Numéro d'ordre: 1281

USTL

50376
1994
49

Année : **1994**

LIFL Laboratoire d'Informatique Fondamentale de Lille

THESE

présentée à

L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

Jean-François Roos

Mise au point d'applications distribuées pour environnement de développement basé sur une technologie objet

Thèse soutenue le 22 Février 1994, devant la commission d'examen

Président:	M. LATTEUX	LIFL
Rapporteurs:	B. PLATEAU	LGI-IMAG
	A. SCHIPER	LSE
Examinateurs:	S. CHAUMETTE	LABRI
	J-M. GEIB	LIFL
	J-F. MEHAUT	LIFL

UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE
U.F.R. d'I.E.E.A. Bât M3. 59655 Villeneuve d'Ascq CEDEX
Tél. 20.43.47.24 Fax. 20.43.65.66

UNIVERSITE DES SCIENCES
ET TECHNOLOGIES DE LILLE

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M. H. LEFEBVRE, M. PARREAU

PROFESSEURS HONORAIRES DES ANCIENNES FACULTES DE DROIT ET SCIENCES ECONOMIQUES, DES SCIENCES ET DES LETTRES

MM. ARNOULT, BONTE, BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, DECUYPER, DEHEUVELS, DEHORS, DION, FAUVEL, FLEURY, GERMAIN, GLACET, GONTIER, KOURGANOFF, LAMOTTE, LASSERRE, LELONG, LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, PEREZ, ROIG, ROSEAU, ROUELLE, SCHILTZ, SAVARD, ZAMANSKI, Mes BEAUJEU, LELONG.

PROFESSEUR EMERITE

M. A. LEBRUN

ANCIENS PRESIDENTS DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

MM. M. PARREAU, J. LOMBARD, M. MIGEON, J. CORTOIS, A.DUBRULLE

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

M. P. LOUIS

PROFESSEURS - CLASSE EXCEPTIONNELLE

M. CHAMLEY Hervé	Géotechnique
M. CONSTANT Eugène	Electronique
M. ESCAIG Bertrand	Physique du solide
M. FOURET René	Physique du solide
M. GABILLARD Robert	Electronique
M. LABLACHE COMBIER Alain	Chimie
M. LOMBARD Jacques	Sociologie
M. MACKE Bruno	Physique moléculaire et rayonnements atmosphériques

M. MIGEON Michel	EUDIL
M. MONTREUIL Jean	Biochimie
M. PARREAU Michel	Analyse
M. TRIDOT Gabriel	Chimie appliquée

PROFESSEURS - 1ère CLASSE

M. BACCHUS Pierre	Astronomie
M. BIAYS Pierre	Géographie
M. BILLARD Jean	Physique du Solide
M. BOILLY Bénoni	Biologie
M. BONNELLE Jean Pierre	Chimie-Physique
M. BOSCQ Denis	Probabilités
M. BOUGHON Pierre	Algèbre
M. BOURIQUET Robert	Biologie Végétale
M. BRASSELET Jean Paul	Géométrie et topologie
M. BREZINSKI Claude	Analyse numérique
M. BRIDOUX Michel	Chimie Physique
M. BRUYELLE Pierre	Géographie
M. CARREZ Christian	Informatique
M. CELET Paul	Géologie générale
M. COEURE Gérard	Analyse
M. CORDONNIER Vincent	Informatique
M. CROSNIER Yves	Electronique
Mme DACHARRY Monique	Géographie
M. DAUCHET Max	Informatique
M. DEBOURSE Jean Pierre	Gestion des entreprises
M. DEBRABANT Pierre	Géologie appliquée
M. DECLERCQ Roger	Sciences de gestion
M. DEGAUQUE Pierre	Electronique
M. DESCHEPPER Joseph	Sciences de gestion
Mme DESSAUX Odile	Spectroscopie de la réactivité chimique
M. DHAINAUT André	Biologie animale
Mme DHAINAUT Nicole	Biologie animale
M. DJAFARI Rouhani	Physique
M. DORMARD Serge	Sciences Economiques
M. DOUKHAN Jean Claude	Physique du solide
M. DUBRULLE Alain	Spectroscopie hertzienne
M. DUPOUY Jean Paul	Biologie
M. DYMENT Arthur	Mécanique
M. FOCT Jacques Jacques	Métallurgie
M. FOUQUART Yves	Optique atmosphérique
M. FOURNET Bernard	Biochimie structurale
M. FRONTIER Serge	Ecologie numérique
M. GLORIEUX Pierre	Physique moléculaire et rayonnements atmosphériques
M. GOSSELIN Gabriel	Sociologie
M. GOUDMAND Pierre	Chimie-Physique
M. GRANELLE Jean Jacques	Sciences Economiques
M. GRUSON Laurent	Algèbre
M. GUILBAULT Pierre	Physiologie animale
M. GUILLAUME Jean	Microbiologie
M. HECTOR Joseph	Géométrie
M. HENRY Jean Pierre	Génie mécanique
M. HERMAN Maurice	Physique spatiale
M. LACOSTE Louis	Biologie Végétale
M. LANGRAND Claude	Probabilités et statistiques

M. LATTEUX Michel	Informatique
M. LAVEINE Jean Pierre	Paléontologie
Mme LECLERCQ Ginette	Catalyse
M. LEHMANN Daniel	Géométrie
Mme LENOBLE Jacqueline	Physique atomique et moléculaire
M. LEROY Jean Marie	Spectrochimie
M. LHENAFF René	Géographie
M. LHOMME Jean	Chimie organique biologique
M. LOUAGE Françis	Electronique
M. LOUCHEUX Claude	Chimie-Physique
M. LUCQUIN Michel	Chimie physique
M. MAILLET Pierre	Sciences Economiques
M. MAROUF Nadir	Sociologie
M. MICHEAU Pierre	Mécanique des fluides
M. PAQUET Jacques	Géologie générale
M. PASZKOWSKI Stéfan	Mathématiques
M. PETIT Francis	Chimie organique
M. PORCHET Maurice	Biologie animale
M. POUZET Pierre	Modélisation - calcul scientifique
M. POVY Lucien	Automatique
M. PROUVOST Jean	Minéralogie
M. RACZY Ladislas	Electronique
M. RAMAN Jean Pierre	Sciences de gestion
M. SALMER Georges	Electronique
M. SCHAMPS Joël	Spectroscopie moléculaire
Mme SCHWARZBACH Yvette	Géométrie
M. SEGUIER Guy	Electrotechnique
M. SIMON Michel	Sociologie
M. SLIWA Henri	Chimie organique
M. SOMME Jean	Géographie
Melle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. THIEBAULT François	Sciences de la Terre
M. THOMAS Jean Claude	Géométrie - Topologie
M. THUMERELLE Pierre	Démographie - Géographie humaine
M. TILLIEU Jacques	Physique théorique
M. TOULOTTE Jean Marc	Automatique
M. TREANTON Jean René	Sociologie du travail
M. TURRELL Georges	Spectrochimie infrarouge et raman
M. VANEECLOO Nicolas	Sciences Economiques
M. VAST Pierre	Chimie inorganique
M. VERBERT André	Biochimie
M. VERNET Philippe	Génétique
M. VIDAL Pierre	Automatique
M. WALLART Françis	Spectrochimie infrarouge et raman
M. WEINSTEIN Olivier	Analyse économique de la recherche et développement
M. ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS - 2ème CLASSE

M. ABRAHAM Francis	
M. ALLAMANDO Etienne	Composants électroniques
M. ANDRIES Jean Claude	Biologie des organismes
M. ANTOINE Philippe	Analyse
M. BALL Steven	Génétique
M. BART André	Biologie animale
M. BASSERY Louis	Génie des procédés et réactions chimiques
Mme BATTIAU Yvonne	Géographie
M. BAUSIERE Robert	Systèmes électroniques
M. BEGUIN Paul	Mécanique
M. BELLET Jean	Physique atomique et moléculaire
M. BERNAGE Pascal	Physique atomique, moléculaire et du rayonnement
M. BERTHOUD Arnaud	Sciences Economiques
M. BERTRAND Hugues	Sciences Economiques
M. BERZIN Robert	Analyse
M. BISKUPSKI Gérard	Physique de l'état condensé et cristallographie
M. BKOUCHE Rudolphe	Algèbre
M. BODARD Marcel	Biologie végétale
M. BOHIN Jean Pierre	Biochimie métabolique et cellulaire
M. BOIS Pierre	Mécanique
M. BOISSIER Daniel	Génie civil
M. BOIVIN Jean Claude	Spectrochimie
M. BOUCHER Daniel	Physique
M. BOUQUELET Stéphane	Biologie appliquée aux enzymes
M. BOUQUIN Henri	Gestion
M. BROCARD Jacques	Chimie
Mme BROUSMICHE Claudine	Paléontologie
M. BUISINE Daniel	Mécanique
M. CAPURON Alfred	Biologie animale
M. CARRE François	Géographie humaine
M. CATTEAU Jean Pierre	Chimie organique
M. CAYATTE Jean Louis	Sciences Economiques
M. CHAPOTON Alain	Electronique
M. CHARET Pierre	Biochimie structurale
M. CHIVE Maurice	Composants électroniques optiques
M. COMYN Gérard	Informatique théorique
Mme CONSTANT Monique	Composants électroniques et optiques
M. COQUERY Jean Marie	Psychophysiologie
M. CORIAT Benjamin	Sciences Economiques
Mme CORSIN Paule	Paléontologie
M. CORTOIS Jean	Physique nucléaire et corpusculaire
M. COUTURIER Daniel	Chimie organique
M. CRAMPON Norbert	Tectolique géodynamique
M. CURGY Jean Jacques	Biologie
M. DANGOISSE Didier	Physique théorique
M. DE PARIS Jean Claude	Analyse
M. DECOSTER Didier	Composants électroniques et optiques
M. DEJAEGER Roger	Electrochimie et Cinétique
M. DELAHAYE Jean Paul	Informatique
M. DELORME Pierre	Physiologie animale
M. DELORME Robert	Sciences Economiques
M. DEMUNTER Paul	Sociologie
Mme DEMUYNCK Claire	Physique atomique, moléculaire et du rayonnement
M. DENEL Jacques	Informatique
M. DEPREZ Gilbert	Physique du solide - christallographie

M. DERIEUX Jean Claude	Microbiologie
M. DERYCKE Alain	Informatique
M. DESCAMPS Marc	Physique de l'état condensé et cristallographie
M. DEVRAINNE Pierre	Chimie minérale
M. DEWAILLY Jean Michel	Géographie humaine
M. DHAMELINCOURT Paul	Chimie physique
M. DI PERSIO Jean	Physique de l'état condensé et cristallographie
M. DUBAR Claude	Sociologie démographique
M. DUBOIS Henri	Spectroscopie hertzienne
M. DUBOIS Jean Jacques	Géographie
M. DUBUS Jean Paul	Spectrométrie des solides
M. DUPONT Christophe	Vie de la firme
M. DUTHOIT Bruno	Génie civil
Mme DUVAL Anne	Algèbre
Mme EVRARD Micheline	Génie des procédés et réactions chimiques
M. FAKIR Sabah	Algèbre
M. FARVACQUE Jean Louis	Physique de l'état condensé et cristallographie
M. FAUQUEMBERGUE Renaud	Composants électroniques
M. FELIX Yves	Mathématiques
M. FERRIERE Jacky	Tectonique - Géodynamique
M. FISCHER Jean Claude	Chimie organique, minérale et analytique
M. FONTAINE Hubert	Dynamique des cristaux
M. FORSE Michel	Sociologie
M. GADREY Jean	Sciences économiques
M. GAMBLIN André	Géographie urbaine, industrielle et démographie
M. GOBLOT Rémi	Algèbre
M. GOURIEROUX Christian	Probabilités et statistiques
M. GREGORY Pierre	I.A.E.
M. GREMY Jean Paul	Sociologie
M. GREVET Patrice	Sciences Economiques
M. GRIMBLOT Jean	Chimie organique
M. GUELTON Michel	Chimie physique
M. GUICHAOUA André	Sociologie
M. HAIMAN Georges	Modélisation,calcul scientifique, statistiques
M. HOUDART René	Physique atomique
M. HUEBSCHMANN Johannes	Mathématiques
M. HUTTNER Marc	Algèbre
M. ISAERT Noël	Physique de l'état condensé et cristallographie
M. JACOB Gérard	Informatique
M. JACOB Pierre	Probabilités et statistiques
M. JEAN Raymond	Biologie des populations végétales
M. JOFFRE Patrick	Vie de la firme
M. JOURNEL Gérard	Spectroscopie hertzienne
M. KOENIG Gérard	Sciences de gestion
M. KOSTRUBIEC Benjamin	Géographie
M. KREMBEL Jean	Biochimie
Mme KRIFA Hadjila	Sciences Economiques
M. LANGEVIN Michel	Algèbre
M. LASSALLE Bernard	Embryologie et biologie de la différenciation
M. LE MEHAUTE Alain	Modélisation,calcul scientifique,statistiques
M. LEBFEVRE Yannic	Physique atomique,moléculaire et du rayonnement
M. LECLERCQ Lucien	Chimie physique
M. LEFEBVRE Jacques	Physique
M. LEFEBVRE Marc	Composants électroniques et optiques
M. LEFEVRE Christian	Pétrologie
Melle LEGRAND Denise	Algèbre
M. LEGRAND Michel	Astronomie - Météorologie
M. LEGRAND Pierre	Chimie
Mme LEGRAND Solange	Algèbre
Mme LEHMANN Josiane	Analyse
M. LEMAIRE Jean	Spectroscopie hertzienne

M. LE MAROIS Henri	Vie de la firme
M. LEMOINE Yves	Biologie et physiologie végétales
M. LESCURE François	Algèbre
M. LESENNE Jacques	Systèmes électroniques
M. LOCQUENEUX Robert	Physique théorique
Mme LOPES Maria	Mathématiques
M. LOSFELD Joseph	Informatique
M. LOUAGE Francis	Electronique
M. MAHIEU François	Sciences économiques
M. MAHIEU Jean Marie	Optique - Physique atomique
M. MAIZIERES Christian	Automatique
M. MANSY Jean Louis	Géologie
M. MAURISSON Patrick	Sciences Economiques
M. MERIAUX Michel	EUDIL
M. MERLIN Jean Claude	Chimie
M. MESMACQUE Gérard	Génie mécanique
M. MESSELYN Jean	Physique atomique et moléculaire
M. MOCHE Raymond	Modélisation,calcul scientifique,statistiques
M. MONTEL Marc	Physique du solide
M. MORCELLET Michel	Chimie organique
M. MORE Marcel	Physique de l'état condensé et cristallographie
M. MORTREUX André	Chimie organique
Mme MOUNIER Yvonne	Physiologie des structures contractiles
M. NIAY Pierre	Physique atomique,moléculaire et du rayonnement
M. NICOLE Jacques	Spectrochimie
M. NOTELET Francis	Systèmes électroniques
M. PALAVIT Gérard	Génie chimique
M. PARSY Fernand	Mécanique
M. PECQUE Marcel	Chimie organique
M. PERROT Pierre	Chimie appliquée
M. PERTUZON Emile	Physiologie animale
M. PETIT Daniel	Biologie des populations et écosystèmes
M. PLIHON Dominique	Sciences Economiques
M. PONSOLLE Louis	Chimie physique
M. POSTAIRE Jack	Informatique industrielle
M. RAMBOUR Serge	Biologie
M. RENARD Jean Pierre	Géographie humaine
M. RENARD Philippe	Sciences de gestion
M. RICHARD Alain	Biologie animale
M. RIETSCH François	Physique des polymères
M. ROBINET Jean Claude	EUDIL
M. ROGALSKI Marc	Analyse
M. ROLLAND Paul	Composants électroniques et optiques
M. ROLLET Philippe	Sciences Economiques
Mme ROUSSEL Isabelle	Géographie physique
M. ROUSSIGNOL Michel	Modélisation,calcul scientifique,statistiques
M. ROY Jean Claude	Psychophysiologie
M. SALERNO Françis	Sciences de gestion
M. SANCHOLLE Michel	Biologie et physiologie végétales
Mme SANDIG Anna Margarette	
M. SAWERYSYN Jean Pierre	Chimie physique
M. STAROSWIECKI Marcel	Informatique
M. STEEN Jean Pierre	Informatique
Mme STELLMACHER Irène	Astronomie - Météorologie
M. STERBOUL François	Informatique
M. TAILLIEZ Roger	Génie alimentaire
M. TANRE Daniel	Géométrie - Topologie
M. THERY Pierre	Systèmes électroniques
Mme TJOTTA Jacqueline	Mathématiques
M. TOURSEL Bernard	Informatique
M. TREANTON Jean René	Sociologie du travail

M. TURREL Georges	Spectrochimie infrarouge et raman
M. VANDIJK Hendrik	
Mme VAN ISEGHEM Jeanine	Modélisation,calcul scientifique,statistiques
M. VANDORPE Bernard	Chimie minérale
M. VASSEUR Christian	Automatique
M. VASSEUR Jacques	Biologie
Mme VIANO Marie Claude	
M. WACRENIER Jean Marie	Electronique
M. WARTEL Michel	Chimie inorganique
M. WATERLOT Michel	géologie générale
M. WEICHERT Dieter	Génie mécanique
M. WERNER Georges	Informatique théorique
M. WIGNACOURT Jean Pierre	
M. WOZNIAK Michel	Spectrochimie
Mme ZINN JUSTIN Nicole	Algèbre

Remerciements

Je remercie les membres du jury :

- Monsieur **Michel LATTEUX**, professeur à l'Université de Lille I pour m'avoir fait l'honneur de présider ce jury ;
- Madame **Brigitte PLATEAU**, professeur à l'ENSIMAG de grenoble, d'avoir accepté de rapporter cette thèse. Ses remarques m'ont permis d'améliorer la rédaction du document ;
- Monsieur **André SCHIPER**, professeur à l'EPFL de Lausanne, d'avoir bien voulu être rapporteur de ce travail ;
- Monsieur **Serge CHAUMETTE**, maître de conférences à l'Université de Bordeaux I, pour son examen attentif et ses remarques précieuses ;
- Monsieur **Jean-Marc GEIB**, professeur à l'Université de Lille I, qui a, par ses lectures critiques et répétées, participé grandement à la qualité de ce travail ; les discussions que nous avons eues tout au long de cette thèse ont apporté de nouvelles idées et ses nombreux conseils ont permis de mener à bien ce travail ;
- Monsieur **Jean-François Méhaut**, maître de conférences à l'Université de Lille I, d'avoir accepté de diriger mes recherches ; il m'a suivi et conseillé, durant ces quatre années, tant sur le plan de la conception que celui de la réalisation ; son amitié m'a également été précieuse.

Je tiens également à remercier **Luc COURTRAI**, maître de conférences à l'université de Rennes I, qui a passé sa thèse l'année dernière dans notre équipe. Nous avons travaillé ensemble pendant trois ans dans un réel climat d'amitié.

Je remercie les membres de l'équipe, **Cédric DUMOULIN**, **Christophe GRANSART**, **Chrystel GRENOT**, **Fred HEMERY**, **Philippe MERLE** et **Raymond NAMYST**, et tous ceux, thésards et enseignants (même depuis Bordeaux), qui font du LIFL un laboratoire où il est agréable de travailler par l'ambiance générale de sympathie et de chaleur qui y règne.

Table des matières

Introduction

Environnement

Les travaux de cette thèse ont été réalisés au sein de l'équipe PVC-BOX. Eric Delattre et Jean-Marc Geib avaient lancé en 1989 PVC-BOX, projet qui avait pour objectif la conception et la réalisation d'un environnement de développement à objets pour architectures fortement distribuées. Dans le cadre de cet environnement, Eric Delattre m'avait proposé d'étudier la phase de mise au point (déverminage) des programmes, tâche particulièrement difficile dans un contexte distribué.

Le projet PVC-BOX comprend deux parties:

- PVC, un support système pour le développement d'applications distribuées. Luc Courtrai [Cou92] a proposé et implanté lors de sa thèse la notion de CAC (Composant Actif de Communication) comme outil de structuration et de programmation. Luc Courtrai a également montré que le modèle CAC est adapté pour supporter l'exécution de programmes à objets parallèles.

- BOX, un langage de programmation à objets. La principale caractéristique de ce langage est qu'il permet d'aborder naturellement le paradigme objet et celui des acteurs communicants. Les CAC fournissent un support adapté pour la gestion distribuée des objets et des acteurs communicants de BOX. Une première version du compilateur est disponible.

Deux autres thèses sur le projet PVC-BOX vont être soutenues prochainement :

- Christophe Gransart, « Fragmentation d'objet pour machines fortement distribuées », sur la définition et conception du langage BOX ;

- Fred Hémery, « Définition et réalisation d'un outil de répartition d'objets pour multi-ordinateur », sur les problèmes de répartition dynamique de charge dans notre environnement.

Les travaux de l'équipe et cette thèse sont financés et s'inscrivent dans le cadre de GANYMEDE, projet régional de recherche inter-laboratoire qui a pour thème la communication avancée.

Thème de la thèse

La programmation parallèle et distribuée est unanimement reconnue comme difficile. Notre équipe défend la position affirmant que la technologie objet facilitera la programmation et l'exploitation des architectures distribuées. Cependant, même dans un contexte objet, les erreurs de programmation restent encore possibles et nous proposons donc de définir des solutions pour la mise au point d'applications distribuées basées sur des objets.

La programmation parallèle et distribuée

Les besoins des utilisateurs en terme de puissance de calcul sont de plus en plus importants. Pour faire face à ces besoins, des progrès sensibles ont été réalisés au niveau des microprocesseurs. Les récentes évolutions concernent essentiellement les processeurs RISC, par exemple le degré d'intégration des composants, l'accroissement des fréquences d'horloge ou bien encore l'exécution pipe-linée des instructions.

Ces progrès restent malheureusement insuffisants pour satisfaire complètement les besoins des utilisateurs, d'où l'intérêt sans cesse croissant que suscitent les architectures parallèles et distribuées. On peut distinguer deux classes de machines parallèles:

- les machines synchrones (SIMD) où les proceseurs sont synchronisés par une unité de contrôle. Ces architectures n'entrent pas dans le cadre de nos investigations.

- les machines asynchrones (MIMD) qui nous concernent directement. Nos prototypes ont été réalisés sur un multi-ordinateur Parsytec (Machine à base de Transputer T800), et aussi sur un réseau de stations de travail SUN4.

La programmation séquentielle ne permet pas d'exploiter efficacement ces nouvelles architectures, d'où un nouveau mode de programmation appelée programmation parallèle. Deux styles de programmation peuvent être utilisées :

- la programmation parallèle synchrone ou bien la programmation « data parallel » qui permet de programmer efficacement les architectures synchrones. Ce mode de programmation n'a pas été étudié dans le cadre de cette thèse.

- la programmation asynchrone basée sur le concept de processus communicant telle qu'elle a été proposé initialement par Hoare [Hoa78]. Cette programmation introduit de nouvelles structures de programmation qui vont permettre la création d'activités pour exploiter le parallélisme physique des architectures. Au niveau de la programmation sont également définies des structures de communication pour permettre la communication entre les activités. Ces communications sont souvent basées sur l'envoi de message ou bien encore sur le rendez-vous.

La difficulté de la programmation parallèle est due au niveau de complexité pendant la phase de conception. Contrairement à la programmation séquentielle où le programmeur ne conçoit qu'une seule activité (flux d'instruction), dans un contexte parallèle, il doit concevoir plusieurs activités qui vont s'exécuter concurremment. De plus, il doit mettre en œuvre tous les aspects coopération et communication entre ces différentes activités. Les erreurs pourront être de différentes origines, soit des erreurs classiques de programmation à l'intérieur d'activités, soit, plus souvent, sur les aspects coopération inter-activités. Les nouveaux problèmes peuvent être alors des interblocages, des attentes infinies ou bien encore des erreurs dans le contenu des messages...

La programmation parallèle à objets

La technologie objet apporte à la programmation parallèle une modélisation naturelle. L'objectif est de faire bénéficier la programmation parallèle des qualités du modèle objet. La programmation objet apporte une prise en compte de l'aspect données quasiment absente de la programmation parallèle où le programmeur se concentrait essentiellement sur les activités. Le programmeur va pouvoir décrire son programme en terme d'objets (données) et d'activités. La création d'activités pourra provenir, suivant les langages, de la création d'objets, de l'exécution de méthodes ou bien encore de la création d'objets processus. La synchronisation se fait généralement au niveau des méthodes, soit sur l'appel, soit sur l'exécution.

Le premier chapitre de la thèse de Luc Courtrai [Cou92] fait une synthèse très complète des langages parallèles à objets (LPO). BOX est le langage parallèle à objets qui a été conçu dans le cadre de notre équipe; une première présentation est faite dans le chapitre 3, une description et une justification plus complète seront très prochainement disponibles dans la thèse de Christophe Gransart [Gra94].

La mise au point de programmes séquentiels

Le problème de la mise au point de programmes a toujours suscité un grand intérêt. La mise au point peut se faire à plusieurs moments :

Avant l'exécution : les langages évolués doivent généralement être compilés avant l'exécution. Le contrôle de type à la compilation prévient le programmeur des erreurs et des incohérences, et d'une certaine façon, contribue à la mise au point. D'autre part, l'analyse statique du source du programme permet aussi de prévoir des erreurs qui pourraient se produire pendant l'exécution. Un des exemples les plus connus est l'outil *lint* du système UNIX. *lint* effectue un contrôle plus strict que celui des compilateurs C, au niveau des types, des paramètres de fonction, de la non utilisation de variables déclarées...

Pendant l'exécution : l'environnement de développement des systèmes fournit des outils qu'on appelle des dévermineurs (débogueurs) qui vont permettre au programmeur d'interagir avec l'exécution des programmes. Les premiers dévermineurs étaient trop rustiques et de bas-niveau. Le programmeur pouvait visualiser le contenu des registres du processeur, la mémoire en spécifiant des adresses physiques ou virtuelles. Ces dévermineurs sont particulièrement adaptés pour les programmes en assembleur (exemple adb). Avec le succès de la programmation

en langage évolué est apparu le besoin de dévermineurs symboliques où le programmeur interagit directement avec son programme en cours d'exécution et son programme source. Les principales fonctionnalités des dévermineurs sont :

- la visualisation des valeurs de variables ; le programmeur peut consulter le contenu d'une variable en donnant son nom au dévermineur ; il contrôle ainsi la validité du contenu des variables pour déterminer l'origine des erreurs ;
- l'arrêt de l'exécution ; le programmeur a souvent besoin d'arrêter le programme, par exemple après avoir exécuté une procédure ; il peut ainsi ensuite contrôler la validité de l'état de l'application ;
- l'exécution du programme en pas à pas ; ceci permet au programmeur d'analyser finement l'exécution de chaque instruction et de se rendre compte de leurs effets sur l'état de l'application.

Après l'exécution (Postmortem) : certains dévermineurs permettent au programmeur de consulter l'état du programme après l'arrêt. C'est particulièrement intéressant dans le cas des programmes qui se terminent anormalement ; le programmeur arrive alors à connaître l'endroit exact où l'exécution s'est arrêtée et aussi à visualiser l'état à cet instant (visualiser l'état des variables, l'état de la pile et les contextes des procédures).

Dans le cadre de la programmation séquentielle, il faut aussi ajouter que l'exécution d'un programme est, la plupart du temps, complétement déterministe, sauf en cas d'utilisation de la valeur de l'horloge ou d'un générateur de nombres aléatoire. Deux exécutions d'un même programme, avec les mêmes données en entrée, produiront toujours le même résultat. Soit le programme est correct et répond au problème, alors les résultats des exécutions pour cette donnée seront toujours bons. Soit le programme est faux et ne répond donc pas au problème, alors les résultats des exécutions pour ces données seront toujours mauvais.

La mise au point de programmes parallèles

Dans un contexte parallèle, à cause du non-déterminisme, deux exécutions d'un même programme pour une même donnée peuvent produire deux résultats différents. Soit le programme parallèle est correct, alors le résultat des exécutions sera toujours bon. Soit le programme parallèle est faux, plusieurs exécutions d'un programme faux avec la même donnée peuvent donner des résultats différents; certains résultats peuvent être bons, d'autres mauvais. A la seule vue d'un résultat d'une exécution de programme parallèle, on ne peut pas décider si le programme donnera toujours le bon résultat.

Le problème du non-déterminisme dans l'exécution de programme parallèle provient du fait que l'ordre des événements peut différer d'une exécution à l'autre. Les programmes faux peuvent produire des résultats différents si deux événements se produisent dans un ordre différent. Les causes du non-déterminisme sont liées à plusieurs problèmes : le non-déterminisme dans les langages de programmation parallèle, la charge du réseau de communication, la charge de calcul des processeurs.

Le déverminage de programmes parallèles peut se faire, avant, pendant ou après l'exécution :

Avant l'exécution : Par analyse statique du source des programmes, il est possible de

détecter certains problèmes qui pourraient se produire pendant l'exécution, par exemple certains interblocages. On peut aussi citer les recherches plus théoriques qui se font dans le domaine de la validation et de la preuve de programmes parallèles. Le déverminage avant exécution n'a pas été abordé au cours de cette thèse.

Pendant l'exécution : Le déverminage risque alors de perturber le comportement du programme parallèle. Il semble naturel d'imposer que les exécutions d'un programme avec et sans déverminage produisent le même résultat, avec la même séquence d'événements. Cette exigence rend difficile ce type de déverminage.

D'autre part, les fonctionnalités des dévermineurs de programmes séquentiels sont difficilement réalisables avec des programmes parallèles s'exécutant sur des machines parallèles :

- Visualiser les variables d'un programme parallèle. La visualisation est plus délicate dans un cadre où la mémoire est distribuée.
- Arrêter l'exécution d'un programme. Beaucoup de questions se posent pour arrêter un programme parallèle : l'arrêt doit-il être complet sur toutes les activités du programme? ou alors, l'arrêt peut-il être partiel et ne concerner qu'une ou plusieurs activités? Généralement, l'arrêt des programmes est complet et concerne toutes les activités. Le problème est alors de stopper, simultanément ou presque, toutes les activités du programme.
- Exécuter un programme en pas à pas. Que pourrait signifier une exécution en pas à pas sur un programme parallèle? Au niveau d'une activité, au niveau de l'ensemble des activités du programme?

Beaucoup de questions sont posées. Il est difficile d'y répondre et de proposer des réalisations. Dans le cadre de notre projet, nous nous sommes résolument orientés vers le déverminage de programmes après une exécution de référence.

Après l'exécution : Beaucoup d'outils de déverminages sont basés sur le principe de la génération de traces pendant l'exécution de programmes. La perturbation du comportement de l'application pendant la génération de traces doit rester faible, voire nulle. Les traces permettent ensuite, suivant les environnements, soit d'analyser l'exécution et d'essayer d'en détecter les anomalies, soit de réexécuter les programmes.

Dans le cadre de cette thèse, nous avons retenu le principe de la réexécution ; pendant une première exécution, nous enregistrons les événements qui se sont produits dans une trace. La réexécution consiste en une exécution dirigée par la trace : les événements doivent se produire dans le même ordre que celui observé pendant la première exécution.

Pendant la réexécution, nous proposons de donner au programmeur les mêmes possibilités que celles existant pour le déverminage pendant l'exécution. Le programmeur pourra arrêter la réexécution et consulter l'état du programme, des activités et des variables. Dans un contexte de réexécution, le système doit garantir que les événements se produiront toujours dans le même ordre.

Plan de la thèse

Cette thèse comprend trois chapitres:

- Le premier chapitre est consacré à une étude bibliographique de la mise au point d'applications parallèles et distribuées. Nous avons particulièrement étudié les systèmes avec analyse post-mortem et les techniques de réexécution de programmes parallèles.

- Dans le second chapitre, nous présenterons d'abord le modèle CAC qui est le support système de notre environnement. L'interface de programmation est décrite dans l'annexe A. Nous avons défini et réalisé deux mécanismes de réexécution: une réexéexécution totale de l'application à partir d'une trace réduite, et une réexécution partielle à partir d'une trace complète générée pendant une première réexécution totale.

- Le dernier chapitre est consacré au déverminage d'applications développées en BOX. Dans un premier temps, nous décrirons les caractéristiques générales du langage en les illustrant par un exemple. Nous préciserons ensuite les particularités de la réexécution des programmes BOX. Nous définirons la notion d'événement dans l'exécution d'un programme BOX. Ces événements pourront être associés à des points d'arrêt. Nous présenterons enfin les mécanismes distribués de détection d'événements et le mécanisme d'arrêt de l'application.

Chapitre 1

Mise au point et parallélisme

LA MISE AU POINT est l'une des étapes les plus délicates dans la conception d'une application parallèle. Nous expliquons dans cette introduction en quoi le parallélisme complique la tâche du programmeur. Le déverminage est défini comme la localisation, l'analyse et la correction des erreurs. On appelle erreur un état où le programme n'effectue pas la fonction désirée. L'erreur est rarement située à l'endroit de sa manifestation. Le programmeur doit acquérir des informations complémentaires à l'aide de traces ou de points d'arrêt dans des exécutions successives du programme. Il localise ainsi l'erreur petit à petit. Cette technique est appelée déverminage cyclique. Cette approche est utilisable uniquement si le comportement du programme est le même durant les exécutions successives.

Pour les programmes séquentiels, des outils efficaces utilisant le déverminage cyclique existent, comme dbxtool [Sun86] sur stations SUN. Mais les mêmes techniques ne peuvent être appliquées pour les programmes parallèles. Ils peuvent avoir un comportement non déterministe : deux exécutions successives, avec des données identiques, peuvent produire des résultats différents éventuellement suivant des chemins différents. Le non déterminisme peut avoir plusieurs causes :

- des constructions du langage de programmation comme le SELECT de ADA [D.o83] ;
- le support d'exécution : les communications entre processus par variables partagées ou par messages ;
- le programmeur par l'appel à des fonctions générant des nombres aléatoires ou récupérant la valeur de l'horloge ;
- la charge de la machine au moment de l'exécution.

Le fait même de déverminer le programme produit des perturbations sur les relations temporelles, pouvant empêcher l'occurrence de l'erreur précédemment détectée. Les

solutions consistant à associer un déverminer séquentiel à chaque processus ne sont donc pas acceptables.

Un autre problème concerne les architectures sans mémoire commune: l'état du système est physiquement distribué et consiste en l'ensemble des états de chaque site et l'état des liens de communications. Il est difficile, voire impossible, de déterminer l'ordre précis dans lequel surviennent des événements sur des processeurs distincts. Cela rend les erreurs beaucoup plus complexes à comprendre et à localiser.

Nous aborderons ici les différentes méthodes de déverminage utilisées pour répondre aux problèmes spécifiques posés par le parallélisme. L'analyse statique est une technique permettant de déterminer la structure d'un programme sans exécution en examinant le texte source ou le code objet. Elle permet essentiellement de détecter certaines des erreurs potentielles d'interblocage, de famine et les erreurs d'accès aux données. Cette solution ne sera pas abordée ici. Pour plus d'information sur l'analyse statique, on peut se référer à l'article de McDowell et Helmbold [MH89]. Nous nous intéresserons au déverminage dynamique, pour lequel trois méthodes sont principalement proposées. Nous utiliserons la terminologie employée par E. Leu et A. Schiper dans leur synthèse [LS91] sur le déverminage des programmes parallèles.

La première méthode est basée sur l'observation du comportement du programme: des informations sont récoltées lors de l'exécution. Celles ci sont présentées à l'utilisateur par animation, statistiques ou autres. La détection des erreurs se fait donc visuellement, à posteriori et sans aucune interaction avec le programme. Les deux autres méthodes permettent à l'utilisateur d'interagir directement avec son programme lors de l'exécution ou indirectement lors d'une réexécution « identique » à l'exécution ayant produit l'erreur.

La mise en œuvre du déverminage peut être effectuée de plusieurs façons:

- par récolte d'événements pour les trois méthodes (observation, interaction directe, interaction indirecte);
- par la sauvegarde régulière de l'état global du programme sur disque, produisant ce que l'on appelle des instantanés d'exécution pour l'interaction indirecte;
- par l'intégration de tâches de déverminage au programme pour l'interaction directe: modification du code ou processus lié dynamiquement au programme, technique conduisant le plus souvent à utiliser des points d'arrêt.

Nous présenterons d'abord dans ce chapitre la notion d'événement, principalement utilisée pour les trois méthodes, et les moyens de les ordonner. Les différentes méthodes seront ensuite décrites, en précisant leurs qualités et leurs limites.

1.1 Les événements

Les techniques présentées ici sont essentiellement basées sur la récolte d'événements. Un événement peut être défini comme un comportement atomique caractéristique de l'exécution d'un programme. Nous verrons d'abord les différents types d'événements, les moyens de les ordonner et aussi les techniques de détection et d'enregistrement.

1.1.1 Types d'événements

On peut distinguer plusieurs types d'événements.

Evénements de base

Ils correspondent souvent à des événements « système » concernant par exemple la gestion de processus (création ou destruction de processus), ou la communication inter-processus (envoi et réception de messages, accès à des variables partagées). L'ensemble des événements de base doit être représentatif du système car il définit son niveau d'observation le plus élémentaire. Ils sont générés indépendamment de l'application par appel au noyau du système.

L'utilisateur peut définir ses propres événements de base mais doit placer lui même leur détection dans le programme. Il peut ainsi utiliser des événements spécifiques à son application. Par exemple, si le programmeur utilise des fichiers, il pourra détecter lui même des événements comme l'ouverture, la lecture ou la fermeture d'un fichier.

Evénements composés

Dans certains systèmes, le programmeur a la possibilité de composer des événements de base entre eux, par l'intermédiaire d'opérateurs permettant d'exprimer le parallélisme et la séquence, et de définir ainsi des événements de plus haut niveau d'abstraction se rapprochant de celui de l'application et du niveau conceptuel. Pour P. BATES, l'occurrence d'un événement composé dépend également de l'information véhiculée par les événements primitifs. Il définit ces événements grâce à EDL [BW83] (Event Description Language) (cf. figure 1.1).

```
event BlockingSend is
    primitive "BlockingSend"
  with
    sender_id   : integer
    receiver_id : integer
    send_time   : time
end

event Answer is
    primitive "Answer"
  with
    sender_id   : integer
    receiver_id : integer
    send_time   : time
end

    event Communication is
        BlockingSend 0 Answer
      cond
        BlockingSend.sender_id  == Answer.receiver_id;
        BlockingSend.receiver_id == Answer.sender_id;
      with
        Delay := Answer.send_time-BlockingSend.send_time;
    fin.
```

FIG. 1.1 - *Exemple de définition d'événements composés avec EDL*

L'exemple définit trois événements dont deux événements de base, *BlockingSend*, correspondant à un envoi de message bloquant, et *Answer* correspondant à l'envoi d'une

réponse. Ces événements possèdent des attributs définis après la clause *with*. L'événement *Communication* est un événement composé à partir des deux événements de base. L'opérateur *0* permet d'indiquer que *BlockingSend* doit précéder *Answer* (d'autres opérateurs sont disponibles : le ou, la concurrence ou l'itération). La clause *cond* permet de filtrer les événements intéressants avec des contraintes sur les attributs des événements constitutifs. Ici on s'assure qu'il s'agit d'une communication et donc que la réponse correspond bien à l'envoi.

1.1.2 Datation des événements

Dans les architectures distribuées, la notion d'horloge globale n'existe pas toujours. Chaque site possède son horloge non synchronisée avec celle des autres sites. Il est donc difficile de dégager un ordre complet sur les événements. L. LAMPORT [Lam78] a défini la relation « précède » sur les événements. Elle peut être vue comme une relation de causalité. La relation, notée $<$, est définie de la manière suivante :

$a<b$ si et seulement si l'une des trois conditions suivantes est vérifiée :

- a et b sont deux événements du même processus et a survient avant b ;
- ou a est l'émission d'un message par un processus et b la réception du même message par un autre processus ;
- ou il existe un événement e tel que $a<e$ et $e<b$.

Deux événements distincts a et b sont dits concurrents si $\neg(a<b)$ et $\neg(b<a)$. La relation est un ordre partiel.

Nous allons maintenant détailler plusieurs définitions et utilisations d'horloge permettant d'utiliser la relation « précède » sur les événements.

L'horloge de Lamport

Une horloge logique sera en fait le moyen de numéroter un événement. Ce numéro sera considéré comme la date d'occurrence de l'événement. A chaque processus est assignée une horloge logique qui peut être implémentée comme un simple compteur. Le mécanisme de datation fonctionne de la manière suivante :

- chaque processus incrémente son horloge entre deux événements successifs ;
- si un événement est l'envoi d'un message par un processus, alors le message est estampillé par l'horloge du processus au moment de l'envoi ;
- à la réception d'un message, le processus met à jour son horloge de telle sorte qu'elle soit plus grande ou égale à sa valeur actuelle et plus grande que l'estampille du message.

L'exemple de la figure 1.2 présente l'utilisation des horloges de Lamport avec deux processus A et B. Le processus B a recalé son horloge à la réception du message, événement b2, avec la valeur de l'horloge de A qui estampillait le message.

Si h est la relation qui associe à un événement sa date et a, b deux événements, on peut établir la relation suivante : $a<b \Rightarrow \mathrm{h}(a)<\mathrm{h}(b)$. La relation inverse n'est pas

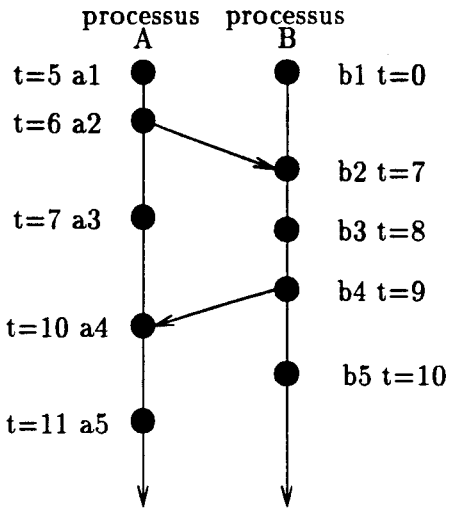

FIG. 1.2 - *Utilisation des horloges de Lamport*

vérifiée. En effet, cela voudrait dire que deux événements concurrents ont toujours exactement la même horloge. Si on reprend l'exemple de la figure 1.2, les événements *a3* et *b3* sont concurrents et pourtant h(*a3*)<h(*b3*). L'horloge de LAMPORT définit en fait un ordre total sur les événements en ordonnant également les processus par un numéro. La connaissance seule de la date de deux événements obtenue avec l'horloge de LAMPORT, pour des processus distincts, ne permet donc pas de les ordonner avec la relation « précède ». Il faut connaître des informations supplémentaires comme le type des événements, le processus qui les a générés ...

Les horloges vectorielles

Pour pouvoir comparer deux événements en connaissant uniquement leur date, C. FIDGE et MATTERN [Fid89, MvdW89] ont défini une horloge comme un vecteur d'entiers, une composante du vecteur pour chaque processus. Le vecteur dans son ensemble représente la date courante du processus. La mise à l'heure s'effectue de la même manière que précédemment, avec un changement pour la réception de message :

- chaque processus P_i incrémente la i^e composante de son vecteur horloge entre deux événements successifs ;
- si un événement est l'envoi d'un message m par un processus P, alors le message est estampillé par T_P, l'horloge de P (le vecteur complet) après incrémentation de l'horloge ;
- à la réception d'un message m envoyé par le processus P_i et reçu par P_j, le processus P_j met à jour son horloge : en incrémentant la j^e composante, puis en conservant la plus grande valeur pour chaque composante entre celle de l'estampille et celle de l'horloge.

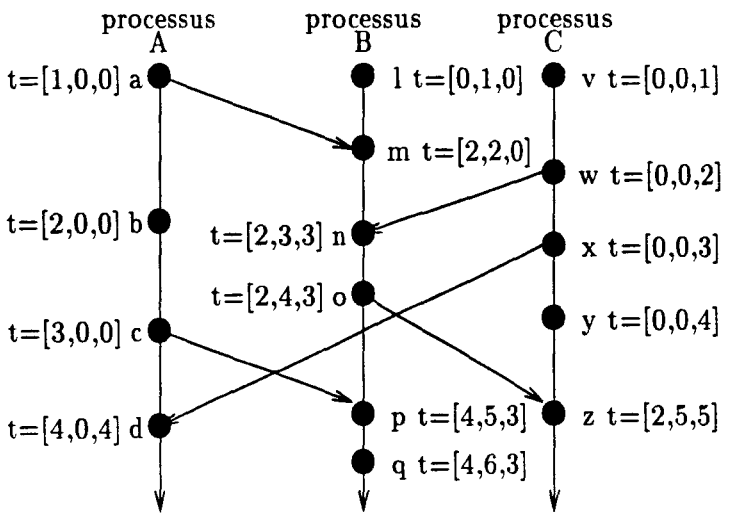

FIG. 1.3 - *Utilisation des horloges de Mattern/Fidge*

Pour comparer deux événements, on procède de la façon suivante. Soit a un événement du processus P_i et b un événement du processus P_j. Soient T_a et T_b leur date respective. L'événement a précède l'événement b si et seulement si la i^e composante de T_a est inférieure à la i^e composante de T_b. La figure 1.3 présente un exemple de l'utilisation des horloges de MATTERN/FIDGE : $a<z$ car $1<2$, $v<o$ car $1<3$, $a<c$ car $1<3$, et $\neg(b<o)$ car $\neg(2<2)$, $\neg(z<a)$ car $\neg(5<0)$. Une variante des horloges de MATTERN/FIDGE a été définie par G.J.W. VAN DIJK et A.J. VAN DER WAL [vDvdW91] pour la communication par rendez-vous.

Les horloges de LAMPORT ordonnent totalement les événements grâce à l'estampillage des messages entre processus. L'information date seule ne permet donc pas d'ordonner les événements selon la relation « précède ». Les horloges vectorielles définissent un ordre partiel et permettent de résoudre ce problème. Mais elles sont plutôt lourdes à gérer :

- la taille de l'horloge devient non négligeable si l'application comporte un grand nombre de processus ;
- la gestion de l'horloge devient problématique en cas de création dynamique de processus.

1.1.3 Collecte d'événements

La récolte d'événements a pour but de constituer une trace qui soit une collection des événements générés pendant l'exécution du programme. Mais cette récolte doit être effectuée sans induire de perturbations sur l'exécution : l'ordre des événements ne doit pas en être modifié. Une certaine perturbation peut être introduite par la récolte lors de la détection et lors de la sauvegarde sur disque. La récolte peut aussi introduire

de nouveaux points de synchronisation. La trace générée peut être constituée d'un ou plusieurs fichiers.

Les événements peuvent être récoltés de manière logicielle ou matérielle. Notons que la récolte non intrusive de traces n'est pas un problème spécifique au déverminage mais existe également pour l'analyse de performances dont les outils sont souvent basés sur une récolte d'événements significatifs. Nous pouvons citer, par exemple, le projet ALPES [KTP92] développé à Grenoble au LGI et celui développé à Lyon au LIP [PTV92].

Récolte logicielle

On peut distinguer plusieurs méthodes. Le programmeur peut effectuer lui même la sauvegarde et la détection en insérant du code dans le programme. Il peut également utiliser une bibliothèque d'appels système modifiés en conséquence. C'est l'option utilisée par E. LEU [LSZ91]. Une autre méthode est l'utilisation de processus espions. Un tel processus observe l'activité d'un ou plusieurs processus et doit détecter et sauvegarder les événements générées par les processus espionnés. L'utilisation de processus espions sera illustrée par Parasight [AG89] à la section 1.4.1. S. CHAUMETTE a introduit la notion de « greffe » [Cha92], un processus greffé sur un lien de communication pouvant ainsi espionner les messages échangés ou même les modifier. Il a montré que ses espions ne changent pas l'ordre des événements s'ils ne font qu'observer et ne modifient pas les messages.

Récolte matérielle

Dans le cas d'une récolte logicielle, on ne peut empêcher une certaine perturbation sur l'exécution du programme: une modification temporelle de l'exécution du programme est donc possible. L'idéal est de disposer d'un processeur attaché à chaque nœud de la machine parallèle dédié à la reconnaissance d'événements. Le système MAD [RRZ89] propose ainsi de doubler la machine parallèle pour réaliser la détection, le filtrage et la sauvegarde des événements. A chaque processeur est donc associé un processeur espion. Celui-ci possède plusieurs liens de communication particuliers avec le processeur de calcul et un lien avec le cache du processeur de calcul. Il peut ainsi détecter et récolter les événements sans perturber le processeur de calcul. Mais c'est une solution vraiment coûteuse.

Récolte mixte

L'utilisation d'une récolte matérielle, souvent réalisée à l'aide d'une surveillance du bus, produit des événements de très bas niveau qui ne sont pas facilement utilisables par les outils de mise au point. C'est pourquoi certains systèmes allient la méthode logicielle et la méthode matérielle. Ainsi, avec The Flight Recorder [Gor91], la détection de l'événement est réalisée de manière logicielle par insertion de code, permettant de détecter des événements de plus haut niveau. Mais le traitement, le filtrage et la sauvegarde des événements sont réalisés matériellement à l'aide d'un coprocesseur.

1.2 Visualisation et animation de traces

La solution principalement retenue pour présenter une trace à l'utilisateur est une représentation graphique des informations contenues dans la trace. Cela peut être réalisé par une animation qui est une visualisation dynamique du comportement du programme. L'idée est de présenter au programmeur une vision schématique de l'exécution du programme pour l'aider à détecter des erreurs de synchronisation et de communication et à comprendre le fonctionnement général du programme. L'outil le plus connu est PARAGRAPH [HE91] mais est plutot orienté mise au point de performances.

Nous allons décrire plusieurs outils de visualisation qui différent de par la nature des informations présentées, et la manière de les visualiser. Nous pouvons dégager deux catégories de visualisation :

- une qui privilégie une vision du comportement au niveau de l'exécution du programme parallèle ;
- une qui privilégie une représentation graphique de l'algorithme : les données du programme, les actions et la sémantique.

Nous présenterons les systèmes suivants : Belvédère [HC87], Vici [CC91], Concurrency Map [Sto89], VISTOP [BB92], Event-Based Models of Behavior [Bat89, BW83], Voyeur [SBN89] et ChaosMon [Kil91]. Nous présenterons d'abord des outils mettant l'accent sur le programme et ensuite des outils permettant de s'adapter aux problèmes.

La plupart des outils présentés ici fonctionnent avec une configuration statique de programmes, ils supportent seulement un modèle statique de processus. C'est suffisant pour de nombreuses applications mais pas assez général. Les outils de visualisation doivent également prendre en compte la grande quantité potentielle d'informations à présenter. Certains outils, par exemple, semblent adaptés à un nombre restreint de processus, mais, si l'application en comporte un grand nombre, l'affichage peut rapidement devenir incompréhensible.

Les techniques d'animation présentent plusieurs limites par rapport à une visualisation classique : les dépendances temporelles sont difficiles à analyser car elles ne sont présentées que temporairement ; la granularité des dépendances temporelles est limitée car on ne peut présenter deux événements au même endroit en même temps.

1.2.1 Présentation orientée programme

Plusieurs niveaux d'abstraction peuvent être utilisés. Les outils de visualisation peuvent baser leur présentation sur le code source, le graphe de processus ou encore le modèle de programmation. Pour les outils plus orientés système, la structure de la machine (topologie, nombre de processeurs ...) ou les objets du système d'exploitation sont les plus importants. Nous présentons plusieurs outils : le premier basé sur le graphe de communication, le deuxième et le troisième sur le graphe de processus et le quatrième sur les objets primitifs du modèle de programmation. Mais d'autres outils auraient pu servir d'exemple, signalons par exemple les systèmes PV [KS92] et Maritxu [ZT92a, ZT92b] qui offre des vues animées sur l'usage des ressources des processeurs (charge, mémoire, liens de communication), les communications.

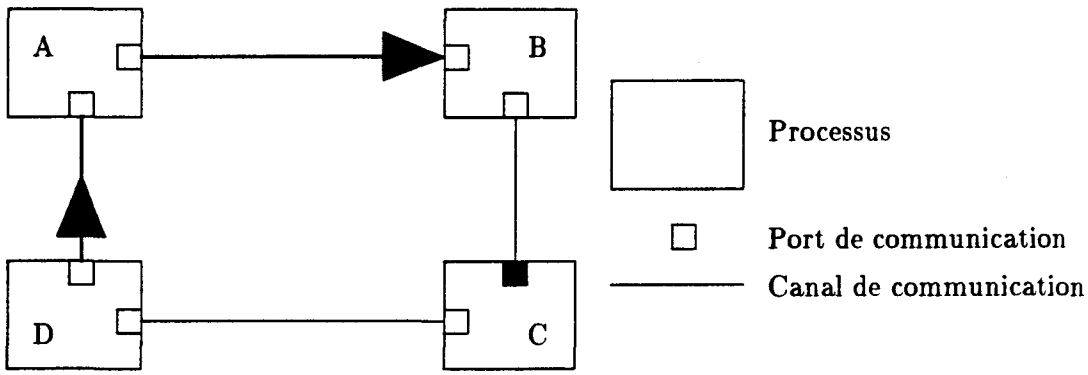

FIG. 1.4 - *Animation avec Belvédère*

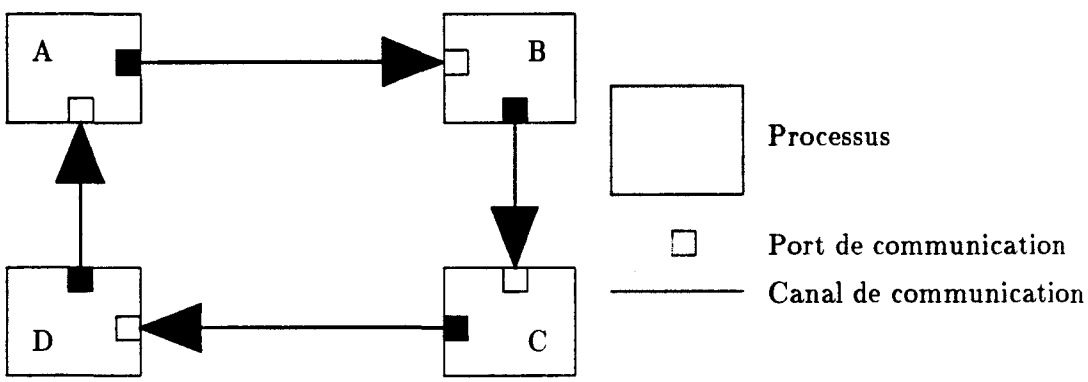

FIG. 1.5 - *Exemple d'interblocage détecté grâce à Belvédère*

Belvédère

Belvédère, réalisé par HOUGH et CUNY [HC87], est un outil d'animation graphique basé sur une trace générée par une exécution du programme. La trace est composée des événements primitifs du système. L'utilisateur a aussi la possibilité de réaliser une animation avec des événements plus abstraits en les définissant avec EDL. Les techniques d'animation consistent généralement en une représentation statique du médium de communication sur lequel sont superposés dynamiquement les événements intéressants. Voici la structuration des informations présentées à l'utilisateur. Chaque nœud du graphe représente un processus et les arcs relient deux processus pouvant communiquer. Une communication entre deux processus est représentée par la mise en évidence de l'arc et des nœuds concernés. La figure 1.4 présente une animation avec quatre processus. Le processus A a envoyé un message au processus B, qui ne l'attend pas (le port de communication n'est pas en inversion vidéo). Le processus C attend un message du processus B (le port de communication est en inversion vidéo). Un message a été envoyé par le processus D vers le processus A (il n'est pas encore arrivé: la flèche est au milieu

du lien). Il est alors facile de visualiser le contenu du message si cette information a été enregistrée. L'utilisateur peut ainsi suivre l'évolution de l'exécution du programme. Il est facile, par exemple, de détecter une situation d'interblocage (cf. figure 1.5).

VICI

VICI [CC91] est un outil de trace postmortem développé par S. CHAUMETTE. Il ne s'agit pas ici d'animation mais de présenter au programmeur un graphe des communications en fonction du temps. Ce graphe est visualisé à partir de fichiers trace. Plusieurs fonctionnalités sont proposées à l'utilisateur :

- il peut obtenir une légende explicitant les symboles utilisés dans le graphe ;
- il peut filtrer les types d'événements qu'il désire visualiser ;
- il peut sélectionner un arc symbolisant une communication et visualiser l'information transmise.

Un exemple de ce qui est présenté à l'utilisateur est donné figure 1.6. Trois processus de nom *elem* doivent envoyer chacun une valeur au processus *sum*. Celui ci en effectue la somme et envoie le résultat au processus *collect*. On s'aperçoit que le processus *sum* n'a reçu que deux messages au lieu de trois : il n'a pas attendu le troisième message et c'est pourquoi *collect* produit un mauvais résultat.

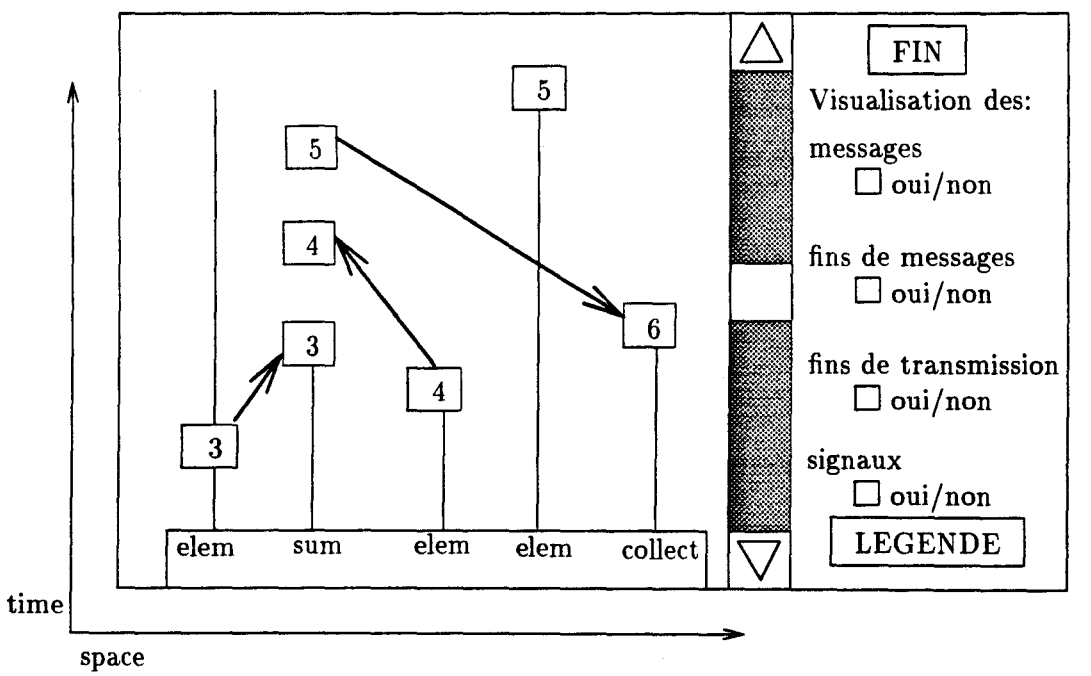

FIG. 1.6 - *Exemple d'utilisation de Vici*

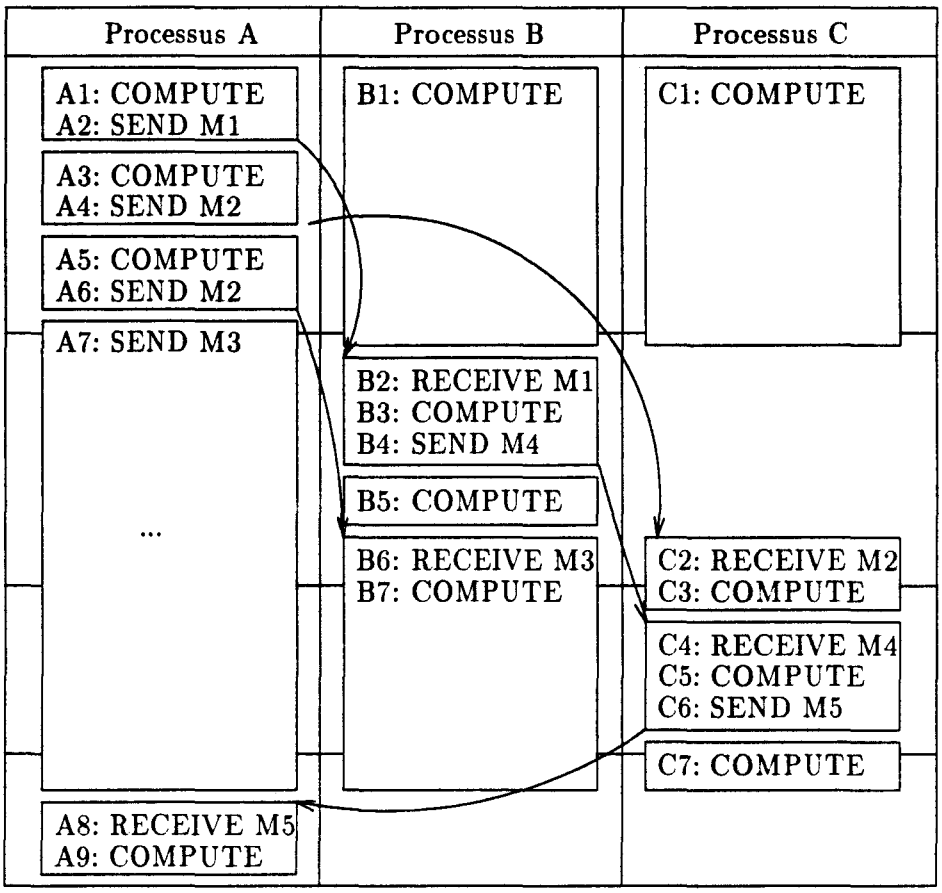

FIG. 1.7 - *Exemple de Concurrency Map*

Concurrency Map

L'idée de J.M. Stone [Sto89] consiste à représenter graphiquement la concurrence potentielle existant entre événements de différents processus contrairement aux outils précédents qui mettent l'accent sur les dépendances entre processus. L'historique des événements de chaque processus est représenté dans un tableau appelé « Concurrency Map », chaque colonne correspondant à un processus et chaque ligne délimitant un intervalle de temps. Tous les événements apparaissant dans une même rangée et dans des colonnes différentes peuvent s'exécuter concurremment. Ainsi, tous les états globaux possibles de l'application sont représentés: le graphique présente l'ensemble des comportements possibles du programme préservant l'ordre partiel défini par l'historique de chaque processus et les dépendances temporelles. Un exemple de Concurrency Map d'un programme pour une architecture à mémoire répartie avec communication par envoi de messages asynchrone est présenté à la figure 1.7. On peut ainsi constater visuellement que A2 se produit avant C4, car il y a une dépendance à travers B2; C1 doit se terminer avant que A8 ne commence; A7 peut être concurrent avec n'importe quel événement du processus C.

VISTOP

VISTOP [BB92] est un outil d'animation de programmes parallèles appartenant à l'environnement TOPSYS. Il visualise la structure d'une application avec les objets de base du modèle de programmation : les tâches, les boîtes aux lettres et les sémaphores. L'outil propose trois vues différentes :

- « Concurrency view » : les objets sont représentés par des icônes ; devant et derrière les icônes représentant les boîtes aux lettres sont représentés les queues des tâches attendant un message ou envoyant un message ; ainsi, l'icône représentant une tâche voulant déposer un message dans une boîte se déplacera de sa position actuelle pour venir s'insérer dans la queue devant la boîte ; l'animation est réalisée de la même façon pour l'accès aux sémaphores ;
- « Object creation view » : c'est une animation de la création des objets ; les objets sont présentés hiérarchiquement suivant leur parenté et l'animation est réalisée à la faveur des créations et destructions dynamiques ;
- « System view » : il s'agit de la présentation de la distribution des objets à l'aide d'un tableau, chaque colonne contenant les objets détenus par un processeur.

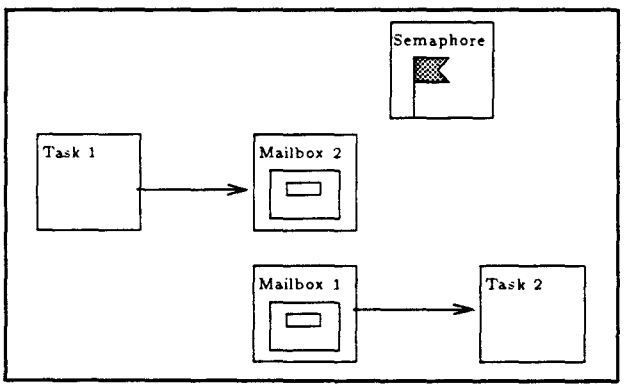

FIG. 1.8 - *Exemple de Concurrency view avec Vistop*

1.2.2 Présentation orientée algorithme

Souvent, il existe un écart important entre la représentation abstraite d'un problème par le programmeur et la structure d'un programme parallèle le résolvant. Les outils que nous allons présenter permettent au programmeur de définir lui même les informations qui lui seront présentées, se rapprochant ainsi du niveau d'abstraction du problème traité par son application. Il décide également de la manière dont elles seront visualisées.

Mais plusieurs critiques peuvent être opposées à ces systèmes. La visualisation n'est plus automatique : l'utilisateur doit la programmer. C'est pourquoi la plupart des outils proposent une bibliothèque d'objets graphiques de base facilement adaptables à de

nouveaux algorithme. Il en va de même pour les modèles de comportement de P. BATES. Quand une erreur est détectée avec ces outils, le programmeur ne peut pas savoir si elle provient du programme ou de la visualisation qu'il a programmée.

Nous allons présenter deux outils de visualisation : le premier permet au programmeur de définir comment les informations lui seront présentées et le deuxième propose en plus l'utilisation d'un mécanisme proche de celui de P BATES.

« Event-Based Models of Behavior »

P. BATES utilise les événements définis par EDL pour une approche de haut niveau du déverminage: EBMB [Bat89, BW83] (Event-Based Models of Behavior). Il s'agit de créer des modèles de comportement à partir de l'exécution et de les comparer aux comportements espérés. Les effets observables et les interactions entre les composants du système sont caractérisés par une suite d'événements décrits avec EDL. Le programmeur peut construire des modèles de comportement à plusieurs niveaux en définissant des événements par agrégation d'événements déjà définis. Il a également la possibilité de filtrer les événements pour éliminer les événements qui ne relèvent pas du modèle considéré en utilisant des conditions dans la définition des événements. En variant le niveau de granularité des événements et en décrivant différents composants du système à travers des points de vue variés, toute l'activité du système peut être décrite et donc observée. Le programmeur construit ainsi une véritable base de modèles d'événements. Les événements de base sont générés par le système et stockés. Un « reconnaisseur d'événements », utilisant la trace et les descriptions des modèles existant dans la base, permet de signaler l'occurrence d'événements composés. Le programmeur interagit avec le « reconnaisseur d'événements » en spécifiant les modèles l'intéressant. La figure 1.9 présente l'architecture du système.

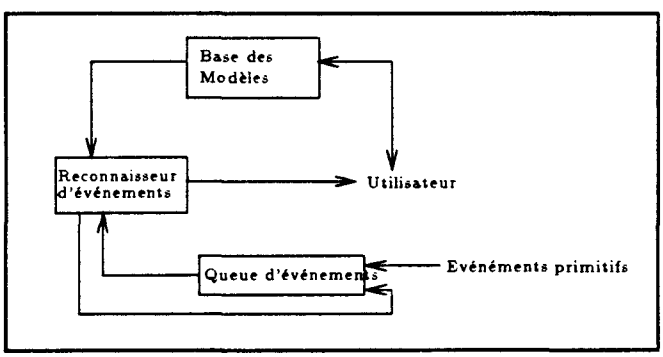

FIG. 1.9 - *Architecture de EBBA*

Voyeur

Voyeur [SBN89] permet à l'utilisateur de construire facilement une vue graphique du programme parallèle (ou même une hiérarchie de vues) spécifique à son application. Les vues se rapprochent donc de la modélisation du problème effectuée par le programmeur. Elles sont définies de manière orientée objet et organisées en hiérarchies de classes de vues : la réutilisabilité est accrue par rapport aux programmes parallèles et aux systèmes parallèles. La vue est dirigée par les événements. C'est le programmeur qui doit les générer en insérant du code dans son programme. Ils peuvent être utilisés par l'intermédiaire d'un fichier trace, mais également directement lors de l'exécution.

ChaosMon

Cet outil [Kil91] regroupe les idées de P. BATES et de Voyeur. Le programmeur doit spécifier un modèle de son programme à l'aide d'un langage de description. Il décrit ensuite différentes vues sur le modèle du programme, ce qui correspond à la définition de modèles pour BATES, et ensuite spécifie des objets d'affichage qui sont mis en relation avec les vues. Tout cela est géré à l'aide d'une base de données et les modèles, vues et objets d'affichages peuvent être réutilisés.

1.2.3 Conclusion

Les outils de visualisation permettent essentiellement de détecter des erreurs de synchronisation et apportent une bonne compréhension du comportement général du programme. Cependant la grande quantité d'information à visualiser reste le problème principal de ces outils : l'outil doit rester utilisable si le nombre de processus augmente.

1.3 Déverminage postmortem

Le déverminage postmortem se déroule en deux phases. Une première phase consiste à récolter des événements et des informations associées sous forme de traces lors de l'exécution. La deuxième phase consiste à analyser cette trace pour trouver les erreurs. Les outils de visualisation présentés sont en majorité des outils postmortem. Cette méthode offre deux avantages principaux :

- l'analyse postmortem répond au problème du non déterminisme des programmes parallèles en fixant une exécution à travers un historique d'événements : l'analyse peut être reproduite autant de fois que nécessaire ;
- les outils postmortem ne nécessitent pas l'usage de la machine parallèle ou distribuée.

Mais l'analyse postmortem n'offre pas que des avantages :

- une analyse fine de l'exécution réclame l'enregistrement d'un grand nombre d'événements et d'informations associées, comme par exemple le contenu des messages, ce qui implique une trace de taille importante ; l'enregistrement de cette trace risque donc de perturber le comportement de l'application ; il faut signaler les travaux de MILLER et CHOI [MCN88] qui proposent un mécanisme de trace incrémentale pour diminuer la perturbation de l'exécution ;

- les informations accessibles au programmeur sont figées dans la trace; on ne peut analyser que ce qui a été enregistré; ceci peut être corrigé en utilisant des mécanismes similaires à ceux de P. Bates, mais tout l'état de l'application ne pourra pas être enregistré à chaque événement.

1.4 Déverminage durant l'exécution

Avec l'analyse postmortem, aucune interaction entre le programmeur et l'exécution n'est possible. Or le programmeur a besoin de connaître plus en détail le fonctionnement du programme : l'interprétation des données pour la mise au point doit inclure un retour de la part du programmeur.

Nous allons examiner les méthodes mises en œuvre pour la mise au point durant l'exécution. En premier lieu, nous verrons les méthodes utilisées directement pendant l'exécution et ensuite les techniques de réexécution.

1.4.1 Déverminage directement durant l'exécution

La méthode la plus répandue pour interagir avec l'exécution d'un programme est le point d'arrêt. Les points d'arrêt peuvent être positionnés de différentes manières :

- ils peuvent être ajoutés au code : soit dans le source, soit pendant l'exécution ;
- ils peuvent être associés à des événements ;
- ils peuvent être associés à un état particulier du programme : par exemple arrêter si $a >= 10$ et $x <= 0$.

Nous allons d'abord présenter deux outils utilisant cette technique, l'un dans le cadre d'une architecture à mémoire partagée, l'autre dans le cadre d'une architecture distribuée. Nous montrerons leurs limites et ensuite les solutions proposées.

Architectures à mémoire commune

Dans une architecture à mémoire commune, les processus communiquent par variables partagées. Les points d'arrêt seront donc généralement des conditions sur la valeur de ces variables. La première méthode pour détecter ce genre d'états est de rajouter des instructions dans le code source et de tester les variables à chaque modification.

La technique développée par Z. ARAL et I. GARTNER [AG89] dans Parasight est plus élaborée. Le programmeur peut lier dynamiquement des fonctions au programme à déverminer. Les endroits dans le code où le programmeur a choisi de lier ses fonctions sont appelés « points d'observation ». Des processus « parasites » sont également liés dynamiquement à l'application : ce sont les outils de déverminage. Ils peuvent s'exécuter sur d'autres processeurs. Ils peuvent ainsi espionner le comportement du programme et communiquent avec les fonctions liées aux « points d'observation » par variables partagées. Mais ils peuvent également, par exemple, stopper le programme. Voici un petit exemple d'utilisation : le programmeur veut stopper son programme si **max** appels à la fonction `f` ont été effectués; aux endroits où `f` est appelée, il lie une fonction

qui incrémente une variable ; il lance également un processus parasite qui sera chargé d'arrêter le programme quand cette variable aura atteint la valeur **max**.

Mais les « points d'observation » sont plus généraux que les points d'arrêt. Un petit exemple présenté figure 1.10, tiré de [AG89], montre comment réaliser une trace des valeurs de deux variables. La commande *run* lance l'exécution du programme *test* ; *load* permet de charger le module parasite *parasite1* ; la commande *start* lance le processus espion *trace_watcher* ; *insert* crée un « point d'observation » à la ligne cinq du programme *test* et la commande *set* lie la fonction *trace_a_b* au « point d'observation ». A chaque fois que le programme *test* rencontrera le « point d'observation », la fonction *trace_a_b* sera appelée. Cette fonction enregistre dans un buffer circulaire les valeurs des variables partagées *a* et *b*. Le processus *trace_watcher* permettra, une fois la variable *outT* positionnée à 1, d'afficher une trace des valeurs prises par *a* et *b*.

Commandes de Parasight :

```
para: run test
para: load parasite1
para: start trace watcher
para: insert test:5
para: set test:5 trace_a_b EBABLE
```

Processus espion :

```
trace_watcher()
{
while(1) if(outT)
  for(i=0;i<TraceSize;i++)
    printf("time %x, trace= %d %d\n",
           TraceBuffer[i].scan_time,
           TraceBuffer[next_entry].data1,
           TraceBuffer[next_entry].data1);
}
```

Fonction liée au point d'observation :

```
trace_a_b()
{
extern int a,b;
 if(startT) {
   TraceBuffer[next_entry].scan_time = *timeptr;
   TraceBuffer[next_entry].data1 = a;
   TraceBuffer[next_entry].data2 = b;
   next_entry = ++next_entry % BUF_SIZE;
 }
}
```

FIG. 1.10 - *Utilisation de Parasight*

Le principal problème que posent de tels points d'arrêts conditionnels est l'évaluation de la condition globale : une décision d'arrêt peut être prise alors qu'une partie de la condition n'est plus vérifiée. Supposons que la condition soit $a > 10$ et $b < 0$, a et b étant deux variables partagées par deux processus. Une fois la première partie de l'expression évaluée, $a > 10$, et pendant l'évaluation de $b < 0$, un des deux processus peut modifier la valeur de a : ainsi $a > 10$ et $b < 0$ pourra être faussement évaluée à vrai.

Architectures à mémoire distribuée

Dans une architecture à mémoire distribuée, les processus communiquent par messages. Les points d'arrêt seront donc ici liés aux événements d'émission et de réception de messages. La détection ne pose pas de problèmes particuliers sur un processus donné. Mais il est plus intéressant de pouvoir arrêter le programme en tenant compte des interactions entre les différents sites, d'un état plus complexe du programme. Pour cela a été introduite la notion de points d'arrêts distribués : ces points d'arrêt sont définis sur l'état de plusieurs processus ou associés à des événements composés. La détection de tels événements impose une communication supplémentaire entre les différents sites impliqués dans la génération de ces événements: un certain délai est inévitable entre le moment où un événement est détecté sur un site et celui où un autre site est prévenu de sa détection. Des événements tels que **`e1 et e2 se produisent au même instant`**, `e1` et `e2` étant deux événements se produisant sur deux sites différents, sont indétectables. Nous discuterons plus en détail des problèmes relatifs à la détection de tels événements dans le chapitre 3 où nous présentons notre définition et implémentation des points d'arrêt distribués pour le langage BOX.

B. MILLER et J.D. CHOI [MC88] définissent des points d'arrêt distribués et un algorithme pour les détecter. Un point d'arrêt distribué dépendra de prédicats définis sur des nœuds différents. Ils décomposent leur définition en plusieurs niveaux :

- S.P. : les simples prédicats. Ils sont basés uniquement sur le comportement ou l'état d'un seul processus, par exemple *le processus A a reçu un message*;

- D.P. : les prédicats disjonctifs. Il s'agit d'une combinaison de S.P. avec l'opérateur disjonctif, par exemple *le processus A a reçu un message ou le processus B a envoyé un message*;

- C.P. : les prédicats conjonctifs. Il s'agit d'une combinaison de S.P. avec l'opérateur de conjonction, par exemple *le processus A a reçu un message et le processus B a envoyé un message*;

- L.P. : les prédicats liés. Il s'agit d'une succession de D.P. ordonnés par la relation « précède », par exemple *(le processus A a envoyé un message ou le processus B a envoyé un message) « précède » le processus C a reçu un message.*

Une restriction est à apporter pour les C.P. En effet, comme nous l'avons remarqué précédemment, il est difficile d'interpréter dans un système distribué la notion de simultanéité introduite par l'utilisation de l'opérateur de conjonction. MILLER et CHOI imposent donc aux différents S.P. d'être ordonnés par la relation « précède ». Ils définissent un algorithme distribué de détection de point d'arrêt et un algorithme d'arrêt dérivé de celui de CHANDY et LAMPORT [CL85] défini pour la sauvegarde d'instantanés d'exécution (cf. 1.4.2 Instantanés d'exécution).

Le problème de l'arrêt

La politique d'arrêt peut être différente lorsque l'on rencontre un point d'arrêt :

- arrêter tous les processus, ce qui est difficile à réaliser dans un court laps de temps ;
- arrêter un processus ou un groupe de processus, auquel cas la perturbation sur l'exécution est importante.

De plus, si on suspend l'exécution d'un programme, il faut pouvoir l'arrêter dans un état cohérent, ce qui n'est pas simple à réaliser dans le cadre d'une simple exécution [1] : les processus ne peuvent être stoppés simultanément. On appelle état cohérent, un état dans lequel les états de deux processus arrêtés ne dépendent pas l'un de l'autre, c'est à dire qu'ils ne sont pas liés par la relation « précède ».

Conclusion

La principale limite de l'interaction directe reste donc le non déterminisme des programmes parallèles. L'interaction directe perturbe beaucoup l'exécution du programme et peut donc empêcher la production de l'erreur observée lors d'une précédente exécution. Mais tous les outils décrits ici peuvent être utilisés dans le cadre d'une réexécution.

1.4.2 Déverminage durant une réexécution

Pour répondre au problème du non déterminisme des programmes parallèles, une solution est de forcer une réexécution du programme identique à une exécution initiale. Pour pouvoir réexécuter le programme, il faut avoir préalablement sauvegardé les événements générateurs du non déterminisme lors de l'exécution de référence. En utilisant cette trace lors de la réexécution, on obtient alors une exécution déterministe et semblable à l'exécution de référence. La réexécution est fidèle si elle garantit le même ordre partiel d'occurrence des événements que l'exécution initiale. Signalons que la réexécution est fidèle, mais non parfaitement identique.

On peut distinguer deux types de réexécution : l'une dirigée par les données ou réexécution partielle que nous détaillerons d'abord, l'autre dirigée par les événements de synchronisation ou réexécution totale. Nous verrons ensuite le mécanisme de reprise d'exécution utilisant des instantanés d'exécution qui permet de réexécuter le programme depuis divers points de l'exécution et non pas uniquement depuis le début. Nous illustrerons par un exemple représentatif chaque type de réexécution. Mais signalons que d'autres exemples auraient pu être choisis, [CW82, Wit89] pour la réexécution dirigée par les données et [GHP+93, Jam93, LA93, Ber92] pour la réexécution dirigée par le contrôle.

Réexécution dirigée par les données

On appelle réexécution dirigée par les données, une réexécution dirigée par les informations associées aux événements. Toute l'information associée à un événement doit

1. MILLER et CHOI atteignent un état cohérent avec l'algorithme de CHANDY et LAMPORT.

être sauvegardée: les accès aux variables partagées sont interceptés et les valeurs correspondantes ainsi que les messages reçus par le processus stockés sur fichier. Toutes les entrées effectuées par un processus doivent être enregistrées. Lors de la réexécution, ce processus sera considéré séparé des autres processus et devra utiliser cette sauvegarde. En effet, cette méthode est principalement utilisée pour ne réexécuter qu'un seul processus: ceci est appelé réexécution partielle.

Cette technique offre plusieurs avantages:

- souvent, la réexécution d'un seul processus est utile au déverminage: le processus que l'on soupçonne d'abriter l'erreur;
- la réexécution sur un seul processeur est possible et ne nécessite donc pas l'utilisation de l'architecture parallèle ou distribuée;
- le fait que l'information destinée à un processus ne doit pas être reproduite permet un gain de temps lors de la réexécution.

L'inconvénient majeur de cette méthode est l'importance de la trace générée lors de la phase de sauvegarde: il faut en effet sauvegarder les valeurs des variables partagées et le contenu des messages reçus. La perturbation sur l'exécution du programme peut donc être importante.

Réexécution dirigée par le contrôle

Le but est ici de réduire la taille de la trace nécessaire pour la réexécution. Le principe de cette technique a été donné avec Instant Replay par T. LEBLANC et J. MELLOR-CRUMMEY [LMC87] pour une machine à mémoire partagée: si chaque processus reçoit les mêmes données, il produira les mêmes résultats dans le même ordre qui pourront servir de données pour d'autres processus. Toutes les entrées d'un processus n'ont donc pas à être enregistrées. Durant la phase de sauvegarde, les informations à enregistrer sont les entrées externes et l'ordre des communications interprocessus: accès aux variables partagées et envoi et réception de messages. Il n'est pas utile d'enregistrer toute l'information associée aux événements comme par exemple le contenu d'un message lors d'une réception car elle sera reproduite lors de la réexécution. La réexécution doit reproduire le même ordre pour les événements de communication interprocessus. Le choix d'une réexécution dirigée par les événements de synchronisation impose de réexécuter tous les processus. Par rapport à la méthode précédente, cette technique permet de réduire considérablement le volume d'informations à enregistrer et, par conséquent, le temps nécessaire pour le faire, donc la perturbation sur le programme lors de l'exécution.

L'utilisation de cette méthode oblige à contrôler toute l'exécution du programme. Ce contrôle peut être effectué par le processus lui même comme on le décrira dans les exemples, mais on peut envisager également une réexécution séquentielle de tous les processus du programme sous le contrôle d'un superviseur. Le choix du processus actif est effectué grâce aux informations contenues dans le fichier événements: le seul processus actif sera le processus responsable de la génération du prochain événement. Dans le cas des architectures à mémoire répartie, un superviseur par site est nécessaire. Mais pour délivrer les messages dans le même ordre lors de la réexécution, le

superviseur devra utiliser des informations associées aux événements comme l'identité de l'expéditeur du message lors d'une réception.

Exemple de réexécution dirigée par les données

D. PAN et M. LINTON utilisent une telle technique pour leur outil Recap [PL89]. Ils utilisent un mécanisme d'instantanés d'exécution. Le programmeur a la possibilité d'examiner l'état d'un processus à l'endroit qu'il désire. Le processus doit être réexécuté depuis le plus proche instantané d'exécution jusqu'à l'état désiré. Les processus communiquent par appels système ou par variables partagées. Pour le déverminage, le programme utilisera une bibliothèque particulière d'appels système qui permet l'enregistrement sur fichier des informations liées aux événements lors de l'exécution, et l'extraction du fichier de ces informations lors de la réexécution (cf. figure 1.11). Un booléen permet de choisir l'une ou l'autre action. Pour les variables partagées, le compilateur aura détecté les accès à ces variables et rajouté des instructions dans le code pour sauvegarder sur fichier leurs valeurs lors de l'exécution, ou pour extraire du fichier les valeurs de ces variables lors de la réexécution.

```
get_message(m)
{
  if(replay)
    extract_from_file(my_pid,m);
  else {
    get(m);
    store_to_file(my_pid,m);
  }
}
```

FIG. 1.11 - *Exemple d'appel système: réception de message*

Exemples de réexécution dirigée par le contrôle

Avec Instant Replay, toutes les interactions entre processus sont vues comme des opérations sur des objets partagés, l'envoi de message vers un processus étant considéré comme un accès à une boîte aux lettres. Une série de modifications sur ces objets est représentée comme une séquence totalement ordonnée de versions, chacune ayant un numéro unique pour un objet particulier. Durant l'exécution, on enregistre un ordre partiel des accès à chaque objet partagé qui est spécifié par une séquence de numéros de versions. Pour enregistrer cet ordre partiel, le système maintient un numéro de version courante pour chaque objet et le nombre de lecteurs pour chaque version de chaque objet. En plus, chaque processus enregistre la version de chaque objet partagé auquel il accède. Durant la réexécution, chaque processus est autorisé à produire ses résultats qui seront les entrées des autres processus. L'enregistrement des accès aux objets est utilisé pour être sûr que la même version de l'objet est utilisée lors de la phase de réexécution et de la phase d'enregistrement.

Un exemple de fonctionnement, tiré de [LS91], est donné aux figures 1.12 pour la phase d'exécution et 1.13 pour la phase de réexécution. La phase d'exécution se déroule

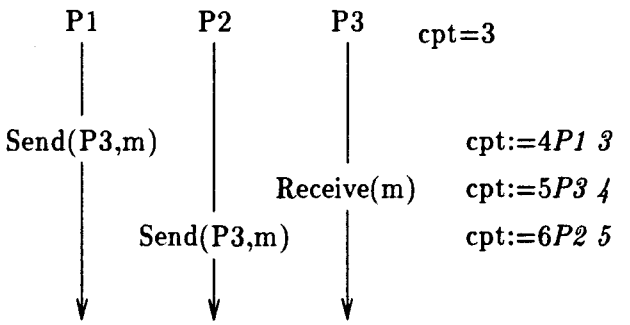

FIG. 1.12 - *Phase d'exécution avec Instant Replay*

ainsi : le compteur, correspondant à un numéro de version de la boîte aux lettres de p3, initialement vaut 3. Il est incrémenté après l'envoi du message de p1 à p3, la réception par p3 et l'envoi de p2 à p3. La trace générée est en italique. Dans la trace de chaque processus est enregistré le numéro de version à laquelle il accède. Durant la phase de réexécution, cpttrace représente la valeur du compteur dans la trace. Le processus p2 est le premier à envoyer son message. La valeur du compteur associé à cet événement dans sa trace étant de 5 et la valeur courante du compteur de 3, p2 est suspendu. p3 est aussi suspendu. Lors de l'envoi de p1 à p3, le compteur est incrémenté et p3 est réveillé et reçoit son message. p2 est alors réveillé.

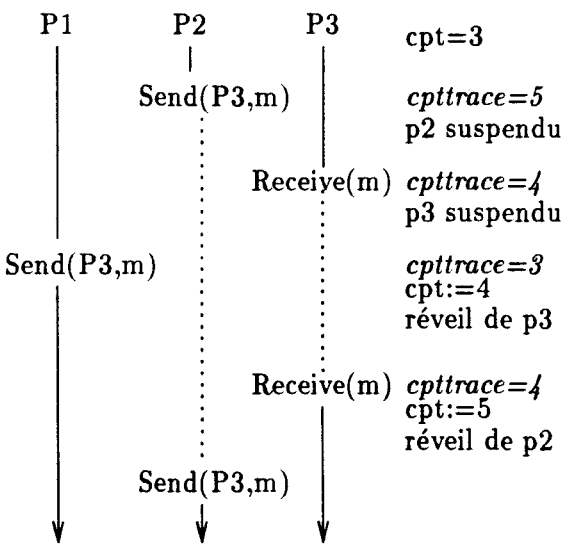

FIG. 1.13 - *Phase de réexécution avec Instant Replay*

E. LEU, A. SCHIPER et A. ZRAMDINI [LSZ91] proposent une méthode de réexécution pour architecture distribuée qui prend en compte l'envoi de messages synchrone

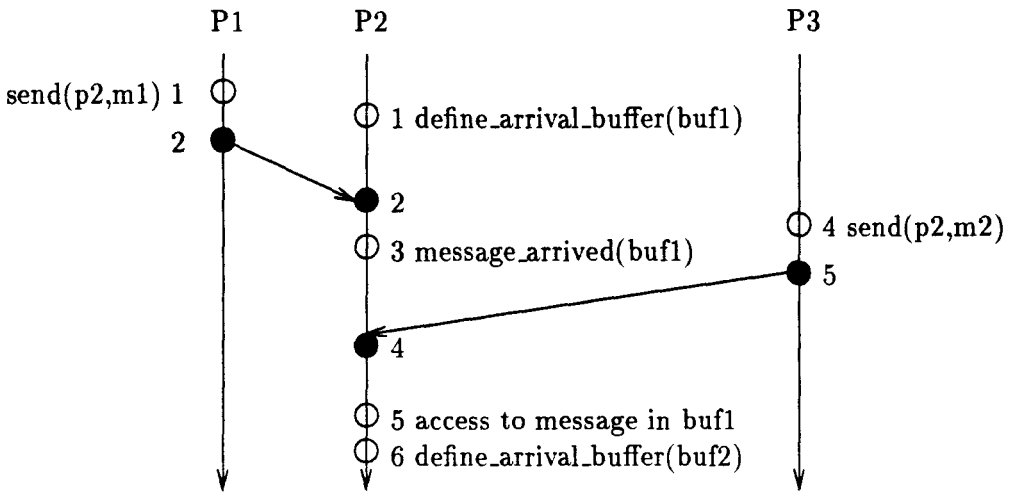

FIG. 1.14 - *Exemple d'exécution de trois processus*

P1	P2	P3
(2,P2,DEPARTURE)	(2,P1,ARRIVAL)	(5,P2,DEPARTURE)
	(4,P3,ARRIVAL)	

FIG. 1.15 - *Traces générées*

et asynchrone. Leur mécanisme est inspiré de la technique précédente. Il repose sur la distinction entre événements explicites (appels systèmes) et événements implicites (arrivée ou départ d'un message). La figure 1.14 présente un exemple d'exécution de programme avec trois processus. Les événements explicites sont représentés par des disques blancs et les événements implicites sont représentés par des disques noirs. Les primitives utilisées pour la communication asynchrone sont :

- **`define_arrival_buffer(buffer)`** qui fournit un buffer au système dans lequel un message peut être stocké ; si un message est déjà arrivé, il est immédiatement transféré du buffer système au buffer du processus ;
- **`message_arrived(buffer)`** qui retourne vrai si un message est contenu dans le buffer du processus ;
- **`define_departure_buffer(dest,buffer)`** qui fournit au système le processus destinataire et le buffer contenant le message à envoyer ;
- **`message_sent(buffer)`** qui retourne vrai si le buffer peut être réutilisé.

Les événements sont numérotés et seuls les événements implicites sont enregistrés. La figure 1.15 présente les traces générées pour le programme de la figure 1.14. Le mécanisme de génération de la trace est présenté à la figure 1.16. Dans la procédure

```
procedure define_arrival_buffer(var buf:buffer_type)
  begin
    event_counter := event_counter + 1;
    "execute the primitive define_arrival_buffer(buf) ";
end define_arrival_buffer;

procedure message_arrival(sender_proc:process_type)
  begin
    event_counter := event_counter + 1;
    SaveEventInFile(event_counter,sender_proc,ARRIVAL);
end message_arrival;
```

FIG. 1.16 - *Génération des traces*

`define_arrival_buffer`, à chaque événement explicite, le compteur d'événement est incrémenté. A chaque occurrence d'événement implicite, une procédure est appelée pour l'enregistrer sur fichier : `message_arrival` est présenté comme exemple pour une arrivée de message.

Lors de la réexécution, pour effectuer un événement explicite, un processus doit attendre l'occurrence de tous les événements implicites le précédant. Pour diriger la réexécution, il faut appeler la procédure `Before_Explicit_Event` (voir figure 1.17) avant d'exécuter un événement explicite ce qui permet d'attendre tous les événements implicites devant le précéder. Examinons le programme de la figure 1.14 et la trace de la figure 1.15. Lors de la réexécution, le processus p2 veut effectuer la primitive `message_arrived()`, `event_counter` vaut 2 et `next_recorded_event` correspond à `p1,ARRIVAL`. p2 devra donc attendre l'arrivée du message en provenance de p1.

Instantanés d'exécution

Si le programme est long, il peut être pénalisant de reprendre l'exécution depuis le début. Un instantané d'exécution est une photographie de l'état du programme à un instant donné : une copie de l'espace mémoire des processus et de l'état des processeurs. La sauvegarde est effectuée régulièrement en arrêtant les processus du système. Le programmeur a ainsi la possibilité de reprendre l'exécution du programme à certains endroits de l'exécution. Mais pour assurer une réexécution fidèle entre deux instantanés d'exécution, il faut également sauvegarder les événements à caractère non déterministe. Cette technique est également utilisée pour les systèmes tolérants aux fautes.

D. PAN et M. LINTON basent le système de réexécution de leur outil Recap [PL89] sur ce principe. La réexécution d'un processus n'est possible qu'à partir d'un instantané d'exécution. La sauvegarde de l'état de tous les processus n'est ici pas nécessaire puisqu'on ne s'occupe de la réexécution d'un seul processus. Les instantanés d'exécution sont implémentés par un processus suspendu créé avec une primitive équivalente au Fork UNIX. Le processus père continue et le processus fils est suspendu jusqu'à son utilisation pour une réexécution. A la réexécution, le processus suspendu se duplique de nouveau pour éviter de perdre l'instantané d'exécution.

L'utilisation d'instantanés d'exécution dans une architecture répartie est plus déli-

```
procedure Before_Explicit_Event
  begin
    while event_counter = next_recorded_event'NUMBER do
      if next_recorded_event'CLASS = DEPARTURE then
        wait for release of the buffer containing the
        message destinated to next_recorded_event'DEST_PROC
      else
        wait for the message sent by
        next_recorded_event'SENDER_PROC
      endif;
      next_recorded_event = next recorded event in the trace;
      event_counter := event counter + 1;
    end while;
end Before_Explicit_Event;

procedure define_arrival_buffer(var buf:buffer_type)
  begin
    event_counter := event_counter + 1;
    Before_Explicit_Event;
    "execute the primitive define_arrival_buffer(buf) ";
end define_arrival_buffer;
```

FIG. 1.17 - *Contrôle de la réexécution*

cate : il faut sauvegarder un état cohérent du programme. Supposons que deux processus communiquent par message. Si l'état du premier processus est sauvegardé avant l'envoi de son message et l'état du second sauvegardé après la réception du message, l'état du programme est incohérent. K. CHANDY et L. LAMPORT donnent un algorithme de sauvegarde pour machines à mémoire répartie [CL85]. Ils modélisent le programme parallèle comme un ensemble de processus connectés par des canaux de communication unidirectionnels possédant des tampons infinis, supposés sans erreurs et délivrant les messages dans l'ordre de leur envoi. L'état global d'un système distribué ne sera plus uniquement composé de l'ensemble des états des processus mais également de l'état des canaux de communication : l'instantané d'exécution ne sera plus seulement constitué d'une sauvegarde de l'état des processus mais aussi d'une sauvegarde de l'état des canaux de communication. Cette sauvegarde se fait à l'aide de marqueurs spéciaux envoyés à travers les canaux, un par instantané d'exécution. Le principe de l'algorithme est donné figure 1.18.

1.5 Synthèse et Conclusion

La mise au point des programmes parallèles est compliquée par :

- leur non déterminisme potentiel ;
- la perturbation engendrée par toute observation ;

envoi d'un marqueur par un processus p :

```
p sauvegarde son etat
pour chaque canal c en sortie de p
  p envoie un marqueur a travers c
  avant d'envoyer d'autres messages
```

réception d'un marqueur par un processus q par le canal c :

```
si q n'a pas sauvegarde son etat
 alors debut
   q enregistre son etat
   q enregistre l'etat de c comme vide
 fin sinon
   q enregistre l'etat de c comme la
   sequence des messages recus par c
   apres avoir enregistre son etat et
   avant de recevoir le marqueur
finsi
```

FIG. 1.18 - *Algorithme de Chandy et Lamport*

- la difficulté de contrôler leur exécution (par exemple pour les arrêter) en particulier dans le cadre d'architectures distribuées.

La mise au point de programmes parallèles doit donc au moins avoir les propriétés suivantes :

- toute analyse doit être reproductible ;
- l'analyse doit être interactive bien que la collecte et la présentation des données puissent être automatisées.

L'approche qui consiste à mettre au point le programme directement durant l'exécution se heurte aux trois problèmes précédemment cités : elle ne prend pas en compte le non déterminisme ; peut perturber grandement le comportement du programme et arrêter le programme dans un état cohérent n'est pas aisé.

L'utilisation d'outils postmortem permet de répondre au problème du non déterminisme en fixant une exécution à travers une trace. Ils permettent d'obtenir une bonne compréhension du comportement du programme et peuvent être utilisés sur une machine autre que la machine parallèle. Mais ils ont besoin d'une grande quantité d'information. La génération et la sauvegarde de cette information peuvent provoquer un changement dans le comportement du programme. De plus, ils ne permettent pas au programmeur d'accéder à toute l'information désirée.

La réexécution de programmes semble être la voie la plus prometteuse. Elle permet de s'abstraire du problème du non déterminisme tout en permettant au programmeur d'interagir avec l'exécution du programme: elle autorise l'utilisation du déverminage cyclique. Le problème reste la collecte et la sauvegarde d'événements pour diriger la réexécution. En utilisant une réexécution dirigée par la synchronisation, la perturbation durant cette phase est considérablement réduite. Cette méthode nous semble donc la base pour la construction d'outils de mise au point. Au dessus de la réexécution peuvent être utilisés des outils de visualisation ou bien encore des points d'arrêts.

Chapitre 2

Un mécanisme de réexécution pour le projet PVC/BOX

NOUS AVONS DÉCRIT au chapitre précédent différents moyens de mise en œuvre de la mise au point de programmes parallèles. Cette étude nous a permis de conclure que la réexécution était une bonne approche pour la conception d'outils de déverminage.

Nous avons donc proposé et réalisé un mécanisme de réexécution dans le cadre du projet PVC/BOX. Nous présenterons d'abord dans ce chapitre le modèle d'exécution des programmes à déverminer qui sont composés de tâches comparables à des processus UNIX communiquant par message. Ces tâches abritent des entités actives comparables à des processus légers communiquant également par message. La seconde partie de ce chapitre sera consacrée à la présentation du mécanisme de réexécution. Nous décrirons en premier la phase d'enregistrement des événements et ensuite le contrôle de la réexécution. Nous terminerons par l'évaluation de notre mécanisme par des mesures effectuées sur les deux architectures utilisées.

2.1 La couche PVC

Le projet a débuté en 1990 par la notion de Processeur Virtuel de Classe (PVC) proposée par ERIC DELATTRE et JEAN-MARC GEIB [CDG91]. Un Processeur Virtuel de Classe est un serveur encapsulant une classe et ses instances. La proposition était basée sur le principe de localité du code et des données et d'une fragmentation des objets par l'héritage. Cette architecture logicielle devait servir de support pour les langages à objets parallèles.

Le projet a ensuite évolué avec la définition des Composants Actifs de Communication par LUC COURTRAI [CRGM92a]. Le but était de développer un environnement

pour faciliter la conception et la construction d'applications parallèles et surtout pour servir de support d'exécution de langages orientés objets parallèles.

Les principaux objectifs des CAC étaient :

- l'indépendance vis à vis des systèmes d'exploitation distribués ;
- la facilité d'adaptation à différentes architectures (multicomputers, réseaux de stations de travail, ...) ;
- la transparence de l'architecture ;
- une distribution statique du code et dynamique des activités.

Nous présenterons tout d'abord les Composants Actifs de Communication (CAC). Ce sont des entités actives et autonomes permettant la construction d'applications parallèles. Elles apportent une vision structurante des données et des activités. C'est le principe objet appliqué aux systèmes. Le CAC est une entité de programmation regroupant une activité (processus léger), un environnement local et une boîte aux lettres. Cette structure est facilement réalisable au dessus des fonctionnalités traditionnelles des systèmes d'exploitation.

Nous présenterons ensuite la notion de module permettant la distribution du code et des activités. Le module est le support pour la répartition des données sur les différents nœuds, mais aussi le support pour la répartition des activités.

Nous décrirons finalement les différentes implémentations du run-time PVC.

2.1.1 Les Composants Actifs de Communication

Les Composants Actifs de Communication définissent un grain unique de décomposition des applications parallèles en unités autonomes et concurrentes. Cette structure de base est d'un niveau plus évolué que le processus : elle intègre une activité, une boîte aux lettres et un environnement privé. La programmation et la manipulation de ces composants sont réalisées par l'utilisation d'une bibliothèque spécifique. Une application est définie par la spécification du comportement des différents CAC nécessaires et de la coopération des CAC, celle ci se fondant sur la communication asynchrone par envoi de messages.

L'environnement des Composants Actifs de Communication permet :

- la gestion d'un parallélisme massif: tout CAC possède une activité autonome et toute application est décrite uniquement à l'aide de CAC ;
- la structuration des données et des activités : tout CAC possède une activité et un environnement local ;
- une description des applications parallèles indépendamment des spécificités du système d'exploitation et de l'architecture cible.

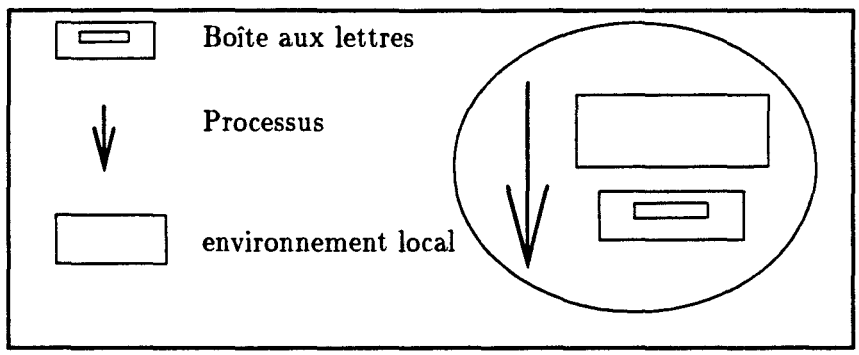

FIG. 2.1 - *structure d'un CAC*

Structure d'un CAC

La structure d'un Composant Actif de Communication est proche de celle d'un Acteur [Agh86]. Elle se compose d'un processus (partie traitement), d'une boîte aux lettres (partie communication), et d'un environnement local (mémoire locale du composant) (c.f. figure 2.1).

Le processus : chaque composant possède une activité autonome dont le code représente le comportement du composant. Cette activité est programmée par l'utilisateur en décrivant une fonction comportementale. Elle gère les données locales du composant et les communications avec les autres composants.

La boîte aux lettres : à chaque composant est associée une boîte aux lettres. Elle stocke tous les messages destinés au composant. Les communications sont asynchrones et s'effectuent en déposant un message dans la boîte aux lettres du composant destinataire. Le processus du composant peut explicitement extraire les messages déposés dans sa boîte pour les traiter. La boîte aux lettres est créée dès la création du composant et son nom identifie le composant dans le système. Les messages contenus dans la boîte d'un composant sont stockés dans l'ordre d'arrivée, mais les primitives de communication permettent au composant de filtrer et d'extraire les messages dans un ordre quelconque.

L'environnement local : l'environnement est créé dynamiquement par le processus du composant. Il regroupe toutes les données locales du composant. L'allocation mémoire s'effectue dans l'espace du processeur où évolue le CAC.

La désignation

L'environnement CAC requiert la désignation de deux entités distinctes : les comportements pour la création des CAC et les CAC eux mêmes pour autoriser la communication entre eux.

Les comportements : la création d'un CAC nécessite la désignation d'un comportement. La création sur un nœud quelconque impose une désignation globale

des comportements sur l'ensemble du système. La création doit respecter une contrainte, dite de localité : le code du comportement doit résider sur le nœud de création.

Les composants : La primitive de création d'un composant renvoie un nom global au système pour permettre la communication entre CAC. Ce nom est celui de la boîte aux lettres du composant créé. La primitive d'envoi de messages laisse le message au système qui est chargé de le déposer dans la boîte aux lettres du CAC destinataire : la connaissance de la localisation du CAC destinataire n'est pas nécessaire.

```
typedef struct {       /* structure decrivant les parametres */
  Component retour;    /* du comportement somme              */
  int borne_inf;
  int borne_sup;
} som_param;

#define B_SOM 1        /* nom global du comportement somme   */

void somme(Component me, som_param *arg)
{
 Mess_Ptr mes_rep1,mes_rep2;
 som_param parametres;
 int result;
 Component fils_droit, fils_gauche;

  if (arg->borne_inf == arg->borne_sup) {
    SendReply(me,arg->retour,sizeof(int),&(arg->borne_inf));
  } else {
    parametres.retour = me;
    parametres.borne_inf = arg->borne_inf;
    parametres.borne_sup = (arg->borne_inf+arg->borne_sup)/2;
    fils_gauche = NewComponent(me,B_SOM,VOID,sizeof(som_param),&parametres);
    parametres.borne_inf = ((arg->borne_inf+arg->borne_sup)/2)+1;
    parametres.borne_sup = arg->borne_sup;
    fils_droit = NewComponent(me,B_SOM,VOID,sizeof(som_param),&parametres);
    GetMessage(me,me,&mes_rep1);
    result = *(int *)GetFirstArg(mes_rep1,sizeof(int));
    GetMessage(me,me,&mes_rep2);
    result += *(int *)GetFirstArg(mes_rep2,sizeof(int));
    SendReply(me,arg->retour,sizeof(int),&result);
    FreeMessage(mes_rep1);FreeMessage(mes_rep2);
  }
  EndComponent(me);
}
```

FIG. 2.2 - *Exemple de fonction comportementale.*

La programmation

Le programmeur décrit une application par un ensemble de comportements modélisant les activités parallèles de son application. La programmation des fonctions comportementales est réalisée en langage C. Toutes les structures de contrôle du langage sont ainsi réutilisées. Les primitives du run-time CAC sont accessibles à travers une bibliothèque de fonctions.

La bibliothèque comprend les primitives d'accès aux services du système et les primitives de communication entre CAC. Des primitives de gestion de boîtes aux lettres sont également proposées au programmeur autorisant un CAC à posséder plusieurs boîtes aux lettres locales, ce qui facilite la modélisation et la programmation des applications parallèles.

Un exemple de fonction comportementale est présenté figure 2.2. Ce comportement permet de calculer par dichotomie la somme des entiers d'un intervalle. Cette méthode de calcul peut être facilement réalisée par les composants actifs de communication. Les CAC de comportement *B_SOM* sont les entités actives de calcul. La primitive *NewComponent()* crée un composant somme dont le processus exécute la fonction comportmentale *somme* décrite en langage C. L'intervalle de calcul est transmis à la primitive ainsi que l'adresse de retour qui est le nom de la boîte aux lettres où doit être envoyé le message de réponse. Un composant client doit lancer l'application. Pour cela, il crée un premier composant de type *somme*, attend un message de ce composant puis affiche le résultat.

2.1.2 Les modules

Nous avons introduit précédemment les CAC comme structure programmable pour la construction d'applications parallèles. Nous présentons maintenant le Module comme le grain logique de répartition du code et des activités. Au départ, un CAC est créé en désignant le code décrivant son comportement. Ce sont les « fonctions comportementales » des CAC utilisés dans l'application. Ces fonctions sont regroupées dans les différents modules utilisés dans l'application. Le code d'une application est ainsi partitionné en modules contenant chacun un sous-ensemble des fonctions comportementales. Cette distribution des comportements en modules peut être retardée jusqu'avant la phase d'exécution de l'application.

Un CAC sera donc créé « dans » un module. Durant la phase d'exécution, un module est instancié sur un des nœuds de la machine cible ; il possède alors le code de certains comportements et un sous ensemble des CAC exécutant ces comportements [1].

Structure d'un module

La mémoire d'un module est structurée en deux espaces (c.f. figure 2.3) :

- un espace code qui prend la forme d'une table des fonctions comportementales du module : cet espace contient plus exactement le code des comportements du module ;

1. Nous verrons plus loin que les CAC de même comportement peuvent être répartis sur plusieurs noeuds.

- un espace mémoire de CAC : chaque module définit une zone mémoire réservée aux CAC du module lors de l'exécution de l'application. La structure d'un CAC est entièrement allouée dans cette zone.

De plus, chaque module possède une activité spécifique gérant les créations de composants dans le module. Ce processus, appelé Composant Gestionnaire de Module (CGM), est implémenté par un CAC spécialisé. Il est chargé de traiter les « requêtes au module », en particulier les demandes de création de composant.

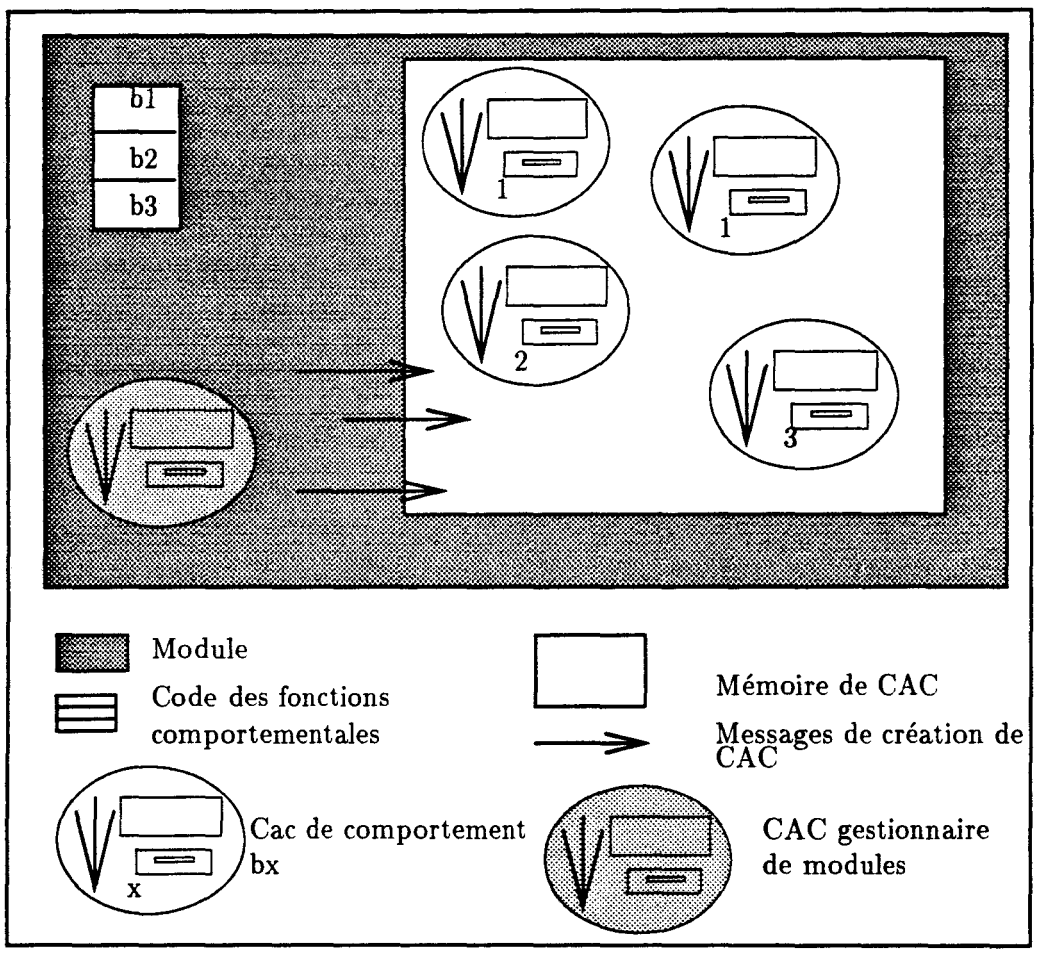

FIG. 2.3 - *structure d'un module*

Fonctionnalités d'un module

Les modules ont deux fonctionnalités :

- un module est une entité de partage de code : il regroupe un ensemble de fonctions comportementales désignées globalement ; ces noms globaux sont utilisés dans un

autre module pour la création à distance de composants ; l'environnement permet difficilement la migration du code à l'exécution et un module réside entièrement sur un nœud de la machine ;

- le module est aussi la structure d'accueil des CAC sur les différents nœuds de la machine cible ; chaque module regroupe un ensemble de CAC ayant chacun un comportement parmi ceux contenus dans le module ; un composant vit dans un module et ne peut migrer sur aucun autre.

Chaque module est nommé par un nom logique qui identifie l'ensemble des fonctions comportementales contenues dans le module. Ce nom logique permettra de manipuler le module lors de la phase d'instanciation sur un nœud.

Distribution des fonctions comportementales

Les critères de regroupement des fonctions comportementales sont libres :

- la phase de répartition des fonctions comportementale peut être retardée jusqu'à l'exécution de l'application pour prendre en compte les contraintes de l'architecture cible ; cette distribution peut être le résultat d'un outil de répartition de code et d'équilibrage de charge ayant pour but de répartir au mieux les CAC ; FRED HEMERY étudie actuellement les problèmes de répartition de charge au sein de l'environnement CAC [HG93] ; cette opération peut être cachée au programmeur ; le module est donc un outil de répartition ;

- le groupement des fonctions peut être lié à leur programmation ; les modules peuvent explicitement être le résultat d'une programmation modulaire au sens classique, comme par exemple une bibliothèque de fonctions comportementales livrée sous forme d'un module compilé ; le module est considéré ici comme un outil de développement.

La distribution des fonctions comportementales dans les modules offre de multiples possibilités. Si un module contient un et un seul comportement, il peut être assimilé à la notion de « classe » de CAC. Mais la notion de module est plus souple que la notion de classe : un module peut contenir plusieurs fonctions comportementales et plusieurs modules peuvent contenir la même fonction comportementale.

Le module est une entité de distribution de code et des objets à l'exécution. La distribution peut être facilement modifiée par le programmeur sans changer le source de l'application.

L'environnement propose un outil supplémentaire de répartition du code : la duplication de modules. Le programmeur découpe une application en un ensemble de modules puis définit le nombre de duplications de chacun d'eux. Cette phase est indépendante de la conception, c'est un paramètre de l'exécution. Les modules seront ainsi dupliqués sur des nœuds différents de la machine. Ce mécanisme offre plusieurs avantages :

- la création des composants sera répartie sur les différents nœuds de la machine abritant les modules dupliqués déchargeant ainsi la mémoire de chaque nœud ;

- le parallélisme réel de l'application est augmenté : sans duplication, les CAC de même comportement sont localisés dans le même module, donc sur le même nœud.

Les modules à l'exécution

Après la compilation des modules, le programmeur peut lui même définir la localisation des modules sur les nœuds de la machine. Il décrit un réseau de modules pour les répartir sur les nœuds de la machine. Cette phase est indépendante de la phase de programmation et permet de prendre en compte les contraintes matérielles de la machine. Il peut aussi laisser cette tâche au système.

Les entrées/sorties

Les entrées/sorties sont réalisées par des CAC spécialisés.

Un CAC réalisant les entrées/sorties standard est créé dans chaque module. Les autres CAC, par l'intermédiaire des primitives d'entrées/sorties décrites en annexe A, envoient un message au CAC spécialisé qui s'occupera de réaliser l'opération. Grâce aux primitives `LockOutput` et `UnLockOutput`, un CAC peut se réserver l'usage du CAC d'entrées/sorties, évitant l'enchevêtrement des messages sur la sortie standard.

Les fichiers sont utilisables dans l'environnement CAC à travers un CAC spécialisé. L'ouverture d'un fichier correspondra à la création d'un CAC chargé de réaliser les opérations d'entrées/sorties sur ce fichier. L'appel aux primitives d'entrées/sorties sur les fichiers (c.f. annexe A) provoque l'envoi d'un message vers le CAC associé au fichier qui réalisera l'opération demandée.

2.1.3 Implémentation

Deux implémentations du run-time CAC ont été effectuées. Une application CAC n'a besoin que d'une recompilation pour s'exécuter sur le MultiCluster ou sur le réseau de stations de travail.

Transputer

Une première version du run-time a été implémentée sur une machine parallèle à base de transputers, le MultiCluster II de Parsytec [Par90], équipée de 32 transputers T800 sous le système d'exploitation distribué Helios [PSL91]. Sur cette architecture, un module est implémenté par une tâche Helios (processus lourd) et l'activité des CAC par un processus Helios (processus léger). Les processus Helios sont créés dans une tâche et partagent l'espace d'adressage de la tâche. La distribution des modules est réalisée par l'intermédiaire du Component Description Language (CDL) [PSL91], outil du système d'exploitation Helios.

Réseau de stations

Le run-time CAC a aussi été porté sur réseau de stations de travail SUN sous le système d'exploitation SUNOS 4.1, utilisant la bibliothèque `lwp` [Sun88] et les communications par sockets. Un module est ici implémenté par un processus Unix et l'activité d'un CAC par un processus léger de la bibliothèque `lwp`. La distribution des modules est également réalisée par l'intermédiaire d'un programme `cdl` qui a été porté sous Unix : un démon sur chaque site est chargé de créer les processus Unix et les liens de communication (socket TCP/IP en mode datagram). Cette implémentation est décrite plus précisément dans [CRD+92].

2.1.4 Conclusion

Nous avons présenté les Composants Actifs de Communication, environnement de programmation simplifiant la modélisation et la conception des applications parallèles. La figure 2.4 présente la structuration des applications dans l'environnement CAC. J'ai participé à la conception et à la réalisation de la version 2.0 de l'environnement CAC [RCGM92] qui est actuellement utilisé dans plusieurs cadres.

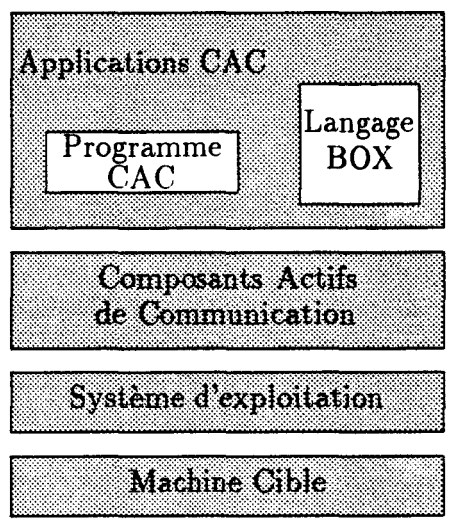

FIG. 2.4 - *Structuration des applications dans l'environnement CAC*

FRED HEMERY étudie la répartition dynamique de la charge en utilisant les CAC comme plate-forme d'expérimentation. Plusieurs de ses stratégies de répartition sont actuellement intégrées au run-time.

Cédric Dumoulin a réalisé pour le run-time CAC un garbage collector distribué de CAC [DRM93]. L'algorithme, dérivé de celui de Kafura [KW91], consiste en l'utilisation de garbage collector locaux à chaque module coopérant pour récupérer les composants inactifs.

Luc Courtrai dans sa thèse a étudié plusieurs modèles de langages orientés objets parallèles implémentés à l'aide des CAC [Cou92, CRGM92b]. Un langage orienté objets parallèle a été défini, le langage Box (c.f. 3.1), dont le modèle d' exécution est basé sur celui des CAC. La version actuelle du compilateur génère du « code CAC » .

Mais l'environnement CAC est également utilisé par d'autres équipes du laboratoire :

- un algorithme de lancer de rayons a été conçu et développé en CAC par l'équipe graphique du LIFL ;
- un simulateur parallèle pour un modèle d'exécution de langage fonctionnel parallèle par l'équipe PALOMA.

Le run-time CAC est en cours de portage sur de nouvelles plate-formes en particulier en utilisant le standard de développement PVM [Sun90, DGMS93].

2.2 La réexécution de programmes CAC

Le modèle CAC possède plusieurs particularités influant sur la conception du mécanisme de réexécution :

- le modèle est conçu pour des machines à mémoire distribuée : les processus coopèrent uniquement par envoi de messages et non par variables partagées ;
- l'envoi de message est asynchrone ;
- le modèle autorise la création dynamique de nombreux processus légers.

Le non déterminisme des applications CAC est donc généré par la communication par message. Seuls les événements liés à la communication sont donc à considérer pour une réexécution. La création dynamique de processus et le grand nombre éventuel de ces créations nous empêchent de générer une trace par processus et obligent à conserver l'identité du CAC responsable de la génération de l'événement dans la trace.

Notre mécanisme de réexécution est basé sur une trace d'événements et est de type « dirigée par le contrôle », ceci afin d'éviter la production d'une trace trop importante et donc de perturber l'exécution. Nous présentons maintenant les deux phases du mécanisme de réexécution : l'enregistrement de la trace et ensuite la phase de réexécution elle même.

2.2.1 La phase d'enregistrement

Nous nous sommes tournés vers une méthode logicielle pour la récolte et la sauvegarde d'événements. En effet, nous ne possédons pas le matériel nécessaire et la réalisation d'un tel matériel ne fait pas partie de notre étude. Le coût d'une telle approche est également très important. De plus, une récolte matérielle produit des événements de bas niveau difficiles à utiliser directement pour une réexécution.

Format de la trace

Trois types d'événements sont nécessaires pour lever le non déterminisme lors de la réexécution. Tout d'abord, deux événements concernant la communication par message qui, selon la terminologie de E. Leu, peuvent être qualifiés d'implicites :

- `MESSARRIVED` : cet événement correspond à l'arrivée effective d'un message dans la boîte aux lettres du CAC destinataire ;
- `MESSDEPARTURE` : cet événement correspond au départ effectif d'un message destiné à un CAC d'un autre module.

Un dernier type d'événement a été défini :

- `EXPLICIT` : cet événement, généré explicitement par l'activité du composant, correspond à l'appel de certaines primitives du run-time comme la consultation d'une boîte aux lettres, la création d'un nouveau CAC ... Ces événements participent au non-déterminisme de l'application : il faut pouvoir ordonner les accès effectués par un CAC à une boîte aux lettres avec les arrivées de message : le CAC doit examiner la boîte aux lettres dans le même état que lors de l'exécution initiale.

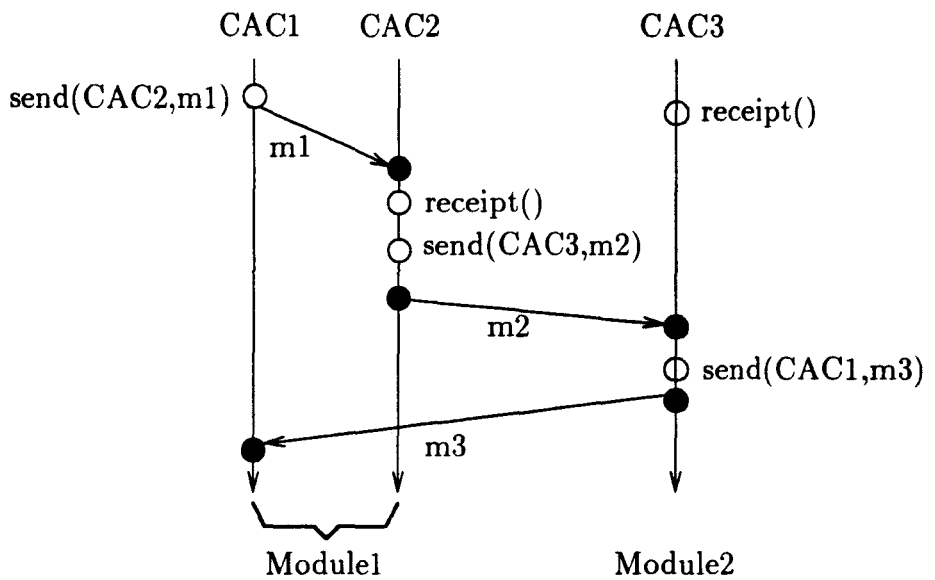

FIG. 2.5 - *Exemple d'exécution d'un programme CAC*

Module1		Module2	
EXPLICIT	CAC1	EXPLICIT	CAC3
MESSARRIVED	CAC1	MESSARRIVED	CAC2
EXPLICIT	CAC2	EXPLICIT	CAC3
EXPLICIT	CAC2	MESSDEPARTURE	CAC2
MESSDEPARTURE	CAC2		
MESSARRIVED	CAC3		

FIG. 2.6 - *Traces du programme de la figure 2.5*

Nous produisons un fichier de trace par module. Chaque élément de la trace a le format suivant : `TYPE_EVENEMENT, CAC_ID`. Voici la signification de `CAC_ID` pour chaque type d'événement :

- `MESSARRIVED` : `CAC_ID` est le nom du CAC expéditeur du message ;
- `MESSDEPARTURE` : `CAC_ID` est le nom du CAC expéditeur du message ;
- `EXPLICIT` : `CAC_ID` est le nom du CAC générateur de l'événement.

La figure 2.5 présente un exemple d'exécution d'un programme CAC. `CAC1` et `CAC2` sont deux composants du même module, `module1`, et `CAC3` est un composant du `module2`. Les événements explicites ont été représentés par un disque blanc. Le fichier de trace généré pour chaque module est présenté à la figure 2.6. L'appel `Send(CAC2,m1)` ne génère pas d'événement de type `MESSDEPARTURE`, puisque CAC1 et CAC2 appartiennent au même module, au contraire de `Send(CAC3,m2)`.

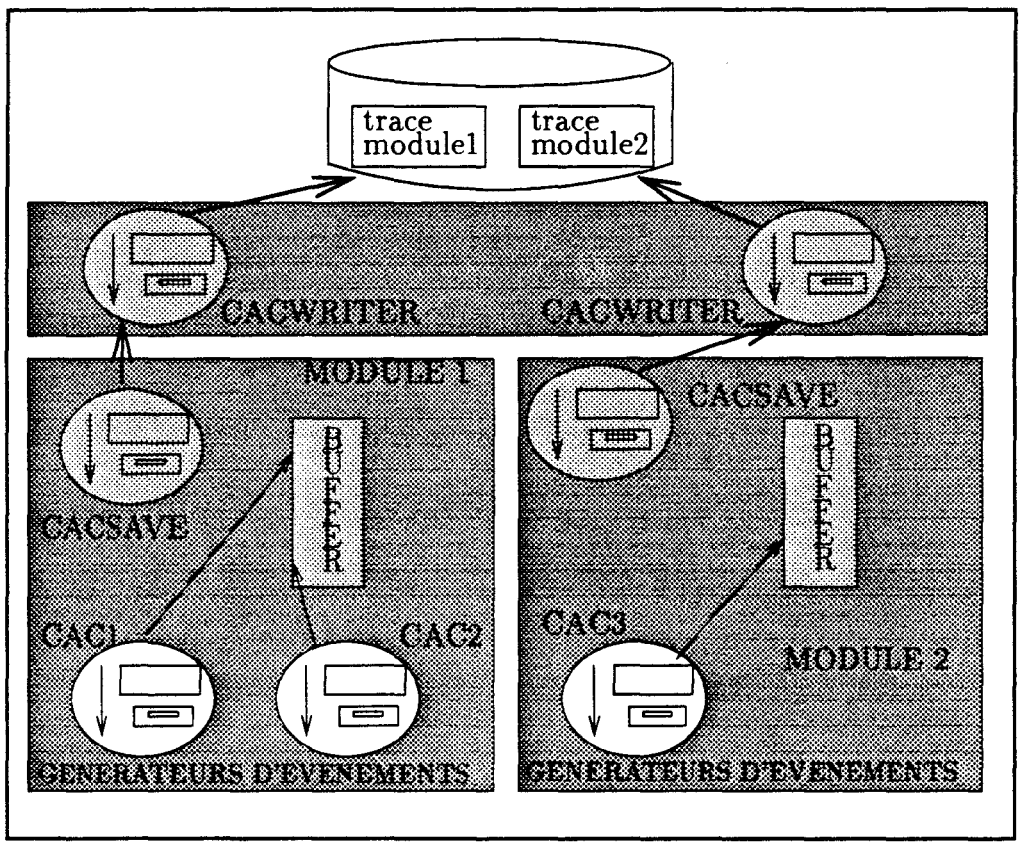

FIG. 2.7 - *Architecture du mécanisme de sauvegarde*

Génération de la trace

L'application n'est en rien modifiée pour générer les traces nécessaires à la réexécution. Un paramètre à l'exécution permet de rendre actif le mécanisme de sauvegarde. Le nom du fichier de trace d'un module est composé du nom de l'exécutable concaténé avec son nom dynamique dans l'application suivi de l'extension « `.sauv` ».

Dans le run-time CAC, la primitive `StoreEvent(type,cacid)` est appelée à chaque génération d'événement. Cette primitive sauvegarde en mémoire dans un buffer l'occurrence de l'événement. La sauvegarde en mémoire par l'intermédiaire d'un buffer évite un nombre de messages trop important, ceci pour diminuer la perturbation de l'application. Le buffer mémoire est donc partagé par les CAC générateurs d'événements (les producteurs) et par le CAC_{SAVE} (consommateur) sauvegardant le buffer. Mais ce mécanisme impose une synchronisation supplémentaire pour protéger l'accès au buffer. Un CAC spécialisé, CAC_{SAVE}, est chargé d'envoyer par un seul message le contenu du buffer, lorsqu'il est plein, à un autre CAC spécialisé, CAC_{WRITER}, chargé lui de la sauvegarde effective sur disque. Un CAC_{WRITER} est associé à chaque module et tous les CAC_{WRITER} sont situés dans un module localisé sur un nœud de la machine pour éviter une perturbation trop important de l'exécution. Cette disposition est intéressante sur

la machine à base de transputers qui ne possède qu'un point d'entrée vers le serveur de fichiers. Les modules de l'application effectuent un envoi de message par les liens de communication des transputers et ne sont pas ralentis par l'accès au serveur de fichiers. S'il existait deux nœuds disposant de liens vers le serveur de fichiers, on pourrait imaginer de répartir les CAC_{WRITER} sur ces deux nœuds. Une autre disposition serait plus intéressante pour le réseau de stations SUN qui dispose du système de fichiers distribué NFS. Dans ce cas, le CAC_{SAVE} de chaque module pourrait sauvegarder directement le buffer sur disque. Mais conserver la configuration avec des CAC_{WRITER} n'introduit aucune pénalisation.

La figure 2.7 présente l'architecture du mécanisme de sauvegarde. La sauvegarde du buffer d'événements présent en mémoire est forcée en plusieurs occasions :

- quand aucun événement n'a été généré depuis un certain laps de temps ;
- quand un signal est propagé dans le module, ceci provoque également l'arrêt de l'application ;
- quand l'application est terminée.

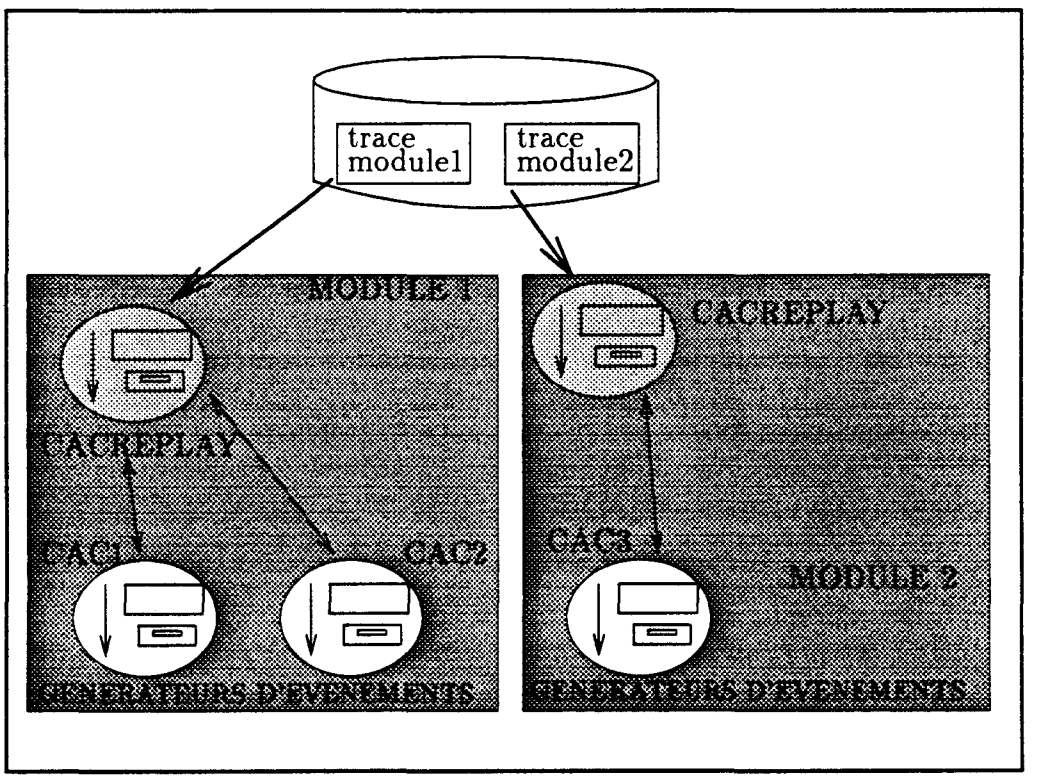

FIG. 2.8 - *Architecture du mécanisme de réexécution*

2.2.2 Réexécution totale

Notre mécanisme de réexécution consiste en une exécution réelle du programme dirigée par les traces générées lors de l'exécution initiale : ces traces servent à lever le non-déterminisme de l'exécution. Le contrôle de la réexécution est distribué dans chaque module. Un CAC spécialisé, CAC_{REPLAY}, est chargé de préserver l'ordre des événements observé durant l'exécution initiale.

Contrôle

Dans le run-time CAC, la primitive `BeforeEvent(type,cacid)`, est appelée avant la génération de chaque événement pendant la phase de réexécution. Cette primitive envoie un message au CAC_{REPLAY} correspondant à une demande d'autorisation de génération d'événement. Le CAC_{REPLAY} donne une autorisation en filtrant sa boîte aux lettres pour trouver la demande correspondant à l'événement courant dans la trace. CAC_{REPLAY} attend ensuite la génération de l'événement avant de donner une nouvelle autorisation. La primitive `AfterEvent(type,cacid)`, après chaque génération d'événement, est appelée afin d'informer le CAC_{REPLAY}. Ainsi, un événement n'est généré qu'après l'occurrence des événements l'ayant précédé lors de l'exécution initiale. La figure 2.8 présente l'architecture du mécanisme de réexécution.

L'algorithme du CAC_{REPLAY} peut être schématisé comme suit :

```
tant que non fin de trace faire
  evt := prochain evenement dans la trace
  attente du message de demande d'autorisation de evt
  donner l'autorisation
  attente du message de terminaison de evt
fait
```

Le CAC_{REPLAY} utilise la possibilité de filtrage de messages dans une boîte aux lettres offerte par le run-time CAC. En effet, les messages ne sont pas uniquement accessibles en mode FIFO, mais tout message d'une boîte aux lettres peut être examiné et extrait quelle que soit sa position. De ce fait, les messages de demande d'autorisation sont extraits de la boîte aux lettres du CAC_{REPLAY} dans l'ordre de la trace. Le CAC_{REPLAY} attendant la fin d'un événement pour autoriser la génération du suivant, les événements d'un module suivent l'ordre de la trace.

Exemple

La figure 2.9 présente un exemple du fonctionnement du contrôle de la réexécution. Deux messages destinés au même CAC arrivent de deux modules différents (1). Deux demandes d'autorisations sont émises vers le CAC_{REPLAY} (2). Celui ci, l'événement courant dans la trace étant `MESSARRIVED M1`, donne l'autorisation de déposer le message `m1` dans la boîte aux lettres du CAC destinataire (3). Le message est alors déposé (4). Puis notification est faite au CAC_{REPLAY} que le message a été déposé (5). Le CAC_{REPLAY} peut alors autoriser l'événement correspondant à l'événement suivant dans la trace : `MESSARRIVED M2`.

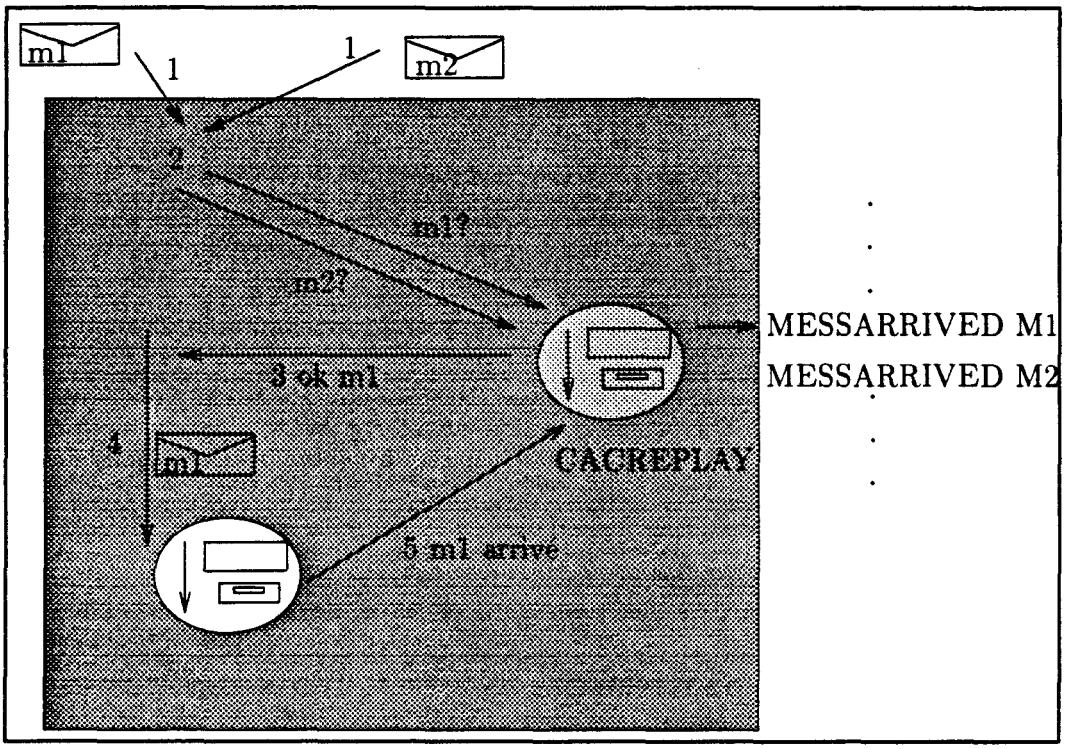

FIG. 2.9 - *Contrôle de la réexécution*

2.2.3 Réexécution partielle

Le mécanisme que nous avons décrit impose une réexécution totale de l'application et donc une utilisation complète de la machine. Nous avons voulu rendre possible une réexécution partielle des applications CAC. Par réexécution partielle, nous entendons la réexécution complète d'un ou plusieurs modules. La réexécution partielle offre plusieurs avantages :

- la réexécution est plus rapide ;
- la quantité d'information à examiner par le programmeur est moins importante ;
- la machine parallèle n'est pas monopolisée.

Notre mécanisme de réexécution partielle n'est possible qu'après une phase de réexécution totale comme le montre la figure 2.10. Les traces générées pour la réexécution partielle, traces de taille importante, le sont lors d'une réexécution totale. Cette méthode évite la récolte des traces importantes pendant une exécution ce qui perturberait beaucoup trop l'application. La récolte pendant une réexécution est à perturbation nulle.

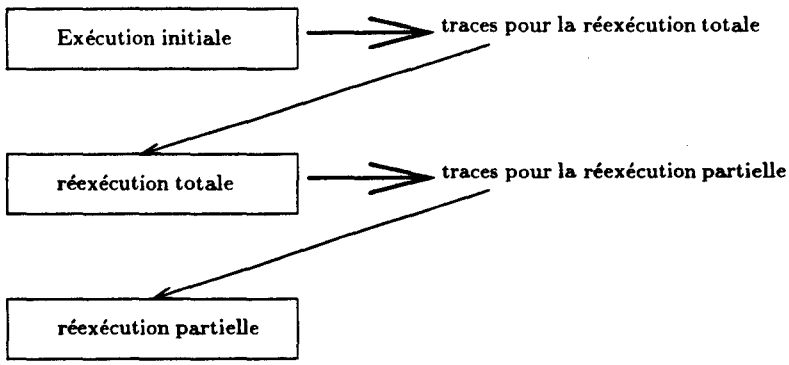

FIG. 2.10 - *Génération de traces pour une réexécution partielle*

Phase de sauvegarde

Durant la réexécution partielle, tous les modules de l'exécution initiale ne seront pas présents. Des informations supplémentaires pour chaque module doivent être sauvegardées pour contrôler la réexécution :

- tous les messages provenant d'autres modules doivent être sauvegardés ;
- un module doit connaître les modules présents lors de l'exécution initiale et quels comportements y étaient définis ; dans une exécution normale, ceci est réalisé par une phase dynamique d'initialisation : le résultat de cette initialisation doit donc être sauvegardé. En effet, le module où doit être créé un comportement est choisi dynamiquement dans le module demandant la création.

Ces informations seront sauvegardées : on obtient un fichier par module dont le nom sera composé du nom de l'exécutable concaténé avec le nom dynamique du module suivi de l'extension « `.part` ».

Pour éviter toute perturbation de l'exécution, les traces nécessaires pour la réexécution partielle (résultats de l'initialisation et messages provenant d'autres modules) sont générées durant une réexécution totale du programme. La figure 2.11 présente l'architecture lors de la sauvegarde pour une réexécution partielle.

Phase de réexécution

Les deux événements `MESSARRIVED` et `MESSDEPARTURE` ne sont pas traités de la même façon lors d'une réexécution partielle et lors d'une réexécution totale :

- `MESSARRIVED` : nous sommes dans le cas où CAC_{REPLAY} attend un événement (`MESSARRIVED`, `CACID`) ; si le message doit provenir d'un autre module, il le lit depuis le fichier de trace de messages ; si le module du CAC expéditeur n'est pas présent lors de la réexécution partielle, CAC_{REPLAY} dépose le message dans la boîte aux lettres du CAC destinataire ; sinon, l'événement est traité comme lors d'une réexécution totale ;

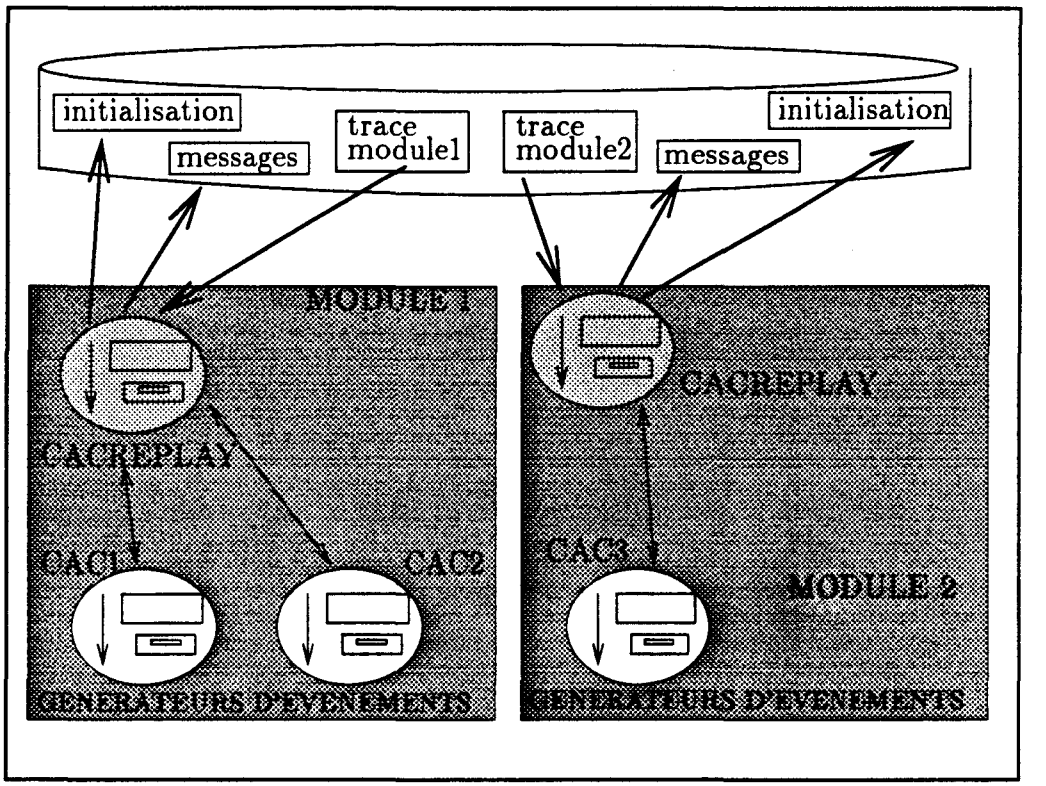

FIG. 2.11 - *Architecture du mécanisme de sauvegarde pour réexécution partielle*

- **MESSDEPARTURE**: nous sommes dans le cas où CAC_{REPLAY} a donné une autorisation pour générer un événement (**MESSDEPARTURE**, **CACID**); la primitive d'envoi de message doit tester la présence du module où réside le CAC destinataire avant d'envoyer le message.

La figure 2.12 présente l'architecture du mécanisme de réexécution partielle.

2.2.4 Problèmes particuliers

Plusieurs problèmes particuliers doivent être pris en compte lors de réexécution, car ils ne peuvent être résolus directement par le mécanisme décrit précédemment; ce sont:

- la répartition dynamique des composants;
- les entrées/sorties;
- l'utilisation par le programme de valeurs non reproductibles.

La méthode de répartition dynamique des composants doit être prise en compte: le choix du module pour la création d'un nouveau composant doit être le même lors de la réexécution et lors de l'exécution initiale. La méthode actuelle du run-time a été

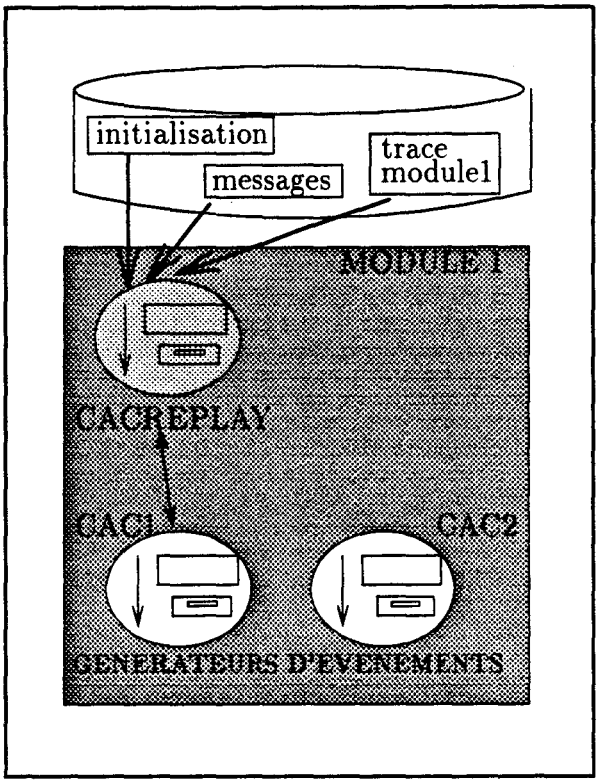

FIG. 2.12 - *Architecture du mécanisme de réexécution partielle*

implémentée par FRED HEMERY. Il utilise plusieurs stratégies, dont certaines utilisent un CAC spécialisé. La réexécution de ce CAC est prise en charge par le mécanisme décrit ci dessus. La sauvegarde d'événements de type `EXPLICIT` a du être rajoutée dans certaines primitives utilisées par la répartition.

Pour obtenir une réexécution conforme à l'exécution initiale, les entrées extérieures doivent être sauvegardées. Les entrées étant réalisées par un CAC spécialisé, le mécanisme de réexécution, décrit ci-dessus, reproduit les accès à l'entrée standard dans le même ordre que l'exécution initiale. Mais le contenu de l'entrée est sauvegardé dans un fichier spécifique. Ce fichier est lu durant la phase de réexécution à la place de l'entrée standard.

Le programmeur peut concevoir des composants dont le comportement non-déterministe n'est plus du au système mais à leur programmation par l'utilisation d'un générateur de nombres aléatoires, de l'horloge système, de valeurs provenant d'un capteur, ... Ces valeurs ont besoin d'être sauvegardées pour obtenir une réexécution conforme. C'est pourquoi le run-time CAC fournit des primitives pour sauvegarder et retrouver dans une phase de réexécution des données non reproductibles. Ces primitives sont détaillées en annexe A. Voici celles pour la sauvegarde:

```
int open_file_save(char *extent);
int write_file_save(int f, int size, char *data);
```

```
int close_file_save(int f, int size, char *data);
int close_file_save(int f);
```

Le fichier aura pour nom le nom de l'exécutable concaténé avec le nom dynamique du module suivi de l'extension fournie par le programmeur.

Facilités de mise au point

Quelleques facilités de mise au point ont été ajoutées au mécanisme de réexécution. L'utilisateur a la possibilité de suspendre l'application. Une fois l'application suspendue, il a la possibilité de lister :

- les boîtes aux lettres d'un module, indiquant également les CAC en attente sur celle-ci ;
- le contenu des boîtes aux lettres ;
- les messages en partance d'un module ;
- les messages arrivant dans un module.

2.2.5 Mesures

Evaluation de la perturbation

L'évaluation de la perturbation due à la sauvegarde d'événements sur le comportement de l'exécution est mesurée en temps d'exécution. La taille du buffer utilisé est de 16 koctets. Nous examinerons deux applications de référence:

- l'application somme ;
- l'application car_wash.

Les deux applications sont présentées dans leur intégralité annexe B. L'application *somme* calcule en parallèle la somme des entiers de 1 à n. Un seul comportement est défini : le comportement *somme*. Celui-ci reçoit en paramètre l'intervalle d'entiers dont il doit calculer la somme. Si l'intervalle est de cardinalité supérieure à 1, le CAC crée deux nouveaux composants de même comportement chargés de calculer chacun la somme pour une moitié de l'intervalle. Le CAC attend les deux réponses, en effectue la somme et envoie le résultat à son créateur. Cette application crée donc dynamiquement un grand nombre de CAC: 2n-1. Nous avons mesuré le temps d'exécution de la somme des 1000 premiers entiers calculée par 1999 CAC et la somme des 2000 premiers entiers calculée par 3999 CAC. Les CAC sont répartis sur 7 modules distribués sur 7 sites différents dans les deux cas.

L'application *car_wash* est la simulation d'une laverie automatique de voitures. Les voitures se déplacent sur une route formée de tronçons jusqu'à la station de lavage. Un contrôleur de la station distribue les voitures sur l'un des postes de lavage. Les CAC sont ici créés statiquement au départ de l'application. Le programme fait appel essentiellement à la communication par message. L'exécution mesurée comporte 30 voitures, 30 tronçons et une station disposant de 4 postes de lavage. Les voitures sont

réparties sur deux modules, ainsi que les postes de lavages et les tronçons. Ces 8 modules sont répartis sur 8 sites différents.

Les programmes que nous avons choisis pour effectuer les mesures n'effectuent aucun calcul. Ils n'effectuent que des communications ou des créations de CAC et sont donc particulièrement sensibles à la perturbation induite par la sauvegarde de l'historique. Quelles sont les actions pouvant entraîner une perturbation de l'exécution pendant une phase de sauvegarde? L'écriture de l'événement dans le buffer est négligeable. Par contre, la synchronisation nécessaire à la protection du buffer vis à vis des accès concurrents peut conduire à une modification de l'ordre des événements. La deuxième cause de perturbation potentielle est l'envoi du buffer plein par message, la perturbation provoquée par la charge du réseau de communication et aussi par le traitement sur le site.

La durée d'une exécution est calculée par la différence entre l'horloge système au début de l'application et à la fin de l'application. Sur le MultiCluster II, aucune perturbation extérieure ne peut modifier les mesures : un réseau de transputer est alloué pour un seul utilisateur. Par contre, sur le réseau de stations de travail, plusieurs paramètres peuvent perturber les mesures : la charge des stations de travail au moment des mesures, ainsi que la charge du réseau ethernet, peut fortement évoluer d'une exécution à l'autre.

TAB. 2.1 - *Evaluation de la perturbation due à la sauvegarde sur le MultiCluster*

Applications	Temps d'exécution en 1/100 s		Perturbation en %	Taille de la trace en octets	
	Normale	Sauvegarde		Total	Moyenne/module
Somme de 1000	2088	2116	1,37	243504	34786
Somme de 2000	5868	5945	1,31	473672	67667
CarWash	1557	1671	7,32	144812	18101

TAB. 2.2 - *Modules ayant généré la plus grande trace*

Applications	Temps d'exécution en 1/100 s		Perturbation en %	Module ayant généré la plus grande trace	
	Normale	Sauvegarde		Taille en octets	Nb événements par 1/100 s
Somme de 1000	2088	2116	1,37	41608	4,91
CarWash	1557	1671	7,32	45716	6,83

Le tableau 2.1 nous donne les résultats pour l'exécution sur le MultiCluster II. Pour l'application *somme*, la perturbation est de 1,37 % pour la somme des 1000 premiers entiers et 1,31 % pour la somme des 2000 premiers entiers. Pour ce type d'application, assez régulière, le fait d'augmenter le travail à réaliser ne modifie pas la perturbation.

Pour l'application *car_wash*, la perturbation est de 7,32 %. Les résultats obtenus sont largement en dessous de la limite de 10 % considérée comme acceptable. Ils montrent que la perturbation dépend de l'application. Mais ils peuvent apparaître comme contradictoires : l'application *somme* pour 1000 entiers produit une trace beaucoup plus importante que l'application *car_wash* par unité de temps, et pourtant subit une plus petite perturbation. Ceci s'explique avec les résultats du tableau 2.2.

L'application *somme* est assez « régulière » : les CAC *somme* sont créés cycliquement dans chaque module, le coût de la sauvegarde est donc parfaitement distribué. Par contre, deux modules sont beaucoup plus sollicités dans l'application *car_wash* : le module abritant les CAC tronçons reçoivent des messages continuellement de la part de toutes les voitures. Nous avons donc calculé le nombre d'événements générés par centième de seconde pour le module ayant généré la plus grande trace. Pour l'application *somme*, ce module a une trace proche de la moyenne alors que pour l'application *car_wash* il possède une trace pratiquement trois fois plus importante que la moyenne. Le nombre d'événements est plus important pour le module de l'application *car_wash* que pour le module de l'application *somme* : la perturbation est donc plus importante.

TAB. 2.3 - *Evaluation de la perturbation due à la sauvegarde sur le réseau de stations*

Applications	Temps d'exécution en 1/100 s		Perturbation en %	Taille de la trace en octets	
	Normale	Sauvegarde		Total	Moyenne/module
Somme	2717	2983	8,68	242224	34603
CarWash	5217	5364	10,34	145432	18179

Le tableau 2.3 présente la perturbation de l'exécution sur le réseau de stations de travail. Nous obtenons des résultats moins bons que sur transputer mais qui restent encore acceptables (8,64% et 10,34%). La différence de résultats entre les deux architectures est due à la différence de performance des réseaux d'interconnexion. L'augmentation de la perturbation pour l'application *car_wash* est moins importante que pour l'application *somme*. En effet, la taille des traces est moins importante, et, de ce fait, le nombre de messages nécessaires pour sauvegarder le buffer est moins important.

TAB. 2.4 - *Tailles des traces pour la réexécution partielle*

Applications	Taille réexécution totale en en octets		Taille réexécution partielle en octets	
	Total	Moyenne/module	Total	Moyenne/module
Somme de 1000	242224	34603	283428	40489
CarWash	145432	18179	232252	29031

Taille des traces pour la réexécution partielle

La taille des traces pour la réexécution partielle est relativement importante puisque le contenu des messages provenant de l'extérieur est sauvegardé pour chaque module.

La différence avec les traces nécessaires à la réexécution totale est de 17% d'augmentation pour l'application *somme*, qui n'est pas très communiquante, et de 59% pour l'application *car_wash*, qui ne fait que communiquer. Signalons que, pour une technique de réexécution dirigée par les données, les traces auraient été encore plus importante: non seulement il est nécessaire de sauvegarder tous les messages provenant de l'extérieur, mais il faut également sauvegarder des événements internes aux modules et l'information associée.

Durée de la réexécution

Le tableau 2.5 détaille les mesures pour la réexécution totale de l'application *somme* et l'application *car_wash*. La durée de la réexécution est 4,6 fois plus importante que celle de l'exécution pour l'application *somme* et 6,6 fois pour l'application *car_wash*. La différence entre les deux applications est due à la même raison que la différence de perturbation.

Nous proposons un mécanisme de réexécution partielle de l'application. L'un de ses avantages est de diminuer la durée de la réexécution. Pour l'application *somme*, nous avons réexécuté le module contenant le premier CAC somme créé, celui qui fournit le résultat. La réexécution de ce seul module est un peu plus rapide qu'une réexécution totale. Mais le travail effectué par chaque module est à peu près équivalent : le gain sur la durée de la réexécution ne peut être important. Nous avons réexécuté, pour l'application *car_wash*, un module contenant des composants de type *voiture*. Ici, la durée de la réexécution partielle est presque deux fois moindre que celle de la réexécution totale. Ceci s'explique par le fait que les modules *Tronçons*, qui effectuent la plus grande part du travail, ne sont plus présents lors de la réexécution partielle.

TAB. 2.5 - *Durée de la réexécution*

Applications	Temps d'exécution en 1/100 s		
	Exécution initiale	Réexécution totale	Réexécution partielle d'un module
Somme de 1000	2088	9663 (x 4,6)	8666 (x 4,1)
CarWash	1557	10245 (x 6,6)	6019 (x 3,8)

2.3 Conclusion

Le mécanisme de réexécution, présenté dans ce chapitre, répond au problème soulevé par le non-déterminisme pour la mise au point des applications parallèles. L'obtention d'exécutions au comportement reproductible permet l'utilisation de la technique de mise au point cyclique (cyclic debugging).

Le choix d'une réexécution dirigée par le contrôle a permis de minimiser la perturbation sur le comportement de l'application lors de la phase de sauvegarde: la section précédente a montré que la perturbation, sur les deux architectures utilisées, restait dans la limite des 10% acceptables.

Notre mécanisme de réexécution est automatique: il ne demande aucune instrumentation particulière de la part du programmeur, ni aucune modification du code, pour la phase de sauvegarde et pour les différentes réexécutions.

Mais la réexécution dirigée par le contrôle conduit à une réexécution totale de l'application, ce qui peut être un inconvénient important si la durée de l'exécution est grande. Dans ce cas, le programmeur peut utiliser le mécanisme de réexécution partielle réduisant ainsi considérablement la durée de la réexécution.

Le développement de la réexécution a été grandement facilité par le modèle CAC. L'uniformité du modèle a permis de réduire le nombre de types d'événements à sauvegarder. De plus, le mécanisme entier (sauvegarde et contrôle de la réexécution) a pu être implémenté par des CAC.

Chapitre 3

La mise au point des programmes BOX

LE LANGAGE BOX est un langage orienté objet parallèle dont le modèle d'exécution est basé sur celui des CAC. Nous proposons dans ce chapitre un outil de mise au point des programmes BOX : des points d'arrêt distribués, utilisés au dessus du mécanisme de réexécution présenté précédemment.

La première partie de ce chapitre présentera le langage BOX et ses particularités. Puis nous expliquerons les modifications introduite dans le run-time pour réexécuter les programmes BOX. Finalement, nous présenterons les points d'arrêt distribués et leur utilisation pour la mise au point des programmes BOX. Les points d'arrêt sont associés à des événements composés dont nous détaillerons l'implémentation et la détection.

3.1 Le langage BOX

Le langage BOX est l'outil principal de l'environnement pour l'exploitation des architectures décentralisées dont la réalisation est le but de l'équipe. C'est un langage objet qui inclut des caractéristiques nouvelles pour prendre en compte le parallélisme et la distribution. Il permet d'utiliser les différents paradigmes de la programmation parallèle à objets.

3.1.1 Description générale

Langage à objets fragmentés

Le langage BOX utilise deux classes de composants élémentaires pour structurer et distribuer les applications : les fragments et les objets.

- **les fragments :** il s'agit d'objets actifs, des entités munies d'une activité propre ; ils sont instances d'une classe de fragments définissant des comportements (l'ac-

tivité du fragment exécutera un de ces comportements) et des attributs (l'environnement d'exécution) ; c'est le grain de décomposition des activités et des données ;

- **les objets :** ce sont des objets passifs, ils correspondent aux objets classiques des langages à objets ; ils sont instances d'une classe d'objets définissant des attributs et des procédures ; ils sont utilisés pour structurer l'ensemble des fragments en abstractions réutilisables et pour représenter des ressources partagées.

Tout attribut, d'un fragment ou d'un objet, peut référencer un objet ou un fragment. Le langage BOX permet ainsi de définir des « objets actifs complexes » mixant des objets et des fragments.

La communication

Le langage BOX permet deux types de communication : l'envoi de message et l'appel de procédure.

- **l'envoi de message asynchrone :** le langage autorise l'utilisation de boîtes aux lettres créées dynamiquement par les objets et les fragments, un fragment étant muni d'une boîte aux lettres implicite ; le fragment est en fait le reflet au niveau du langage de la notion de CAC ;

- **l'appel de procédure :** il correspond à l'appel synchrone procédural d'une méthode sur un objet.

Un objet est un composant passif et ne peut donc être sollicité que par un appel de procédure tandis qu'un fragment, étant un composant actif, ne peut être sollicité que par un envoi de message.

La synchronisation

La création de fragments provoque la création de nouveaux flots d'exécution. Le langage BOX doit offrir des possibilités de synchroniser ces différents flots. Il permet deux types de synchronisation : la synchronisation par objets partagés et la synchronisation sur la réception de message.

- **synchronisation par objets partagés :** les appels de méthodes sur un objet peuvent être concurrents : les procédures peuvent s'exécuter en parallèle ; si l'accès à l'objet a besoin d'être synchronisé, le langage offre une politique de synchronisation de type lecteurs/rédacteurs [CHP71] pour l'accès aux procédures d'un objet ; les procédures sont déclarées lectrices ou rédactrices de l'objet ;

- **synchronisation sur la réception de message :** les flots d'exécution peuvent se mettre en attente sur une opération de réception sélective de message sur une boîte aux lettres.

La distribution des entités

Les objets et les fragments peuvent être répartis sur n'importe quel nœud de la machine. La communication entre entités (objets et fragments), appel procédural et envoi de message, est transparente quelle que soit leur localisation.

La création des objets et fragments est dynamique et synchrone. Elle a pour résultat la référence sur la nouvelle entité créée. La localisation de la création est laissée au système. Le run-time BOX crée un objet sur un nœud où réside le code des méthodes de la classe et la description des attributs. De même, pour un fragment, le run-time choisit un nœud où réside le code du comportement et le descriptif des attributs de la classe de fragment.

Le programmeur a cependant la possibilité de préciser statiquement dans un fichier de configuration la répartition du code de chaque classe.

3.1.2 Exemple de programme

L'exemple que nous présentons, développé par C. GRANSART, est une simulation de magasin composé d'un rayon d'articles et d'un stock d'articles. La simulation est la suivante : les clients retirent des articles du rayon ; lorsque le niveau du rayon atteint un seuil minimum, un manutentionnaire transfère des articles du stock vers le rayon ; lorsque le niveau du stock atteint son seuil minimal, le producteur livre de nouveaux articles et les place dans le stock. La figure 3.1 présente l'architecture générale du programme.

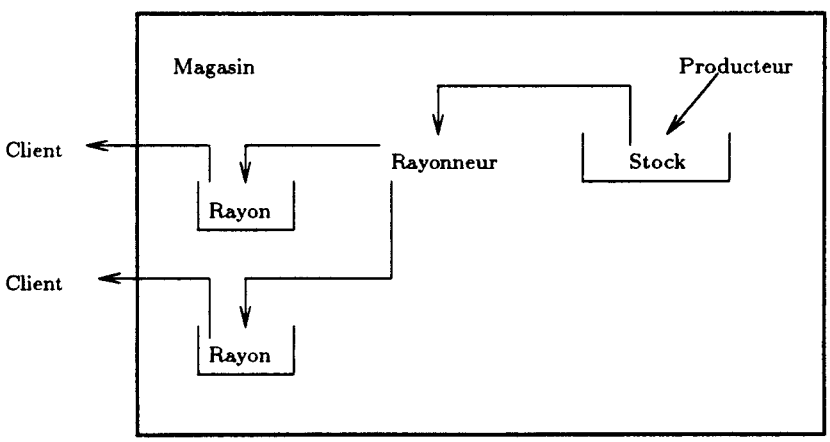

FIG. 3.1 - *Programme Magasin*

Le client, le manutentionnaire et le producteur sont des entités actives et sont implémentés par des fragments, alors que le magasin, le stock et le rayon sont des entités passives et seront implémentés par des objets. Le programme est présenté dans son intégralité en annexe C. Ici sont présentés le code d'une classe d'objets et le code d'une classe de fragments.

La classe **Magasin** (c.f. figure 3.2) est sous-classe de la classe **OBJ**, surclasse de toutes les classes d'objets passifs. Elle possède deux attributs **rayon** et **stock**. La méthode

```
-- Magasin
[OBJ] Magasin ::
  [RAYON] rayon ; -- Rayon ou le serveur se sert
  [STOCK] stock ; -- Stock ou on peut se reapprovisionner

  [SPROC ] retirer_article : [INT] qtte :
    -- Retirer l'article en quantite 'qtte'
    -- Resultat stocke dans un tableau d'articles
  {
     -- Aller dans le rayon
     -- Creer le tableau de resultat
    [FILE <- out !!] out.s ("demande de retrait dans le magasin\n");
    rayon.retirer (qtte) ;
    out.s ("Fin de retrait dans le magasin\n");
  }

  [PROC] init ::
    -- Creation de touts
  {
    stock := [STOCK <- init !!] ;
    rayon := [RAYON <- init ! stock !] ;
  }
```

FIG. 3.2 - *Classe Magasin*

`init` sert à initialiser les deux attributs. La création d'objet est réalisée par un envoi de message vers la classe, l'opération `<-` : les éléments du message étant séparés par `!`.

La méthode `retirer_article` est de type SPROC : l'objet est synchronisé. Il est muni de la politique implicite de synchronisation de type lecteurs/rédacteurs. Les méthodes de type SPROC sont dites rédactrices de l'objet et les méthodes de type PROC lectrices. La méthode « écrit » l'attribut `rayon` avec l'appel `rayon.retirer`.

La classe RAYONNEUR (c.f. figure 3.3) est sous-classe de la classe FRAG, surclasse de toutes les classes de fragment. Elle possède deux attributs `rayon` et `etat`. La méthode `work` de type PROC est le comportement du fragment : c'est le code de l'activité qui sera exécutée à la création du fragment. Le processus du fragment RAYONNEUR attend un message sur sa boîte aux lettres implicite de nom BOX par l'opération `->`. Il récupère un message, `m`, et un nom de procédure, `f`. En effet, un nom de procédure de fragment, FPROC, peut être envoyé par message. Il effectue ensuite l'appel à la procédure `un_retrait`, seule FPROC du fragment.

3.1.3 La réexécution du langage

Le langage BOX est implémenté au dessus du run-time CAC. Un CAC spécialisé serveur d'objets est chargé dans chaque module de la création des objets passifs. L'appel de méthode est réalisé :

- par appel procédural sur le site local ;
- par un mécanisme d'appel de procédure à distance (RPC), implémenté à l'aide de CAC, sur site distant.

```
[FRAG] RAYONNEUR :: -- Chef de rayon
  [INT] etat ;
  [RAYON] rayon ;

  [FPROC] un_retrait ::
  -- Un retrait a eu lieu
  {
    etat := etat - 1 ;
    if etat <= rayon.limite then
      etat := etat + 1 ;
      loop
        if etat = rayon.max then exit end_if ;
        [MANUTENTIONNAIRE <- work ! rayon.stock, rayon !] clarcker ;
        etat := etat + 1 ;
      end_loop ;
    end_if ;
  }

  [PROC] work : [RAYON] ray :
  -- Travail
  {
    rayon := ray ;
    loop
      BOX -> [MESS] m ! [FPROC] f ! ;
      f!m ;
    end_loop ;
  }
```

FIG. 3.3 - *Classe Rayonneur*

L'appel de méthode local, effectué par un appel procédural, n'est pas tracé par le mécanisme de sauvegarde. Cet appel n'introduit pas de non-déterminisme dans l'exécution. Par contre, l'appel à distance peut être source de non-déterminisme dans l'application. Mais ce mécanisme étant implémenté en CAC, il ne nécessite aucun traitement particulier pour le mécanisme de réexécution. Celui ci, développé pour le run-time CAC, convient donc très bien pour la réexécution des programmes BOX en ce qui concerne les objets actifs. Pourtant, deux problèmes supplémentaires apparaissent avec l'introduction des objets passifs BOX :

- la politique de synchronisation implicite de type lecteurs/rédacteurs des objets ;
- les accès concurrents possibles aux attributs d'un objet, générateurs potentiel de non-déterminisme.

La synchronisation des méthodes

Si l'objet est synchronisé (i.e. existence de méthodes de type SPROC), l'appel de méthode, local ou distant, est parenthésé au niveau CAC par l'appel à deux fonctions de synchronisation utilisant des sémaphores pour implémenter le protocole de type lecteurs/rédacteurs. La réexécution doit assurer que l'appel à ces fonctions se fasse dans le même ordre que l'exécution initiale pour conserver l'ordre des appels de méthode.

C'est pourquoi des événements de type EXPLICIT seront générés lors des appels à ces fonctions.

Les accès concurrents aux attributs

Les accès concurrents aux attributs d'un objet ne sont pas pris en compte actuellement par le mécanisme de réexécution. Si une méthode accède à une valeur différente d'un attribut lors de la réexécution, celle-ci peut se bloquer. Deux solutions sont possibles :

- réexécuter de telle sorte que les méthodes accèdent aux mêmes valeurs d'attributs : cela implique, soit de conserver les valeurs accédées, soit un numéro de version comme pour Instant Replay ; cela revient à augmenter considérablement la taille de la trace et donc la perturbation due à la sauvegarde sur le programme BOX ;
- signaler au programmeur pendant la réexécution que l'accès à un objet ne respecte pas le protocole lecteurs/rédacteurs : le compilateur peut « marquer » les méthodes comme « rédactrices » ou « lectrices » de l'objet ; le CAC_{REPLAY} est capable de connaître les différentes méthodes en cours d'exécution sur l'objet ; il peut ainsi signaler si une méthode « rédactrice » est en concurrence avec une autre méthode et qu'un accès concurrent à un attribut est possible.

Ces deux solutions ne sont pas contradictoires mais plutôt complémentaires : si la réexécution assure l'accès aux mêmes valeurs d'attributs, le programmeur doit être prévenu de l'accès concurrent.

3.2 Points d'arrêt distribués

Grâce au mécanisme de réexécution des programmes BOX, le problème issu du non-déterminisme est résolu pour la mise au point. L'utilisation du déverminage cyclique est ainsi devenue possible. Dans l'environnement CAC, le programmeur avait déjà la possibilité d'arrêter l'application, sans connaissance préalable de son état global. Cependant il ne pouvait contrôler l'état dans lequel il désirait suspendre le programme. Le moyen le plus naturel pour interagir avec le programme à déverminer est le point d'arrêt. Nous avons donc défini pour le langage BOX un mécanisme de points d'arrêt distribués.

L'utilisation de points d'arrêt en contexte distribué pose trois problèmes :

- la définition d'un tel point d'arrêt ;
- la détection de l'état devant provoquer l'arrêt ;
- l'arrêt de l'application dans un état cohérent.

L'exécution d'un programme est généralement décrite en termes d'événements qui correspondent à un comportement particulier ou à un changement d'état de l'exécution. En contexte distribué, l'état ou le comportement du programme dépend de plusieurs processus et de plusieurs sites. Deux formes d'événements doivent donc être pris en compte:

- des événements « simples » concernant l'exécution d'un processus et correspondant aux prédicats classiques utilisés pour les programmes séquentiels ;

- des événements distribués définis à partir de plusieurs événements simples.

C'est l'approche utilisée par B. MILLER et J.D. CHOI [MC88] décrite section 1.4.1, et par D. HABAN et W. WEIGEL [HW88]: associer des points d'arrêt aux événements distribués. Ils proposent des algorithmes de détection et un algorithme d'arrêt dérivé de celui de CHANDY et LAMPORT.

Nous associons également la notion de point d'arrêt à celle d'événement. Un point d'arrêt sera atteint lors de l'occurrence de l'événement qui lui est associé. Pour le langage BOX, plusieurs événements de base ont été définis. L'occurrence d'un tel événement ne peut être provoquée que par un flot d'exécution et par un seul site. Mais le programmeur a besoin d'associer un point d'arrêt à un état plus global de l'application. C'est pourquoi il a la possibilité de composer les événements de base afin de définir des événements d'un plus haut niveau d'abstraction et d'y associer des points d'arrêt. Mais la détection d'événements distribués est plus délicate. Nous détaillerons les moyens utilisés pour la détection des événements en BOX.

Nous présenterons d'abord les différents événements, de base et composés. Ensuite, nous présenterons les points d'arrêt pouvant leur être associés et leur utilisation. Nous détaillerons ensuite l'implémentation : la détection d'événement, la décision d'arrêt et l'arrêt effectif de l'application.

3.2.1 Les événements de base

Les événements de base correspondent à des événements générés par le système. Ils ont été définis en rapport avec la structure du langage BOX et son aspect multiparadigme : certains événements concernent les appels de méthodes et correspondent aux événements classiques des langages orientés objet ; d'autres événements prennent en compte l'aspect processus communiquant (envoi et réception de messages). Nous allons d'abord décrire les types d'événements de base du langage BOX et ensuite les différents moyens de filtrage définis pour le programmeur.

Types d'evénements de base

Voici la liste des types d'événements de base du langage BOX. L'ensemble de ces types de base permet une description relativement complète du comportement d'une application BOX. Il faut signaler que ces événements n'ont pas de durée, au contraire de prédicats comme « telle méthode est en cours d'exécution ».

- **Evénements objets :**
  - `OBJCREAT`
    cet événement correspond à la création d'un objet ;
  - `BEGINMETHODE`
    cet événement correspond au début d'exécution d'une méthode ;
  - `ENDMETHODE`
    cet événement correspond à la fin de l'exécution d'une méthode ;
  - `CALLMETHODE`
    cet événement correspond à un appel de méthode du côté appelant.

- **Evénements processus et communications :**
  - STARTFRAGMENT
    cet événement correspond au début d'exécution d'un fragment ;
  - ENDFRAGMENT
    cet événement correspond à la fin de l'exécution d'un fragment ;
  - SENDMESSAGE
    cet événement correspond à l'instruction d'envoi de message ;
  - DEPOSITMESSAGE
    cet événement correspond au dépôt d'un message dans la boîte aux lettres d'un fragment ;
  - WAITMESSAGE
    cet événement correspond à l'instruction d'attente de message ;
  - TAKEMESSAGE
    cet événement correspond à l'extraction effective d'un message d'une boîte aux lettres.

Filtrage des événements

Le programmeur a la possibilité de filtrer les événements de base en restreignant leur domaine de définition précisant, par exemple, une instance particulière de classe d'objets ou de fragments. Voici la liste des paramètres pouvant être ajoutés. Certains paramètres de filtrage sont obligatoires afin d'éviter la définition d'événements par trop généraux. Mais un filtrage plus fin avec des paramètres optionnels est possible. Dans la suite, **<instance objet>** représentera le nom d'une instance particulière de classe d'objets, **<instance fragment>** représentera le nom d'une instance particulière de classe de fragments, **<instance objet_fragment>** pourra être remplacé indifféremment par **<instance objet>** ou **<instance fragment>**, **<instance message>** représentera un nom de message et **<instance boîte aux lettres>** représentera une boîte aux lettres particulière.

- **Evénements objets :**
  - OBJCREAT **<nom de classe objet>** [FOR **<instance objet_fragment>**]
    <nom de classe> précise la classe à laquelle doit appartenir l'instance créée, **<instance objet_fragment>** précise le nom de l'entité demandant la création ;
  - BEGINMETHODE **<nom de classe objet>** **<nom de méthode>**
    [ON **<instance objet>**] [FOR **<instance objet_fragment>**]
    <nom de classe objet><nom de méthode> précise une méthode particulière, ON **<instance objet>** précise le nom de l'instance sur laquelle est exécutée la méthode, FOR **<instance objet_fragment>** précise le nom de l'entité effectuant l'appel de méthode ;
  - ENDMETHODE **<nom de classe objet>** **<nom de méthode>**
    [ON **<instance objet>**] [FOR **<instance objet_fragment>**]
    les paramètres ont la même signification que pour BEGINMETHODE ;

- CALLMETHODE <nom de classe objet> <numéro de ligne source>
  [FOR <instance objet_fragment>]
  <nom de classe objet> <numéro de ligne source> précise un appel de méthode particulier, FOR <instance objet_fragment> précise l'entité effectuant l'appel de méthode.

- **Evénements processus et communications :**

  - STARTFRAGMENT <nom de classe de fragment>
    [FOR <instance objet_fragment>]
    <nom de classe de fragment> précise la classe à laquelle doit appartenir l'instance créée ; <instance objet_fragment> précise le nom de l'entité demandant la création ;
  - ENDFRAGMENT <nom de classe de fragment>
    [<instance fragment>]
    <nom de classe de fragment> précise la classe à laquelle doit appartenir le fragment dont l'activité se termine ; <instance fragment> précise le nom du fragment ;
  - SENDMESSAGE <nom de classe> <numéro de ligne source>
    [FOR <instance objet_fragment>]
    [ON <instance boîtes aux lettres>]
    [ON <instance fragment>]
    <nom de classe> <numéro de ligne source> précise l'instruction d'envoi de message ; <instance objet_fragment> précise l'activité envoyant le message, l'activité du fragment ou celle s'exécutant sur l'objet désigné ; <instance boîtes aux lettres> et <instance fragment> nomment le destinataire du message, le dernier représentant la boîte aux lettres implicite du fragment ;
  - DEPOSITMESSAGE <nom de classe de fragment>
    [ON <instance fragment>]
    [FOR <instance objet_fragment>] [<instance message>]
    ou
    DEPOSITMESSAGE <instance boîtes aux lettres>
    [FOR <instance objet_fragment>] [<instance message>]
    les trois paramètres suivants <nom de classe de fragment>, <instance fragment> et <instance boîtes aux lettres> précisent le destinataire du message ; <instance objet_fragment> précise l'activité ayant envoyé le message, l'activité du fragment ou celle s'exécutant sur l'objet désigné ; <instance message> permet de nommer un message particulier ;
  - WAITMESSAGE <nom de classe><numéro de ligne source>
    [ON <instance objet_fragment>]
    <nom de classe> <numéro de ligne source> précise l'instruction d'attente de message ; <instance objet_fragment> précise l'activité se bloquant en attente de message, l'activité du fragment ou celle s'exécutant sur l'objet désigné ;

- **TAKEMESSAGE <nom de classe><numéro de ligne source>**
  [ON <instance objet_fragment>]
  [ON <instance boîtes aux lettres>]
  [<instance message>]
  <nom de classe>numéro de ligne source> précise l'instruction de réception de message ; **<instance objet_fragment>** précise l'activité retirant le message de la boîte aux lettres, l'activité du fragment ou celle s'exécutant sur l'objet désigné ; **<instance boîtes aux lettres>** précise la boîtes aux lettres de laquelle est retiré le message ; **<instance message>** permet de nommer un message particulier.

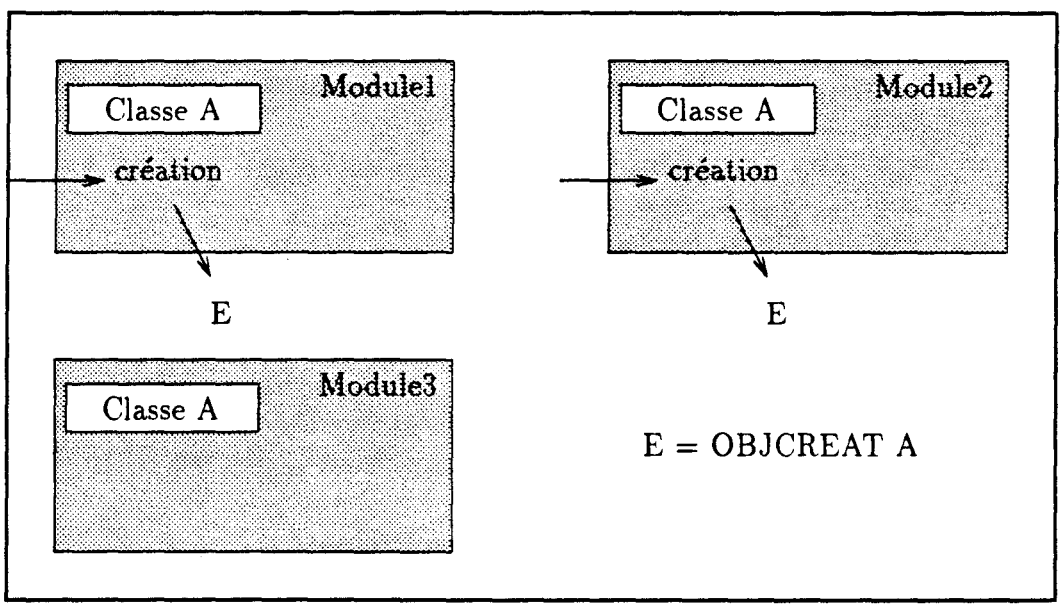

FIG. 3.4 - *Occurence d'un événement de base*

Quand le programmeur définit un événement, la localisation du site de son occurrence n'est pas possible. Si nous prenons l'exemple de la figure 3.4, le programmeur a défini l'événement **E** par **OBJCREAT A** qui correspond à la création d'un objet de la classe **A**. Le code de la classe **A** est ici localisé dans trois modules différents. Ainsi, une occurrence de **E** pourra avoir lieu sur chacun des trois sites. Plus, deux occurrences de **E** peuvent survenir au même instant sur deux sites différents.

3.2.2 Evénements composés

Pour se rapprocher du niveau d'abstraction de l'application, le programmeur a la possibilité de définir des événements composés. Ces événements sont définis en utilisant les opérateurs suivants avec des événements de base ou composés: le *ou* noté $\vee$, la *séquence* notée $\Rightarrow$ et le *et* noté $\wedge$. La sémantique de chaque opérateur est la suivante :

- $\vee$: $EVT = EVT_1 \vee EVT_2 \vee \cdots \vee EVT_n$.
  l'événement EVT correspond à l'occurrence de l'un des événements entrant dans sa définition, chaque événement pouvant être de base ou composé ;

- $\Rightarrow$: $EVT = EVT_1 \Rightarrow EVT_2 \Rightarrow \cdots \Rightarrow EVT_n$.
  l'événement EVT correspond à l'occurrence successive des événements entrant dans sa définition (séquence), chaque événement pouvant être simple ou composé ;

- $\wedge$: $EVT = EVT_1 \wedge EVT_2 \wedge \cdots \wedge EVT_n$.
  une occurrence de l'événement EVT est provoquée par l'occurrence de tous les événements entrant dans sa définition, l'ordre n'important pas ; chaque événement entrant dans la définition de EVT peut être simple ou composé.

Les expressions utilisées pour définir de nouveaux événements sont homogènes : elles ne sont construites qu'à partir d'un seul opérateur. Le programmeur est donc obligé de passer par des événements intermédiaires s'il veut définir des événements utilisant plusieurs opérateurs. Contrairement à MILLER et CHOI, nous autorisons le programmeur à définir des événements composés avec des événements composés, et, contrairement à HABAN et WEIGEL, le programmeur peut composer un nombre quelconque d'événements pour en définir de nouveaux. Signalons que l'occurrence d'un événement ne peut intervenir qu'une seule fois dans la détection d'un événement mais peut intervenir dans la détection de plusieurs événements.

Examinons la sémantique intuitive de ces trois opérateurs et les problèmes que posent les occurrences successives d'un même événement.

L'opérateur *ou* : soit E l'événement défini par $E_1 \vee E_2$. Si E_1 est détecté, une occurrence de E est générée. Mais que doit être le comportement de l'algorithme de détection si E_2 est détecté ensuite ? Générer une nouvelle occurrence de E ou considérer $E_1 \vee E_2$ comme déjà satisfait ? Nous pensons qu'il faut générer une nouvelle occurrence de E et ne pas « oublier » l'occurrence de E_2. En effet, si le programmeur a associé un point d'arrêt à E, une fois l'application suspendue et s'il ne désire qu'une seule occurrence de E, il peut supprimer le point d'arrêt.

L'opérateur *et* : l'événement E est défini par $E_1 \wedge E_2$. Si trois occurrences successives de E_1 sont détectées, les trois occurrences ultérieures de E_2 provoqueront-elles la génération de trois occurrences de E ou d'une seule occurrence ? Nous avons choisi dans ce cas de nous « souvenir » des occurrences de E_1 et de générer trois occurrences successives de E, ce qui nous semble plus correspondre à la sémantique intuitive de l'opérateur.

L'opérateur *séquence* : la même question que pour l'opérateur *et* se pose. L'événement E est défini par $E_1 \Rightarrow E_2$. Supposons la séquence d'événements suivante : E_1 E_1 ... **<grand laps de temps>** E_1 E_2 E_2. Si nous choisissions la même méthode que pour le *et*, la première occurrence de E_2 entraînerait la détection de E, mais la deuxième occurrence également. Mais la sémantique de l'opérateur *séquence* est différente. Le comportement attendu serait plutôt que la deuxième occurrence de E_2 n'ai rien provoqué et que les anciennes occurrences de E_1 aient été oubliées. C'est cette politique de détection que nous avons choisie.

De même que pour les événements de base, la localisation du site de l'occurrence de l'événement n'est pas possible. De plus, l'occurrence d'un événement composé peut faire intervenir plusieurs flots d'exécution sur plusieurs sites. La détection en est donc plus délicate. L'exemple de la figure 3.5 présente l'occurrence d'un événement composé défini avec l'opérateur de séquence : `E = OBJCREAT A ⇒STARTFRAGMENT B` qui correspond à une création d'une instance de la classe d'objet `A` suivie de la création d'un fragment de la classe `B`. Le code de la classe `A` est localisé dans le module1 et celui de la classe `B` dans le module2 et le module3. L'occurrence de `E` fait donc intervenir deux sites, module1 et module2 ou module3, et peut survenir sur deux sites, module2 et module3.

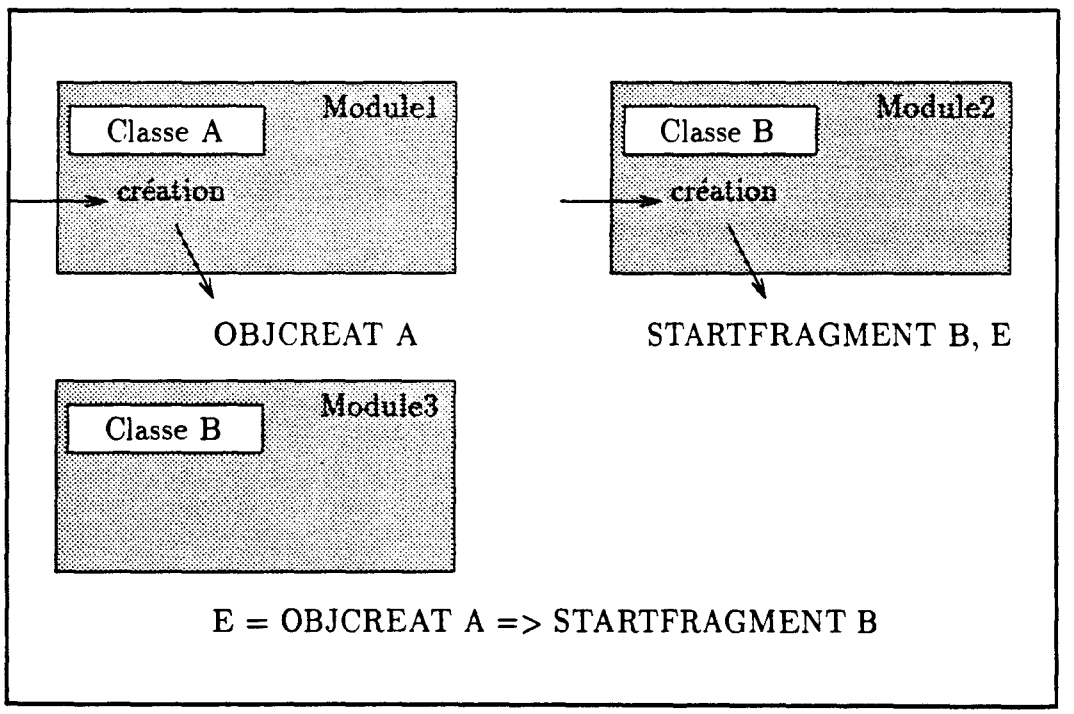

FIG. 3.5 - *Occurence d'un événement composé*

3.2.3 Points d'arrêts et événements

Pour positionner un point d'arrêt, le programmeur doit lui associer un événement. Dès l'occurrence de l'événement détectée, l'application sera suspendue en s'écartant le moins possible de l'état ayant provoqué l'arrêt : si un point d'arrêt est positionné sur un début de méthode, il ne faut pas que l'application soit suspendue après la fin de la méthode. Tout ceci est, bien sûr, réalisé lors d'une phase de réexécution.

Le programmeur ne peut définir un point d'arrêt que si l'application est suspendue. Une fois l'application suspendue, il peut examiner l'état de l'application : les instances créées, les attributs et leur valeur, les messages dans les boîtes aux lettres, les messages en transit ... Tout cela était déjà possible au niveau CAC. C'est en examinant l'état de l'application que le programmeur pourra obtenir des noms d'instances d'objet ou de fragment, de messages ou de boîtes aux lettres pour affiner la définition de ses événements et définir de nouveaux points d'arrêt.

Si le programmeur définit des points d'arrêts associés à des événements pouvant être générés sur plusieurs sites, l'application peut être suspendue par plusieurs points d'arrêt en même temps. Dans l'exemple de la figure 3.6, nous avons deux événements définis par `E1 = OBJCREAT C` et `E2 = OBJCREAT A ⇒ STARTFRAGMENT B` associés tous deux à des points d'arrêt. Le code de la classe `A` se trouve dans le module1, celui de la classe `B` dans le module2 et celui de la classe `C` dans le module3. L'occurrence de `E1` sur le module3 et l'occurrence de `E2` sur le module2 peuvent toutes deux décider de l'arrêt en même temps car sur deux sites différents. Le programmeur sera averti que l'arrêt aura été provoqué par ces deux événements.

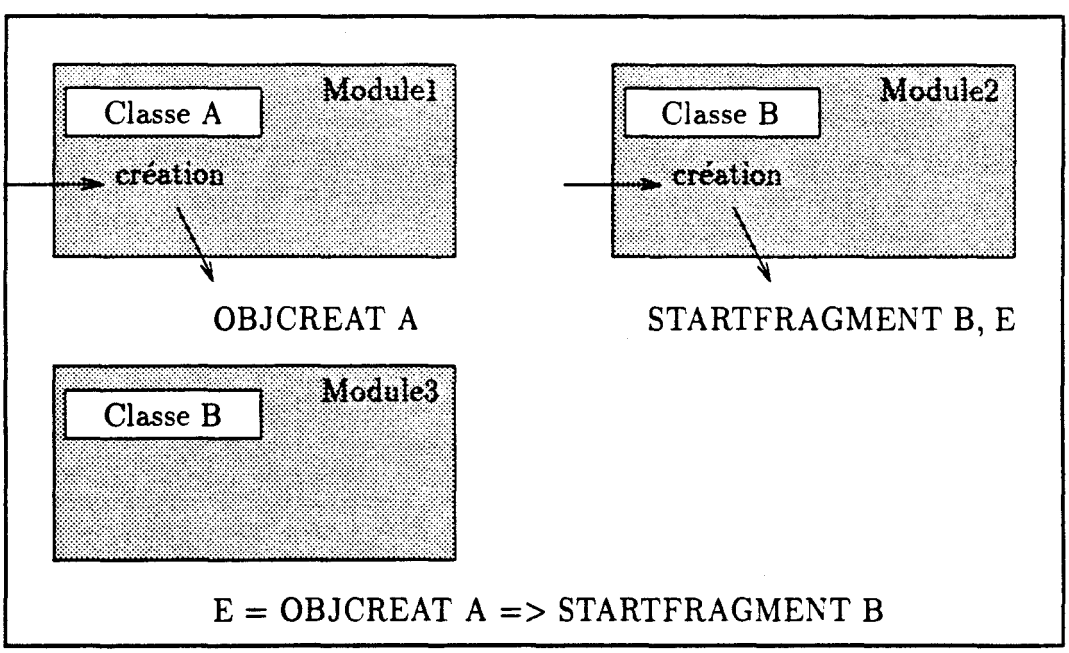

FIG. 3.6 - *Arrêt sur deux points d'arrêt*

Nous présentons maintenant des exemples de point d'arrêts en utilisant le programme **Magasin** décrit partiellement à la section 3.1 et en intégralité annexe C.

Exemple de point d'arrêt défini avec ∨

```
retrait=BEGINMETH rayon.retirer ∨ BEGINMETH stock.extraire
stop at retrait
```

Les deux lignes précédentes définissent un point d'arrêt qui sera atteint lors d'un retrait d'articles du rayon ou du stock.

Exemple de point d'arrêt défini avec ∧

```
rayon_vide=TAKEMESSAGE rayonneur ∧ STARTFRAGMENT manutentionnaire
stop at rayon_vide
```

Le point d'arrêt sera atteint si un article est en train d'être retiré du rayon et si le rayonneur (responsable du rayon) a demandé à un manutentionnaire de remplir le rayon : le rayon a atteint sa valeur minimale.

Exemple de point d'arrêt défini avec ⇒

```
commande=rayon_vide ⇒ STARTFRAGMENT producteur
stop at commande
```

Le point d'arrêt sera atteint si un rayon est vide, événement défini au paragraphe précédent, et si cet événement a déclenché l'activité du producteur.

3.2.4 Implémentation

La gestion des points d'arrêts sera réalisée au dessus du mécanisme de réexécution implanté par des CAC. Rappelons que ce mécanisme est distribué : il existe un CAC_{REPLAY} par module. Malgré la réexécution, l'utilisation de points d'arrêts distribués entraîne de nouveaux problèmes :

- la définition d'un événement peut concerner plusieurs sites : la définition devra donc être distribuée ; la détection d'un événement devra donc être distribuée ;
- la réexécution permet de suspendre le programme dans un état cohérent mais il faut arrêter le programme le plus rapidement possible afin de ne pas s'éloigner de l'état qui a provoqué l'arrêt.

Un module est un site virtuel. Dans la suite, nous utiliserons indifféremment les termes sites et modules.

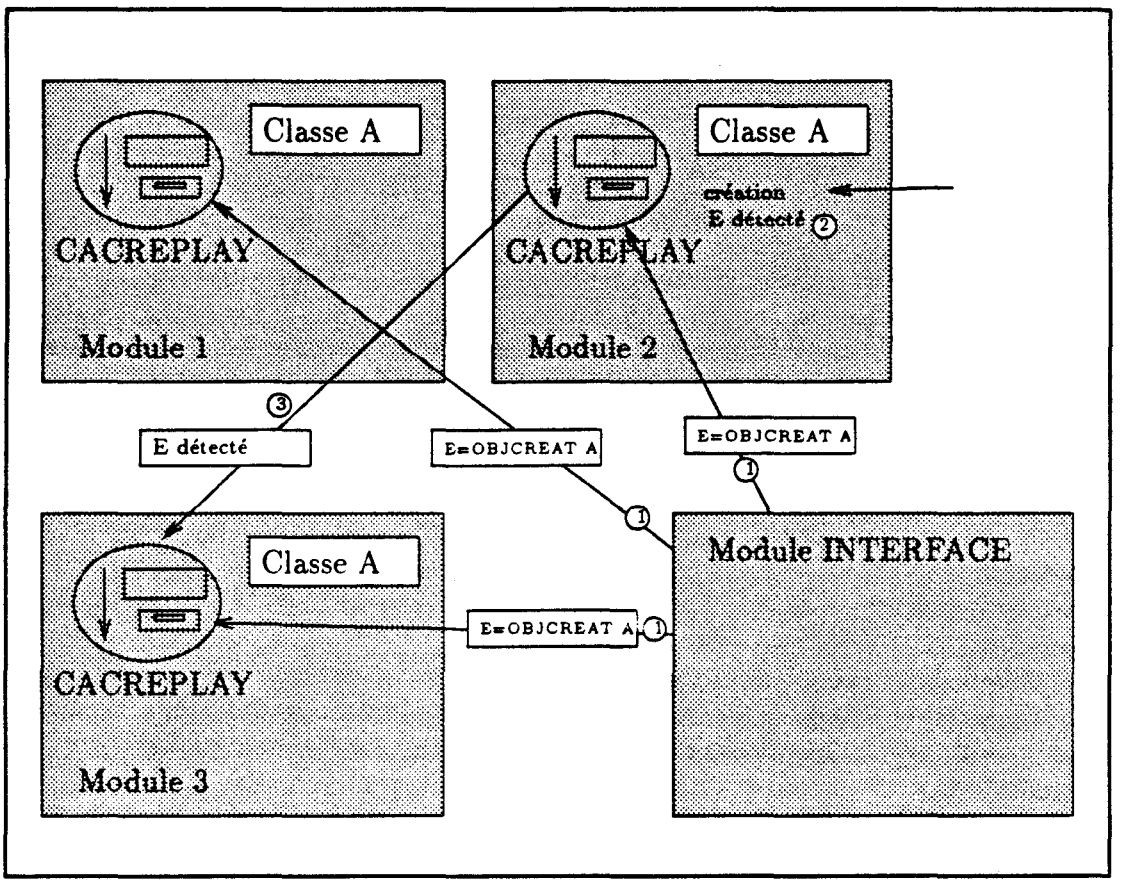

FIG. 3.7 - *Détection d'événements de base*

Evénements de base

La gestion des événements de base est illustrée par l'exemple de la figure 3.7. Un ① indiquera l'endroit dans la figure correspondant à l'explication.

Localisation des événements de base: la définition des événements de base est distribuée sur tous les sites pouvant les générer vers chaque CAC_{REPLAY} et gardée dans le module interface. Dans l'exemple de la figure 3.7, l'événement `E` est défini comme `OBJCREAT A`. La classe A étant définie dans les trois modules, la définition de `E` sera diffusée vers les trois modules depuis le module Interface ①.

Représentation interne des événements de base: la définition d'un événement de base contient les informations suivantes :

- un numéro global l'identifiant ;
- le type de l'événement et les paramètres de filtrage;

- la liste des événements qui dépendent de son occurrence, en fait une liste de couples (numéro d'événement, sites possédant sa définition) ;

- un indicateur permettant de savoir si l'événement est associé à un point d'arrêt.

D'autres données sont associées à l'événement qui pourraient être consultées par le programmeur comme l'identité du générateur de l'événement par exemple.

Algorithme de détection des événements de base: la détection des événements de base est effectuée par le CAC_{REPLAY} de chaque module. Si l'événement est associé à un point d'arrêt, l'application est suspendue. L'algorithme d'arrêt est décrit à la section 3.2.5 et peut être initié de plusieurs sites. Ainsi, l'application peut être suspendue une seule fois sur plusieurs points d'arrêt. Ensuite, les sites possédant des événements dont l'occurrence dépend de l'événement qui vient d'être généré sont prévenus par message de cette génération. Rappelons que l'occurrence d'un événement ne peut intervenir qu'une fois dans la détection d'un événement donné, mais il peut intervenir dans la détection de plusieurs événements. L'application est suspendue d'abord pour éviter de trop s'éloigner de l'état ayant provoqué l'arrêt. Si tous les CAC de l'application sont suspendus, les CAC_{REPLAY} sont toujours actifs.

Reprenons l'exemple de la figure 3.7. L'événement E est détecté dans module2 ②. E n'est pas associé à un point d'arrêt. Le module3 possédant un événement composé à partir de E, un message prévenant de la génération de E est envoyé au CAC_{REPLAY} du module3 ③.

La gestion des événements composés est différente de celle des événements de base. La localisation, la structure et l'algorithme de détection dépendent de l'opérateur utilisé pour les définir. Pour chaque opérateur, *ou*, *et*, *séquence*, nous présenterons l'implémentation, à la fois de la représentation et de la détection.

Evénements composés avec l'opérateur *ou*

La gestion des événements composés avec l'opérateur *ou* est illustrée par l'exemple de la figure 3.8.

Localisation des événements composés avec l'opérateur *ou*: la définition des événements est distribuée sur tous les sites pouvant les générer et gardée dans le module interface. Dans l'exemple de la figure 3.8, l'événement E est défini comme Evt_1 V Evt_2. Evt_1 étant localisé dans le module1 et Evt_2 dans les module2 et module3, la définition de E sera diffusée vers les trois modules depuis le module Interface ①.

Il faut mettre à jour la liste des dépendances de Evt_1 et Evt_2 sur tous les sites les possédant. La liste contiendra le nom de l'événement E associé au site local : l'occurrence d'un des événements composant E suffit pour le générer.

Ainsi, sur le module1, Evt_1 aura comme liste de dépendance le couple (E,module1).

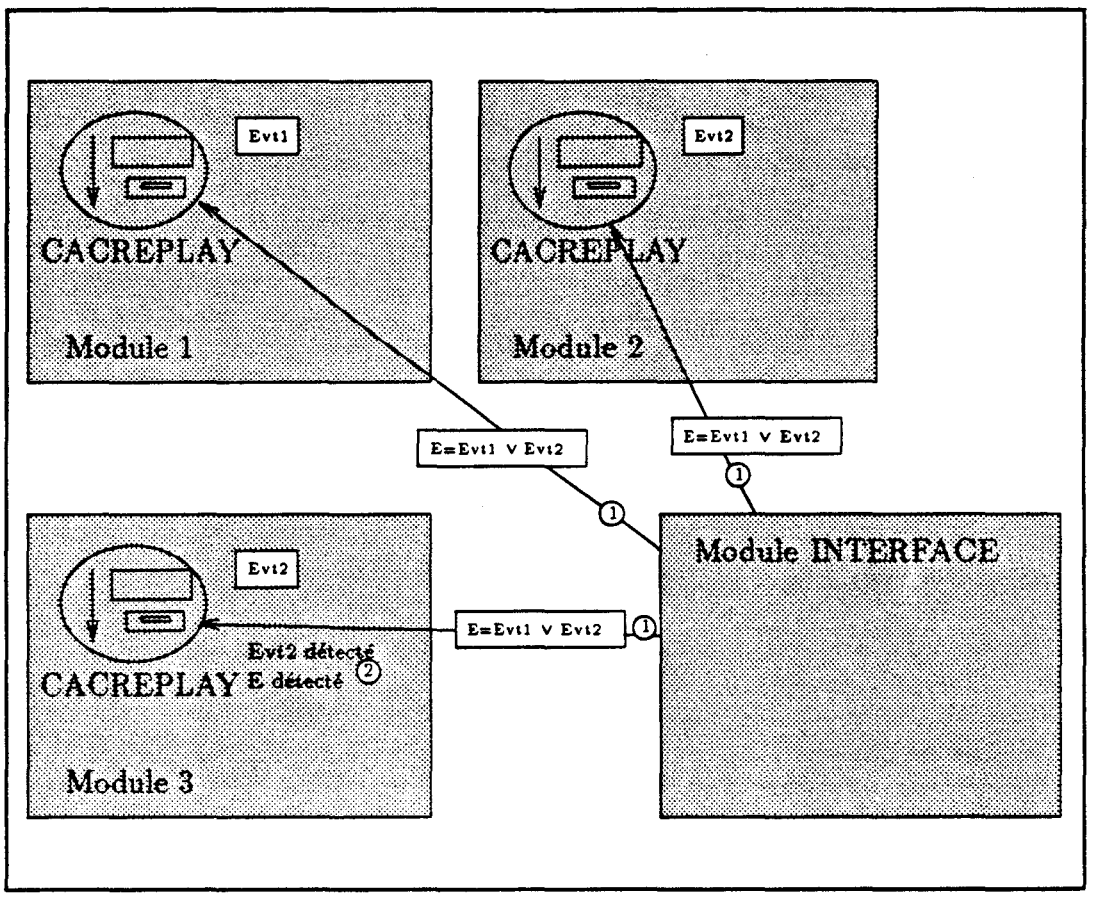

FIG. 3.8 - *Détection d'événements composés avec* ∨

Représentation interne des événements composés avec l'opérateur *ou*: la définition d'un événement composé contient les informations suivantes:

- un numéro global l'identifiant ;
- l'opérateur utilisé pour le définir ;
- la liste des événements qui dépendent de son occurrence, en fait une liste de couples (numéro d'événement, sites possédant sa définition) ;
- la liste des événements dont l'occurrence est attendue ;
- un indicateur permettant de savoir si l'événement est associé à un point d'arrêt.

La liste des événements dont l'occurrence est attendue pour un événement composé avec ∨ est réduite aux seuls événements présents sur le site. Ainsi, la définition de E sur le module1 ne comprendra qu'une liste à un élément : $\mathtt{Evt}_1$.

Algorithme de détection des événements composés avec l'opérateur *ou*: l'événement attendu étant sur le même site que l'événement composé, aucun message n'est généré pour détecter l'événement composé. Une fois l'événement détecté, l'application est suspendue s'il est associé à un point d'arrêt. Ensuite, comme pour les événements de base, les sites possédant des événements dont l'occurrence dépend de l'événement qui vient d'être généré sont prévenus par message de cette génération.

Dans notre exemple, la détection de $\mathbf{Evt_2}$ dans le module3 provoque la détection immédiate de **E** dans ce même module ②. **E** étant situé dans le même module, cela réduit la durée entre le moment où $\mathbf{Evt_2}$ est généré et le moment où l'algorithme d'arrêt est initié si **E** est associé à un point d'arrêt.

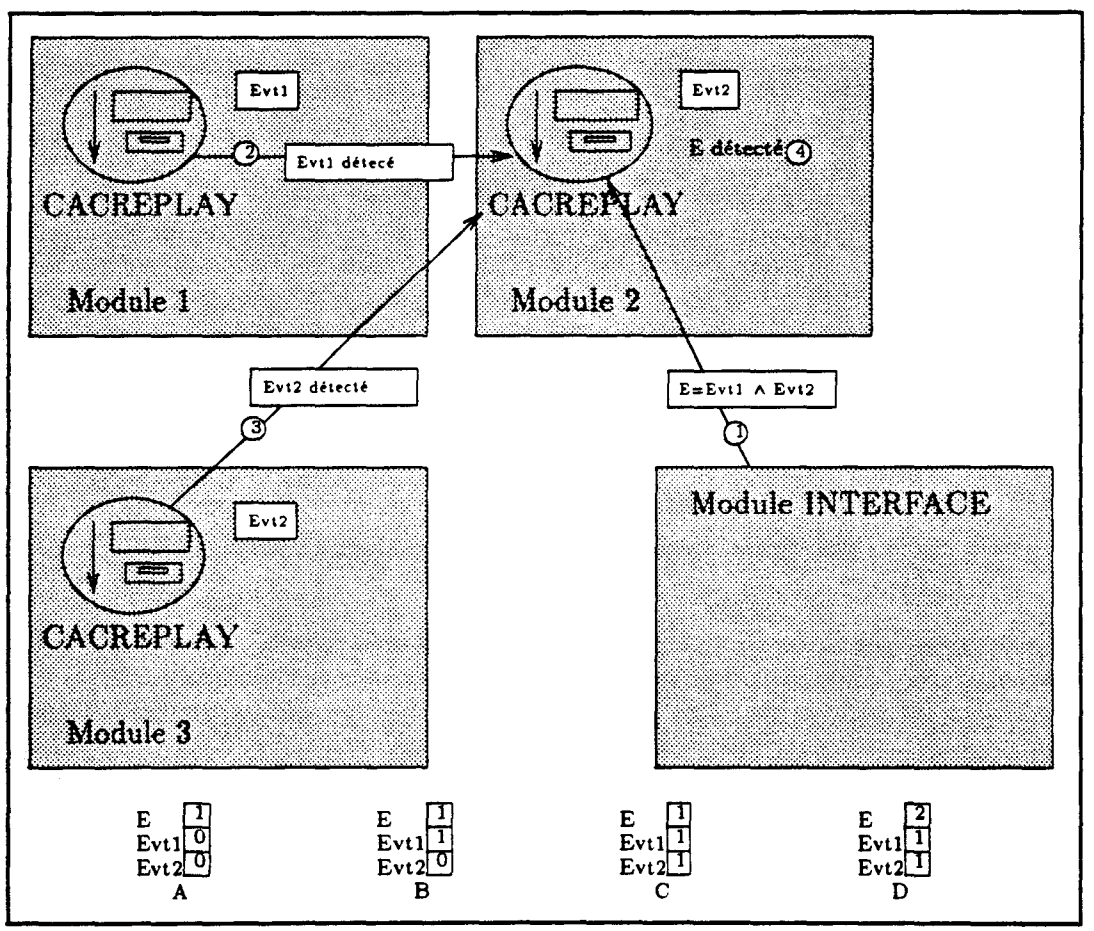

FIG. 3.9 - *Détection d'événements composés avec* ∧

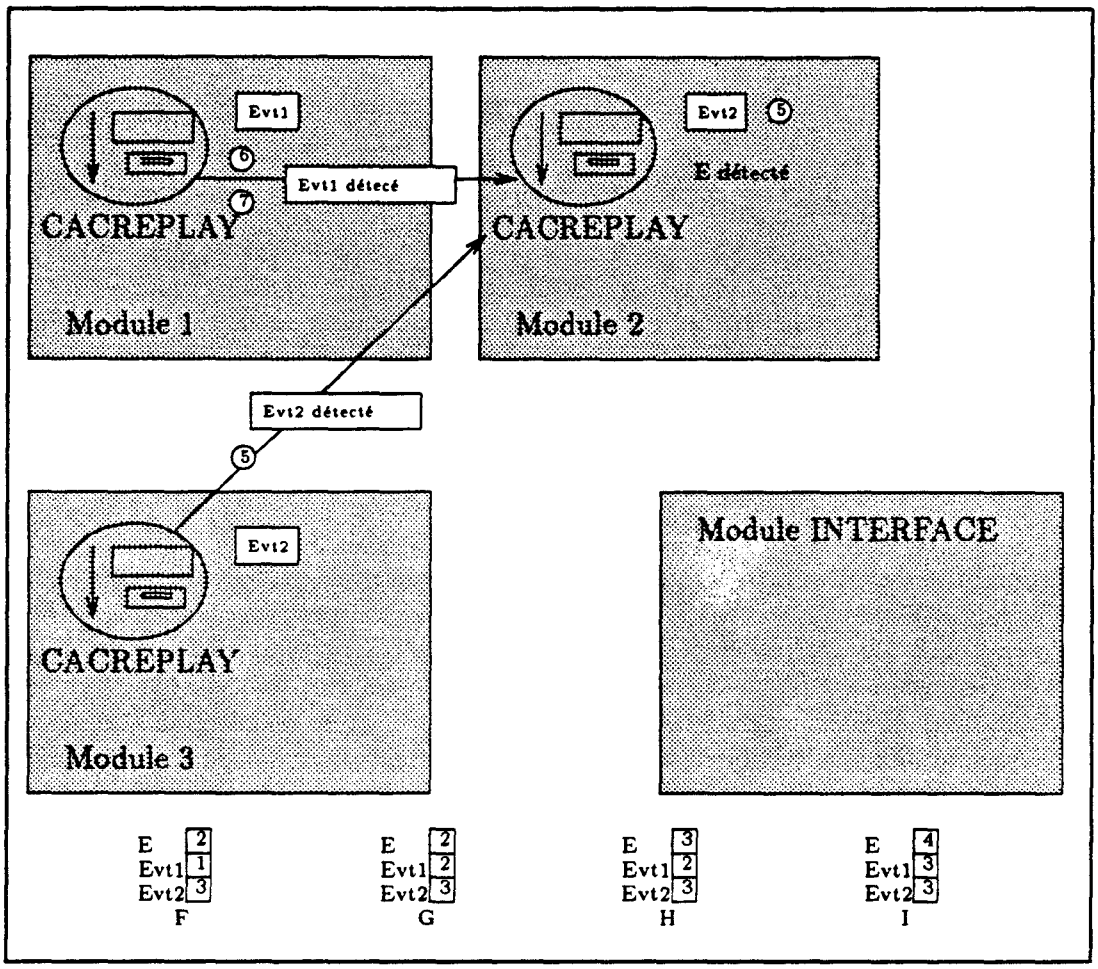

FIG. 3.10 - *Détection d'événements composés avec* $\wedge$

Evénements composés avec l'opérateur *et*

La gestion des événements composés avec l'opérateur *et* est illustrée par l'exemple de la figure 3.9 et 3.10.

Localisation des événements composés avec l'opérateur *et*: la définition des événements est envoyée sur un site choisi au hasard parmi les sites possédant la définition de l'un des événements le composant et gardée dans le module interface. Dans l'exemple de la figure 3.9, l'événement **E** est défini comme $\mathbf{Evt_1} \wedge \mathbf{Evt_2}$. $\mathbf{Evt_1}$ étant localisé dans le module1 et $\mathbf{Evt_2}$ dans les module2 et 3, la définition de **E** sera diffusée vers le module2 depuis le module Interface ①. Il faut mettre à jour la liste des dépendances de $\mathbf{Evt_1}$ et $\mathbf{Evt_2}$ sur tous les sites les possédant mais en indiquant comme site à prévenir seulement le site où se trouve la définition de l'événement. Ainsi, sur le module1, $\mathbf{Evt_1}$ aura comme liste de dépendance le couple `(E,module2)`.

Représentation interne des événements composés avec l'opérateur *et*: la définition contient les mêmes informations que celle d'un événement composé avec ∨ mais aussi :

- le numéro de la prochaine occurrence : c'est un compteur qui correspond au nombre d'occurrences de l'événement plus 1 ;
- un compteur pour chaque événement attendu correspondant au nombre d'occurrences ayant déjà été signalées.

La définition de E au départ comprendra un compteur initialisé à 1 et deux compteurs initialisés à 0, un correspondant au nombre d'occurrences de $\mathtt{Evt}_1$ et l'autre correspondant au nombre d'occurrences de $\mathtt{Evt}_2$.

Algorithme de détection d'événementscomposés avec l'opérateur *et*: quand un événement attendu est signalé, le compteur qui lui correspond est incrémenté. Si tous les compteurs associés aux événements attendus sont supérieurs ou égaux au compteur de l'événement composé, l'événement composé est généré. Une fois l'événement détecté, l'application est suspendue s'il est associé à un point d'arrêt. Ensuite, comme pour les événements de base, les sites possédant des événements dont l'occurrence dépend de l'événement qui vient d'être généré sont alors prévenus par message de cette génération.

Reprenons l'algorithme de détection avec notre exemple (figure 3.9). Les compteurs sont au départ dans la configuration A. L'événement $\mathtt{Evt}_1$ est généré et un message est envoyé au CAC_{REPLAY} de module2 pour en signaler l'occurrence ②. Le compteur correspondant est incrémenté. Les compteurs passent alors dans la configuration B. L'événement $\mathtt{Evt}_2$ est détecté sur module3 et un message est envoyé au CAC_{REPLAY} de module2 pour en signaler l'occurrence ③. Le compteur correspondant est incrémenté. Les compteurs passent alors dans la configuration C. E est alors détecté car les deux compteurs correspondant aux événements attendus sont égaux à celui de E ○ 4. Le compteur correspondant au nombre d'occurrences de E plus un est incrémenté. Les compteurs passent alors dans la configuration D. La suite de l'exemple est illustrée par la figure 3.10. Deux événements $\mathtt{Evt}_2$ sont détectés, l'un sur module2, l'autre sur module3. Leur occurrence est signalée au CAC_{REPLAY} de module2 ⑤. Les compteurs correspondants sont incrémentés. Les compteurs passent alors dans la configuration F. Une première occurrence de $\mathtt{Evt}_1$ fait passer les compteurs dans la configuration G ○ 6 : E est détecté ce qui fait passer les compteurs dans la configuration H. La prochaine occurrence de $\mathtt{Evt}_1$ ⑦ provoquera la détection de E et fera passer finalement les compteurs dans la configuration I. Ce mécanisme de compteur permet de ne perdre aucune information, c'est à dire de prendre en compte toutes les occurrences d'événements : sans lui, une seule occurrence de E aurait été détectée.

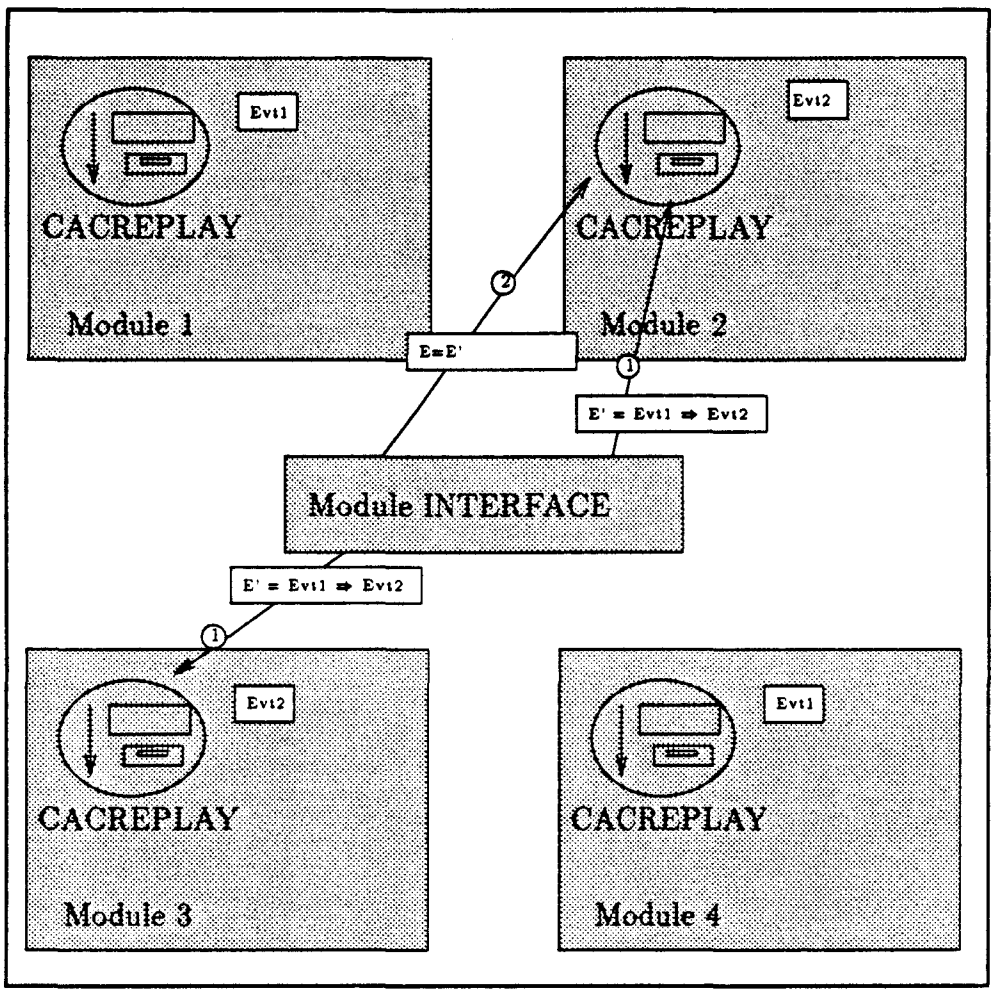

FIG. 3.11 - *Localisation d'événements composés avec* ⇒

Evénements composés avec l'opérateur *séquence*

La gestion des événements composés avec l'opérateur *séquence* est illustrée par l'exemple de la figure 3.11.

Localisation des événements composés avec l'opérateur *séquence* : soit l'événement E défini par $Evt_1 \Rightarrow Evt_2 \ldots Evt_{n-1} \Rightarrow Evt_n$. Pour détecter l'événement complet, nous décomposons l'événement en sous-séquences $E'_1 = Evt_1 \Rightarrow Evt_2$, $E'_2 = E_1 \Rightarrow Evt_2$, ..., $E'_{n-1} = E_{n-1} \Rightarrow Evt_n$. Nous nommerons les événements $E'_1 \ldots E'_{n-1}$ sous-événements. Chaque sous-événement est envoyé sur les sites contenant le dernier événement de la sous-séquence correspondante. Chaque sous-séquence est représentée par un événement dont l'occurrence ne pourra avoir lieu que sur un seul site : nous le nommerons événement intermédiaire et il sera noté $E_1 \ldots E_{n-1}$, E_{n-1} correspondant à E, l'événement composé global. Chaque événement intermédiaire est envoyé sur l'un

des sites, choisi au hasard, contenant le dernier événement de la sous-séquence correspondante. Dans l'exemple de la figure 3.11, l'événement **E** est défini comme $\mathbf{Evt_1} \Rightarrow \mathbf{Evt_2}$. $\mathbf{Evt_1}$ est localisé dans module1 et module4, $\mathbf{Evt_2}$ dans module2 et module3. La définition d'un événement **E'** attendant l'occurrence de $\mathbf{Evt_1}$ et de $\mathbf{Evt_2}$ est envoyée sur module2 et module3 ①. Sur module2, choisi au hasard entre module2 et module3, est envoyée la définition de **E** comme attendant la génération de **E'** ②. Il faut mettre à jour la liste des dépendances en conséquence. Ainsi, sur le module 1, $\mathbf{Evt_1}$ aura comme liste de dépendance les couples **(E',module2)** **(E',module3)** et $\mathbf{Evt_2}$ sur le module2 **(E',module2)**.

Si la même localisation que pour l'opérateur $\wedge$ avait été choisie, de faux événements auraient pu être détectés. La figure 3.12 nous montre pourquoi. **E** est défini comme $\mathbf{Evt_1} \Rightarrow \mathbf{Evt_2} \Rightarrow \mathbf{Evt_3}$. $\mathbf{Evt_1}$ est localisé dans module1, $\mathbf{Evt_2}$ dans module2, $\mathbf{Evt_3}$ dans module3 et **E** dans le module3. Supposons que se produise la séquence suivante : $\mathbf{Evt_2} \Rightarrow \mathbf{Evt_1} \Rightarrow \mathbf{Evt_3}$. Un message prévenant de l'occurrence de $\mathbf{Evt_2}$ et de $\mathbf{Evt_1}$ est envoyé depuis les modules 2 et 1 ①. Le message provenant de module1 est reçu avant celui provenant de module2 par module3 ②. Ceci correspond à la définition de **E** : **E** sera donc détecté faussement.

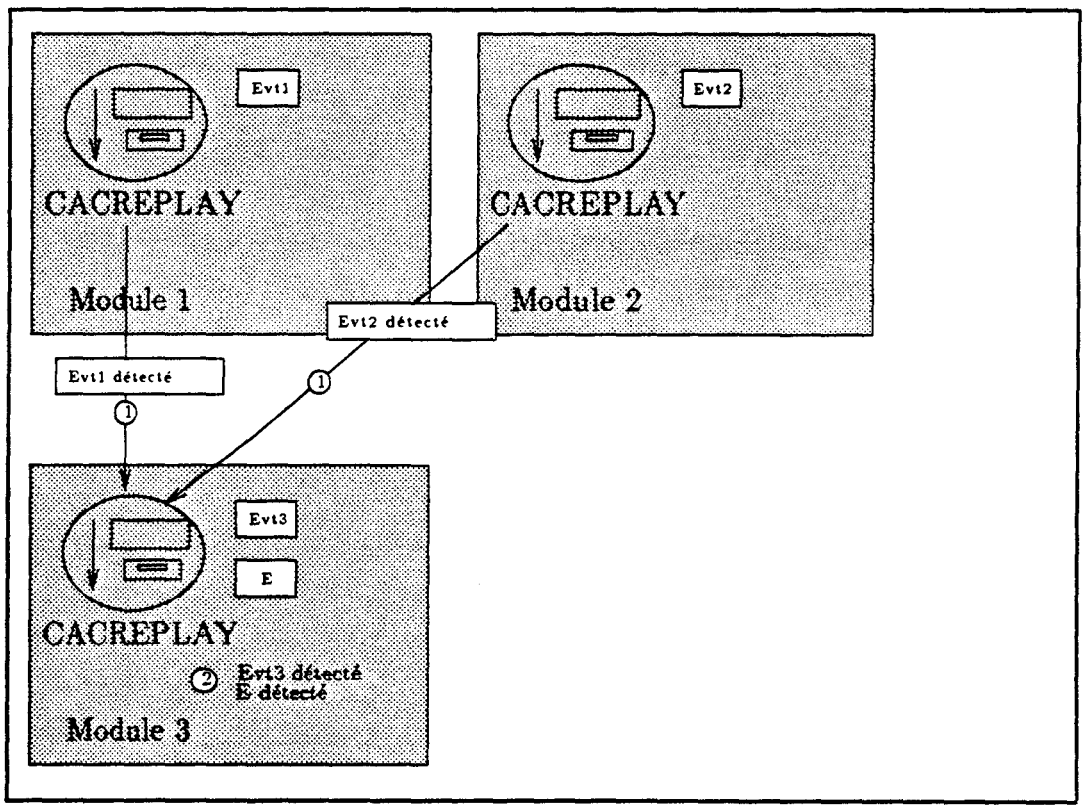

FIG. 3.12 - *Faux événement détecté*

Représentation des événements composés avec l'opérateur *séquence* : La représentation interne des sous-événements comprend les informations suivantes, en plus des informations communes à tous les événements composés :

- le nombre d'occurrences signalées du premier événement de la sous-séquence ;
- le numéro de la dernière occurrence du premier événement ayant généré le sous-événement.

La représentation interne de l'événement intermédiaire comprend uniquement un compteur correspondant au nombre d'occurrences du premier événement de la sous-séquence.

La définition de E' au départ comprendra deux compteurs initialisés à 0 et la définition de E comprendra un compteur initialisé à 0.

Algorithme de détection des événements composés avec l'opérateur *séquence* : nous décrirons l'algorithme pour une sous-séquence $\mathbf{Evt_1} \Rightarrow \mathbf{Evt_2}$, l'événement intermédiaire étant appelé E et le sous-événement E'. Voici l'algorithme de détection du sous-événement E'. Quand une occurrence de $\mathbf{Evt_1}$ est signalée, le compteur qui lui correspond dans la représentation de E' est incrémenté. Lorsqu'une occurrence de $\mathbf{Evt_2}$ est signalée, elle n'est prise en compte que si le nombre d'occurrences de $\mathbf{Evt_1}$ est supérieur au numéro de la dernière occurrence de $\mathbf{Evt_1}$ ayant généré le sous-événement E', ceci afin d'assurer une séquence entre $\mathbf{Evt_1}$ et $\mathbf{Evt_2}$. Dans ce cas, un message, estampillé par le nombre d'occurrences de $\mathbf{Evt_1}$, est envoyé vers le site contenant l'événement intermédiaire E. Ensuite, le compteur contenant le numéro de la dernière occurrence de $\mathbf{Evt_1}$ ayant généré E' est mis à jour en prenant comme valeur le nombre d'occurrences de $\mathbf{Evt_1}$ signalées. Voici l'algorithme de détection de l'événement intermédiaire E. Le site contenant sa représentation reçoit un message prévenant de l'occurrence du sous-événement E'. Si l'estampillage du message est supérieur à la valeur du compteur contenu dans la représentation de E, l'événement E est détecté et le compteur prend la valeur de l'estampillage. Dans le cas contraire, le message est ignoré.

Le fait de transporter le nombre d'occurrences du premier événement dans les messages permet d'éviter de prendre en compte une seule occurrence de cet événement pour générer deux occurrences de la séquence. L'utilisation du numéro de la dernière occurrence du premier événement à la place d'un compteur d'occurrences du deuxième événement permet de résoudre le problème des occurrences multiples de $\mathbf{Evt_1}$ comme cela a été indiqué section 3.2.2.

Voici le déroulement de l'algorithme pour notre exemple, présenté figure 3.13. La représentation des différents événements est au départ dans l'état A. $\mathbf{Evt_1}$ est détecté dans module1 : un message est envoyé vers module2 et module3 prévenant de cette génération ①. Les compteurs correspondants sont incrémentés : les représentations passent à l'état B. $\mathbf{Evt_2}$ ② est détecté dans module2. Les compteurs sont remis à jour pour la représentation de E' dans module2 (état C). La génération de E' ③ est signalée et prise en compte pour E : le nombre d'occurrences de $\mathbf{Evt_1}$ est 1 et le compteur de E est à 0. E est donc généré ④ et les représentations des événements se retrouvent dans l'état D. $\mathbf{Evt_2}$ est détecté dans module3 : il peut être pris en compte pour la génération de E', la

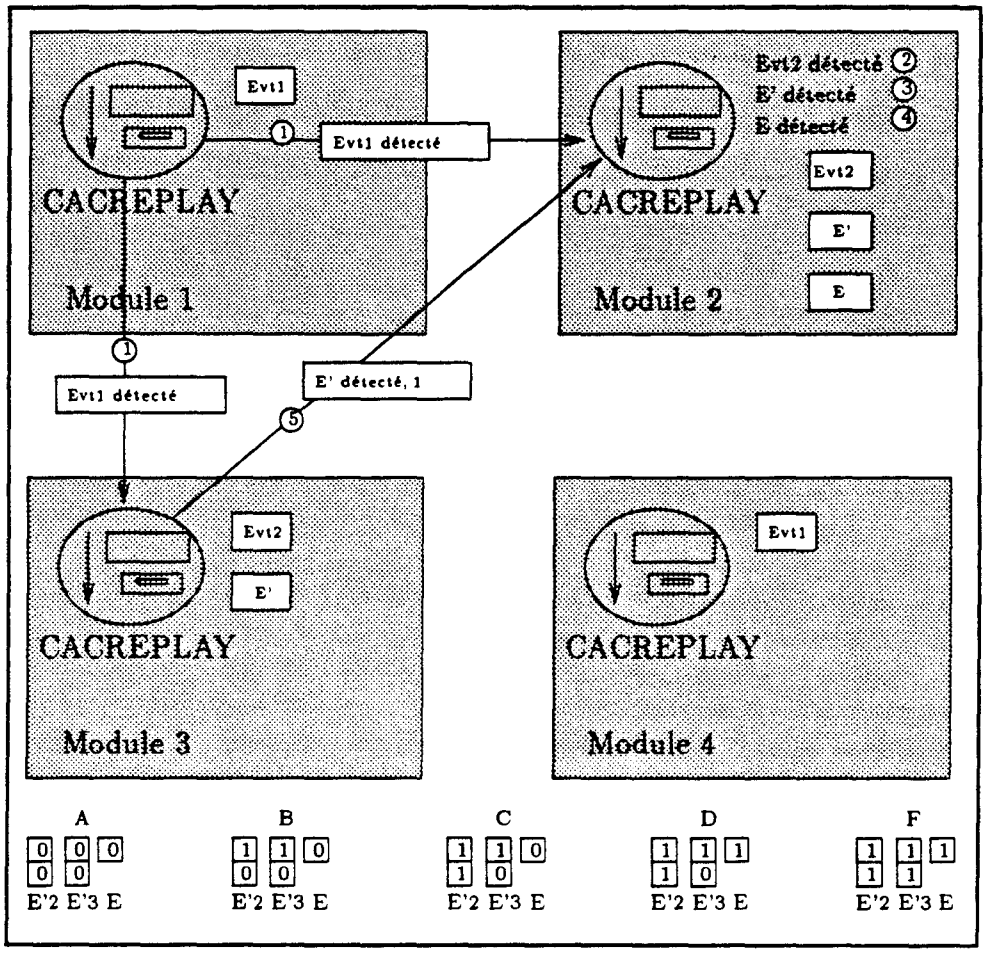

FIG. 3.13 - *Détection d'événements composés avec* $\Rightarrow$

représentation des événements passent à l'état F. Un message estampillé par la valeur 1 est envoyé à module2 ⑤. Mais ce message est ignoré car le compteur dans la représentation de E est à 1. Ainsi, l'occurrence de $\mathbf{Evt}_1$ n'aura provoqué qu'une occurrence de E.

Tous les événements composés avec la séquence ne pourront pas être détectés : si le message prévenant de l'occurrence de $\mathbf{Evt}_1$ arrive dans le module3 après la détection de $\mathbf{Evt}_2$ sur ce même module, E ne sera pas détecté.

Il faut assurer au moins que les événements dépendants l'un de l'autre provoquent une occurrence de l'événement composé. Avec l'algorithme actuel, cela n'est pas vérifié : l'exemple de la figure 3.14 présente le cas où $\mathbf{Evt}_2$ dépend de $\mathbf{Evt}_1$ mais par l'intermédiaire d'un troisième module ①. Le message prévenant de l'occurrence de $\mathbf{Evt}_1$ peut arriver dans le module contenant $\mathbf{Evt}_2$ après le message provenant du troisième module

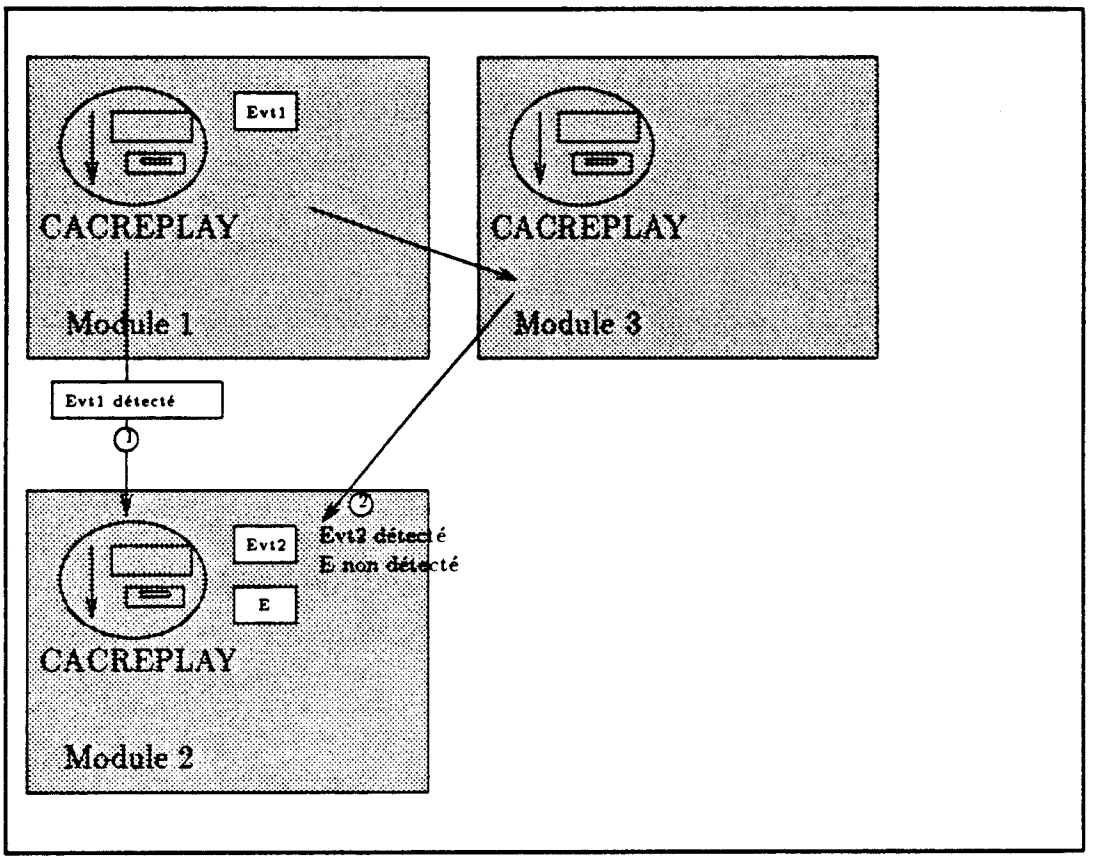

FIG. 3.14 - *Evénements non détectés*

et provoquant $\mathbf{Evt}_2$ ②. Deux solutions différentes ont été étudiées :

- le module contenant $\mathbf{Evt}_1$ diffuse à tous les modules le message prévenant de l'occurrence de $\mathbf{Evt}_1$ et ceux-ci redirigent le message vers le module contenant $\mathbf{Evt}_2$; tout message provenant du premier module ne pourra induire d'autres événements dans le deuxième avant que ce dernier ne soit prévenu de l'occurrence de $\mathbf{Evt}_1$;
- le module contenant $\mathbf{Evt}_1$ envoie un message au module contenant $\mathbf{Evt}_2$, prévenant de l'occurrence de $\mathbf{Evt}_1$, et attend un message d'acquittement ; le premier ne peut ainsi induire aucun événement avant que le deuxième ne soit prévenu.

3.2.5 Arrêt d'une application

L'algorithme d'arrêt est dérivé de celui de CHANDY et LAMPORT [CL85] défini pour la sauvegarde de l'état d'une application distribuée. L'algorithme est initié par le CAC_{REPLAY} d'un module ayant détecté un événement associé à un point d'arrêt ou par l'utilisateur par l'intermédiaire du module **INTERFACE**. Il peut être initié de plusieurs sources en même temps.

Nous présentons l'algorithme en deux parties :

- l'algorithme suivi par le (ou les) CAC_{REPLAY} initiateur(s) de l'arrêt ;
- l'algorithme suivi par un CAC_{REPLAY} non initiateur de l'arrêt.

Algorithme pour le CAC_{REPLAY} initiateur :

```
arret de la reexecution du module
diffusion d'un message demandant l'arret aux autres modules
attente des reponses de tous les modules
prevenir le module INTERFACE que l'application est stoppee
```

Algorithme pour un CAC_{REPLAY} non initiateur :
après réception d'un message de demande d'arrêt :

```
arret de la reexecution du module
diffusion d'un message demandant l'arret aux autres modules-initiateur
attente des reponses de tous les modules sauf l'initiateur
envoi d'un message d'arret au module initiateur
```

3.3 Conclusion

Les points d'arrêt distribués pour le langage BOX dont nous venons de détailler la définition et l'implémentation permettent au programmeur d'utiliser la technique dite du « cyclic debugging » pour mettre au point ses programmes : une technique familière aux programmeurs.

Les points d'arrêt sont associés à des événements :

- des événements de base dont les différents types prennent en compte les deux paradigmes du langage : objet et processus communiquant ;
- des événements composés qui permettent au programmeur de spécifier des conditions d'arrêt d'un plus haut niveau d'abstraction sans se préoccuper de la distribution des entités.

Le filtrage des événements mis à la disposition du programmeur, en spécifiant une entité particulière pendant l'exécution (instance d'objet, instance de fragment, boîte aux lettres, message), lui permet de cerner petit à petit la localisation de l'erreur.

Notre méthode de détection et d'arrêt n'assure pas que le programme se trouvera exactement dans l'état décrit par l'événement associé au point d'arrêt. Mais elle assure que l'événement a bien été généré. De plus, grâce au mécanisme de réexécution, l'état global de l'application suspendue est cohérent. L'état atteint est un état « possible » de l'exécution, mais il n'a pas obligatoirement été atteint lors de l'exécution de référence.

Comme pour la réexécution, nous avons défini un mécanisme de détection distribué des événements composés. Celui-ci peut paraître lourd. Mais il faut prendre en considération que la détection se déroule pendant une phase de réexécution et ne peut affecter l'ordre observé sur les événements durant l'exécution initiale. Un protocole minimal est nécessaire, comme nous l'avons signalé dans la section précédente, est nécessaire

afin d'éviter la détection de « faux événements », la perte de certains événements ou la prise en compte multiple du même événement pour la génération d'un seul événement composé.

Nous avons précisé toutes les spécifications de l'implémentation des algorithmes de détection, l'implémentation est en cours sur le réseau de stations de travail.

Conclusion

Les travaux présentés dans cette thèse ont pour finalité la conception et la réalisation d'outils de mise au point pour l'environnment orienté objet parallèle développé par l'équipe PVC/BOX.

Le déverminage de programmes parallèles et distribués est compliqué par plusieurs faits :

- le non-déterminisme inhérent à la distribution et au parallélisme : deux exécutions successives peuvent ne pas avoir le même comportement et l'observation de l'exécution peut en modifier le comportement ;

- la difficulté d'obtenir un état global du programme qui est composé de l'état de chaque processus et distribué sur chaque site ;

- la difficulté d'arrêter un programme distribué dans un état cohérent ;

- la quantité d'informations à analyser et à présenter est plus importante que pour la programmation séquentielle.

Notre objectif était de permettre au programmeur l'utilisation du déverminage cyclique, impossible par les techniques classiques dans le cadre des applications parallèles et distribuées. Le travail réalisé est constitué de deux parties : un mécanisme de réexécution et l'adaptation de la notion de point d'arrêt pour les applications distribuées du langage BOX. Nous résumerons nos propositions dans la première partie de cette conclusion. Puis, nous exposerons les développements nécessaires et les perspectives offertes par le travail réalisé.

Travail réalisé

Mécanisme de réexécution

Nous avons défini et développé un mécanisme de réexécution pour le modèle CAC, adapté également à la reproduction des exécutions des programmes BOX. Le principal intérêt de ce mécanisme est de reproduire une exécution de référence à partir d'une trace d'événements destinés à lever le non-déterminisme de l'exécution. L'utilisation du déverminage cyclique est ainsi rendue possible.

La réexécution comporte deux principales difficultés :

- la perturbation du comportement de l'application du à la sauvegarde des traces ;

- le contrôle de la réexécution.

Nous avons opté pour une récolte logicielle de la trace. Une perturbation nulle de l'exécution n'était donc pas possible. Il s'agissait de la minimiser. Le choix d'une réexécution dirigée par le contrôle a permis de réduire la taille de la trace à sauvegarder et ainsi la perturbation sur l'exécution. Au chapitre 2, nous avons montré que notre mécanisme offrait de bonnes performances. La perturbation reste en dessous de la limite des 10%, très en dessous sur transputer et proche sur réseau de stations de travail.

Le contrôle de la réexécution est distribué dans chaque module composant l'application. Indépendamment les uns de autres, un CAC spécialisé dans chaque module est chargé de conserver l'ordre des événements internes au module. Nous avons montré que l'ordre des événements observé durant l'exécution de référence était conservé par la réexécution.

La réexécution du langage BOX apporte un problème supplémentaire : l'accès concurrent aux attributs d'un objet passif. Ces accès ne sont pas conservés par le mécanisme de réexécution : une réexécution peut donc être bloquée si les accès ne se font pas dans le même ordre. Mais nous avons montré que la détection des accès concurrents potentiels étaient possibles durant la réexécution et pouvaient être signalés au programmeur.

Le type de réexécution développée oblige une réexécution totale de l'application, ce qui peut être long et fastidieux. C'est pourquoi nous avons développé un mécanisme de réexécution partielle qui permet de ne réexécuter qu'un seul module : cela réduit le temps nécessaire à la réexécution et l'utilisation de l'architecture distribuée. Les traces nécessaires à la réexécution partielle, plus importantes que pour une réexécution totale, sont générées pendant une réexécution totale ce qui ne produit aucune perturbation sur l'application.

Points d'arrêt distribués

Nous avons défini des points d'arrêt distribués pour les programmes écrits en langage BOX. Ces points d'arrêts sont associés à des événements. Des événements de base couvrent les deux aspects du langage BOX : l'aspect objet et l'aspect acteur communiquant. Des possibilités de filtrage permettent au programmeur d'affiner la définition de ses points d'arrêt et de cerner petit à petit l'erreur recherchée. Il a également la possibilité de composer les événements de base pour définir des événements d'un plus haut niveau d'abstraction et d'y associer également des points d'arrêt.

L'utilisation de ces points d'arrêt est réalisée pendant une réexécution de l'application. Nous avons proposé des algorithmes distribués de détection des événements composés. La détection ne peut perturber l'application car elle se déroule pendant une réexécution. Ces algorithmes évitent :

- la prise en compte de l'occurrence d'un même événement plusieurs fois pour la génération d'un événement ;
- la détection de faux événements ;
- la non-détection de certains événements.

Développements et Perspectives

Développements

Nous devons améliorer les outils existants :

- implémenter pour la réexécution, la détection des accès concurrents aux attributs d'un objet passif ;
- prouver les algorithmes de détection d'événements composés ;
- poursuivre l'implémentation de ces algorithmes et de l'utilisation des points d'arrêt au dessus de la réexécution.

Une évaluation auprès de programmeurs BOX de l'utilisation des outils proposés est nécessaire :

- la granularité du filtrage est-elle assez fine ?
- les opérateur proposés permettent-ils de définir tous les événements dont le programmeur a besoin ?

Perspectives

Les environnements de programmation parallèle et distribuée devraient inclure, dès leur conception, des outils de mise au point. En particulier, nous avons montré que la réexécution de programmes étaient la base pour la construction de tels outils :

dévermineur symbolique avec points d'arrêt, outils de visualisation. Nous avons établi qu'un tel mécanisme implémenté de manière logicielle offrait de bonnes performances.

Un outil de visualisation n'a pas été développé pour les programmes BOX. Pourtant, ce type d'outil permet une vision globale des applications parallèles et de bien comprendre son comportement. L'utilisation conjointe d'un outil de visualisation et des points d'arrêt distribués directement au dessus de la réexécution fourniraient un bon environnement de déverminage, l'outil de visualisation pouvant être utilisé seul, les traces qui lui seraient nécessaires seraient alors générées durant une réexécution.

Pour obtenir un environnement complet, nous comptons associer également un dévermineur séquentiel pour le corps des méthodes.

La réexécution peut servir à d'autres tâches que le déverminage :

- observer finement l'exécution d'un programme ;
- effectuer des analyses de performances ;
- analyser le degré de parallélisme.

Annexe A

Interface de programmation C_AC

AddArg

AddArg ajoute un argument dans un message.

Synopsis: int AddArg(Mess_Ptr mes , int size, char *arg);

Paramètres:

mes: pointeur sur le message

size: taille en octets de l'argument à ajouter

arg: pointeur sur les octets à ajouter

Description: Cette fonction ajoute *size* octets pointés par *arg* au message pointé par *mes*. Le message doit être préalablement alloué par *NewMessage*. Il faut utiliser pour le premier paramètre la fonction *AddFirstArg*. *size* est ajouté au champ *offset* du message. Cette fonction renvoie 0 en cas de succès, -1 sinon (ajout de l'argument dépasse la taille allouée pour le message).

Voire aussi: AddFirstArg, NewMessage

AddFirstArg

AddFirstArg ajoute le premier argument dans un message.

Synopsis: int AddFirstArg(Mess_Ptr mes , int size, char *arg);

Paramètres:

mes: pointeur sur le message

size: taille en octets de l'argument à ajouter

arg: pointeur sur les octets à ajouter

Description: Cette fonction ajoute *size* octets pointés par *arg* au message pointé par *mes*. Le message doit être préalablement alloué par *NewMessage*. Il faut l'utiliser avant toute utilisation de *AddArg*. Elle positionne à *size* le champ *offset* du message. Cette fonction renvoie 0 en cas de succès, -1 sinon (ajout de l'argument dépasse la taille allouée pour le message).

Voire aussi: AddArg, NewMessage

ChecKOwner

ChecKOwner teste le propriétaire d'une boîte aux lettres.

Synopsis: word ChecKOwner (WORD bal, word own);

Paramètres:

bal: nom de la boîte aux lettres.

own: nom du CAC propriétaire supposé.

Description: Cette fonction retourne vrai si *own* est le propriétaire de la boîte aux lettres *bal.*

Voire aussi: SetOwner

CloseFlot

CloseFlot ferme un flot associé à un fichier.

Synopsis: int CloseFlot(Component me,Component flot);

Paramètres:

me: nom du CAC demandant la fermeture.

flot: nom du flot.

Description: Cette fonction permet de fermer un flot associé à un fichier. Le CAC gérant le fichier le ferme et s'arrête. Cette fonction renvoie 0 en cas de succès, -1 sinon.

Voire aussi: OpenFlot

ConsultBox

ConsultBox consulte une boîte aux lettres.

Synopsis: int ConsultBox (WORD me,WORD bal, int n, char **message, int *length);

Paramètres:

me: nom du CAC demandant la consultation.

bal: nom de la boîte aux lettres devant être consultée.

n: numéro du message devant être consulté.

message: message devant être consulté.

length: longueur du message devant être consulté.

Description: Cette fonction permet de consulter une boîte aux lettres. ***message** pointera sur le message numéro *n* de la boîte aux lettres et ***length* contiendra sa longueur. Attention, ***message* pointe réellement sur le message de la boîte aux lettres et non sur une copie. Elle renvoie 0 en cas de succès, -1 sinon (boîte distante ou inexistante, message n inexistant).

Voire aussi: DeleteMessageFromBox, ExtractBox, NumberMessageInBox, ReadFromBox

DeleteMessageFromBox

***DeleteMessageFromBox** détruit un message d'une boîte aux lettres.*

Synopsis: int DeleteMessageFromBox (WORD bal, int n);

Paramètres:

bal: nom de la boîte aux lettres.

n: numéro du message devant être détruit.

Description: Cette fonction permet de détruire le message numéro *n* de la boîte aux lettres *bal*. Cette fonction renvoie 0 en cas de succès, -1 sinon (boîte distante ou inexistante, message numéro n inexistant).

Voire aussi: ConsultBox, ExtractBox, NumberMessageInBox, ReadFromBox

DiffuseName

***DiffuseName** lance la phase d'initialisation des modules.*

Synopsis: void DiffuseName(void);

Paramètres:

aucun.

Description: Cette fonction permet d'initialiser les modules. Elle doit être appelée après les appels à *InitModule* et *InsertBehavior*. Le *main* d'un module doit être de la forme:

```
int main(int argc, char ** argv)
{
  InitModule(argc,argv,1);
  InsertBehavior(fac,B_FAC);
  DiffuseName();
  EndModule();
}
```

Voire aussi: InitModule, InsertBehavior

dispose

***dispose** libère la mémoire.*

Synopsis: void dispose (void *p);

Paramètres:

p: pointeur sur la zone mémoire à libérer.

Description: Cette fonction permet de libérer la zone mémoire pointé par *p*.

Voire aussi: new

DumpMessage

***DumpMessage** affiche le contenu d'un message.*

Synopsis: void DumpMessage(Mess_Ptr mes);

Paramètres:
mes: pointeur sur message à afficher.

Description: Cette fonction permet d'afficher le contenu du message pointé par *mes*.

EndComponent

***EndComponent** termine un CAC.*

Synopsis: void EndComponent(Component me);

Paramètres:
me: nom du CAC.

Description: Cette fonction permet de terminer un CAC. Elle doit être appelée en dernier dans le code du CAC de boîte de nom *me*. Elle procède à la libération des ressources utilisées par le CAC.

Voire aussi: NewComponent

EndModule

***EndModule** termine un module.*

Synopsis: void EndModule(void);

Paramètres:
aucun.

Description: Cette fonction permet de terminer un module. Elle doit être appelée en dernier dans le code du *main* du module, qui doit se bloquer sur cette primitive. Le *main* d'un module doit être de la forme:

```
int main(int argc, char ** argv)
{
  InitModule(argc,argv,1);
  InsertBehavior(fac,B_FAC);
  DiffuseName();
  EndModule();
}
```

Voire aussi: DiffuseName, InitModule, InsertBehavior

ExtractBox

***ExtractBox** extraie un message d'une boîte aux lettres.*

Synopsis: int ExtractBox (WORD me,WORD bal, int n, char **message, int *length);

Paramètres:

me: nom du CAC demandant la consultation.

bal: nom de la boite aux lettres.

n: numéro du message devant être extrait.

message: message devant être extrait.

length: longueur du message devant être extrait.

Description: Cette fonction permet d'extraire un message d'une boite aux lettres. **message* pointera sur le message numéro *n* de la boîte aux lettres et **length* contiendra sa longueur. Attention, le message n'est plus contenu dans la boite aux lettres. Cette fonction renvoie 0 en cas de succès, -1 sinon (boite distante ou inexistante, message numéro n inexistant).

Voire aussi: ConsultBox, DeleteMessageFromBox, NumberMessageInBox, ReadFromBox

FFlush

***FFlush** vide le tampon associé au flot.*

Synopsis: int FFlush(Component me,Component flot);

Paramètres:

me: nom du CAC appelant la primitive.

flot: nom du flot.

Description: Cette fonction permet de vider le tampon du fichier associé à un flot. Le flot doit être ouvert en écriture. Cette fonction renvoie 0 en cas de succès, -1 sinon.

Voire aussi: OpenFlot

Flush

***Flush** vide le tampon associé au flot de sortie standard.*

Synopsis: void Flush(Component me);

Paramètres:

me: nom du CAC appelant la primitive.

Description: Cette fonction permet de vider le tampon du fichier associé àu flot de sortie standard.

Voire aussi:

FPrintChar

FPrintChar écrit un caractère sur un flot.

Synopsis: int FPrintChar(Component me,Component flot,char x);

Paramètres:

me: nom du CAC demandant l'écriture.

flot: nom du flot.

x: caractère à écrire.

Description: Cette fonction permet d'écrire le caractère *x* sur le fichier associé à *flot*. Le flot doit être ouvert en écriture. Cette fonction renvoie 0 en cas de succès, -1 sinon.

Voire aussi: FPrintDouble, FPrintFloat, FPrintInt, FPrintString, OpenFlot

FPrintDouble

FPrintDouble écrit un double sur un flot.

Synopsis: int FPrintDouble(Component me,Component flot,double x);

Paramètres:

me: nom du CAC demandant l'écriture.

flot: nom du flot.

x: double à écrire.

Description: Cette fonction permet d'écrire le double *x* sur le fichier associé à *flot*. Le flot doit être ouvert en écriture. Cette fonction renvoie 0 en cas de succès, -1 sinon.

Voire aussi: FPrintChar, FPrintFloat, FPrintInt, FPrintString, OpenFlot

FPrintFloat

FPrintFloat écrit un float sur un flot.

Synopsis: int FPrintFloat(Component me,Component flot,float *x);

Paramètres:

me: nom du CAC demandant l'écriture.

flot: nom du flot.

x: pointeur sur le float à écrire.

Description: Cette fonction permet d'écrire le float **x* sur le fichier associé à *flot*. Le flot doit être ouvert en écriture. Cette fonction renvoie 0 en cas de succès, -1 sinon.

Voire aussi: FPrintChar, FPrintDouble, FPrintInt, FPrintString, OpenFlot

FPrintInt

FPrintInt écrit un entier sur un flot.

Synopsis: int FPrintInt(Component me,Component flot,int x);

Paramètres:

me: nom du CAC demandant l'écriture.

flot: nom du flot.

x: entier à écrire.

Description: Cette fonction permet d'écrire l'entier *x* sur le fichier associé à *flot*. Le flot doit être ouvert en écriture. Cette fonction renvoie 0 en cas de succès, -1 sinon.

Voire aussi: FPrintChar, FPrintDouble, FPrintFloat, FPrintString, OpenFlot

FPrintString

FPrintString écrit une chaine de caractères sur un flot.

Synopsis: int FPrintString(Component me,Component flot,char *x);

Paramètres:

me: nom du CAC demandant l'écriture.

flot: nom du flot.

x: chaine à écrire.

Description: Cette fonction permet d'écrire la chaine pointé par *x* sur le fichier associé à *flot*. Le flot doit être ouvert en écriture. Cette fonction renvoie 0 en cas de succès, -1 sinon.

Voire aussi: FPrintChar, FPrintDouble, FPrintFloat, FPrintInt, OpenFlot

FRead

FRead lit un certain nombre d'octets à partir d'un flot.

Synopsis: int FRead(Component me,Component flot,int nb,char *x);

Paramètres:

me: nom du CAC demandant la lecture.

flot: nom du flot.

nb: nombre d'octets à lire.

x: pointeur sur les octets lus.

Description: Cette fonction permet de lire *nb* octets sur le fichier associé à *flot*. **x* pointera sur ces octets lus et doit déjà être alloué. Le flot doit être ouvert en lecture. Cette fonction renvoie le nombre d'octets lus en cas de succès, -1 en cas d'erreur.

Voire aussi: FWrite

FReadChar

***FReadChar** lit un caractère à partir d'un flot.*

Synopsis: int FReadChar(Component me,Component flot,char *x);

Paramètres:

me: nom du CAC demandant la lecture.

flot: nom du flot.

x: pointeur sur le caractère à lire.

Description: Cette fonction permet de lire un caractère sur le fichier associé à *flot*. **x* contiendra le caractère lu. Le flot doit être ouvert en lecture. Cette fonction renvoie 1 en cas de succès, 0 si le type ne correspond pas, -1 en fin de fichier.

Voire aussi: FReadDouble, FReadFloat, FReadInt, FReadString, OpenFlot

FReadDouble

***FReadDouble** lit un double à partir d'un flot.*

Synopsis: int FReadDouble(Component me,Component flot,double *x);

Paramètres:

me: nom du CAC demandant la lecture.

flot: nom du flot.

x: pointeur sur le double à lire.

Description: Cette fonction permet de lire un double sur le fichier associé à *flot*. **x* contiendra le double lu. Le flot doit être ouvert en lecture. Cette fonction renvoie 1 en cas de succès, 0 si le type ne correspond pas, -1 en fin de fichier.

Voire aussi: FReadChar, FReadFloat, FReadInt, FReadString, OpenFlot

FReadFloat

***FReadFloat** lit un float à partir d'un flot.*

Synopsis: int FReadFloat(Component me,Component flot,float *x);

Paramètres:

me: nom du CAC demandant la lecture.

flot: nom du flot.

x: pointeur sur le float à lire.

Description: Cette fonction permet de lire un float sur le fichier associé à *flot*. **x* contiendra le float lu. Le flot doit être ouvert en lecture. Cette fonction renvoie 1 en cas de succès, 0 si le type ne correspond pas, -1 en fin de fichier.

Voire aussi: FReadChar, FReadDouble, FReadInt, FReadString, OpenFlot

FReadInt

FReadInt lit un entier à partir d'un flot.

Synopsis: int FReadInt(Component me,Component flot,int *x);

Paramètres:

me: nom du CAC demandant la lecture.

flot: nom du flot.

x: pointeur sur l'entier à lire.

Description: Cette fonction permet de lire un entier sur le fichier associé à *flot*. ***x** contiendra l'entier lu. Le flot doit être ouvert en lecture. Cette fonction renvoie 1 en cas de succès, 0 si le type ne correspond pas, -1 en fin de fichier.

Voire aussi: FReadChar, FReadDouble, FReadFloat, FReadString, OpenFlot

FReadString

FReadString lit une chaine de caractères à partir d'un flot.

Synopsis: int FReadString(Component me,Component flot,int size_max, char *x);

Paramètres:

me: nom du CAC demandant la lecture.

flot: nom du flot.

size_max: taille maximale de la chaine à lire.

x: chaine à lire.

Description: Cette fonction permet de lire une chaine de caractères sur le fichier associé à *flot*. *x* contiendra la chaine lue. Le flot doit être ouvert en lecture. Cette fonction renvoie 1 en cas de succès, 0 si le type ne correspond pas, -1 en fin de fichier.

Voire aussi: FReadChar, FReadDouble, FReadFloat, FReadInt, OpenFlot

FreeBox

FreeBox libère une boîte aux lettres.

Synopsis: int FreeBox (WORD bal);

Paramètres:

bal: boîte aux lettres à libérer.

Description: Cette fonction permet de libérer une boîte aux lettres et aussi tous les messages qu'elle pouvait contenir. Cette fonction renvoie 0 en cas de succès, -1 sinon (boîte distante ou inexistante, message numéro n inexistant).

Voire aussi: NewBox

FreeMessage

***FreeMessage** libère un message.*

Synopsis: void FreeMessage(Mess_Ptr mes);

Paramètres:

mes: pointeur sur le message à libérer.

Description: Cette fonction permet de libérer un message.

Voire aussi: NewMessage

FWrite

***FWrite** écrit un certain nombre d'octets sur un flot.*

Synopsis: int FWrite(Component me,Component flot,int nb,char *x);

Paramètres:

me: nom du CAC demandant l'écriture.

flot: nom du flot.

nb: nombre d'octets à écrire.

x: pointeur sur les octets à écrire.

Description: Cette fonction permet d'écrire *nb* octets sur le fichier associé à *flot*. **x* pointe sur ces octets. Le flot doit être ouvert en écriture. Cette fonction renvoie le nombre d'octets écrits en cas de succès, -1 en cas d'erreur.

Voire aussi: FRead

GetFirstArg

***GetFirstArg** renvoie un pointeur sur le premier argument dans un message.*

Synopsis: char *GetFirstArg(Mess_Ptr mes , int size);

Paramètres:

mes: pointeur sur le message

size: taille en octets de l'argument

Description: Cette fonction renvoie un pointeur sur le premier argument du champ donnee du message pointé par *mes*. Il faut l'utiliser avant toute utilisation de *GetNextArg*. Elle positionne à *size* le champ *offset* du message. Cette fonction renvoie NULL en cas d'erreur.

Voire aussi: GetNextArg

GetMessage

***GetMessage** renvoie le premier message d'une boîte aux lettres.*

Synopsis: void GetMessage(Component me,Component c,Mess_Ptr *m);

Paramètres:

me: nom du CAC appelant la primitive

c: nom de la boîte aux lettres

m: contiendra un pointeur sur le message

Description: Cette fonction renvoie le premier message de la boîte aux lettres *c* et **m* pointera sur ce message. Cette primitive est bloquante s'il n'y a pas de messages.

Voire aussi: GetMessageTF, GetMessageTFR, GetReply, GetRequest

GetMessageTF

***GetMessageTF** renvoie le premier message de type et de fonction donnés d'une boîte aux lettres.*

Synopsis: void GetMessageTF(Component me,Component c,int type,int function,Mess_Ptr *m);

Paramètres:

me: nom du CAC appelant la primitive

c: nom de la boîte aux lettres

type: type du message attendu

function: fonction du message attendu

m: contiendra un pointeur sur le message

Description: Cette fonction renvoie le premier message de type *type* et de fonction *function* de la boîte aux lettres *c* et **m* pointera sur ce message. Cette primitive est bloquante s'il n'y a pas de messages correspondant.

Voire aussi: GetMessage, GetMessageTFR, GetReply, GetRequest

GetMessageTFR

***GetMessageTFR** renvoie le premier message de type, de fonction et de requête donnés d'une boîte aux lettres.*

Synopsis: void GetMessageTFR(Component me,Component c,int type,int func,int req, Mess_Ptr *m);

Paramètres:

me: nom du CAC appelant la primitive

c: nom de la boîte aux lettres

type: type du message attendu

func: fonction du message attendu

req: requête du message attendu

m: contiendra un pointeur sur le message

Description: Cette fonction renvoie le premier message de type *type*, de fonction *function* et de requête *req* de la boîte aux lettres *c* et **m* pointera sur ce message. Cette primitive est bloquante s'il n'y a pas de messages correspondant.

Voire aussi: GetMessage, GetMessageTF, GetReply, GetRequest

GetNextArg

***GetNextArg** renvoie un pointeur sur l'argument suivant d'un message.*

Synopsis: char *GetNextArg(Mess_Ptr mes , int size);

Paramètres:

mes: pointeur sur le message

size: taille en octets de l'argument

Description: Cette fonction renvoie un pointeur sur l'argument suivant du champ donnee du message pointé par *mes*. Il faut l'utiliser après un premier appel à *GetFirstArg*. Elle ajoute *size* au champ *offset* du message. Cette fonction renvoie NULL en cas d'erreur.

Voire aussi: GetFirstArg

GetReply

***GetReply** renvoie le premier message de type M_USER et de fonction M_REPLY d'une boîte aux lettres.*

Synopsis: void GetReply(Component me,Component c,Mess_Ptr *m);

Paramètres:

me: nom du CAC appelant la primitive

c: nom de la boîte aux lettres

m: contiendra un pointeur sur le message

Description: Cette fonction renvoie le premier message de type M_USER et de fonction M_REPLY de la boîte aux lettres *c* et **m* pointera sur ce message. Cette primitive est bloquante s'il n'y a pas de messages correspondant.

Voire aussi: GetMessage, GetMessageTF, GetMessageTFR, GetRequest

GetRequest

***GetRequest** renvoie le premier message de type M_USER et de fonction M_REQUEST d'une boîte aux lettres.*

Synopsis: void GetRequest(Component me,Component c,Mess_Ptr *m);

Paramètres:

me: nom du CAC appelant la primitive

c: nom de la boîte aux lettres

m: contiendra un pointeur sur le message

Description: Cette fonction renvoie le premier message de type M_USER et de fonction M_REQUEST de la boîte aux lettres *c* et **m* pointera sur ce message. Cette primitive est bloquante s'il n'y a pas de messages correspondant.

Voire aussi: GetMessage, GetMessageTF, GetMessageTFR, GetReply

GetUser1

***GetUser1** renvoie la valeur du champ user1 d'une boîte aux lettres.*

Synopsis: word GetUser1 (WORD bal);

Paramètres:

bal: nom de la boîte aux lettres

Description: Cette fonction renvoie la valeur du champ user1 de la boîte aux lettres *c*. Ce champ est laissé libre pour l'utilisateur.

Voire aussi: SetUser1

InitModule

InitModule débute l'initialisation d'un module.

Synopsis: void InitModule(int argc,char **argv,int nb_b);

Paramètres:

argc: nombre de paramètres du module

argv: paramètres du module

nb_b: nombre de comportements du module

Description: Cette fonction débute l'initialisation d'un module. Elle doit être appelée en premier dans le code du main du module. Le *main* d'un module doit être de la forme:

```
int main(int argc, char ** argv)
{
  InitModule(argc,argv,n);
  InsertBehavior(fonc1,B_1);
                .
                .
                .
  InsertBehavior(foncn,B_n);
  DiffuseName();
  EndModule();
}
```

Voire aussi: DiffuseName, EndModule, InsertBehavior

InsertBehavior

InsertBehavior déclare un nouveau comportement dans le module.

Synopsis: void InsertBehavior(VoidFnPtr f , int n);

Paramètres:

f: fonction, code du comportement

n: nom du comportement

Description: Cette fonction déclare un nouveau comportement dans le module. Elle doit être appelé entre l'appel à *InitModule* et à *DiffuseName* dans le code du main du module. Le *main* d'un module doit être de la forme:

```
int main(int argc, char ** argv)
{
  InitModule(argc,argv,1);
  InsertBehavior(fac,B_FAC);
  DiffuseName();
  EndModule();
}
```

Voire aussi: DiffuseName, EndModule, InitModule

KillComponent

***KillComponent** envoie un message de type M_SYSTEM, fonction M_REQUEST et requête KILL_C.*

Synopsis: void KillComponent(Component me,Component c);

Paramètres:

me: nom du CAC appelant la primitive

c: nom de la boîte aux lettres du CAC destinataire

Description: Cette fonction envoie un message de type M_SYSTEM, fonction M_REQUEST et requête KILL_C au CAC de boîte aux lettres *c*. Ce message correspond à une demande d'arrêt pour le CAC de boîte aux lettres *c*. Le CAC doit explicitement lire le message de destruction.

Voire aussi:

LockOutput

***LockOutput** réserve la sortie standard.*

Synopsis: void LockOutput(Component me);

Paramètres:

me: nom du CAC appelant la primitive

Description: Cette fonction permet au CAC de boîte aux lettres *me* de se réserver la sortie standard. Ainsi, ses messages ne seront pas entrecoupés par les messages des autres CAC.

Voire aussi: UnLockOutput

new

***new** alloue de la mémoire.*

Synopsis: void *new (int size);

Paramètres:

size: nombre d'octets à allouer

Description: Cette fonction alloue *size* octets de mémoire. Elle protège des accès concurrents à la mémoire.

Voire aussi: dispose

NewBox

***NewBox** crée une boîte aux lettres.*

Synopsis: WORD NewBox (Component me);

Paramètres:

me: boîte aux lettres du CAC créateur

Description: Cette fonction crée une boîte aux lettres dans le module courant et renvoie son nom.

Voire aussi: FreeBox

NewComponent

***NewComponent** crée nouveau CAC.*

Synopsis: Component NewComponent(Component me,int beh,int num,int size,char *args);

Paramètres:

me: boîte aux lettres du CAC demandant la création
beh: nom du comportement (type) du CAC à créer
num: lieu de création
size: taille en octets des paramètres du CAC
args: pointeur sur les paramètres du CAC

Description: Cette fonction crée un CAC de nom *beh* avec *size* octets pointés par *args* comme arguments. La fonction renvoie le nom de la boîte aux lettres du CAC créé. L'endroit de la création est déterminé par la valeur de *num*:

VOID: le choix du module pour la création est laisse au système
LOCAL: le CAC est créé dans le module courant
autre: donne le numéro du module pour la création

Voire aussi: FreeComponent

NewMessage

***NewMessage** alloue un message.*

Synopsis: void NewMessage(Mess_Ptr *mes, int s);

Paramètres:

mes: contiendra un pointeur sur le message
s: taille du champ donnée du message

Description: Cette fonction alloue un message dont le champ donnée a *s* octets.

Voire aussi: FreeMessage

NumberMessageInBox

***NumberMessageInBox** renvoie le nombre de messages d'une boîte aux lettres.*

Synopsis: int NumberMessageInBox (WORD me,WORD bal);

Paramètres:

me: nom du CAC demandant la consultation.
bal: nom de la boîte aux lettres.

Description: Cette fonction renvoie le nombre de messages d'une boîte aux lettres. Attention, l'appel suivant à *WaitOnBox* sera bloquant si aucun nouveau message n'est déposé dans la boîte. Cette fonction renvoie -1 en cas d'erreur.

Voire aussi: ConsultBox, DeleteMessageFromBox, ExtractBox, ReadFromBox, WaitOnBox

OpenFlot

***OpenFlot** ouvre un flot associé à un fichier.*

Synopsis: Component OpenFlot(Component me,int mode,char *name);

Paramètres:

me: nom du CAC demandant l'ouverture.

mode: mode d'ouverture.

name: nom du fichier.

Description: Cette fonction permet d'ouvrir un flot associé à un fichier. Cette ouverture consiste, en fait, à créer un CAC de comportement B_FLOT prédéfini. Il faut donc « insérer » le comportement B_FLOT dans au moins un module par la commande *InsertBehavior(cac_flot,B_FLOT)*. Le mode d'ouverture est soit READ_ONLY pour la lecture, soit WRITE_ONLY pour l'écriture. Cette fonction renvoie NULLCOMPONENT en cas d'erreur.

Voire aussi: CloseFlot

PrintChar

***PrintChar** écrit un caractère sur la sortie standard.*

Synopsis: void PrintChar(Component me,char x);

Paramètres:

me: nom du CAC demandant l'écriture.

x: caractère à écrire.

Description: Cette fonction permet d'écrire le caractère *x* sur la sortie standard.

Voire aussi: PrintDouble, PrintFloat, PrintInt, PrintString

PrintDouble

***PrintDouble** écrit un double sur la sortie standard.*

Synopsis: void PrintDouble(Component me,double x);

Paramètres:

me: nom du CAC demandant l'écriture.

x: double à écrire.

Description: Cette fonction permet d'écrire le double *x* sur sur la sortie standard.

Voire aussi: PrintChar, PrintFloat, PrintInt, PrintString

PrintFloat

PrintFloat écrit un float sur la sortie standard.

Synopsis: void PrintFloat(Component me,float *x);

Paramètres:

me: nom du CAC demandant l'écriture.

x: pointeur sur le float à écrire.

Description: Cette fonction permet d'écrire le float **x* sur la sortie standard.

Voire aussi: PrintChar, PrintDouble, PrintInt, PrintString

PrintInt

PrintInt écrit un entier sur la sortie standard.

Synopsis: void PrintInt(Component me,int x);

Paramètres:

me: nom du CAC demandant l'écriture.

x: entier à écrire.

Description: Cette fonction permet d'écrire l'entier *x* sur la sortie standard.

Voire aussi: PrintChar, PrintDouble, PrintFloat, PrintString

PrintString

PrintString écrit une chaine de caractères sur la sortie standard.

Synopsis: void PrintString(Component me,char *x);

Paramètres:

me: nom du CAC demandant l'écriture.

x: chaine à écrire.

Description: Cette fonction permet d'écrire la chaine pointée par *x* sur la sortie standard.

Voire aussi: PrintChar, PrintDouble, PrintFloat, PrintInt

PutToBox

PutToBox dépose un message dans une boîte aux lettres.

Synopsis: void PutToBox (WORD bal, char *message,int length);

Paramètres:

bal: nom de la boîte aux lettres

message: pointeur sur le message

length: longueur du message (entête + données)

Description: Cette fonction dépose le message *message* dans la boite aux lettres *bal*.

Voire aussi: ConsultBox, DeleteMessageFromBox, ExtractBox, NumberMessageInBox, ReadFromBox, WaitOnBox

ReadChar

***ReadChar** lit un caractère sur l'entrée standard.*

Synopsis: int ReadChar(Component me,char *x);

Paramètres:

me: nom du CAC demandant la lecture.

x: pointeur sur le caractère à lire.

Description: Cette fonction permet de lire un caractère sur l'entrée standard. Cette fonction renvoie 1 en cas de succès, 0 si le type ne correspond pas, -1 en fin de fichier.

Voire aussi: ReadDouble, ReadFloat, ReadInt, ReadString

ReadDouble

***ReadDouble** lit un double sur l'entrée standard.*

Synopsis: int ReadDouble(Component me,double *x);

Paramètres:

me: nom du CAC demandant la lecture.

x: pointeur sur le double à lire.

Description: Cette fonction permet de lire un double sur l'entrée standard. Cette fonction renvoie 1 en cas de succès, 0 si le type ne correspond pas, -1 en fin de fichier.

Voire aussi: ReadChar, ReadFloat, ReadInt, ReadString

ReadFloat

***ReadFloat** lit un float sur l'entrée standard.*

Synopsis: int ReadFloat(Component me,float *x);

Paramètres:

me: nom du CAC demandant la lecture.

x: pointeur sur le float à lire.

Description: Cette fonction permet de lire un float sur l'entrée standard. Cette fonction renvoie 1 en cas de succès, 0 si le type ne correspond pas, -1 en fin de fichier.

Voire aussi: ReadChar, ReadDouble, ReadInt, ReadString

ReadFromBox

ReadFromBox consulte une boîte aux lettres.

Synopsis: int ReadFromBox (WORD me,WORD bal, int n, char **message,int *length);

Paramètres:

me: nom du CAC demandant la consultation.

bal: nom de la boîte aux lettres devant être consultée.

n: numéro du message devant être consulté.

message: message devant être consulté.

length: longueur du message devant être consulté.

Description: Cette fonction permet de consulter une boîte aux lettres. **message* pointera sur le message numéro *n* de la boîte aux lettres et **length* contiendra sa longueur. Attention, **message* pointe sur une copie du message de la boîte aux lettres et non sur le vrai message. Cette fonction renvoie 0 en cas de succès, -1 sinon (boîte distante ou inexistante, message numéro n inexistant).

Voire aussi: ConsultBox, DeleteMessageFromBox, ExtractBox, NumberMessageInBox

ReadInt

ReadInt lit un entier sur l'entrée standard.

Synopsis: int ReadInt(Component me,int *x);

Paramètres:

me: nom du CAC demandant la lecture.

x: pointeur sur l'entier à lire.

Description: Cette fonction permet de lire un entier sur l'entrée standard. Cette fonction renvoie 1 en cas de succès, 0 si le type ne correspond pas, -1 en fin de fichier.

Voire aussi: ReadChar, ReadDouble, ReadFloat, ReadString

ReadString

ReadString lit une chaine de caractères sur l'entrée standard.

Synopsis: int ReadString(Component me,int size_max,char *x);

Paramètres:

me: nom du CAC demandant la lecture.

size_max: taille maximale de la chaine à lire.

x: chaine à lire.

Description: Cette fonction permet de lire une chaine de caractères sur l'entrée standard. Cette fonction renvoie 1 en cas de succès, 0 si le type ne correspond pas, -1 en fin de fichier.

Voire aussi: ReadChar, ReadDouble, ReadFloat, ReadInt

Send

***Send** envoie un message.*

Synopsis: void Send(Component me,Component c,int size,char *arg);

Paramètres:

me: nom du CAC effectuant l'envoi.

c: nom de la boîte aux lettres destinataires

size: nombre d'octets à envoyer

arg: pointeur sur les octets à envoyer.

Description: Cette fonction construit un message de type M_USER, de fonction M_INFO, de requête VOID et dont le champ données est constitué des *size* octets pointés par *arg*. L'envoi est asynchrone.

Voire aussi: SendaMessage, SendaMessageTFR, SendReply, SendRequest

SendaMessage

***SendaMessage** envoie un message.*

Synopsis: void SendaMessage(Component me, Component c, Mess_Ptr m);

Paramètres:

me: nom du CAC effectuant l'envoi.

c: nom de la boîte aux lettres destinataires

m: pointeur sur le message à envoyer.

Description: Cette fonction dépose le message pointé par *m* dans la boîte aux lettres de nom *c*.

Voire aussi: Send, SendaMessageTFR, SendReply, SendRequest

SendaMessageTFR

***SendaMessageTFR** envoie un message.*

Synopsis: void SendaMessageTFR(Component me,Component c, int t,int f,int r,int size, char *arg);

Paramètres:

me: nom du CAC appelant la primitive

c: nom de la boîte aux lettres

t: type du message

f: fonction du message

r: requête du message

size: nombre d'octets à envoyer

arg: pointeur sur les octets à envoyer.

Description: Cette fonction construit un message de type *t*, de fonction *f*, de requête *r* et dont le champ données est constitué des *size* octets pointés par *arg*. L'envoi est asynchrone.

Voire aussi: Send, SendaMessage, SendReply, SendRequest

SendReply

SendReply envoie un message.

Synopsis: void SendReply(Component me,Component c,int size,char *val);

Paramètres:

me: nom du CAC effectuant l'envoi.

c: nom de la boîte aux lettres destinataires

size: nombre d'octets à envoyer

arg: pointeur sur les octets à envoyer.

Description: Cette fonction construit un message de type M_USER, de fonction M_REPLY, de requête VOID et dont le champ données est constitué des *size* octets pointés par *arg*. L'envoi est asynchrone.

Voire aussi: Send, SendaMessage, SendaMessageTFR, SendRequest

SendRequest

SendRequest envoie un message.

Synopsis: void SendRequest(Component me, Component c, int r,int size,char *arg);

Paramètres:

me: nom du CAC effectuant l'envoi.

c: nom de la boîte aux lettres destinataires

size: nombre d'octets à envoyer

arg: pointeur sur les octets à envoyer.

Description: Cette fonction construit un message de type M_USER, fonction M_REQUEST, requête VOID et dont le champ données est constitué des *size* octets pointés par *arg*. L'envoi est asynchrone.

Voire aussi: Send, SendaMessage, SendaMessageTFR, SendReply

SetHeader

SetHeader prépare un message.

Synopsis: void SetHeader(Mess_Ptr mes,Component me,Component c,int t,int f,int r);

Paramètres:

mes: pointeur sur le message.

me: nom du CAC envoyeur

c: boîte destinataire du message

t: type du message

f: fonction du message

r: requête du message

Description: Cette fonction initialise l'entête du message *mes*. Celui ci doit être préalablement alloué avec *NewMessage*.

Voire aussi: SetMessage

SetMessage

***SetMessage** prépare un message.*

Synopsis: void SetMessage(Mess_Ptr mes,Component s,Component d,int t,int f,int r,int size, char * arg);

Paramètres:

mes: pointeur sur le message.
s: nom du CAC envoyeur
d: boîte destinataire du message
t: type du message
f: fonction du message
r: requête du message
size: nombre d'octets à envoyer
arg: pointeur sur les octets à envoyer.

Description: Cette fonction initialise l'entête du message *mes* et le champ données avec *size* octets pointés par *arg*. Le message doit être préalablement alloué avec *NewMessage*.

Voire aussi: SetHeader

SetOwner

***SetOwner** positionne le propriétaire d'une boîte aux lettres.*

Synopsis: void SetOwner (WORD me,WORD bal, word own);

Paramètres:

me: nom du CAC appelant la primitive
bal: nom de la boîte aux lettres.
own: nom du propriétaire supposé.

Description: Cette fonction positionne le champ propriétaire de la boîte aux lettres *bal* à *own*.

Voire aussi: ChecKOwner

SetUser1

***SetUser1** positionne le champ user1 d'une boîte aux lettres.*

Synopsis: void SetUser1 (WORD bal, word val);

Paramètres:

bal: nom de la boîte aux lettres.
val: valeur à donner au champ user1.

Description: Cette fonction positionne le champ *user1* de la boîte aux lettres *bal* à *val*.

Voire aussi: GetUser1

StopModules

StopModules arrête l'application.

Synopsis: void StopModules(Component me);

Paramètres:

me: nom du CAC appelant la primitive

Description: Cette fonction arrête l'application. Le CAC qui l'appelle ne doit plus appeler autre chose que *EndComponent*.

Voire aussi:

UnLockOutput

UnLockOutput libére la sortie standard.

Synopsis: void UnLockOutput(Component me);

Paramètres:

me: nom du CAC appelant la primitive

Description: Cette fonction permet au CAC de boîte aux lettres *me* de libérer la sortie standard qu'il s'était réservé par un appel à *LockOutput*.

Voire aussi: LockOutput

WaitOnBox

WaitOnBox attend un nouveau message.

Synopsis: int WaitOnBox (WORD me,WORD bal);

Paramètres:

me: nom du CAC demandant la consultation.

bal: nom de la boîte aux lettres.

Description: Cette fonction renvoie le nombre de messages d'une boîte aux lettres. Attention, l'appel suivant à *WaitOnBox* est bloquante si aucun nouveau message n'est déposé dans la boîte depuis le dernier appel à *NumberMessageInBox* ou *WaitOnBox*. Cette fonction renvoie -1 en cas d'erreur.

Voire aussi: ConsultBox, DeleteMessageFromBox, ExtractBox, NumberMessageInBox, ReadFromBox

Annexe B

Exemples de programme CAC

B.1 L'application somme

L'application *somme* calcule en parallèle la somme des entiers de 1 à n. Un seul comportement est défini : le comportement *B_SOM*. Celui-ci reçoit en paramètre l'intervalle d'entiers dont il doit calculer la somme. Si l'intervalle est de cardinalité sup 'erieure à 1, le CAC crée deux nouveaux composants de même comportement chargés de calculer chacun la somme pour une moitié de l'intervalle. Le CAC attend les deux réponses, en effectue la somme et envoie le résultat à son créateur.

L'application est découpé en deux types de modules : `Module_Dia` qui est chargé de démarrer l'application et de récupérer le résultat, et `Module_Somme` qui abritera les CAC de comportements `B_SOM`.

B.1.1 Le module *Module_Dia*

```
/*************************************************************************/
/*                                                                       */
/*                          Module_Dia.c                                 */
/*                                                                       */
/*************************************************************************/

#include <Behavior.h>
#include <Prim_Cac.h>

typedef struct sum_param {
  Component reply;
  int inf;
  int sup;
} sum_param;
```

```
void lance(Component me, int *arg)
{
 /* comportement demarrant l'application */
 unsigned int Time;
 Component root;
 int limite, result, tps;
 Mess_Ptr mes;
 sum_param parameters;

  PrintString(me,"\n*****APPLI*****    \n\n");
  PrintString(me,"Limite :");Flush(me);
  ReadInt(me,&limite);

  Time = _cputime();

  parameters.reply = me;
  parameters.inf = 1;
  parameters.sup = limite;

  /* creation du premier composant de comportement B_SOM */
  root = NewComponent(me,B_SOM,VOID,sizeof(sum_param),(char *)&parameters);

  /* attente du resultat */
  GetMessage(me,me,&mes);
  result = *(int *)GetFirstArg(mes,sizeof(int));

  tps = _cputime() - Time;

  PrintString(me,"result summ  of 1 to ");
  PrintInt(me,limite);
  PrintString(me," = ");
  PrintInt(me,result);
  PrintString(me,"\n\n\n");
  PrintString(me,"execution time ");
  PrintInt(me, tps);
  PrintString(me," 1/100 s \n");
  Flush(me);

  /* arret de l'application */
  StopModules(me);
  EndComponent(me);
}


int main(int argc, char ** argv)
{
 word create;
  InitModule(argc,argv,1);
  InsertBehavior(lance,B_DIA);
  DiffuseName();
  create = NewBox(0);
  NewComponent(create,B_DIA,LOCAL,0,NULL);
  EndModule();
}
```

B.1.2 Le module *Module_Somme*

```
/*****************************************************************************/
/*                                                                           */
/*                          Module_Somme.c                                   */
/*                                                                           */
/*****************************************************************************/
#include <Behavior.h>
#include<Prim_Cac.h>

typedef struct sum_param {
  Component reply;
  int inf;
  int sup;
} sum_param;

void sum(Component me, sum_param *arg)
{
 Component left, right;
 int result;
 Mess_Ptr message;
 sum_param parameters;
  if (arg->inf == arg->sup)
    /* envoi du resultat au CAC createur */
    SendReply(me,arg->reply,sizeof(int),(char *) &(arg->inf));
  else {
    /* creation d'un CAC pour calculer la premiere partie de la somme */
    parameters.reply = me;
    parameters.inf = arg->inf;
    parameters.sup = (arg->inf+arg->sup)/2;
    left = NewComponent(me,B_FAC,VOID,sizeof(fac_param),(char *) &parameters);
    /* creation d'un CAC pour calculer la deuxieme partie de la somme */
    parameters.inf = ((arg->inf+arg->sup)/2)+1;
    parameters.sup = arg->sup;
    right = NewComponent(me,B_FAC,VOID,sizeof(fac_param),(char *) &parameters);
    /* attente du resultat de l'un des CAC */
    GetMessage(me,me,&message);
    result = *(int *)GetFirstArg(message,sizeof(int));
    FreeMessage(message);
    /* attente du resultat du dernier CAC */
    GetMessage(me,me,&message);
    result += *(int *)GetFirstArg(message,sizeof(int));
    FreeMessage(message);
    /* envoi du resultat au CAC createur */
    SendReply(me,arg->reply,sizeof(int),(char *) &result);
  }
  EndComponent(me);
}

int main(int argc, char ** argv)
{
  InitModule(argc,argv,1);
  InsertBehavior(sum,B_SOM);
  DiffuseName();
  EndModule();
}
```

B.2 L'application car_wash

Nous simulons une station de lavage comprenant plusieurs postes de lavage de voitures. Les voitures se déplacent sur une route formée de tronçons. Un contrôleur distribue les voitures sur l'un des postes de lavage. Une fois lavées, les voitures vont sur un tronçon final.

B.2.1 Le module *Module_Auto*

Ce module comprend le comportement *voiture*.

```
/*************************************************************************/
/*                          Module_Auto.c                                */
/*************************************************************************/
#include <Behavior.h>
#include <Prim_Cac.h>
#include <auto.h>
#include <troncon.h>

void voiture(Component me, auto_param *arg)
{
 Mess_Ptr rep;
 Component troncon_suivant;
 Component boite_locale;
 troncon_message tr_mess;
  boite_locale = NewBox(me);
  do {
    tr_mess.code = 0;                    /* demande le nom du troncon suivant */
    tr_mess.boite_envoyeur = boite_locale;
    tr_mess.num_voit = arg->numero_voiture;
    tr_mess.nom_troncon = arg->troncon;
    Send(me,arg->troncon,sizeof(troncon_message),&tr_mess);
    GetMessage(me,boite_locale,&rep);
    if((troncon_suivant = *ArgMess(rep,1)) != VOID) {/* test fin du parcours */
      FreeMessage(rep);          /* demande l'entree dans le troncon suivant */
      tr_mess.code = 1;
      Send(me,troncon_suivant,sizeof(troncon_message),&tr_mess);
      GetMessage(me,boite_locale,&rep);
      arg->troncon = troncon_suivant;
    }
    FreeMessage(rep);
  } while (troncon_suivant != VOID);
  FreeBox(boite_locale);
  LockOutput(me); PrintString(me,"Sortie  de la voiture ");
  PrintInt(me,arg->numero_voiture); PrintChar(me,'\n'); Flush(me);
  UnLockOutput(me);
  Send(me,arg->root_cac,0,NULL);
  EndComponent(me);
}

int main(int argc, char ** argv)
{
  InitModule(argc,argv,1);
  InsertBehavior(voiture,B_AUTO);
  DiffuseName();
  EndModule();
}
```

B.2.2 Le module *Module_Car_Wash*

Ce module comprend le comportement *car_wash* qui représente le lavoir automatique et le comportement *controleur* qui est chargé de répartir les voitures sur les différents postes de lavage.

```
/*************************************************************************/
/*                    Module_Car_Wash.c                                  */
/*************************************************************************/
#include <Behavior.h>
#include <Prim_Cac.h>
#include <carwash.h>
#include <troncon.h>
#include <station.h>

void car_wash(Component me, carwash_param *arg)
{
 Mess_Ptr mes;
 Component controleur;
 control_param ctr_par;
 troncon_message *tr_mess;
 control_message ctr_mess;
  ctr_par.nb_laveurs = arg->nb_stations;
  ctr_par.tmps = arg->tmps;
  controleur=NewComponent(me,B_CONTROLEUR,VOID,sizeof(control_param),&ctr_par);
  while (1) {
    GetMessage(me,me,&mes);
    tr_mess = (troncon_message *)GetFirstArg(mes,sizeof(troncon_message));
    switch (tr_mess->code) {
      case 0 : /*demande le nom du troncon suivant */
        Send(me,tr_mess->boite_envoyeur,sizeof(Component),&(arg->troncon_suiv));
        break;
      case 1 : /* demande l'entree dans le troncon */
               LockOutput(me);
               PrintString(me,"Entree de la voiture ");
               PrintInt(me,tr_mess->num_voit);
               PrintString(me," dans le controleur ");
               PrintInt(me,arg->numero_troncon);
               PrintChar(me,'\n');
               Flush(me);
               UnLockOutput(me);
               ctr_mess.num_voit = tr_mess->num_voit;
               ctr_mess.boit_env = tr_mess->boite_envoyeur;
               Send(me,controleur,sizeof(control_message),&ctr_mess);
               if (tr_mess->nom_troncon != VOID) {
                 /*libere le troncon precedent */
                 int code = 2;
                   Send(me,tr_mess->nom_troncon,sizeof(int),&code);
               }
               break;

      case 2 :
               break;
    }
    FreeMessage(mes);
  }
  EndComponent(me);
}
```

```
void controleur(Component me, control_param *arg)
{
 Mess_Ptr mes1,mes2;
 Component acces_station;
 int i;
 station_param sta_par;
 Component station;
  acces_station = NewBox(me);
  sta_par.controleur = acces_station;
  sta_par.tmps = arg->tmps;
  for (i = 1; i <= arg->nb_laveurs; i++) {
    sta_par.numero_station = i;
    NewComponent(me,B_STATION,VOID,sizeof(station_param),&sta_par);
  }
  while (1) {
    GetMessage(me,me,&mes1);
    GetMessage(me,acces_station,&mes2);
    station = *(Component *)GetFirstArg(mes2,sizeof(Component));
    SendaMessage(me,station,mes1);
    FreeMessage(mes2);
  }
  EndComponent(me);
}

int main(int argc, char ** argv)
{
  InitModule(argc,argv,2);
  InsertBehavior(car_wash,B_CAR_WASH);
  InsertBehavior(controleur,B_CONTROLEUR);
  DiffuseName();
  EndModule();
}
```

B.2.3 Le module *Module_Station*

Ce module comprend le comportement *station* qui représente un poste de lavage.

```
/*************************************************************************/
/*                    Module_Station.c                                   */
/*************************************************************************/
#include <Behavior.h>
#include <Prim_Cac.h>
#include <station.h>
#include <carwash.h>
#include <troncon.h>

void station(Component me, station_param *arg)
{
 Mess_Ptr mes;
 control_message *ctr_mess;
 int i;
  while (1) {
    /* demande un travail au controleur */
    Send(me,arg->controleur,sizeof(Component),&me);
    GetMessage(me,me,&mes);
    ctr_mess = (control_message *)GetFirstArg(mes,sizeof(control_message));
    LockOutput(me);
    PrintString(me,"Lavage de la voiture ");
    PrintInt(me,ctr_mess->num_voit);
    PrintString(me," dans la station ");
    PrintInt(me,arg->numero_station);
    PrintChar(me,'\n');
    Flush(me);
    UnLockOutput(me);
    for (i = 0 ; i < arg->tmps; i++); /* simule un travail */
    /* libere la voiture */
    Send(me,ctr_mess->boit_env,0,NULL);
    FreeMessage(mes);
  }
  EndComponent(me);
}

int main(int argc, char ** argv)
{
  InitModule(argc,argv,1);
  InsertBehavior(station,B_STATION);
  DiffuseName();
  EndModule();
}
```

B.2.4 Le module *Module_Troncon*

Ce module comprend le comportement *troncon* qui représente un « morceau » de route ne pouvant contenir qu'une voiture.

```
/*****************************************************************************/
/*                      Module_Troncon.c                                     */
/*****************************************************************************/
#include <Behavior.h>
#include <Prim_Cac.h>
#include <troncon.h>

void troncon(Component me, troncon_param *arg)
{
 Mess_Ptr mes,req;
 Component buffer;
 int libre;
 troncon_message *tr_mess;
  buffer = NewBox(me);
  libre = VOID;
  while (1) {
    GetMessage(me,me,&mes);
    tr_mess = (troncon_message *)GetFirstArg(mes,sizeof(troncon_message));
    switch (tr_mess->code) {

      case 0 : /*demande le nom du troncon suivant */
        Send(me,tr_mess->boite_envoyeur,sizeof(Component),&(arg->troncon_suiv));
        break;

      case 1 : /* demande l'entree dans le troncon */
         if ((libre == VOID) || (arg->troncon_suiv == VOID) ){
           /* le tampon est libre */
           LockOutput(me);
           PrintString(me,"Entree de la voiture ");
           PrintInt(me,tr_mess->num_voit);
           PrintString(me," dans le troncon ");
           PrintInt(me,arg->numero_troncon);
           PrintChar(me,'\n');
           Flush(me);
           UnLockOutput(me);
           libre = tr_mess->num_voit;
           if (tr_mess->nom_troncon != VOID) {
             /*libere le troncon precedent */
             int code = 2;
              Send(me,tr_mess->nom_troncon,sizeof(int),&code);
           }
           /* previent la voiture de l'entree dans le troncon */
           Send(me,tr_mess->boite_envoyeur,0,NULL);
         } else {
           /* empile la voiture dans le buffer */
           LockOutput(me);
           PrintString(me,"Empile la demande de la voiture ");
           PrintInt(me,tr_mess->num_voit);
           PrintString(me," dans le troncon ");
           PrintInt(me,arg->numero_troncon);
           PrintChar(me,'\n');
           Flush(me);
           UnLockOutput(me);
```

```
          Send(me,buffer,sizeof(troncon_message),tr_mess);
        }
        break;

      case 2 :
        /* teste l'existance d'autre voiture en attente */
        if (NumberMessageInBox(me,buffer) > 0) {
          GetMessage(me,buffer,&req);
          tr_mess = (troncon_message*)GetFirstArg(req,sizeof(troncon_message));
          LockOutput(me);
          PrintString(me,"Depile la demande de la voiture ");
          PrintInt(me,tr_mess->num_voit);
          PrintString(me," dans le troncon ");
          PrintInt(me,arg->numero_troncon);
          PrintChar(me,'\n');
          Flush(me);
          UnLockOutput(me);
          libre = tr_mess->num_voit;
          if (tr_mess->nom_troncon  != VOID) {
            /*libere le troncon precedent */
            int code = 2;
             Send(me,tr_mess->nom_troncon,sizeof(int),&code);
          }
          /* previent la voiture de l'entree dans le troncon */
          Send(me,tr_mess->boite_envoyeur,0,NULL);
          FreeMessage(req);
        } else
          /* pas de voiture en attente */
          libre = VOID;
        break;
    }
    FreeMessage(mes);
  }
  EndComponent(me);
}

int main(int argc, char ** argv)
{
  InitModule(argc,argv,1);
  InsertBehavior(troncon,B_TRONCON);
  DiffuseName();
  EndModule();
}
```

B.2.5 Le module *Module_Dia*

Ce module comprend le comportement *lance* qui démarre l'application, crée les différents CAC de l'application.

```
/*************************************************************************/
/*                      Module_Dia.c                                     */
/*************************************************************************/
#include <Behavior.h>
#include <Prim_Cac.h>
#include <troncon.h>
#include <carwash.h>
#include <auto.h>

void lance(Component me, int *arg)
{
 Mess_Ptr mes;
 int temps_travail, nb_voiture, nb_troncon, nb_station;
 int top_depart, i, tps;
 troncon_param tr_par;
 carwash_param cw_par;
 auto_param aut_par;
 Component dernier, suivant;
  PrintString(me,"\n*****APPLI*****    \n\n");
  PrintString(me,"entrer le temps par voiture : ");Flush(me);
  ReadInt(me,&temps_travail);
  PrintString(me,"entrer le nombre de voiture : ");Flush(me);
  ReadInt(me,&nb_voiture);
  PrintString(me,"entrer le nombre de troncon : ");Flush(me);
  ReadInt(me,&nb_troncon);
  PrintString(me,"entrer le nombre de station : ");Flush(me);
  ReadInt(me,&nb_station);
  top_depart = _cputime();

  tr_par.numero_troncon = nb_troncon;
  tr_par.troncon_suiv = VOID;
  dernier = NewComponent(me,B_TRONCON,VOID,sizeof(troncon_param),&tr_par);

  cw_par.numero_troncon = nb_troncon-1;
  cw_par.troncon_suiv = dernier;
  cw_par.nb_stations = nb_station;
  cw_par.tmps = temps_travail;
  suivant = NewComponent(me,B_CAR_WASH,VOID,sizeof(carwash_param),&cw_par);

  for (i=2; i <= nb_troncon; i++) {
    tr_par.numero_troncon = nb_troncon - i;
    tr_par.troncon_suiv = suivant;
    suivant = NewComponent(me,B_TRONCON,VOID,sizeof(troncon_param),&tr_par);
  }

  aut_par.troncon = suivant;
  aut_par.root_cac = me;
  for (i=1; i <= nb_voiture; i++) {
    aut_par.numero_voiture = i;
    NewComponent(me,B_AUTO,VOID,sizeof(auto_param),&aut_par);
  }

  for (i=1; i <= nb_voiture; i++) {
```

```
    GetMessage(me,me,&mes);
    FreeMessage(mes);
  }
  tps = _cputime() - top_depart ;
  PrintString(me,"temps execution ");
  PrintInt(me,tps);
  PrintString(me," 1/100 s \n");
  Flush(me);
  StopModules(me);
  EndComponent(me);
}

int main(int argc, char ** argv)
{
 word create;
  InitModule(argc,argv,2);
  InsertBehavior(lance,B_DIA);
  InsertBehavior(cac_flot,B_FLOT);
  DiffuseName();
  create = NewBox(0);
  NewComponent(create,B_DIA,VOID,0,NULL);
  EndModule();
}
```

Annexe C

Exemple de programme BOX: le magasin

C.1 La classe Magasin

```
-- Magasin
[OBJ] Magasin ::
  -- Un magasin ou on peut retirer des articles
  [RAYON] rayon ; -- Rayon ou le serveur se sert
  [STOCK] stock ; -- Stock ou on peut se reapprovisionner

  [SPROC ] retirer_article : [INT] qtte :
  -- Retirer l'article du type 'type' en quantite 'qtte'
  -- Resultat stocke dans un tableau d'articles
  -- Il n'y a qu'un serveur
  {
    -- Aller dans le rayon
    -- Creer le tableau de resultat
    [FILE <- out !!] out.s ("demande de retrait dans le magasin\n");
    rayon.retirer (qtte) ;
    out.s ("Fin de retrait dans le magasin\n");
  }

  [PROC] init ::
  -- Creation de touts
  {
    stock := [STOCK <- init !!] ;
    rayon := [RAYON <- init ! stock !] ;
  }

||||||||||

[OBJ] ARTICLE :: -- Un article
```

C.2 La classe Rayon

```
[OBJ] RAYON ::
  -- Le rayon ou il y a des articles dans une certaine quantite
  const [INT] max = 10 ;    -- Nbre max d'article en rayon
  const [INT] limite = 2 ; -- Qtte minimale
  [INT] qtte_courante ; -- Quantite courante en rayon
  [STOCK] stock ; -- Adresse du stock d'articles ou il faut recommander
  [RAYONNEUR] rayonneur ; -- Personne responsable de la gestion du rayon
  [FILE] out ;

  [PROC] retirer : [INT] qtte :
  -- Retirer l'article en quantite 'qtte'
  {
    -- Essayer de retirer
    -- Recommander au STOCK si pas assez
    out.s ("demande de retrait dans le rayon nb articles : ");
    out.i (qtte) ;
    out.nl ;
    status ;
    [INT <- qtte] i ;
    loop
      if i = 0 then exit end_if ;
      rayonneur <- un_retrait !! ;
      [ARTICLE] aa := buf_a.get ;
      i := i - 1;
    end_loop ;
    status ;
  }

  [PROC] deposer : [ARTICLE] art :
  -- Ajouter un article dans le rayon
  {
    buf_a.put (art) ;
  }

  [PROC] init : [STOCK] stk :
  -- Initialisation
  {
    out := [FILE <- out !!] ;
    buf_a := [BUFFER [ARTICLE] <- init ! !] ;
    rayonneur := [RAYONNEUR <- work ! self ! ] ;
    stock := stk ;
  }

  -- Implementation
  [BUFFER [ARTICLE]] buf_a ;

  [PROC] status ::
  -- Status du rayon
  {
    out.m ([MESS <- ! "\t\t\t\t\t Rayon ", buf_a.size, "\n" ! ] ) ;
    out.m ([MESS <- ! "\t\t\t\t\t\t\t Stock ", stock.buf_a.size, "\n" ! ] )
  }
```

C.3 La classe Stock

```
[OBJ] STOCK ::
  -- Stock d'articles
  const [INT] max = 30 ;     -- Nbre max d'article en stock
  const [INT] limite = 10 ; -- Qtte minimale
  [INT] qtte_courante ;      -- Quantite courante en stock
  [STOCKEUR] stockeur ; -- Personne responsable de la gestion du stock

  [PROC [ARTICLE]] extraire ::
  -- Extraire un article
  {
    stockeur <- un_retrait !! ;
    result := buf_a.get ;
  }

  [PROC] ajouter : [ARTICLE] art :
  -- Ajouter un article
  {
    buf_a.put (art) ;
  }

  [PROC] init ::
  -- Initialisation
  {
    buf_a := [BUFFER [ARTICLE] <- init ! !] ;
    stockeur := [STOCKEUR <- work ! self ! ] ;
  }

  -- Implementation

  [BUFFER [ARTICLE]] buf_a ;
```

C.4 La classe Rayonneur

```
-- Rayonneur ou chef de rayon
-- Responsable du reaprovisionnemnt des rayons, stocks
[FRAG] RAYONNEUR ::
  -- Chef de rayon
  [INT] etat ;
  [RAYON] rayon ;

  [FPROC] un_retrait ::
  -- Un retrait a eu lieu
  {
    etat := etat - 1 ;
    if etat <= rayon.limite then
      etat := etat + 1 ;
      loop
        if etat = rayon.max then exit end_if ;
        [MANUTENTIONNAIRE <- work ! rayon.stock, rayon !] clarcker ;
        etat := etat + 1 ;
      end_loop ;
    end_if ;
  }
```

```
  [BPROC] work : [RAYON] ray :
  -- Travail
  {
    rayon := ray ;
    loop
      BOX -> [MESS] m ! [FPROC] f ! ;
      f!m ;
    end_loop ;
  }
```

C.5 La classe Manutentionnaire

```
[FRAG] MANUTENTIONNAIRE ::
  -- Manutentionnaire qui deplace un ARTICLE du STOCK vers le RAYON
  [STOCK] stock ;
  [RAYON] rayon ;

  [BPROC] work : [STOCK] stk ; [RAYON] ray :
  -- Comportement d'un travailleur
  {
    -- Initialisation
    stock := stk ;
    rayon := ray ;
    [ARTICLE] art ;
    art := stock.extraire ;
    -- J'ai un article
    rayon.deposer (art) ;
    [FILE <- out !!] out.s("\t\t\tdeplacement\n") ;
    -- Le boulot est fini, auxdieux !
  }
```

C.6 La classe Stockeur

```
[FRAG] STOCKEUR ::
  -- Chef de stock
  [INT] etat ;
  [STOCK] stock ;

  [FPROC] un_retrait ::
  -- Un retrait a eu lieu
  {
    etat := etat - 1 ;
    if etat <= stock.limite then
      etat := etat + 1 ;
      loop
        if etat = stock.max then exit end_if ;
        [PRODUCTEUR <- produire ! stock.buf_a !] mousse ;
        etat := etat + 1 ;
      end_loop ;
    end_if ;
  }

  [BPROC] work : [STOCK] sto :
  -- Travail
  {
```

```
    stock := sto ;
    loop
      BOX -> [MESS] m ! [FPROC] f ! ;
      f!m ;
    end_loop ;
  }
```

C.7 La classe Producteur

```
[FRAG] PRODUCTEUR ::
  -- Creation d'un nouvel article et depot dans le stock

  [BPROC] produire : [BUFFER [ARTICLE]] buf_a :
  -- Produire un article et le deposer dans le stock
  {
    -- Producteur: il est cree pour le boulot et il meurt
    -- apres.
    [ARTICLE <-] art ; -- Creation d'un article
    buf_a.put (art) ;
    [FILE <- out !!] out.s("production\n") ;
  }
```

Bibliographie

[AG89] Ziya Aral and Ilya Gertner. High-level debugging in parasight. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):151-162, January 1989.

[Agh86] G. Agha. *Actors. A Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1986.

[Bat89] Peter Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):11-22, January 1989.

[BB92] Thomas Bemmerl and Peter Braun. Visualization of Message Passing Parallel Programs. In Cosnard, editor, *Proceedings of CONPAR*, pages 79-90, 1992.

[Ber92] Anton Beranek. Execution replay and data race detection for the debugging of MOSKITO applications. In W. Joosen and E. Milgrom, editors, *Proceedings of the European Workshops on Parallel Computing "From Theory to Sound Practice"*, pages 80-91. IOS Press, March 1992.

[BW83] Peter C. Bates and Jack C. Wileden. High-level debugging of distributed systems: The behavioral abstraction approach. *Journal of Systems and Software*, 3(4):255-264, 1983.

[CC91] S. Chaumette and M.C. Counilh. A development environment for distributed systems. In Arndt Bode, editor, *Proceedings of Distributed Memory Computing 2^{nd} European Conference, EDMCC2*, pages 110-119. Springer-Verlag, April 1991.

[CDG91] Luc Courtrai, Eric Delattre, and Jean-Marc Geib. A Server Based Architecture to Support Object-Oriented Languages on Multicomputers (V 2.0). Technical Report ERA-95, Université des Sciences et Technologies de Lille, Laboratoire d'Informatique Fondamentale de Lille, April 1991.

[Cha92] Serge Chaumette. *Outils de développement et de mise au point pour la classe des machines parallèles à mémoire distribuée.* PhD thesis, Université Bordeaux I, Laboratoire Bordelais de Recherche en Informatique, Avril 1992.

[CHP71] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent Control with « Readers » and « Wwriters ». *Communications of the ACM*, 14(10), 1971.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed Snaphots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[Cou92] Luc Courtrai. *Les Composants Actifs de Communication: Outils pour la conception et l'implantation de langages parallèles à objets actifs pour machines MIMD.* PhD thesis, Université des Sciences et Technologies de Lille, Laboratoire d'Informatique Fondamentale de Lille, Octobre 1992.

[CRD+92] Luc Courtrai, Jean-Francois Roos, C. Dumoulin, P. Merle, Jean-Marc Geib, and Jean-Francois Méhaut. Communicating Active Components: Support for Parallel Object Languages on Distributed Architectures. In *Proceedings of EUSUG'92 European Sun User Group Conference*, Wiesbaden, November 1992.

[CRGM92a] Luc Courtrai, Jean-Francois Roos, Jean-Marc Geib, and Jean-Francois Méhaut. Communicating Active Components: an Environment for Concurrent Applications on Parallel Machines. *Proceedings of EUROMICRO Hardware and Software Design Automation, published in Microprocessing and Microprogramming*, 35(1-5):47–54, September 1992.

[CRGM92b] Luc Courtrai, Jean-Francois Roos, Jean-Marc Geib, and Jean-Francois Méhaut. An Environment to Support Fragmented Active Objects. In Luis-Felipe Cabrera and Eric Jul, editors, *Proceedings of Second I-WOOOS International Workshop on Object Orientation in Operating Systems*, pages 124–128. IEEE Computer Society Press, September 1992.

[CW82] R.S. Curtis and Larry D. Wittie. BugNet: A debugging system for parallel programming environments. In *Proceedings of the 3^h International Conference on Distributed Computing Systems*, October 1982.

[DGMS93] Jack Dongarra, G. A. Geist, Robert Manchek, and V. S. Sunderam. Integrated PVM Framework Supports Heterogeneous Network Computing. *Computers in Physics*, 7(2):166–175, 1993.

[D.o83] D.o.D. (U.S. Department of Defense), The Pentagon, Washington. *Ada Reference Manual for the Ada programming language*, 1983.

[DRM93] Cedric Dumoulin, Jean-Francois Roos, and Jean-Francois Méhaut. Le ramasse-miettes du projet pvc-box. In Laboratoire d'Informatique de Brest (équipe Armen), editor, *5emes Rencontres du Parallélisme*, pages 105–108. PRC C^3, PRC ANM, France TELECOM, LIFL, USTL, Mai 1993.

[Fid89] C. J. Fidge. Partial orders for parallel debugging. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):183–194, January 1989.

[GHP+93] O. Gerstel, M. Hurfin, N. Plouzeau, M. Raynal, and S. Zaks. On the fly replay: a practical paradigm and its implementation for distributed debugging. Technical Report 731, Institut de Recherche en Informatique et Systèmes Aléatoires, Campus Universitaire de Beaulieu, Rennes, May 1993.

[Gor91] Michael M. Gorlick. The Flight Recorder: An Architectural Aid for System Monitoring. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 26(12):175–183, December 1991.

[Gra94] Christophe Gransart. *Fragmentation d'objets pour Machines Fortement Distribuées.* PhD thesis, Université des Sciences et Technologies de Lille, Laboratoire d'Informatique Fondamentale de Lille, à paraître 1994.

[HC87] Alfred A. Hough and Janice E. Cuny. Belvedere: Prototype of a pattern-oriented debugger for highly parallel computation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 735–738, University Park PA, 1987.

[HE91] M. Heath and J. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, September 1991.

[HG93] Fred Hémery and Jean Marc Geib. Simulation pour l'Aide au Placement d'Entités Actives Communicantes. In Laboratoire d'Informatique de Brest (équipe Armen), editor, *5emes Rencontres du Parallélisme*, pages 195–199. PRC C^3, PRC ANM, France TELECOM, LIFL, USTL, Mai 1993.

[Hoa78] C.A.R. Hoare. Communicating Sequential Process. *Communications of the ACM*, 21(8):666–677, August 1978.

[HW88] Dieter Haban and Wolfgang Weigel. Global events and global breakpoints in distributed systems. In *Proceedings of the 21th International Conference on System Sciences*, pages 166–175, January 1988.

[Jam93] H. Jamrozik. Aide à la mise au point des applications parallèles et réparties à base d'objets persistants. In *Actes des Journées des Jeunes Chercheurs en Systèmes Informatiques Répartis*, pages 125–129. DRED MRE-CNRS, IMAG, April 1993.

[Kil91] Carol Kilpatrick. ChaosMON–Application-Specific Monitoring and Display of Performance Information for Parallel and Distributed Systems. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 26(12):57–67, December 1991.

[KS92] Doug Kimelman and Gerald Sang'udi. Program Visualization by Integration of Advanced Compiler Technology with Configurable Views. In *Proceedings of the Environments and Tools For Parallel Scientific Computing Workshop*, pages 73–84. CNRS - NSF, LIP-IMAG ENS, September 1992.

[KTP92] Joao Paulo Kitajima, Cécile Tron, and Brigitte Plateau. ALPES: A Tool for the Performance Evaluation of Parallel Programs. In *Proceedings of the Environments and Tools For Parallel Scientific Computing Workshop*, pages 213–228. CNRS - NSF, LIP-IMAG ENS, September 1992.

[KW91] D. Kafura and D. Washabough. Progress in the Garbage Collection of Aactive Objects. *ACM OOPS Messager*, 2(2):55–58, April 1991.

[LA93] Luk J. Levrouw and Koenraad M. R. Audenaert. An Efficient Record-Replay Mechanism for Shared Memory Programs. In *Proceedings of the Euromicro Workshop on Parallel and Distributed Processing*, pages 169–176. IEEE, January 1993.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LMC87] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[LS91] Eric Leu and André Schiper. Techniques de déverminage pour programmes parallèles. *T.S.I. Technique et Science Informatique*, 10(1):5–21, Janvier 1991.

[LSZ91] Eric Leu, André Schiper, and Abdelwahab Zramdini. Efficient execution replay technique for distributed memory architectures. In Arndt Bode, editor, *Proceedings of Distributed Memory Computing 2^{nd} European Conference, EDMCC2*, pages 315–324. Springer-Verlag, April 1991.

[MC88] Barton P. Miller and Jong-Deok Choi. Breakpoints and halting in distributed programs. In *Proceedings of the 8^{th} International Conference on Distributed Computing Systems*, pages 316–323, 1988.

[MCN88] Barton P. Miller, Jong-Deok Choi, and Robert Netzer. Techniques for debugging parallel programs with flowback analysis. Technical Report 786, University of Wisconsin, Madison, Computer Sciences Department, August 1988.

[MH89] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[MvdW89] G.J.W. Mattern and A.J. van der Wal. Partial ordering of synchronization events for distributed debugging in tightly-coupled multiprocessors systems. In M. Cosnard et al., editors, *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V. (North Holland), 1989.

[Par90] Parsytec GmbH, Juelicher straBe 338, D-5100 Aachen. *MultiCluster-2. Technical documentation. Installation, expansion and maintenance manual*, Rev 1.1 edition, May 1990.

[PL89] Douglas Z. Pan and Mark A. Linton. Supporting Reverse Execution of Parallel Programs. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):124–129, January 1989.

[PSL91] Perihelion Software LTD, editor. *The Helios Parallel Operating System.* Prentice Hall, 1991.

[PTV92] S. Poinson, B. Tourancheau, and X. Vigouroux. Distributed monitoring for scalable massively parallel machines. In *Proceedings of the Environments and Tools For Parallel Scientific Computing Workshop*, pages 85–101. CNRS - NSF, LIP-IMAG ENS, September 1992.

[RCGM92] Jean-Francois Roos, Luc Courtrai, Jean-Marc Geib, and Jean-Francois Méhaut. Les Composants Actifs de Communication: Manuel de Programmation (V 2.0). Technical Report ERA-115, Université des Sciences et Technologies de Lille, Laboratoire d'Informatique Fondamentale de Lille, Septembre 1992.

[RCM93] Jean-Francois Roos, Luc Courtrai, and Jean-Francois Méhaut. Execution Replay of Parallel Programs. In *Proceedings of the Euromicro Workshop on Parallel and Distributed Processing*, pages 429–434. IEEE, January 1993.

[RRZ89] Robert V. Rubin, Larry Rudolph, and Dror Zernik. Debugging parallel programs in parallel. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):216–225, January 1989.

[SBN89] David Socha, Mary L. Bailey, and David Notkin. Voyeur: Graphical Views of Parallel Programs. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):206–215, January 1989.

[Sto89] Janice M. Stone. A graphical representation of concurrent processes. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):226–235, January 1989.

[Sun86] Sun Microsystems. *News Preliminary Technical Overview*, 1986.

[Sun88] Sun Microsystems. *Systems Services Overview Chapter 6: Lightweight Processes part number 800-1753*, 1988.

[Sun90] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice & Experience*, 2(4):315–339, December 1990.

[vDvdW91] G.J.W. van Dijk and A.J. van der Wal. Partial ordering of synchronization events for distributed debugging in tightly-coupled multiprocessors systems. In Arndt Bode, editor, *Proceedings of Distributed Memory Computing 2nd European Conference, EDMCC2*, pages 100–109. Springer-Verlag, April 1991.

[Wit89] Larry D. Wittie. Debugging Distributed C Programs by Real Time Replay. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):57–67, January 1989.

[ZT92a] Eugenio Zabala and Richard Taylor. Maritxu: Visualising the run-time behaviour of transputer networks. In W. Joosen and E. Milgrom, editors, *Proceedings of the European Workshops on Parallel Computing "From Theory to Sound Practice"*, pages 100–103. IOS Press, March 1992.

[ZT92b] Eugenio Zabala and Richard Taylor. Process and processor interaction: Architecture independent visualisation schema. In *Proceedings of the Environments and Tools For Parallel Scientific Computing Workshop*, pages 55–71. CNRS - NSF, LIP-IMAG ENS, September 1992.