* (§ VI-3.2) .

La fonction interne *DEPLACEMENT* , calcule le déplacement de l'article courant *NAC* dans la page courante *NPC* selon la technique classique"en quatre étapes" décrite au § VI-2.3 .

Au corps de toutes les fonctions externes du module , le compilateur attachera le traitement standard par défaut pour défaillance (§ VI-5) .

```
0   class  VAS;
1   % env  ENVCAT,ENVSEG,ENVD,ENVAS %
2   const  NB=4096,NS=1024 ,NP=8, LMAXF=128, LA=32, LNOMX=8, NA=16;
3   type   NOMBLOC=1..NB;NOMSEGMENT=1..NS;NOMPAGE=1..NP; TAILLE=1..LMAXF;
4          MODEACCES=(L,LE);
```

Figure 15 (début)

```
5       DISKBLOC=univ *array [1..512] of char;
6       ARTICLE=array[1..LA]of char;
7       NOMX=array[1..LNOMX]of char;
8       MAPPING=array[NOMPAGE]of NOMBLOC;
9       SEGMENT=record T:TAILLE;
10                     M:MAPPING
11              end;    %fin env %
12      PAGE=array [1..NA]of boolean ; % 16 articles par page %
13      ETAT-INTERNE=record NOM:NOMX; % nom externe du fichier accédé %
14                          MA:MODEACCES;
15                          NAC:0..LMAXF; % Numéro de l'Article Courant; 0=n
16                          NPC:0..NP; % Numéro de Page Courante; 0=nil %
17                          S:SEGMENT; % descripteur du segment du fichier
                                         accédé %
18                          P:PAGE; % image de la page courante %
19                          ECRIT:boolean% vrai si la page courante a été éc
20                    end;
21  %shared CAT:CATALOGUE(NS); %
22  ref function CAT.DEMANDE-ACCES(N:NOMX; MACC:MODEACCES):NOMSEGMENT;
                 signals INEXISTENT,CONFLICT;
23  ref procedure CAT.FIN-ACCES(N:NOMX; MACC:MODEACCES);
24  %shared SEG:SEGMENTS(NS,NB,NP);
25  ref function SEG.LIRE(N:NOMSEGMENT):SEGMENT;signals RANGE-ERROR,ILLEGAL;
26  %shared D:DISQUE; %
27  ref function D.READ(B:NOMBLOC):DISKBLOC; *
28  ref procedure D.WRITE(B:NOMBLOC; BUF:DISKBLOC*); signalsBV;
29  var DCB:ETAT-INTERNE;
30  ext procedure OUVRIR(N:NOMX; AC:MODEACCES); signals INEXISTENT, CONFLIC
31  with DCB do
32  begin S:=SEG.LIRE(CAT.DEMANDE-ACCES(N,AC))[INEXISTENT:signal INEXISTENT,
33                                             CONFLICT: signal CONFLICT];
34        NOM:=N;MA:=AC; ECRIT:=false;
```

Figure 6.15 (SUITE)

Si le mot-clé *univ* précède une déclaration de paramètre formel(ou résultat de fonction)la seule vérification qui sera faite à la compilation sur le paramètre effectif(ou variable à affecter)conçerne la longueur de sa représentation interne qui doit être égale à celle du paramètre formel(ou du résultat).L'introduction de *univ*pour affaiblir dans certains cas les vérifications de type,est nécessaire dan tout langage destiné à implémenter des systèmes [Brinch Hansen 77].En effet,dans un système il est souvent nécessaire de pouvoir sauvegarder sur un même périphérique (disque,bande) l'état de certains objets qui ont été déclarés de type différent. Ceci est possible technologiquement puisque les opérations d'entrée /sortie physiques portent sur la représentation interne de ces objets.Ainsi,une activation de *D.WRITE*(lignes 38,49) avec le paramètre effectif *DCB.P* de type *PAGE* est légale,car la longueur de la représentation en mémoire centrale d'une page(32x16=512 caractèr est la même que la longueur de la représentation interne d'un bloc disque de type *DISKBLOC*.

```
35          NAC:=0; PAGE:=0;
36    end;
37    ext procedure FERMER; signals VAV; %Voie d'Accès Verrouillée %
38    with DCB do begin if ECRIT then D.WRITE(S.M[NPC],P)[BV:reset VAV];
39                       CAT.FIN-ACCES(NOM,MA);
40                end;
41    function DEPLACEMENT:1..NA; signals EOF,VAV; % End Of File %
42    var NEWPAGE:integer;
43    with DCB do
44    begin NAC:=NAC+1;
45          if NAC>S.T then signal EOF;
46          NEWPAGE:=NAC div 16+1;
47          if NEWPAGE≠NPC then
48          begin if ECRIT then
49                begin D.WRITE(S.M[NPC],P)[BV: signal VAV];
50                      ECRIT:=false;
51                end;
52                NPC:=NEWPAGE;
53                P:=D.READ( NEWPAGE );
54          end;
55          DEPLACEMENT:=NAC mod 16+1
56    end;
57    ext procedure COURANT(ACC:TAILLE); signals EOF;
58    begin if ACC>DCB.S.T then signal EOF;
59          DCB.NAC:=ACC-1;
60    end,
61    ext function LIRE:ARTICLE; signals EOF, VAV;
62    LIRE:=DCB.P[DEPLACEMENT] [EOF:signal EOF,VAV:reset VAV];
63    ext procedure ECRIRE(A:ARTICLE); signals EOF,VAV,INTERDIT;
64    with DCB do
65    begin if MA=L then signal INTERDIT;
66          P[DEPLACEMENT]:=A [EOF: signal EOF,VAV:reset VAV]
67          ECRIT:= true
68    end;
69    begin % partie initialisation vide %
70    end VAS .
```

Figure 6.15 (FIN)

VI-3.4 LE TYPE *TRANSACTION* .

TRANSACTION , construit à l'aide de *GES,VAS* et *UTILISATEUR* (fig-6.6 du § VI· importe de ces modules les environnements *ENVGES,ENVAS* et *ENVU* , et exporte · un *PROCESSUS* l'environnement *ENVU* contenant la description syntaxique des commandes de l'utilisateur interactif .

Comme nous ne voulons pas entrer dans les détails de structure fine d'*UTILISATEUR* , nous dirons simplement que pendant l'activation de *LIRE* , il e possible dedétecter toute commande syntaxiquement illégale de l'utilisateur . Ainsi , à la suite d'une activation de *UTILISATEUR.LIRE* , dans *TRANSACTION*, on reçoit :

- ou bien une valeur de type *COMMANDE* (§VI-2.2)
- ou bien un des signaux d'exception: *CI* (Commande Incorrecte Syntaxiquem ou *CD* (Communication Défaillante avec la console de l'utilisateur) .

Dans la figure 6.16 , nous nous sommes limité à décrire uniquement la structure fine de la transaction qui nous intéresse : *MAJ* . Les procédures internes à *MAJ* : *TINEXISTENT*, *TCONFLICT*, *TVAV*, *TEOF* et *CC* , sont des traitants associés aux signaux d'exceptions pouvant être envoyés vers *MAJ* par *VAS* ou *UTILISATEUR* . Certains de ces traitants peuvent finir par signaler l'exception *DEFAILLANCE* dans le corps de la fonction *MAJ* . Ces possibles signaux de *DEFAILLANCE* seront tous traités par le traitant par défaut pour *DEFAILLANCE* , attaché par le compilateur au corps de *MAJ*.

Dans la figure 6.17 , nous donnons la procédure de connexion *NIVEAU4* , décrivant comment il faut connecter entre eux les composants passifs du système pour offrir aux processus les opérations définies sur les types *TRANSACTION* et *UTILISATEUR* .

```
1   class  TRANSACTION;
2  %env  ENVGES,ENVAS,ENVU %
3   const  LNOMX=8, LMAXF=128, LA=32;
4   type  NOMX=array [1..LNOMX] of char;
5         TAILLE=1..LMAXF;
6         ARTICLE=array [1..32] of char;
7         MODEACCES=(L, LE);
8         COD:(DT,FT,AT,C,D,A,AC,L,E);
9         COMMANDE: record case CODE:COD of
10                  DT,FT;AT:(  );
11                  C:(NOM:NOMX, T:TAILLE);
12                  D:(NOM:NOMX);
13                  A:(NOM:NOMX, ACCES :MODEACCES);
14                  AC:(NAC:TAILLE);
15                  L:(  );
16                  E:(ART:ARTICLE)
17                  endcase;
18                  end;
19 %unique VA:VAS; %
20  ref procedureVA.OUVRIR(N:NOMX; ACC:MODEACCES); signals INEXISTENT,CONFLICT;
21  ref procedure VA.FERMER; signals VAV;
22  ref procedure VA.COURANT(ARC:TAILLE); signalsEOF;
23  ref function  VA.LIRE:ARTICLE; signals EOF, VAV;
24  ref procedure VA.ECRIRE(A:ARTICLE) signals EOF, VAV, INTERDIT;
25 %unique  U:UTILISATEUR(&ADRTTYU); %
26  ref function  U.LIRE:COMMANDE; signals CI, CD;
27  ref procedure  U.ENVOYER(M:MESSAGE); signals CD;
28  ext procedure MAJ(N:NOMX);
29  var  BUF:ARTICLE;
30       C:COMMANDE;
31  procedure TINEXISTENT; % taitement de VA.INEXISTENT %
32  begin  U.ENVOYER ("fichier inexistent,transaction annulée");
33         signal DEFAILLANCE;
34  end;
35  procedure  TCONFLICT; % traitement de VA.CONFLICT %
36  begin  U.ENVOYER ("conflit d'accès,transaction annulée,
                       revenez plus tard s.v.p" );
37         signal DEFAILLANCE;
38  end;
39  procedure  TVAV; % traitement de VA.VAV.%
40  begin  U.ENVOYER ("un des disques est défaillant et la maintenance a été
42                     demandée ,transaction annulée,revenez plus tard s.v.p");
42         signal  DEFAILLANCE;
43  end;
```

Figure 6.16

```
44  procedure  TEOF; % traitement de VA.EOF %
45  begin  U.ENVOYER ("fin de fichier")
46  end;
47  function  CC:COMMANDE; % Commande Correcte %;
48  var  compteur : integer;
49  begin compteur :=13;
50        if compteur > 0 then
51        CC:=U.LIRE [CI: begin U.ENVOYER ("commande incorrecte");
52                           compteur :=compteur -1;
53                           retry
54                     end];
55        if compteur =0 then begin U.ENVOYER("transaction annulée");
56                                  signal  DEFAILLANCE
57                            end;
58  end;
59  begin % corps de la fonction externe MAJ %
60        VA.OUVRIR(N,LE)[INEXISTENT:TINEXISTENT, CONFLICT:TCONFLICT];
61        C:=CC;
62        with C do
63        while CODE≠ FT do
64        begin case  CODE of
65              DT,C,D,A:U.ENVOYER ("commande erronée");
66              AT: reset  DEFAILLANCE;
67              AC:VA.COURANT(NAC) [EOF:TEOF];
68              L: begin BUF:=VA.LIRE [EOF:TEOF, VAV:TVAV];
69                       U.ENVOYER(BUF);
70                 end;
71               E:VA.ECRIRE(ART) [EOF:TEOF, VAV:TVAV,
72                                 INTERDIT:U.ENVOYER ("mode d'accès interdit"
73              end case;
74              C:=CC;
75        end;
76        VA.FERMER(N,LE) [VAV:TVAV];
77  end;[DEFAILLANCE : reset DEFAILLANCE]
      .
      .
    %le corps des autres fonctions externes CREATION,DESTRUCTION,LECTURE %
      .
      .
    begin  % partie initialisation vide %
    end  TRANSACTION.
```

Figure 6.16 (FIN)

```
connection  NIVEAU4 (&ADRTTYU);  %&ADRTTYU=adresse du télétype d'un utilisateur U%
env  ENVD, ENVCAT, ENVSEG,
     ENVAS, ENVGES, ENVU;

shared  CAT : CATALOGUE(NS);
        SEG:SEGMENTS(NB, NS, NP);
        D : DISQUE;

unique  U : UTILISATEUR(&ADRTTYU);
        VA : VAS;
        GE : GES;
        T : TRANSACTION;

ext  procedure T.CREATION(N:NOMX; T:TAILLE);
               T.DESTRUCTION(N:NOMX);
               T.MAJ(N:NOMX);
               T.LECTURE(N:NOMX);
               U.LIRE:COMMANDE; signals CI, CD;
               U.ENVOYER(M:MESSAGE); signals CD;

init  CAT, SEG, D, U;
end  NIVEAU4.
```

Figure 6.17

VI-3.5 PROGRAMME D'UN PROCESSUS.

Chaque processus P_i peut appeler les opérations externes spécifiées sur U_i:*UTILISATEUR* et T_i:*TRANSACTION* (figure 6.7 du §VI-2.4) . Mais si l'on veut écrire un seul modèle de programme pour tous les processus du système, (ce qui est souhaitable, car les algorithmes des processus sont identiques) il ne faut pas faire apparaître, dans l'expression des droits d'accès d'un processus, le nom de ses variables abstraites , car le nom de ces variables est distinct , pour des processus distincts . L'utilisation du concept de référence externe non résolue (repérée par le mot-clé dummy [Cheval 76]) permet de donner une solution élégante à ce problème .

La figure 6.18 décrit un modèle de processus , et dans la figure 6.19 , nous décrivons tout le système *MULTIACCES* par un programme de connexion .

```
1   process  PROCESSUS;
2   %env  ENVU; %
3   const LNOMX=8, LMAXF=128, LA=32;
4   type  NOMX= array [1..LNOMX] of char;
5         TAILLE= 1..LMAXF;
6         ARTICLE= array [1..32] of char;
7         MODEACCES= (L,LE);
8         COD = (DT, FT, AT,C,D,A, AC,L, E);
9         COMMANDE = record case  CODE of
10                              DT, FT, AT, :( );
11                              C: (NOM:NOMX, T:TAILLE);
12                              D: (NOM:NOMX);
13                              A: (NOM:NOMX, ACCES:MODEACCES);
14                              AC (NAC:TAILLE);
15                              L: ( );
16                              E: (ART:ARTICLE)
17                              end case;
18                      end;
19  %unique  T:TRANSACTION %
20  dummy  procedure  CREATION (N:NOMX; T:TAILLE);
21  dummy  procedure  DESTRUCTION (N:NOMX);
22  dummy  procedure  MAJ (N:NOMX);
23  dummy  procedure  LECTURE (N:NOMX);
24  %unique  U:UTILISATEUR %
25  dummy   function  LIRE:COMMANDE; signals CI, CD;
26  dummy  procedure  ENVOYER (M:MESSAGE); signals CD;
27  var  C:COMMANDE;

28  begin  % algorithme du processus %
29          while true do
30          begin C:= LIRE[CI: begin envoyer("DT attendu");
                                goto FIN end, CD: goto FIN];
31                if C.CODE ≠ DT then begin ENVOYER("DT attendu"); goto FIN end;
32                C:=LIRE[CI:begin ENVOYER("commande erronée;recommencez");
                                goto FIN end,
33                              CD: goto FIN];
34                  with C do %ici,C contient une commande correcte syntaxiquen
35                  begin case  CODE  of
36                        DT,FT,AT,AC,L,E:ENVOYER("commande erronée,recommencez
37                        C:CREATION(NOM,T);
38                        D:DESTRUCTION(NOM);
```

```
39                      A: case ACCES of
40                        L: LECTURE(N);
41                        LE:MAJ(N)
42                        endcase;
43                    endcase;
44              end  [DEFAILLANCE:  ];
45              FIN:
46        end;
47  end P.
```

Figure 6.18(FIN)

```
system  MULTIACCES;
 env  ENVU;
 const  N= % nombre maximal d'utilisateurs interactifs %
        ADC= array [1..N]of ADRTTY1, ADRTTY2,..,ADRTTYN; %adresses physiques
                des consoles des utilisateurs %
 for I:=1 to N do  % déclarations des composants du système  %
 begin  unique V[I]: NIVEAU4 (ADC[I]);
        process  PR [I] : PROCESSUS;
 end;
 for I:=1 to N do %declaration des connexions à réaliser entre les composants %
 begin  PR[I].CREATION:= V[I].CREATION;
        PR[I].DESTRUCTION:= V[I].DESTRUCTION;
        PR[I].MAJ          := V[I].MAJ;
        PR[I].LECTURE      :=V [I].LECTURE;
        PR[I].LIRE         :=V [I].LIRE;
        PR[I].ENVOYER      :=V [I].ENVOYER;
end;
% initialisation du système %
for  I:=1 to N do
begin  init  V[I];
       start PR[I]
end;
end  MULTIACCES.
```

Figure 6.19

VI-4 LA TRANSACTION *MAJ* EST UNE OPERATION ATOMIQUE .

Le but du chapitre VI est de montrer que le mécanisme d'exception du chapitre est un outil commode pour la construction des types abstraits ayant des opérati atomiques . Il serait trop long de démontrer que toutes les opérations définies le type *TRANSACTION* sont atomiques . Nous nous limiterons à montrer comment il est possible de prouver l'atomicité de *MAJ* , étant entendu que la preuve de l'atomicité des autres transactions peut être faite de manière analogue .

Pour démontrer que *MAJ* est atomique, il est d'abord nécessaire de déterminer l'ensemble *EP* des Exceptions Pouvant être détectées pendant l'exécution de *MAJ* La figure 6.20 contient la liste des opérations abstraites activables durant une exécution de *MAJ* .

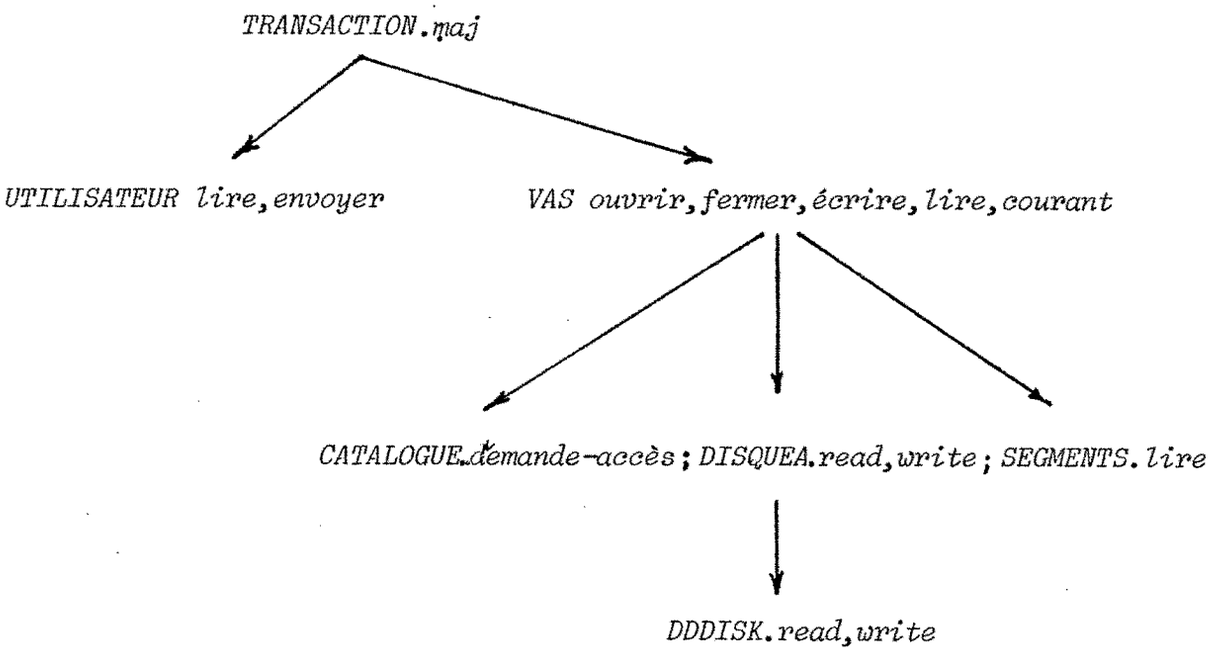


Figure 6.20

Ces opérations abstraites contiennent à leur tour des appels à des opérations concrètes spécifiées pour des types de niveau 0:

{ *integer, char, boolean, scalar, interval, array, record, TTY, DISK* }

Soit E_0 l'ensemble d'exceptions spécifiées pour les opérations de niveau 0 utilisées par *MAJ* .

E_0 = {*ARITMETIC-OVERFLOW, RANGE-ERROR, TAGFIELD-TYPE-BAD, NO-CASE-PROVIDED-FOR-THI -VALUE, DIVISION-BY-ZERO, ILLEGAL-CHARACTER, INCORRECT-LINE, TTY.TRANSMISSION INCORRECT-BLOCK, DISK.TRANSMISSION, INTERVENTION* }

En utilisant les méthodes d'analyse statique, on peut démontrer qu'un certain nombre d'exceptions $ET_0 \subset E_0$ ne peuvent pas survenir à l'exécution, parceque leur précondition est toujours fausse . Soit $EP_0 = E_0 - ET_0$, l'ensemble des exceptions de niveau 0 qui ne peuvent pas être évitées par des moyens statiques . De manière analogue,on peut déterminer les ensembles EP_1 , EP_2 , EP_3 et EP_4 des exceptions de niveau 1, 2, 3 et respectivement 4 qui ne peuvent pas être évitées statiquement .

$$EP = EP_0 \cup EP_1 \cup EP_2 \cup EP_3 \cup EP_4$$

Conformément à la définition du § III-2.1 , pour prouver que *MAJ* est atomique , il faudrait prouver que *MAJ* est tolérante à toute exception de *EP* . Cela revient à démontrer *card (EP)* lemmes plus simples , concernant la tolérance de *MAJ* pour chaque exception de *EP* .
Par la suite , nous n'allons pas démontrer toutes ces lemmes , car cela allongerait trop notre exposé . Nous nous contenterons d'illustrer le principe de telles démonstrations partielles , en montrant :

- au § VI-4.1 que *MAJ* est en général fortement tolérante à *DISK.TRANSMISSION* $\in EP_0$
- au § VI-4.2 que *MAJ* est faiblement tolérante à TTY.TRANSMISSION $\in EP_0$ ou *CAT.CONFLICT* $\in EP_1$
- au § VI-4.3 que *MAJ* est faiblement tolérante à une *DEFAILLANCE* $\in EP_4$:

VI-4.1 TOLERANCE FORTE A CERTAINES EXCEPTIONS D'ENTREE / SORTIE .

1) *DISK.TRANSMISSION* :

Une erreur de cadence ou une erreur de parité , provoquant l'émissio du signal *TRANSMISSION*, surviennent dans certaines circonstances particulière de fonctionnement d'un périphérique . Par exemple, dans le cas de l'ancien ordinateur CII 10070 de l'IMAG , où deux disques et une imprimante étaient gérés par un même canal, une erreur de cadence pouvait survenir quand les trois périphériques étaient lancés en même temps par le canal, mais ne survenait jamais quand le degré de parallélisme réel était inférieur ou égal à 2 [Moalla 79]. De la même façon, une certaine imprécision mécanique dans le déplacement d'une tête de disque peut provoquer de temps en temps un phénomène de diaphonie (détecté par une erreur de parité) lorsque la tête en question est suffisamment décalée par rapport à la piste voulue pour subir l'influence magnétique de la piste voisine [Moalla 79]. La probabilité d'occurrence de telles situations étant assez faible, (dans le cas des tois périphériques du 10070 , d'environ 0,006), il y a de fortes chances que lorsqu'on re-essaye un transfert qui a mal fini, on ne retombe plus sur les mêmes circonstances "critiques" .

Comme *DISK.TRANSMISSION* ne peut pas être évitée par des moyens statique elle est bien incluse dans l' ensemble EP_0 . Puisque *MULTIACCES* utilise deux variables concrètes *&D1* , *&D2* de type *DISK*, nous allons distinguer une exception *TRANSMISSION* suivant qu'elle est signalée par le disque physique d'adresse *&D1* ou *&D2*, en utilisant les noms qualifiés *&D1.TRANSMI* et *&D2*.

La levée d'une exception *&D2.TRANSMISSION (I=1,2)* en *DI.DDDISK* (fig-6.11 lignes 19,31) correspond à la détection (implicite) de l'exception *DI.PCF(I* Pour les raisons évoquées précédemment, les tentatives de masquage de *DI.PCF* de la figure 6.11 , seront dans la plupart des cas couronnées de succès, et la propagation *&DI.TRANSMISSION* ⟶ *DI.PCF (I = 1,2)* s'arrêtera en général au niveau des modules *DI*.

Si toutefois, les douzes tentatives de masquage contenues dans un certain *DI (I=1,2)* ne réussissent pas, le bloc physique à transférer est laissé dans un état incohérent, la néparation de *DI* est demandée par un appel à *OP.ALARM* (fig-6.11, lignes 23,35)et l'exception *DI.PCF* est levée dans *D:DISQUEA* (figure 6.13) . Si les hypothèses HI) et H2) du § VI-3.2 ,(sur lesquelles la construction du type abstrait *DISQUEA* est basée) sont valides * , alors nous pouvons démontrer que les opérations

- *D.READ* et *D.$WRITE* sont fortement tolérantes à *&D1.TRANSMISSION* et *&D1.TRANSMISSION* .
- *D.WRITE* est faiblement tolérante à *&D1.TRANSMISSION* , et fortemen tolérante à *&D2.TRANSMISSION*.

a) Si *D1.PCF* est levée en *D.DISQUEA* pendant l'activation de *D.READ* ou *D.$WRITE* , l'effet standard de ces procédures est disponible . En effet, si *D1.PCF* est signalée par *D1* , H1) et H2) entraînent que le disque *D2* n'est pas verrouillé , et que *D2.PCF* ne surviendra pas. Par conséquent, le traitan de *D1.PCF* (ligne 15,figure 6.13) pourra affecter à *D.READ* la valeur de *D2.READ*, et réaliser l'effet standard . De la même façon, dans *D.$WRITE*, l'appel à *D2.WRITE* (ligne 33, FIGURE 6.13) , aboutira à la réalisation de l'effet standard .

b) Si *D1.PCF* est levée en *D* pendant l'activation *D.WRITE*, l'effet standard ne sera pas disponible, mais *D* gardera l'état externe d'avant l'activation de *D.WRITE* et signalera l'exception *D.BV* à son utilisateur *VAS*.

---

* Si H1) et H2) deviennentinvalideslors d'une activation de :
- *DISQEA.$WRITE*, alors l'exception *DEFAILLANCE* sera détectée (lignes 30,32, dans une fonction d'annulation , ce qui provoquera l'arrêt immédiat de *MULTIACCES* (voir § IV.4) .
- *DISQUEA .READ* ou *DISQUEA.WRITE*, alors l'exception *DEFAILLANCE* (ligne 15,16 sera levée en *VAS* . Ceci provoque l'activation d'un traitant par défaut attaché à une des procédures externes de *VAS* . Ce traitant exécute un *res* qui active à son tour *DISQUEA.$WRITE* , conduisant à l'arrêt de *MULTIACCES*

En effet, le traitant de *D1.PCF* (ligne 24, fig- 6.13) verrouille le bloc *B* sur *D1*, mais comme *B* n'est pas verrouillé sur *D2* , l'état du bloc"abstrait" *B* reste égal à l'état du bloc physique *B* du disque *D2*, qui n'a pas été altéré depuis l'activation de *D.WRITE* .

c) Si *D2.PCF* est levée en *D*, l'effet standard de *D.READ*, *D.$WRITE* ou *D.WRITE* sera disponible. En effet, H1) et H2) entraînent que le disque *D1* est en bon état au moment où l'exception *D2.PCF* est levée en *D* . Dans ces conditions, *D2.PCF* ne peut pas survenir pendant une activation de *D.READ* , car, si la première lecture sur *D1* marche, on ne lit plus le disque *D2* . L'effet standard de *D.READ* est donc disponible .Supposons que *D2.PCF* survient pendant une activation de *D.WRITE* (ligne 25 de la figure 6.13). Le traitant de *D2.PCF* se contente de verrouiller le bloc *B* sur *D2* . A l'entrée dans *D.WRITE* , le *VERROU(B,1)* était forcément faux, et durant l'activation de *D1.WRITE* (ligne 31), l'exception *D1.PCF* n'a pas pu survenir, en vertu de H1) et H2) . L'état externe du bloc "abstrait" *B* , égal à l'état externe du bloc *B* de *D1*, a changé comme voulu , donc le service standard de *D.WRITE* est disponible . De la même façon, si *D2.PCF* est levée pendant l'exécution de *D.$WRITE*, comme l'écriture sur *D1* a marché, à cause de H1) et H2) , l'effet standard de *D.$WRITE* sera disponible .

En conclusion a), b)et c) montrent que :

- La propagation *&DI.TRANSMISSION* ⟶ *DI.PCF* ⟶ *D.BV* *(I=1,2)* s'arrête toujours au niveau de *D* , sauf dans le cas où il y a eu détection de *D1.PCF* pendant une activation de *D.WRITE*, et dans ce cas, la propagation peut se poursuivre vers le module *VAS*, mais *D* garde l'état abstrait d'avant l'appel à *D.WRITE* .

Considérons maintenant l'éventualité où *D.BV* est levée en *VAS* (ligne 38 ou 49 de la figure 6.15) . Cette levée correspond à une détection implicite de l'exception *VAV* dans *VAS* . Le traitant de *D.BV* de la ligne 38, propage *VAV* dans *TRANSACTION.MAJ* (ligne 76 , figure 6.16) . Le traitant de la ligne 49 , local à la fonction interne *DEPLACEMENT* , lève *VAV* en *VAS.LIRE* ou *VAS.ECRIRE* (lignes62 et 66 de la figure 15) . Les taitants pour *VAV* déclarés dans ces procédures , propagent *VAV* dans *TRANSACTION.MAJ* (ligne 68 et 71 de la figure 6.16). Dans tous les cas, l'état interne restauré pour *VAS* est équivalent à l'état d'avant l'appel à *VAS.FERMER*, *VAS.LIRE* ou *VAS.ECRIRE* (parceque tous les traitants activés contiennent un appel à *reset*) . Donc,les opérations *VAS.FERMER*, *VAS.LIRE* et *VAS.ECRIRE* , pendant l'exécution desquelles *D.BV* peut être levée, sont bien faiblement tolérantes à cette exception .

La procédure interne *TVAV* (lignes 39-43 de la figure 6.16) qui est le traita de *VAV* (lignes 67,70,75 de la figure 6.16),envoie un message à l'utilisateur interactif en annonçant que la transaction de mise à jour sera annulée à cause d'une défaillance de la mémoire secondaire,et lève l'exception *DEFAILLANCE* dans le corps de la procédure *MAJ* (l'instruction bloc étalée entre les lignes 59 et 77 de la figure 6.16).Ceci entraîne l'activation du traitant pour *DEFAILLANCE* associé au corps de la procédure *MAJ* (ligne 77). Le *reset* contenu dans ce traitant va restaurer l'état des blocs du fichier *N* qui ont été décrits depuis le début de l'exécution de *MAJ* et qui sont devenus des résidus par des appels implicites à *D.$WRITE*. Dans les hypothèses H1) et H2),l'effet standard de toutes les activations de *D.$WRITE* est disponible. *reset* appelle également la fonction d'annulation *CAT.$DEMANDE-ACC* pour libérer l'accès à *N*, mais fait également un travail inutile : restaure pour la variable d'état concrète *DCB* de *VAS* l'état d'avant *VAS.OUVRIR*.(Ceci n'est pas nécessaire car *VAS.OUVRIR* est une opération d'initialisation).

En conclusion : *TRANSACTION.MAJ* est , dans la plupart des cas, fortement tolérante à *DISK.TRANSMISSION* $\in EP_0$.

Dans certaines circonstances, *MAJ* est faiblementtolérante à cette exception et émet le signal d'exception *MAJ.DEFAILLANCE* vers le *PROCESSUS* qui exécute *MAJ* .

VI-4.2 TOLERANCE FAIBLE A D'AUTRES EXCEPTIONS .

1) *TTY.TRANSMISSION* .

Comme dans le cas de *DISK.TRANSMISSION* (§VI-4.1), cette exception ne peut pas être évitée statiquement . Supposons que durant l'exécution de *V[I].MAJ* par un processus *PR[I](I=1,2,..N)* , une exception de niveau 0 *ADRTTYI.TRANSMISSION* est levée , soit en *U[I].LIRE* , soit en *U[I].ENVOYER*. Pour des raisons de simplicité, nous n'avons pas décrit la structure fine du type abstrait *UTILISATEUR*, mais nous pouvons admettre que l'écriture d'un traitant pour *TTY.TRANSMISSION* dans *UTILISATEUR* , dont le rôle se limite à propager l'exception de niveau 1 *UTILISATEUR.CD* (correspondant à *TTY.TRANSMISSION*) en *TRANSACTION* (lignes 32,36,40;45,51,55,65,69,72 figne pose aucun problème . Ce traitant ne doit pas restaurer un état interne cohérent pour *UTILISATEUR* , car cet état n'est pas conservé entre deux appels successifs à *LIRE* ou *ENVOYER* .

Dans le programme de *TRANSACTION* (figure 6.16), aucun traitant explicite pour *CD* n'est prévu , donc, si cette exception est levée dans *MAJ*, le mécanisme d'exception donnera le contrôle au traitant pour *DEFAILLANCE*

attaché au corps de la procédure externe *MAJ* . En conformité avec la définition de sa sémantique, au § IV-4, l'exécution du *reset*, contenu dans ce traitant (ligne 77, figure 6.16), restaure pour *V[I]* un état interne équivalent à l'état d'avant l'appel à *V[I].MAJ* . Cette restauration reste invisible pour les autres processus *PR[J],J≠I* , car les résidus laissés dans les moniteurs de *NIVEAU4* sont restaurés par des appels à des fonctions explicites d'annulation, qui garantissent le respect des invariants concrets de ces moniteurs (voir §IV.2) . L'exécution du traitant par défaut finit par lever l'exception *DEFAILLANCE* en *PR[I]* (ligne 44 de la fig 6.18) , ce qui aura pour effet d'activer le traitant pour *DEFAILLANCE* associé à l'instruction *with* (lignes 34-44, figure 6.18). L'unique effet de l'activation de ce traitant (vide) , est de forcer *PR[I]* à revenir en début de cycle . A partir de ce moment, l'utilisateur peut tenter de recommencer une autre transaction , ou peut partir de sa console ; dans ce cas, le processus *PR[I]* sera mis en attente passive sur l'instruction *LIRE*, (ligne 30, figure 6.18).

2) *CAT.CONFLICT*.

Cette exception est une exception d'entrée dans le moniteur *CAT* ,qui ne peut pas être évitée statiquement . Supposons qu'elle soit détectée à la suite d'une activation de *CAT.DEMANDE-ACCES* par le module *VA[I].OUVRIR*, qui à son tour, était appelé par *V[I].MAJ* . Sa levée dans *VA[I].OUVRIR* (ligne 33, figure 6.15) indique que le fichier *N* (que l'utilisateur *V[I]* a l'intention de mettre à jour), a été déjà ouvert en lecture / écriture par un autre utilisateur interactif . Au moment où *CAT.CONFLICT* est levée, l'état interne de *VA[I]* est identique à l'état interne d'avant *VA[I].OUVRIR* . Donc le traitement au niveau du module *VA[I]* se réduit à la propagation de l'exception *VA[I].CONFLICT* (qui correspond à *CAT.CONFLICT)* dans *V[I].MAJ* (ligne 60, figure 6.16) . Le traitant associé dans *V[I].MAJ* à *VA[I].CONFLICT* est *TCONFLICT* (lignes 35-38, figure 6.16). Ce traitant notifie à l'utilisateur interactif la détection d'un conflit d'accès, lui annonce que la transaction à peine entammée sera annulée, et lève l'exception *DEFAILLANCE* dans le corps de la procédure externe *MAJ* . Comme au point d'activation de *TCONFLICT* (ligne 60, figure 6.16) aucun traitant pour *DEFAILLANCE* n'est attaché , le contrôle sera donné par le mécanisme d'exception au traitant par défaut, attaché à la procédure *MAJ* toute entière (ligne 77) . Comme entre le début de l'activation de *V[I].MAJ* et l'activation de l'instruction *reset* incluse dans ce dernier traitant, aucune modification de l'état de *V[I]* n'a eu lieu, le cache est vide, et

le seul effet de *reset DEFAILLANCE* est de lever *V[I].DEFAILLANCE* dans *PR[I]* (ligne 41, figure 6.18). Une *DEFAILLANCE* levée pendant l'exécution de l'instruction-bloc qui s'étend entre les lignes 35 et 44 de la figure 6.18, donne le contrôle au traitant (vide) pour *DEFAILLANCE* , situé ligne 44. Le rôle de ce traitant est de replacer *PR[I]* en début de cycle, c'est à dire en attente d'une nouvelle demande de transaction .

VI-4.3 TOLERANCE FAIBLE A UNE *DEFAILLANCE* .

Nous avons montré au chapitre VI § 4.1 et 4.2 que *TRANSACTION.MAJ* est faiblement tolérante aux exceptions *VAS.VAV* , *UTILISATEUR.CD* et *VAS.CONFLICT* , dont la levée en *MAJ* conduit à une détection de l'exception *TRANSACTION.DEFAILLANCE* $\in EP_4$. En fait, l'insertion (implicite) par le compilateur du traitant par défaut pour *DEFAILLANCE* dans la ligne 77, fig-16, assure la tolérance faible de *TRANSACTION.MAJ* à toute autre exception pouvant être levée par les modules utilisés dans *MAJ* et non explicitement traitée dans *MAJ* . La propriété de faible tolérance est conditionnée ::

- Par le bon fonctionnemnet du mécanisme d'exception, qui doit activer le traitant de la ligne 76 (figure 6.16), dans tous les cas où un module utilisé par *TRANSACTION.MAJ* lève dans *MAJ* une exception qui n'est pas explicitement traitée dans *MAJ*.

- Par la validité des hypothèses H1, H2, H3 du § IV-1, sur lesquelles le fonctionnement du mécanisme de restauration est basé (au § VI-4.1 , nous avons vu que si H1 devient invalide pour *DISQUE*, le système *MULTIACCES* s'arrête) .

7. CONCLUSION

En admettant l'utilité de disposer de langages de programmation modulaires (offrant à leurs utilisateurs la possibilité d'enrichir un ensemble de types préexistants par des nouveaux types abstraits), nous avons abordé le problème du traitement des exceptions dans ces langages. Ce domaine particulier de la conception des langages fait l'objet de débats actuels. Mais ces débats ne sont facilités ni par l'absence d'une terminologie commune, ni par l'absence d'un consensus sur ce qu'est le rôle du traitement d'exceptions dans la construction des programmes fiables.
A notre avis, le but du traitement des exceptions dans les machines modulaires que l'on peut construire à l'aide des langages à types abstraits est la préservation des propriétés d'intégrité et d'abstraction (§III.3) en présence d'occurrences dynamiques d'exceptions. Une fois ce but posé, nous avons essayé de définir d'une façon aussi précise que possible les concepts qui nous semblaient utiles pour la compréhension de ce qu'est le traitement des exceptions. Nous avons montré que la propriété d'intégrité peut être atteinte, si les types utilisés pour construire une telle machine ont des opérations atomiques (faiblement ou fortement tolérantes aux exceptions pouvant être détectées à l'exécution). Pour résoudre les problèmes liés à la restauration des résidus induits dans des modules périphériques, nous avons proposé d'associer aux opérations qui changent l'état des périphériques, des opérations d'annulation. La propriété d'abstraction est liée au partage d'une machine modulaire par plusieurs processus qui s'exécutent en parallèle, et fait intervenir le concept de variable partagée. En SESAME, le mécanisme de synchronisation qui résoud les conflits d'accès à une telle variable est le moniteur [Hoare 74]. Si des opérations d'annulation sont associées aux opérations qui changent l'état des moniteurs, alors il est possible d'atteindre la propriété d'abstraction.

Pour pouvoir spécifier les effets exceptionnels des opérations définies sur un type, nous avons suggéré au §II.2 une extension de la technique de spécification opérationnelle des types, proposée dans [Wulf 76]. Pour l'implémentation des effets exceptionnels, nous avons proposé un mécanisme d'exception au chapitre cinq. L'introduction de l'exception prédéfinie DEFAILLANCE et des traitants par défaut implicitement attachés par le compilateur aux procédures externes de modules, permet de garantir la compatibilité de ce nouveau mécanisme avec les autres mécanismes d'abstraction du langage.

Au §III.2.5 nous avons remarqué que la présence d'un tel mécanisme dans le langage permet la faible tolérance aux exceptions DEFAILLANCE dues aux fautes de codage dont la période de latence ne dépasse pas une activation de module. Il est cependant admis que la probabilité de production des fautes de codage croît avec la complexité du langage de programmation. Peut-on alors justifier l'introduction de ce nouveau mécanisme,si sa présence contribue à accroître la complexité du langage ? Citons à ce sujet [Horning 78]: "La réponse doit être un "oui" nuancé. Oui, si le mécanisme est suffisamment simple pour maintenir la complexité qu'il introduit à un niveau jugé acceptable, et si sa disponibilité dans le langage permet de réduire d'une façon substantielle la complexité typique des programmes de traitement d'exceptions. Un mécanisme d'exception est nécessaire dans un langage à partir du moment où ce langage n'offre aucun autre moyen simple et sûr pour programmer le traitement des exceptions, dont l'occurrence ne peut pas être évitée par des moyens statiques". Le lecteur qui a une certaine expérience du traitement des exceptions dans des programmes assez complexes(systèmes d'exploitation, systèmes de gestion de bases de données) peut apprécier la simplicité et la lisibilité des programmes de traitement d'exception contenus dans le système multi-accès du chapitre six, ainsi que l'avantage d'avoir une séparation nette entre les algorithmes standard et les algorithmes d'exception.

Il serait possible de continuer le travail présenté dans cette thèse dans plusieurs directions. Sur un plan pratique,il est nécessaire de réaliser une implémentation du mécanisme, pour pouvoir l'utiliser à construire des programmes "réels". Au §V.7 nous avons remarqué qu'une implémentation pouvant offrir des services "à la carte" (propagation des exceptions + restauration des ensembles d'incohérence, ou seulement : propagation des signaux d'exception) permettra une plus grande souplesse d'utilisation. Des mesures quantitatives sont nécessaires pour évaluer les conséquences économiques de l'intégration du mécanisme dans le langage. Au §V.7 nous avons suggéré au moins deux études à faire : une, du style de celle qui est décrite dans [Gannon 75], pour évaluer les avantages de l'introduction du mécanisme dans le langage, une deuxième pour mesurer l'augmentation du temps d'exécution due à l'interférence des composants restauration et contrôle du mécanisme avec les autres primitives du langage.

Sur un plan théorique, il serait utile d'étudier une technique de spécification permettant de décrire l'effet exceptionnel d'une opération à la fois sur l'état des variables et sur le contrôle (§II.2.1). De plus, la sémantique du mécanisme d'exception lui-même, devrait pouvoir être spécifiée formellement. Il reste encore à élaborer un modèle conceptuel pour comprendre les problèmes qui se posent dans la re-synchronisation des processus coopérants, et éventuellement à imaginer des outils linguistiques pour la résolution de ce problème.

REFERENCES

[BERT 79] D. Bert : La programmation générique : construction de logiciel, spécification algébrique et vérification. Thèse d'Etat, IMAG, 1979.

[BRINCH HANSEN 73] P. Brinch Hansen : Operating Systems Principles. Prentice-Hall, 1973.

[BRINCH HANSEN 75] P. Brinch Hansen : The Programming Language Concurrent Pascal. Summer School on Language Hierarchies and Interfaces, Munich, 1975.

[BRINCH HANSEN 77] P. Brinch Hansen : The Architecture of Concurrent Programming. Prentice-Hall, 1977.

[BRON 76] C. Bron, M. Fokkinga, A. De Haas : A Proposal for Dealing with Abnormal Termination of Programs. Technical Report, Twente University of Technology, 1976.

[BURSTALL 77] R. Burstall, J. Goguen : Putting Theories Together to Make Specifications. Proc. IJCAJ-77, MIT, 1977.

[CHEVAL 76] J.L. Cheval, F. Cristian, S. Krakowiak, Ma. Lucas, J. Montuelle, J. Mossière : Un système d'aide à l'écriture des systèmes d'exploitation, Congrès AFCET, Paris 1976.

[CHEVAL 77] J.L. Cheval, F. Cristian, S. Krakowiak, J. Montuelle, J. Mossière : An Experiment in Modular Program Design. Congres IFIP, TORONTO, 1977

[COUSOT 78] P. Cousot : Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes. Thèse d'Etat, IMAG, 1978.

[CRISTIAN 76] F. Cristian, F. GAUDUEL : Réalisation d'un BIBLIOTHECAIRE pour le projet SESAME. Rapport de 3e année ENSIMAG, Grenoble, 1976

[CRISTIAN 77] F. Cristian : A Case Study in Modular Design. Proc. International Computing Symposium, Liège, 1977.

[CRISTIAN 79] F. Cristian : A Recovery Mechanism for Modular Software RR 146, IMAG, 1979. (aussi dans Proc. of the 4th International Conf. on Software Engineering, Munich, 1979).

[DAVIES 78] C.T. Davies : Data Processing Integrity. Advanced Course on Computing Systems Reliability, University of Newcastle upon Tyne, 1978.

[DAHL 72] O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare : Structured Programming Academic Press, 1972.

[DEC 74] Digital Equipment Corporation : BLISS-11 Programmer's Manual. Maynard, Mass., 1974.

[DE REMER 75] F. De Remer, H. Kron : Programming in the Large Versus Programming in the Small. SIGPLAN Notices, 10,6, 1975.

[DIJKSTRA 68] E.W. Dijkstra : Co-operating Sequential Processes. In Programming Languages, F. Genuys (ed), Academic Press, 1968.

[ENGLAND 74] D.M. England : Capability Concept, Mechanism and Structure in System 250. International Workshop on Protection in Operating Systems, IRIA, 1974.

[FLOYD 67] R. Floyd : Nondeterministic Algorithms. Journal of the ACM, 14,4, 1967.

[GANNON 75] J. Gannon, J.J. Horning : The Impact of Language Design on the Production of Reliable Software, SIGPLAN Notices, 10,6, 197

[GOODENOUGH 75] J.B. Goodenough : Exception Handling-Issues and a Proposed Notation. Communications of the ACM, 18,12, 1975.

[GRAY 76] J. Gray, R. LORIE, G. Putzolu, J. Traiger : Granularity of .ooks and Degrees of Consistency in a Shared Data Base. In Modeling in Data Base Management Systems. North Holland Publishing Comp., 1976.

[GRAY 78] J.N. Gray : Notes on Data Base Operating Systems. In operating Systems, an Advanced Course, Lecture Notes in Comp. Science, Springer-Verlag, Vol. 60, 1978.

[GUTTAG 75] J.V. Guttag : The Specification and Application to Programming of Abstract Data Types. Ph. D. Thesis, University of Toronto, 1975.

[GUTTAG 77] J.V. Guttag, E. Horowitz, D. Musser : Some Extentions to Algebraic Specifications. Sigplan Notices, Vol.12, n°3, 1977.

[HOARE 69] C.A.R. Hoare : An axiomatic Basis for Computer Programming. Communications of the ACM, 12,10, 1969.

[HOARE 72] C.A.R. Hoare : Proof of Correctness of Data Representations. Acta Informatica, 1,4, 1972.

[HOARE 74] C.A.R. Hoare : Monitors - An Operating System Structuring Concept. Communications of the ACM, 17,10, 1974.

[HORNING 74] J.J. Horning, H. Lauer, M. Melliar-Smith, B. Randell : A Program Structure for Error Detection and Recovery. Lecture Notes in Comp. Science, Vol.16, Springer Verlag, 1974.

[HORNING 78] J.J. Horning : Programming Languages. Advanced Course on Computing Systems Reliability, University of Newcastle upon Tyne, 1978.

[IBM 70] IBM Corporation : PL1 Language Reference Manual. IBM Corporation, 1970.

[KAISER 74] C. Kaiser, S. Krakowiak : Analyse de quelques pannes d'un système d'exploitation. Colloque International sur les Systèmes d'Exploitation, IRIA, 1974.

[KAISER 76] C. Kaiser, D. Lanciaux : Un modèle pour le traitement des erreurs dans un système à domaines. IRIA, RR 173, 1976.

[KRAKOWIAK 76] S. Krakowiak, M. Lucas, J. Montuelle, J. Mossière : A modular Approach to the Structured Design of Operating Systems. Proc. MRI Symposium on Computer Software Engineering, Polytechnic Institute, New York, 1976.

[LAMPSON 74] B. Lampson, J. Mitchell, E. Satterthwaite : On the Transfer of Control between Contexts. Lecture Notes in Comp. Science, Vol.19, Springer Verlag, 1974.

[LEVIN 77] R. Levin : Program Structures for Exceptional Condition Handling. Ph.D. Thesis, Carnegie-Mellon University, 1977.

[LISKOV 74] B.H. Liskov, S. Zilles : Programming with Abstract Data Types. SIGPLAN Notices, 9,4, 1974.

[LOMET 77] D.B. Lomet : Process Structuring, Synchronisation and Recovery using Atomic Actions. SIGPLAN Notices 12,3, 1977.

[LONDON 75] R. London, M. Shaw, W. Wulf : Abstraction and Verification in Alphard-A Symbol Table Example. RR. 76-51, University of Southern California, 1976.

[MELLIAR-SMITH 77] M. Melliar-Smith, B. Randell : Software Reliability - the Role of Programmed Exception Handling. SIGPLAN Notices 12, 3, 1977.

[MITCHELL 78] J.G. Mitchell, W. Maybury, Richard Sweet : Mesa Language Manual. XEROX, Palo Alto Research Center, CSL-78-1, 1978.

[MOALLA 79] M. Moalla : Discussion privé sur les erreurs transitoires dans les périphériques. IMAG, 1979.

[MONTUELLE 77] J. Montuelle : Conception modulaire de systèmes d'exploitation - Méthode et Exemple d'Application. Thèse de Docteur-Ingénieur, IMAG, 1977.

[MOSSIERE 77] J. Mossière : Méthode pour l'écriture des systèmes d'exploitation. Thèse d'Etat, IMAG, 1977.

[PARNAS 72] D.L. Parnas : Response to Detected Errors in Well-Structured Programs, Carnegie-Mellon University, Dept. of Comp. Science, Report, 1972.

[PARNAS 74] D.L. Parnas : On a Buzzword - Hierarchical Structure. Proc. IFIP Congress, 1974.

[PARNAS 76] D.L. Parnas, H. Wurges : Response to Undesired Events in Software Systems. Forshungsbericht BS I 77/1, T.H. Darmstadt, 1976.

[PRENER 72] C.J. Prener & all : An implementation of Backtracking for Programming Languages. ACM Annual Conference, 1972.

[RANDELL 75] B. Randell : System Structure for Fault Tolerance. SIGPLAN Notices, 10,6, 1975.

[RANDELL 78] B. Randell, P. Lee, P. Treleaven : Reliability Issues in Computing System Design. Computing Surveys, Vol.10, n°2, 1978.

[RUSSEL 75] D.L. Russel, T.H. Bredt : Error Resynchronisation in Producer Consummer Systems, Proc. of the 5th Operating System Symp, SIGOPS Review, 1975.

[SHRIVASTAVA 78] S.K. Shrivastava, J.P. Banâtre: Reliable Resource Allocation between Unreliable Processes. IEEE Trans. on Soft. Eng, Vol.SE-4, n°3, 1978.

[SHRIVASTAVA 78b] S.K. Shrivastava : Sequential Pascal with Recovery Blocks. Software Practice and Experience, Vol.8, 1978.

[SHRIVASTAVA 78c] S.K. Shrivastava : Concurrent Pascal with Backward Error Recovery. RR. n° 127, University of Newcastle upon Tyne, 1978.

[SHRIVASTAVA 78d] S.K. Shrivastava, A. Akinpelu : Fault-Tolerant Sequential Programming. Proc. of the 8th Int. Symp. on Fault Tolerant Computing, Toulouse, 1978.

[WULF 71] W.A. Wulf, D. Russel, A.N. Habermann : BLIS - a Language for System Programming. Communications of the ACM, 14, 12, 1971.

[WULF 75] W. Wulf : Reliable Hardware-Software Architecture. SIGPLAN Notices, 10,6, 1975.

[WULF 76] W. Wulf, R. London, M. Shaw : Abstraction and Verification in Alphard-Intorduction to the Methodology. Carnegie-Mellon University, Dept. of Comp. Science, Report,1976.