Mise au point de programmes repartis. Application au systeme Chorus

Frederic Ruget

▶ **To cite this version:**

Frederic Ruget. Mise au point de programmes repartis. Application au systeme Chorus. Réseaux et télécommunications [cs.NI]. Université Joseph-Fourier - Grenoble I, 1994. Français. NNT : . tel-00005110

HAL Id: tel-00005110
https://theses.hal.science/tel-00005110

Submitted on 25 Feb 2004

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

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

THÈSE

présentée par

Frédéric RUGET

pour obtenir le titre de

Docteur de l'Université
Joseph Fourier – Grenoble-1

(arrêté ministériel du 5 juillet 1984 et du 30 mars 1992)

Spécialité INFORMATIQUE

Mise au Point de Programmes Répartis

Application au Système CHORUS

Thèse soutenue le 22 Novembre 1994 devant la commission d'examen

MM. :	Jean-Pierre VERJUS	Président
	Jean-Pierre BANÂTRE	Rapporteurs
	André SCHIPER	
	Friedemann MATTERN	Examinateurs
	Roger NEEDHAM	
	Marc ROZIER	
	Sacha KRAKOWIAK	

Thèse préparée au sein du Laboratoire Bull-IMAG et de la société Chorus systèmes

Remerciements

Je remercie Jean-Pierre VERJUS, Professeur à l'Institut National Polytechnique de Grenoble et Directeur de l'IMAG, qui me fait l'honneur de présider ce jury.
Je remercie Sacha KRAKOWIAK, Professeur à l'Université de Grenoble-1, qui a dirigé cette thèse. Il m'a accordé une grande autonomie, tout en m'apportant un soutien engagé et des conseils toujours avisés.
Je remercie Marc ROZIER, Docteur de l'Institut National Polytechnique de Grenoble et directeur du département micro-noyau chez Chorus systèmes, qui m'a orienté, conseillé et amicalement encouragé tout au long de mon travail de recherche. Grâce à lui, cette thèse a pu être menée à bien.
Je remercie Jean-Pierre BANÂTRE, Professeur à l'Université de Rennes-1 et Directeur de l'IRISA, et André SCHIPER, Professeur à l'École Polytechnique Fédérale de Lausanne d'avoir bien voulu accepter la charge de rapporteurs.
Je remercie Friedemann MATTERN, Professeur à l'Université de la Sarre d'avoir accepté de juger mon travail.
Je remercie Roger NEEDHAM, Professeur et Directeur du Laboratoire d'Informatique à l'Université de Cambridge d'avoir accepté de juger mon travail, et de m'avoir chaleureusement accueilli pendant une année au «Computer Lab». Ces remerciements s'adressent également au Docteur Ian LESLIE qui s'est chargé de mon encadrement pendant cette année.
Je remercie enfin les «chorusiens», chacun en particulier, pour l'intérêt qu'ils ont porté à mes recherches, pour leur gentillesse et pour l'aide qu'ils ne m'ont jamais refusée malgré des emplois du temps chargés, et la société Chorus systèmes en général, qui m'a offert des conditions de travail idéales pour mener à bien cette thèse.

Table des matières

Table des figures

Liste des tableaux

Chapitre I

Introduction

La phase de validation est une étape importante du développement d'un logiciel. Elle représente généralement une part significative de son coût, tant en terme de temps passé par le programmeur qu'en terme de matériel mis en œuvre. Cela est d'autant plus vrai aujourd'hui, avec la complexité croissante des systèmes informatiques. Les systèmes récents sont constitués de nombreux sites répartis sur un réseau de communication, chaque site pouvant lui-même comprendre plusieurs processeurs fonctionnant en parallèle. Les difficultés liées au parallélisme et à la répartition de l'exécution des applications écrites pour de tels systèmes posent un défi que doivent relever les outils de mise au point de demain.

Le problème de la mise au point des applications parallèles et réparties a offert depuis déjà quelques années un cadre pour des recherches assez variées. Du côté le plus théorique, on trouve l'analyse statique qui consiste à déterminer certaines propriétés d'un programme parallèle ou réparti à partir de son code, ou d'une abstraction de celui-ci – par exemple des propriétés concernant des problèmes de synchronisation. Cette approche est en principe la plus rigoureuse et la plus satisfaisante pour l'esprit, car elle permet de prendre en compte tous les comportements possibles du programme. Malheureusement elle a seulement connu un succès limité, car d'une part elle impose une procédure et un formalisme qui peuvent dérouter le programmeur non spécialiste, et d'autre part elle conduit facilement à une «explosion combinatoire» lors de l'exploration de l'espace des états potentiels du programme.

Des approches plus pragmatiques ont dernièrement reçu la faveur des chercheurs. On peut distinguer trois axes généraux. Le premier, peu «formel» mais néanmoins fondamental, est la recherche de représentations adéquates de l'exécution d'un programme parallèle ou réparti, permettant au programmeur d'en avoir une compréhension plus intuitive : représentation graphiques, sonores (!), interactives, plus ou moins détaillées... Le deuxième axe est la détection au vol de propriétés d'une exécution particulière du programme (et non de toutes les exécutions possibles, contrairement à l'approche statique). Les recherches orientées en ce sens ont permis de préciser la notion d'observation et de propriété d'une exécution répartie ; en particulier elles ont mis en évidence que certaines formulations «intuitives» de propriétés n'avaient pas de sens dans le cadre ré-

parti. Malheureusement elles n'ont pas encore tenu toutes leurs promesses, en particulier parce que la détection des propriétés intéressantes (i.e. suffisamment expressives) coûte cher. Le dernier axe de recherche consiste à développer des outils pour l'analyse post-mortem d'une exécution répartie : photographie et développement de l'état global du programme, ré-exécution, «voyage» temporel... Cet axe pourrait déboucher rapidement (nous l'espérons) sur des applications pratiques, car il réalise une véritable symbiose avec les outils de mise au point classiques et permet d'étendre à un moindre coût leur champ d'utilisation aux programmes répartis.
Notre recherche s'inscrit dans ce dernier axe. Plus précisément, notre «défi» a été de réaliser un outil permettant l'enregistrement puis la reproduction de l'exécution d'un programme réparti. Notre thèse est qu'avec les techniques et les matériels actuels, il est faisable et relativement peu coûteux de réaliser un tel outil, et ceci de façon transparente pour les programmes ré-exécutés (c'est à dire que la reproduction de l'exécution d'un programme quelconque peut être mise en œuvre sans instrumentation préalable du programme.)
Les aspects appliqués de notre travail ont eu comme cadre le système d'exploitation à micro-noyau CHORUS de la société Chorus systèmes. Très concrètement, notre objectif était de réaliser un outil de mise au point permettant d'enregistrer et de reproduire l'exécution d'applications distribuées composées d'acteurs multi-activités CHORUS répartis sur un réseau de machines mono-processeur de type PC. Notre travail s'est réparti en deux tâches plus ou moins imbriquées. La première tâche a consisté à étendre le micro-noyau CHORUS pour offrir le support nécessaire à la ré-exécution. Elle a fourni une contribution pour le projet européen ESPRIT N^0 6603 «OUVERTURE». La deuxième tâche a consisté à concevoir et à implanter l'outil de ré-exécution au dessus de la version étendue du micro-noyau.
Le plan de la thèse est le suivant:
Cette introduction constitue le premier chapitre.
Le deuxième chapitre dresse un état de l'art de la mise au point dans les systèmes parallèles et répartis, et développe les points discutés aux premiers paragraphes de l'introduction.
Le troisième chapitre est consacré aux extensions du micro-noyau CHORUS qui fournissent un support aux outils de ré-exécution. Il en présente les concepts, l'interface et les performances. Il faut noter que les dites extensions ne sont pas réservées au seul usage des outils de ré-exécution, et qu'elles ont déjà été utilisées avec succès pour d'autres applications (outils de monitoring du système CHORUS).
Le quatrième chapitre présente CDB, l'outil de ré-exécution proprement dit : les abstractions qu'il fournit à l'utilisateur, son interface, ses fonctions et limitations, son architecture et les algorithmes qu'il utilise, ses performances. Il propose également une comparaison succincte avec d'autres outils de ré-exécution décrits dans la littérature.
Le cinquième chapitre décrit des protocoles d'horloges logiques optimisées développés pour l'usage de CDB. Il s'agit de deux algorithmes efficaces d'horloges matricielles. Cette partie de la thèse, relativement théorique et tardive, n'a pas encore été complètement intégrée à CDB. Néanmoins, si les développements sur

CDB se poursuivent, et en particulier s'il s'avère nécessaire d'en améliorer les performances, elle pourra constituer un bon point de départ.
Nous concluons au sixième chapitre.

Chapitre II

État de l'art de la mise au point des applications parallèles et réparties

Les systèmes informatiques récents sont constitués de nombreux sites répartis (ou distribués) sur un réseau de communication, chaque site pouvant lui-même comprendre plusieurs processeurs fonctionnant en parallèle. Les problèmes liés au parallélisme et à la répartition de l'exécution des applications écrites pour de tels systèmes posent un défi que doivent relever les outils de mise au point de demain. Ce chapitre, présente les techniques les plus récentes de validation et de mise au point des logiciels parallèles et répartis. Nous commençons par une introduction très rapide aux méthodes de validation des programmes par preuves formelles, pour nous concentrer ensuite sur les techniques de mise au point fondées sur l'analyse d'une exécution particulière d'un programme.
En section II.2, nous décrivons les différents moyens mis à la disposition du programmeur pour observer (sonder) le comportement d'un programme et pour enregistrer les informations recueillies dans des journaux d'événements. Nous considérons le problème de la convivialité des représentations adoptées pour visualiser ces informations.
En section II.3, nous présentons les techniques de mise au point fondées sur la détection de propriétés de l'exécution d'un programme. Nous introduisons le modèle des exécutions réparties qui est la base théorique de cette approche et nous décrivons les deux grandes catégories de propriétés qui présentent un aspect pratique pour la mise au point : celles fondées sur l'analyse des flots de contrôle de l'exécution, et celles fondées sur l'analyse des états globaux de l'exécution.
En section II.4, nous présentons les techniques fondées sur l'analyse post-mortem d'une exécution, notamment les techniques de ré-exécution, qui font plus particulièrement l'objet de cette thèse.
Nous concluons en section II.5.

II.1 Différentes approches

Signalons tout d'abord l'existence de travaux de synthèse traitant de la mise au point des programmes parallèles et répartis: l'article de Cheung [CBM90] pour une description des aspects «humains» de l'activité de mise au point ; celui de McDowell et Helmbold [MH89] pour un panorama des techniques de validation formelle ; les synthèses de [LS91, Mou90, Val90] ; et plus récemment la bibliographie de Gu et al. [GVS94] pour une classification des techniques d'observation des exécutions, ainsi que la synthèse de Raynal [Ray94] à propos des techniques de ré-exécution et d'analyse de propriétés. Il nous faut enfin mentionner la thèse de Jamrozik [Jam93], à laquelle nous avons fait de nombreux emprunts pour les sections II.2 et II.4.
Citons Raynal [Ray94] :

> «les outils de validation de logiciels peuvent être classés en deux catégories. Dans la première, on trouve les techniques [...] qui permettent de certifier que tous les comportements possibles d'un programme vérifient une propriété donnée. Cette vérification se fait par une preuve assertionnelle du programme ou par l'exploration exhaustive du graphe des états qui lui est associé. Dans la seconde, on trouve des techniques de test et de mise au point qui permettent de constater si une exécution particulière du programme vérifie ou non une propriété.»

Nous commençons par une brève présentation de la première catégorie.

II.1.1 Analyse statique

L'analyse statique est une technique qui permet de déterminer les caractéristiques (e.g. structurelles) d'un programme, sans que le code ne soit exécuté. On pourra se reporter à [MH89] pour une revue de ce domaine.
Dans le cadre de la programmation distribuée, l'analyse statique permet essentiellement de détecter deux classes d'erreurs :

- les erreurs de synchronisation,
- les erreurs d'accès aux données.

Les erreurs de synchronisation sont (par exemple) celles qui conduisent à un interblocage ou à une famine. Les erreurs d'accès aux données sont d'une part l'accès à des variables non initialisées (un problème connu en programmation séquentielle) et d'autre part l'accès concurrent non contrôlé à un même objet – les erreurs d'accès aux données sont donc un cas particulier des erreurs de synchronisation.
L'analyse statique se fonde généralement sur la modélisation du programme sous forme de graphes : graphes d'états, graphes d'événements, réseaux de Pétri, etc. Pour donner l'intuition de ce que représentent ces graphes et de l'usage qui peut en être fait, nous présentons sommairement la technique proposée par Appelbe [AM88] : le programme – qui spécifie l'exécution parallèle de processus

séquentiels – est d'abord converti en un graphe du flot de contrôle, dans lequel on ne conserve que les événements de synchronisation. Le graphe obtenu est appelé graphe de concurrence CHG (*Concurrency History Graph*). Chaque nœud du graphe correspond à un état de concurrence du programme, et spécifie l'état de synchronisation de chaque processus. La construction du graphe de concurrence est illustrée par les figures II.1 et II.2 (reprises de [LS91]).

```
PROCESSUS T;

    VAR x;

    PROCESSUS T1;
    DEBUT (*T1*)
       x:= ...;
       ...;
       SI condition ALORS
          Reception(message);
       FIN (*SI*)
       StopProcessus;
    FIN (*T1*)

    DEBUT (*T*)
       Creer(T1);
       x:= ...;
       Envoi(T1,message);
       StopProcessus;
    FIN (*T*)
```

FIG. II.1 - *Fragment de programme parallèle*

L'analyse du graphe de concurrence permet notamment de détecter des erreurs de synchronisation. Par exemple, un blocage correspond à un nœud du graphe ayant des tâches actives mais pas de successeur (nœud (6) sur la figure II.2). Notons que si l'on complète les nœuds du graphe d'états avec les informations concernant les variables partagées consultées et mises à jour entre deux opérations de synchronisation, on peut alors également se servir du graphe pour détecter les erreurs d'accès au variables.

L'analyse statique est malheureusement confrontée à un problème d'explosion combinatoire: le nombre des états dans lesquels peut passer le programme est souvent exponentiel et la construction du graphe des états n'est alors pas réalisable.

Pour résoudre ce problème, des techniques d'exécution symbolique [YT88] et d'interprétation abstraite des programmes ont été développées pour diminuer le nombre d'états à prendre en compte. L'interprétation abstraite [Cou90] par exemple, consiste à simplifier l'état du programme en ne considérant pas les valeurs exactes des variables, mais seulement leur abstraction, c'est à dire en quelque sorte leur «type[1]». L'interprétation abstraite a été utilisée avec succès

1. En réalité, la notion d'abstraction est plus fine que la notion de type (dans le sens qui

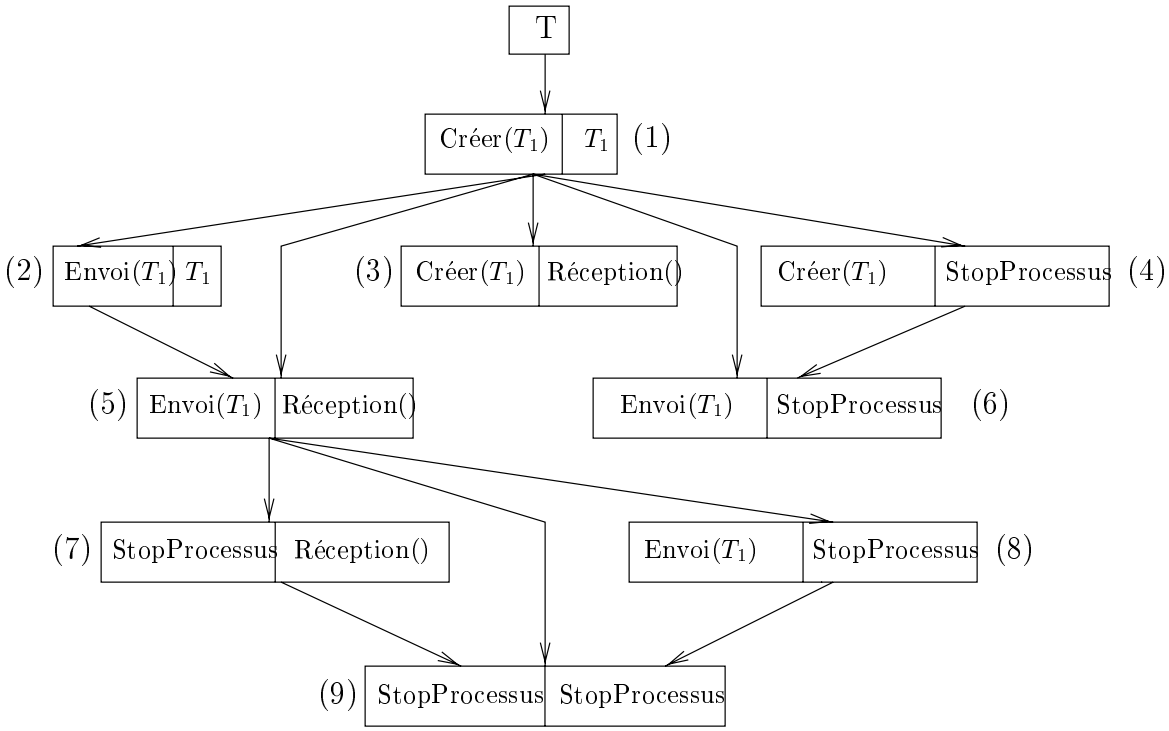


FIG. II.2 - *Graphe de concurrence*

dans le contexte de langages de type CSP [Hoa85] pour réduire la complexité des preuves de programmes [CC80].
Malgré ces tentatives pour faciliter l'usage de l'analyse statique, cette technique ne peut en général pas être utilisée seule, et le programmeur doit avoir recours à la technique moins «pure» d'analyse d'une exécution particulière du programme. Notons cependant que l'analyse statique peut intervenir en complément aux techniques d'analyse d'une exécution particulière, pour permettre par exemple de réduire le volume des données qui doivent être recueillies durant l'exécution. Cette approche prometteuse est suivie dans [DS90, CS91, SG94].

II.1.2 Analyse d'une exécution particulière

Les techniques d'analyse d'une exécution particulière d'un programme parallèle ou réparti sont les héritières des techniques «traditionnelles» utilisées pour les programmes séquentiels et mises en œuvres par des outils comme DBX[AM86] ou GDB[Sta86].
À l'origine, ces techniques traditionnelles sont fondées sur la possibilité pour le programmeur (1) d'interrompre l'exécution du programme pour en examiner l'état, (2) d'exécuter le programme pas à pas, (3) de recommencer autant de fois que nécessaire l'exécution, jusqu'à ce que l'erreur soit localisée. Cette méthode de mise au point est qualifiée d'interactive et de cyclique.
Les premières tentatives pour appliquer les techniques traditionnelles de mise au point aux programmes répartis ont consisté à associer un débogueur (metteur au point) séquentiel à chaque processus (séquentiel) du programme. À condition de disposer d'un environnement multi-fenêtre (tel que Suntools [Sun86] ou X-Window [SG86]), on pouvait ainsi associer une fenêtre différente à chaque processus de façon à pouvoir bien distinguer les informations provenant des différents débogueurs.
Il est apparu qu'il était de plus nécessaire de disposer d'un mécanisme pour coordonner les actions des différents débogueurs séquentiels. Par exemple, si l'on désire interrompre l'exécution du programme, il faut stopper l'ensemble des processus. Il est souhaitable que l'interruption des processus soit réalisée automatiquement, afin de réduire le plus possible le laps de temps entre l'interruption du premier et du dernier processus, de sorte que l'exécution du programme soit perturbée le moins possible. Par exemple, en réduisant ce laps de temps, on diminue la probabilité qu'un processus non encore stoppé tente de communiquer avec un processus déjà stoppé, ce qui perturberait l'exécution.
Le débogueur «réparti» pdbx [Pdb86] met en œuvre ce principe en permettant à l'utilisateur de commander l'ensemble des metteurs au point séquentiels à partir d'une seule fenêtre. Les commandes pdbx s'appliquent en effet (1) soit à un processus distingué appelé processus courant, (2) soit à un processus explicitement désigné dans la commande, (3) soit à tous les processus susceptibles d'être intéressés par la commande. Par exemple, la commande *Continuer* s'applique à tous les processus suspendus. Griffin [Gri87] généralise ce mécanisme en

est généralement donné à ce mot en programmation). Cependant, Monsuez a montré [Mon93] qu'il est possible de retrouver les types «standards» à l'aide de l'interprétation abstraite.

permettant à l'utilisateur de contrôler des ensembles de processus modifiables dynamiquement.
Ces premières tentatives apportent une solution simple mais partielle au problème de la mise au point d'applications parallèles et réparties. Elle permettent de contrôler dans une certaine mesure le cours d'une exécution et d'examiner l'état du programme, mais introduisent une perturbation importante de l'exécution (due à la façon dont les points d'arrêt sont réalisés). Par ailleurs, elles ne donnent pas la garantie que la méthode de mise au point cyclique pourra être mise en œuvre. En effet, elles ne prennent pas en compte le fait que des exécutions successives d'un même programme parallèle ou réparti ne sont en général pas identiques.
Nous dressons ici une brève liste des difficultés liées au caractère parallèle ou réparti de l'exécution, qui rendent malaisée la généralisation des techniques traditionnelles de mise au point.

Difficultés liées au parallélisme

- Le comportement d'un programme parallèle est souvent non-déterministe. Il dépend en effet des conditions particulières d'exécution (ordonnancement des processus, charge des média de communication, etc.), qui peuvent modifier d'une exécution à un autre l'ordre des opérations de communication et de synchronisation. Considérons par exemple deux processus non synchronisés concurrents tels que l'un modifie la valeur d'une variable partagée et l'autre la lit : l'exécution du système va dépendre de la façon (aléatoire) dont les lectures sont ordonnées par rapport aux écritures.

  Le non-déterminisme des programmes parallèles rend plus difficile la localisation des erreurs. En effet, la reproduction d'un comportement erroné n'est pas assurée : le comportement erroné se produit de façon aléatoire, en fonction des conditions d'exécution particulières. Par conséquent, il n'est pas facile d'appliquer la méthode de mise au point cyclique.

- Le simple fait d'observer le programme constitue une perturbation qui risque de modifier son exécution.

  C'est l'expression informatique du principe d'incertitude d'Heisenberg : « Le déroulement d'un phénomène physique qui est observé est affecté par cette observation. » Dans le domaine de la mise au point, ce phénomène est connu sous le nom d'effet de sonde (*probe effect* [Gai85]). En effet, l'observation du programme risque de modifier les vitesses relatives d'exécution des processus, et par suite l'ordre des événements de l'exécution.

Difficultés liées à la répartition

- Dans le cas d'une exécution répartie sur plusieurs sites, il n'est pas possible d'observer de l'état global de l'exécution de façon instantanée (i.e. l'état de tous les processus et du système de communications.) En effet, chaque

site a son propre référentiel de temps, indépendant des autres. Le système n'a pas d'horloge globale et les processus peuvent se synchroniser que par l'intermédiaire du système de communication – rappelons ici que le cadre général de notre travail est celui des systèmes distribués asynchrones. Pour des systèmes distribués synchrones, il serait en effet être possible de construire un référentiel de temps «réel» commun à tous les processus et suffisamment précis, comme cela est montré dans [Cri89].

– Le contrôle de l'utilisateur sur le programme réparti ne peut pas être immédiat, en raison des délais de communication.

Dans la suite de ce chapitre, nous présentons les techniques «modernes» qui permettent de mettre au point des applications parallèles et réparties, en résolvant notamment les problèmes soulevés plus haut.

II.2 Outils d'observation

La possibilité d'observer l'exécution d'un programme est une condition sine qua non à sa mise au point. Dans cette section (qui reprend en partie la présentation faite par Jamrozik dans sa thèse [Jam93]), nous exposons les différentes techniques permettant l'observation d'une exécution : sondes, journaux et points d'arrêt. Nous abordons ensuite le problème de la présentation conviviale des observations.

II.2.1 Sondes

Un outil de mise au point qui vise à analyser une exécution particulière d'un programme doit tout d'abord être capable d'observer cette exécution. Pour ce faire, l'outil de mise au point doit placer des «sondes» qui vont lui permettre de recueillir des informations concernant l'exécution du programme. Ces sondes sont introduites à l'aide de techniques d'instrumentation variées (nous utilisons le terme instrumentation pour signifier l'ajout de capteurs, indifféremment à un programme, à un code exécutable ou à un matériel.) Par exemple :

- Par instrumentation du code source du programme .

  Le principe est d'ajouter du code d'observation à des endroits appropriés dans le code source du programme. Ce code d'observation a pour but de recueillir des informations concernant l'exécution du programme et éventuellement de communiquer ces informations à l'outil de mise au point.

  L'ajout de ce code peut être effectué de manière manuelle ou automatique. L'ajout manuel [MCWB91] est une méthode très fastidieuse, demandant beaucoup de temps et d'effort au programmeur et qui peut en outre être la cause d'erreurs supplémentaires. L'ajout automatique de code [LR89, LCSM90, TA91] s'effectue par l'intermédiaire d'un langage de spécification d'événements qui est compilé avec le programme cible. La compilation de ces spécifications ajoute automatiquement au source du programme le code nécessaire pour observer les événements définis.

- Par instrumentation du code objet.

  Ici, c'est le compilateur lui-même qui instrumente le code objet du programme [MRA+89, MCWB91]. Cette technique est similaire à la technique classique d'instrumentation du code objet d'un programme utilisée pour obtenir un profil de l'exécution du programme (*code profiling*).

- Par instrumentation des primitives d'une librairie.

  L'exécution du programme peut être observée en remplaçant les librairies standard par des librairies modifiées de façon à jouer un rôle de sonde. Cette approche est utilisée dans [JLSU87, MCWB91]. Elle permet d'ajouter et d'ôter facilement les sondes. Il suffit pour cela de lier le programme avec la version appropriée des librairies. Une recompilation du programme n'est pas nécessaire. En revanche, la fonction exercée par les sondes est

fixée une fois pour toute. Cette méthode est donc moins flexible que les deux précédentes.

– Par utilisation de programmes espions, s'exécutant parallèlement au programme cible et collectant périodiquement des informations de trace [AG89]. L'inconvénient de cette technique est son caractère asynchrone : il peut se passer un certain temps entre l'occurrence d'un événement et sa détection par le programme espion. Les événements «instables» peuvent même passer inaperçus.

– Par instrumentation du système.

  Le système lui-même (par conception, ou après une instrumentation convenable) peut fournir un certain support pour l'observation de l'exécution d'un programme.

  Par exemple, dans le cas décrit par [SBN89], le système rapporte directement au processus observé les informations concernant son exécution. Cette façon de procéder ne nous semble pas idéale car elle n'est pas transparente pour le programme observé. L'approche décrite par [Tok89] nous semble préférable : le système rapporte les observations à un processus indépendant chargé de l'enregistrement.

  Cette approche, également suivie par [Hab87, EA88, ZPS88] présente l'avantage de ne nécessiter aucune instrumentation du programme observé. En revanche, seuls les événements relatifs au système sont observables, ce qui se traduit par un niveau d'abstraction assez grossier, pas nécessairement adéquat pour décrire le comportement du programme cible.

– Par instrumentation matérielle.

  L'observation de l'exécution peut être confiée à un matériel spécialisé qui va exercer automatiquement le rôle de sonde. Cette approche est intéressante parce qu'elle permet d'éliminer totalement ou partiellement l'effet perturbateur des sondes logicielles. En revanche, elle est coûteuse, peu portable et d'une mise en œuvre assez lourde. Lorsqu'elle est mise en œuvre, c'est le plus souvent sur un parc limité de machines spécialisées, ce qui constitue une contrainte supplémentaire. Enfin, les informations qu'elle permet d'obtenir sont souvent de trop bas niveau, et elles sont difficiles à exploiter.

  La littérature décrit cependant des réalisations à base de matériel spécialisé. Par exemple l'utilisation d'une machine «espion» qui observe directement le bus d'une machine cible [Lin86, Hab87, RR89, BHL89] ou qui récupère des observations concernant la machine cible en communicant avec celle-ci au moyen de canaux de communications spécialisés (de façon à limiter le plus possible l'effet de sonde) [RR89].

  Une telle approche peut être suivie pour obtenir un profil (*code profiling*) très précis et non intrusif de l'exécution d'un programme : la machine espion enregistre les événements d'apparition des points d'entrée et de sortie des procédures du programme sur le bus d'adresses de la machine cible,

et leur associe la date donnée par une horloge physique. Cette technique est présentée dans [McR93].

Bacon [BG91] propose une architecture matérielle et logicielle qui permet d'observer l'ordre des références mémoires des processus d'un programme parallèle en enregistrant un sous-ensemble du trafic entre la mémoire et les antémémoires (*caches*) des processeurs. L'utilisation de matériel spécialisé permet de réaliser cette observation de façon très peu intrusive, même lorsque les processus font un usage très fin (*fine grain*) et très fréquent de la mémoire partagée. Une telle observation ne pourrait pas être réalisée de façon purement logicielle, sinon à un coût prohibitif (en verrouillant et déverrouillant à chaque fois les zones de mémoire accédées par les processus.) Notons cependant que le débogueur « Instant Replay» – qui se passe de tout support matériel – utilise une telle technique logicielle: tout objet accédé par le programme cible est au préalable verrouillé par une procédure qui effectue entre autres une mise à jour de numéros de version. Pour cette raison, Instant Replay est plutôt adapté à la mise au point de programmes à base d'objets de taille moyenne ou grande (*coarse grain*).

En général, lorsque la fonction réalisée par un matériel spécialisé présente un intérêt suffisant, les constructeurs de machines finissent par l'intégrer en standard à leurs processeurs. C'est par exemple le cas de la fonction d'exécution pas à pas du processeur, où de point d'arrêt en cas d'accès à une adresse particulière de la mémoire (ces fonctions sont présentes sur le processeur i386 d'Intel [Int92], qui est très largement diffusé).

On peut donner un exemple de fonction réalisée par le matériel qui, bien qu'encore peu proposée par les constructeurs, va certainement connaître une large diffusion. Il s'agit du compteur d'instruction. Le compteur d'instruction est un module qui mémorise le nombre d'instructions (machine) exécutées par le processeur. Il permet d'observer (et de reproduire) le moment précis de l'apparition d'événements asynchrones (e.g. interruption matérielle ou changement de contexte). Des réalisations expérimentales de compteur d'instructions matériels ont été proposées [For89], le constructeur Hewlett Packard en propose un pour l'architecture de précision RISC HP [HP 89], et le futur processeur «P6» d'Intel devrait également en comporter un.

À notre avis, les techniques d'instrumentation au niveau système ou matériel doivent être préférées car elles permettent une meilleure transparence (et donc une moindre intrusion) vis à vis du programme observé. Cependant ces techniques sont plus difficiles, voire impossibles à mettre en œuvre (e.g. dans le cas où on n'a pas accès au code source du système.) Elles sont également moins portables – particulièrement les instrumentations matérielles.

II.2.2 Traces

Les observations produites par les sondes doivent être collectées et analysées par le système de mise au point, ou bien enregistrées dans des journaux pour une

analyse ultérieure. Dans les systèmes bien structurés, la collecte de ces «traces» d'exécution est souvent effectuée par un module dédié appelé moniteur local ou résident. Il peut y avoir un moniteur local par site d'exécution du programme, voire un moniteur local par processus. Dans un deuxième temps, les observations collectées par les moniteurs locaux peuvent être rassemblées (après un premier filtrage) par un second module appelé moniteur central ou relationnel [GVS94]. Le moniteur central peut être situé sur une machine différente et communiquer avec les moniteurs locaux à travers un réseau.
L'utilisation de deux niveaux de collecte présente deux avantages :

- Les particularités liées au système ou au site d'exécution sont prises en compte uniquement par les moniteurs locaux.
- Les moniteurs locaux peuvent filtrer les informations collectées ou en faire une première analyse (partielle). Cela simplifie la tâche du moniteur central et permet d'éviter un goulet d'étranglement dans le cas où le programme cible comprend un grand nombre de processus ou de sites.

Lorsque l'observation est couplée à un système de réponse en «temps-réel», il peut même être utile de structurer le moniteur en de plus nombreuses couches, par exemple : sondes, moniteurs locaux, moniteur central. Les données collectées peuvent être partiellement analysées à chaque niveau, et la réponse peut être générée le plus tôt possible [OSS93, GMGK84].
Si les informations de trace produites par l'observation de l'exécution du programme ne sont pas analysées au vol, elles doivent être enregistrées dans un journal, aussi appelé historique de l'exécution. Le problème de la journalisation mérite une thèse à lui seul [Ruf92] ; nous allons en présenter certains aspects pertinents pour la mise au point.
L'historique doit contenir suffisamment d'information pour permettre la localisation d'erreurs potentielles par le programmeur. C'est pourquoi il a tendance à être volumineux. Cependant, un historique trop volumineux présente de nombreux désavantages:

- La collecte et la journalisation des informations de trace peut rarement être réalisée de manière totalement non intrusive. Par conséquent, plus on journalise d'information, plus on risque de perturber l'exécution du programme.
- Au bout du compte, l'historique est destiné à être analysé. Cette analyse sera d'autant plus longue que l'historique sera volumineux.
- En pratique, le facteur qui limite le plus la création d'historiques trop volumineux est le simple fait que l'outil de mise au point ne dispose pas de moyens de stockage suffisants.

Plusieurs travaux ont eu pour but la recherche de moyens pour réduire le volume des informations conservées dans l'historique :

- Garcia-Molina [GMGK84] propose de ne conserver que les informations importantes, sous la forme d'une trace ordonnée d'événements de haut niveau. Ces événements correspondent à des actions macroscopiques faisant

intervenir le système (e.g. la création et la destruction d'un processus, l'envoi et la réception de messages, l'allocation et la libération d'une ressource, etc.). En revanche, les actions internes au programme ne sont pas journalisées (e.g. le changement de valeur d'une variable).

- Une autre technique consiste à enregistrer des numéros de version des objets observés, plutôt que leur état complet. Cette technique est appliquée par le débogueur distribué *Agora* [For89], ainsi que par le débogueur *Instant Replay* [LMC87].

- Des techniques diverses ont été développées [MC88c, MC88a, CS91, MC93, SG94] afin d'utiliser des informations de compilation pour réduire le volume d'information journalisée. Parmi celles-ci, nous présentons à titre d'exemple la technique de trace incrémentale (*incremental tracing*) introduite par Miller et Choi [MC88c, MC88a].

  La technique de trace incrémentale a tout d'abord été développée pour des programmes séquentiels, puis étendue aux programmes parallèles. Elle consiste à générer dans un premier temps une trace succincte, suffisante pour caractériser l'exécution tracée, mais pas pour être exploitée directement par le programmeur. Dans un deuxième temps, cette trace peut être affinée sur requête de l'utilisateur, en utilisant une technique semblable à la ré-exécution : à partir de la trace succincte et d'informations spécifiques générées à la compilation, on peut re-calculer l'état précis du programme à tout instant. Ainsi, le coût de création de traces exhaustives est reporté sur les phases non critiques de compilation et d'interaction «programmeur-outil de mise au point». Cette approche peut être qualifiée de paresseuse (i.e. les informations ne sont générées qu'à la demande).

  La technique de trace incrémentale suppose qu'un certain travail soit réalisé au moment de la compilation : une analyse syntaxique du programme permet de le découper en blocs équivalents, selon des critères de taille et de temps d'exécution probable. Au début et à la fin de chaque bloc, le compilateur insère du code pour produire à l'exécution des enregistrements prologue (*prelog*) et épilogue (*postlog*) contenant les valeurs des variables respectivement susceptibles d'être lues ou modifiées dans le bloc. La détermination des variables en question est effectuée au moyen d'une analyse statique du programme (cf. section II.1.1). Le compilateur associe également à chaque bloc une portion de code capable de simuler l'exécution du bloc et de produire une trace fine d'exécution à partir d'un simple enregistrement prologue.

  Pour utiliser cette technique avec des programmes parallèles, il faut en plus connaître la valeur des variables partagées au moment de leur utilisation dans le bloc. Ces variables peuvent en effet être modifiées entre le moment de l'entrée dans le bloc et le moment de leur utilisation. La valeur des variables partagées est donc sauvegardée avec la trace d'exécution, au moment où elles sont accédées.

- Dans certains cas, il existe des algorithmes spécifiques permettant d'op-

timiser la journalisation d'un type d'événement particulier. Par exemple Netzer et Miller [NM92, Net93] proposent un algorithme «optimal» (suivant une métrique qu'ils définissent) pour journaliser les événements de réception de messages dans un système réparti : ils déterminent localement si deux messages reçus consécutivement sont dans une situation de course (*race*), et ne journalisent que les messages qui sont dans une telle situation.

- Une technique particulièrement intéressante consiste à utiliser des heuristiques pour prédire le contenu du journal. Si l'observation à journaliser correspond à l'information prédite par l'heuristique, on ne journalise rien («0 logging»). Cette technique a été appliquée pour implémenter efficacement la journalisation pour des protocoles de tolérance aux fautes [SBY88].

- Enfin, il est toujours possible de réduire la taille du journal en utilisant un des nombreux algorithmes de compression de données disponibles dans la littérature.

II.2.3 Points d'arrêt

Les techniques présentées plus haut permettent de tracer l'exécution d'un programme parallèle ou réparti. Les informations de trace représentent le flot de données ou de contrôle du programme durant une exécution particulière. Une approche de l'observation légèrement différente consiste à s'intéresser non plus à des flots d'exécution, mais à l'état du programme à un instant donné (son image mémoire, l'état de ses différents processus...). Cette approche est l'héritière directe de la technique «traditionnelle» dite du *point d'arrêt*. Dans cette section, nous montrons comment la notion traditionnelle de point d'arrêt peut être étendue aux exécutions parallèles et réparties.
Comme nous venons de l'indiquer, l'arrêt de l'exécution permet au programmeur d'examiner et de modifier l'état du programme (la valeur des données utilisées, l'état des processus), et de contrôler la suite de l'exécution (exécution pas à pas, ajout de points d'arrêt, reprise de l'exécution, lancement d'une nouvelle exécution).
Dans le cas parallèle ou réparti, on peut envisager plusieurs politiques d'arrêt de l'exécution [MH89] :

- Arrêter tous les processus participant à l'exécution du programme.

- Arrêter un seul processus, ou un groupe de processus (les autres continuant leur exécution).

L'arrêt de tous les processus peut être difficile à réaliser dans un intervalle de temps réduit, et l'arrêt d'un seul processus ou d'un groupe de processus peut provoquer de graves perturbations pour les systèmes avec des contraintes de temps réel.

Suivant l'architecture du système sur lequel s'exécute le programme, l'arrêt de l'exécution peut être plus ou moins facile à réaliser.

- Dans les architectures à mémoire partagée, les processus sont fortement couplés et il est facile de leur faire partager un référentiel temporel commun. Dans ces conditions, il est aisé de suspendre l'exécution de tous les processus de façon atomique.

- Dans les architectures à mémoire répartie, les processus coopèrent principalement en échangeant des messages. L'absence d'horloge physique commune rend impossible l'arrêt simultané de tous les processus. Une première façon de résoudre ce problème consiste à simuler un temps physique global : certains algorithmes permettent d'obtenir une bonne approximation [Jéz89, Cri89]. Une autre façon consiste à utiliser un temps logique à la place du temps physique [CL85, NT86]. Cette dernière technique permet d'arrêter les processus dans un état qui n'est pas réellement un instantané de l'exécution, mais qui satisfait des contraintes de cohérence. Nous reviendrons sur la définition de ces états cohérents répartis en section II.3.1.

L'arrêt des processus d'un programme peut être cause de perturbations importantes, en particulier si le programme fait explicitement référence à une horloge physique. Pour limiter l'importance de ce problème, Cooper [Coo87] propose de substituer à l'horloge physique des δ-horloges (qu'il appelle horloges logiques, terme que nous réservons aux horloges linéaires, vectorielles et matricielles présentées au chapitre V.) Durant l'exécution normale du programme, les δ-horloges comptent le temps à la même vitesse que l'horloge physique. En revanche, leur évolution est bloquée pendant toute la durée du point d'arrêt. Cette technique permet donc de rendre le point d'arrêt transparent au programme cible.

II.2.4 Présentation des informations

Une bonne présentation des informations de trace est primordiale pour simplifier le travail du programmeur [CBM90]. Par exemple, il est toujours souhaitable de présenter les informations de trace en utilisant le langage même dans lequel le programme a été écrit [2] :

- Dans le cas de programmes classiques (e.g. écrits en C), un bon outil de mise au point devrait permettre de visualiser les entités «module», «procédure», «ligne de programme», «variable», etc. Des outils standard comme DBX [AM86] ou GDB [Sta86] s'acquittent bien de cette tâche pour des programmes séquentiels (ils utilisent pour cela des tables générées au moment de la compilation : table des symboles, table de correspondance instruction-ligne, informations concernant la structure de la pile, etc.)

- Dans le cas de programmes rédigés dans un langage à base d'objets, un bon outil de mise au point devrait permettre de visualiser les objets invoqués lors de l'exécution, leur état et leurs interactions [Jam93].

2. Ce n'est pas toujours facile, e.g. si le programme a été optimisé, cf. [BHS92, HCU92].

– Dans le cas de programmes à base de processus communiquant par messages, un bon outil de mise au point devrait permettre de visualiser les entités «processus», «porte de communication» et «message», y compris dans le cas où ces entités ont finalement disparu du code exécutable (e.g. suite à une optimisation tirant parti d'une zone de mémoire partagée).

Parmi les autres techniques qui permettent de faciliter la compréhension des historiques d'exécution, on peut citer :

– Le filtrage

  Pour faciliter la manipulation des historiques volumineux, il est possible d'appliquer une technique de filtrage (par type d'événement et par processus). L'historique peut aussi être conservé dans une base de données interrogeable en langage relationnel [Sno87, Sno88, GMGK84] ou logique [LP85].

– La représentation sous forme de graphe

  Il est aussi possible de représenter l'historique sous forme de graphes. Graphe du flot de données du programme par exemple, dont les nœuds représentent des blocs du programme et les arcs indiquent le flot des données (c'est la «flowback analysis» de Miller et Choi [MC88a]). Où bien graphes de causalité, dont les nœuds correspondent aux opérations sur les objets partagés, et les arcs aux relations d'antériorité entre ces opérations [FL89].

La représentation graphique de l'historique d'une exécution est particulièrement intéressante dans le cas de petits programmes, ou dans un but pédagogique. Elle permet de saisir le fonctionnement du programme et détecter certaines erreurs (de synchronisation par exemple). Malheureusement, dans le cas de programmes réels, les historiques générés sont volumineux, et leur représentation graphique souvent difficile à manipuler.

II.3 Détection de propriétés

La détection de propriétés est une technique qui permet de vérifier si l'exécution d'un programme est conforme à un comportement attendu. Par exemple, on peut vouloir vérifier que la valeur de telle variable reste comprise entre des bornes données, ou que tel événement se produit toujours avant tel autre (e.g. l'initialisation d'une variable avant sa première lecture).
La détection de telles propriétés est relativement aisée pour des programmes séquentiels ou parallèles. En effet, les exécutions séquentielles et parallèles présentent :

- une «unité de lieu» : il existe une mémoire centrale accédée par le ou les différents processus.
- une «unité de temps» : il existe une horloge globale qui peut servir de référence.

Il est donc possible d'accéder directement à l'état de global de l'exécution, et de connaître sans ambiguïté l'ordre dans lequel les événements se produisent.
En revanche, les exécutions réparties sont caractérisées par l'absence de référentiel temporel ou spatial unique qui pourrait servir de lieu d'observation privilégié. La répartition de l'exécution sur plusieurs sites et l'incertitude sur les délais de communication peuvent donner lieu à plusieurs observations différentes de la même exécution du programme. Avant de se poser le problème de la détection de propriétés, il faut donc se poser en premier lieu la question de la définition formelle de la notion d'observation d'une exécution. Ensuite seulement, on pourra proposer des règles pratiques de vérification de propriétés.

II.3.1 Modélisation des exécutions réparties

Nous présentons ici le modèle d'exécution répartie asynchrone avec communication par message. Ce modèle simple est le plus utilisé dans la littérature concernant la mise au point distribuée. Il nous permettra de faire une description plus formelle des techniques de détection de propriétés.

Modèle réparti asynchrone avec communication par message

Nous nous inspirons de la présentation faite dans [Ray94]. Un programme réparti est composé de n processus séquentiels $P_1, ..., P_n$, qui ne communiquent et ne se synchronisent que par échanges de messages[3].
Un processus peut exécuter trois types d'actions :

- une action interne.

3. Il existe d'autres modèles qui ne supposent pas un nombre borné de processus. Ces modèles sont fondés sur l'existence d'une primitive de duplication (*fork*) qui permet à un processus d'engendrer un processus fils s'exécutant en parallèle [NR88]. L'hypothèse simplificatrice d'un nombre borné de processus permet de faire un exposé plus concis. Par ailleurs cette hypothèse ne constitue pas forcément une restriction très gênante dans la pratique. En effet, si on peut imaginer que le nombre de processus ne soit pas borné, le nombre de processeurs du système – qui réalisent la concurrence réelles des exécutions – est quant à lui le plus souvent fixé.

- une action d'émission qui consiste à envoyer un message vers un autre processus. L'action d'émission est non bloquante, ce qui veut dire que le processus peut continuer à exécuter d'autres actions sans attendre que le processus destinataire ait effectivement reçu le message.
- une action de réception. Lors d'une action de réception, le processus se bloque jusqu'à l'arrivée d'un message en provenance de n'importe lequel des autres processus. Le contenu du premier message arrivé est délivré au processus qui peut alors reprendre son exécution.

On peut faire différentes hypothèses concernant le système de communication qui achemine les messages d'un processus à un autre. Par exemple :

- Deux processus quelconques peuvent toujours communiquer (directement par un canal de communication, où indirectement après routage à travers un certains nombre d'autres processus). Cette propriété est le non partitionnement du système de communication.
- Le système de communication ne perd pas de messages, ne duplique pas de messages, ne crée pas de messages intempestifs. Ces hypothèses caractérisent la fiabilité du système de communication.
- Le système de communication est tel que les messages transitant entre deux processus donnés sont reçus dans l'ordre de leur émission (communication «FIFO»).
- Le système de communication est tel que si deux messages sont reçus par le même processus et qu'il y a une relation de causalité entre l'émission du premier et du second message, alors nécessairement, la réception du second message à lieu après la réception du premier (communication «causale»).

Par la suite, et sauf précision explicite, nous ferons seulement l'hypothèse de la fiabilité et du non partitionnement du système de communication.
L'exécution d'un processus P_i produit une séquence d'événements dits primitifs. Chaque événement primitif est :

- soit un événement interne (exécution d'une action interne)
- soit un événement de communication (exécution d'un envoi ou d'une réception de message).

Cette séquence est appelée l'histoire H_i de P_i et est notée :

$$H_i = e_i^0 e_i^1 e_i^2 \dots e_i^x e_i^{x+1} \dots$$

où e_i^x est le x-ième événement primitif exécuté par P_i; e_i^0 est un événement fictif qui initialise l'état de P_i. Les événements sont atomiques.
On peut définir une relation d'ordre partiel, appelée *précédence causale* [Lam78] sur l'ensemble H des événements de l'exécution du programme réparti. Pour cela :

- les événement locaux à chaque processus sont ordonnés par numéro de séquence.

– les événements de réception de messages sont ordonnés après les événements d'émission des messages correspondants.

L'ordre de précédence causale est la fermeture transitive des ordres locaux aux processus et de l'ordre sur les événements de communication.
Une exécution répartie est exactement représentée par l'ensemble H de ses événements primitifs, partiellement ordonné par la relation de précédence causale.
La figure II.3 (reprise de [Ray94]) représente une exécution répartie sous forme d'un diagramme «espace-temps» : les axes horizontaux représentent l'évolution des processus ; les points noirs et blancs représentent les événements ; les flèches reliant deux événements produits par des processus différents représentent les messages.

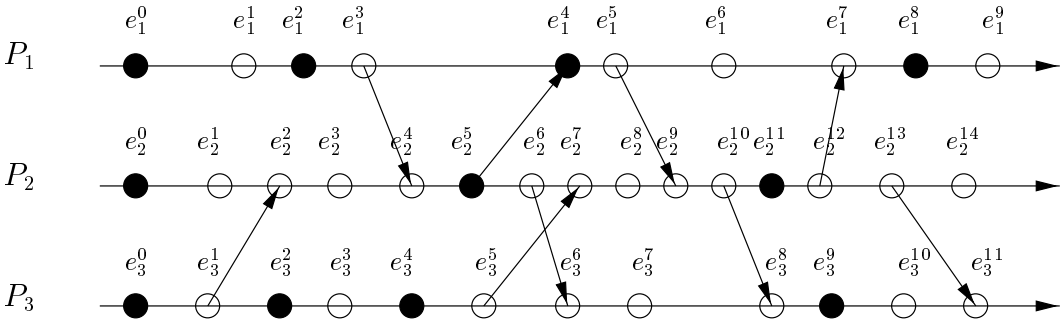


FIG. II.3 - *Une exécution vue au niveau d'abstraction des événements primitifs*

Selon le problème à résoudre, seulement un sous-ensemble des événements primitifs peuvent être significatifs pour l'utilisateur. Par exemple, si on considère la détection d'un prédicat global portant sur les variables de plusieurs processus, seules les modifications de ces variables constituent des événements significatifs (ce sont les seuls à pouvoir modifier la valeur de vérité du prédicat). On appelle R l'ensemble des événements significatifs.
L'ensemble R des événements significatifs hérite de l'ordre partiel de précédence causale définit sur l'ensemble H des événements primitifs. Par exemple, on a représenté en figure II.4 le diagramme espace-temps de la figure II.3 en faisant abstraction des événements non significatifs (les événements significatifs sont indiqués par les points noirs). On constate que même si les événements de communication ont disparu en tant que tels, ils subsistent encore sous forme de relations d'ordre supplémentaires entre les événements significatifs.

Quelques définitions et propriétés

Dans toute cette section, on a choisi un niveau d'abstraction des événements. L'ensemble des événements correspondant au niveau d'abstraction choisi est noté E, et il est muni de la relation de précédence causale.

Notation 1 *Dans la littérature, la relation de précédence causale est souvent indiquée par le symbole «$\rightarrow$». Ici, on utilisera plutôt les symboles classiques en*

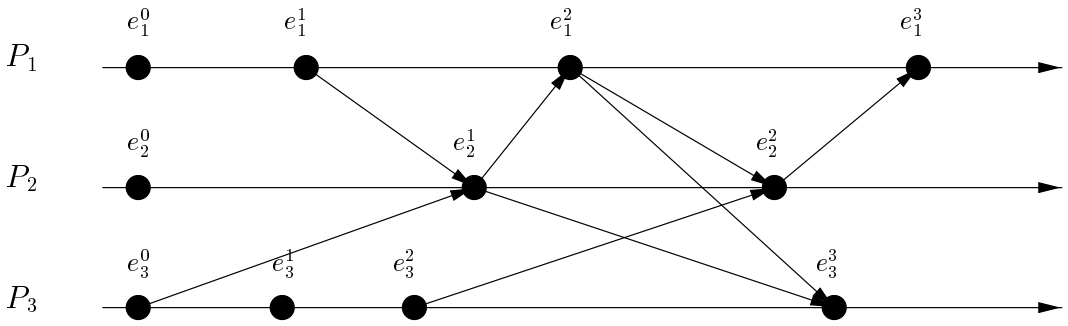


FIG. II.4 - *Une exécution vue à un niveau d'abstraction «utilisateur»*

théorie de l'ordre : «$\leq$» et «$<$». □

Définition 1 *Deux événements e_i et e_j sont* concurrents, *ou* indépendants, *ou encore* incomparables *(noté $e_i \| e_j$) si aucun ne précède causalement l'autre:*

$$e_i \| e_j \iff \neg(e_i \leq e_j) \wedge \neg(e_j \leq e_i)$$

□

Définition 2 *On dit qu'un événement e_i est* prédécesseur immédiat *d'un événement e_j (noté $e_i -\!\!< e_j$) si e_i précède causalement e_j et s'il n'existe aucun événement compris causalement entre e_i et e_j.*

$$e_i -\!\!< e_j \iff e_i < e_j \ \wedge \ \neg\exists e, e_i < e < e_j$$

□

On définit maintenant formellement la notion d'état local d'un processus et d'état global de l'exécution.

Définition 3 *On confond l'état local d'un processus avec la séquence des événements locaux au processus qui ont conduit à cet état local. Par abus de notation on note e_i^x l'état local du processus P_i juste après que l'événement e_i^x se soit produit.* □

La définition des états locaux est illustrée par la figure II.5 qui reprend l'exécution de la figure II.3. Comme les événements, les états locaux sont partiellement ordonnés :

Définition 4 *L'ordre de précédence causale induit un ordre strict sur les états locaux des processus. Cet ordre (noté «$\overset{w}{\rightarrow}$») est appelé précédence faible par Fromentin et Raynal [FR94b] :*

$$e_i^x \overset{w}{\rightarrow} e_j^y \iff e_i^x < e_j^y$$

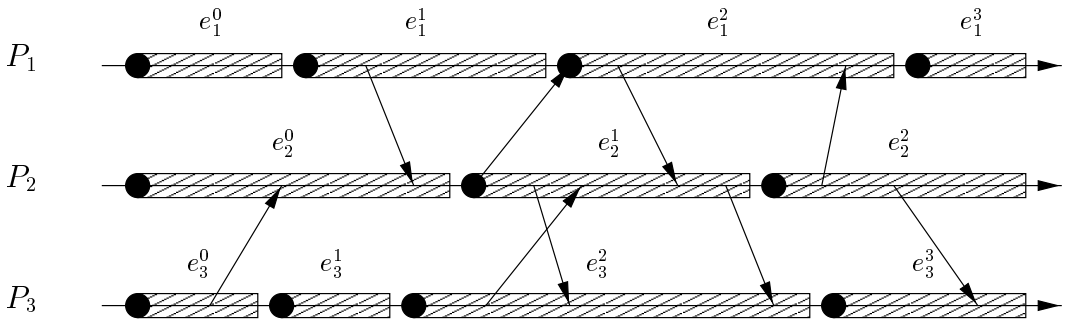


FIG. II.5 - *États locaux au niveau d'abstraction «utilisateur»*

(intuitivement, l'état e_i^x a commencé d'exister avant l'état e_j^y).
Fromentin et Raynal définissent une autre relation d'ordre strict sur les états locaux. Cet ordre (noté «$\xrightarrow{s}$») est appelé précédence forte et est défini par :

$$e_i^x \xrightarrow{s} e_j^y \iff e_i^{x+1} \leq e_j^y$$

(intuitivement, l'état e_i^x a cessé d'exister lorsque l'état e_j^y a commencé d'exister). □

Un état global est défini par la donnée d'un état local pour chaque processus de l'exécution. Un état global est dit cohérent s'il a «réellement pu exister» au cours de l'exécution. Formellement :

Définition 5 *Si l'état global est défini par l'ensemble $\Sigma = (s_1, ..., s_n)$ des états des processus de l'exécution, alors la cohérence de Σ est caractérisée par la relation suivante :*

$$\Sigma \text{ est cohérent} \iff \forall i, j, \neg(s_i \xrightarrow{s} s_j)$$

□

Une autre façon de définir les états globaux cohérents est de s'intéresser à l'ensemble des événements locaux qui ont conduit à un état global :

Définition 6 *On appelle coupure de l'exécution un ensemble d'événements définissant un état local pour chaque processus (cf. définition 3). Une coupure C est dite cohérente si l'ensemble C est clos pour la relation de précédence causale :*

$$C \text{ est cohérente} \iff \forall e \in C, \forall e' \in E, e' \leq e \implies e' \in C$$

□

Il y a correspondance bijective entre les états globaux cohérents et les coupures cohérentes de l'exécution.

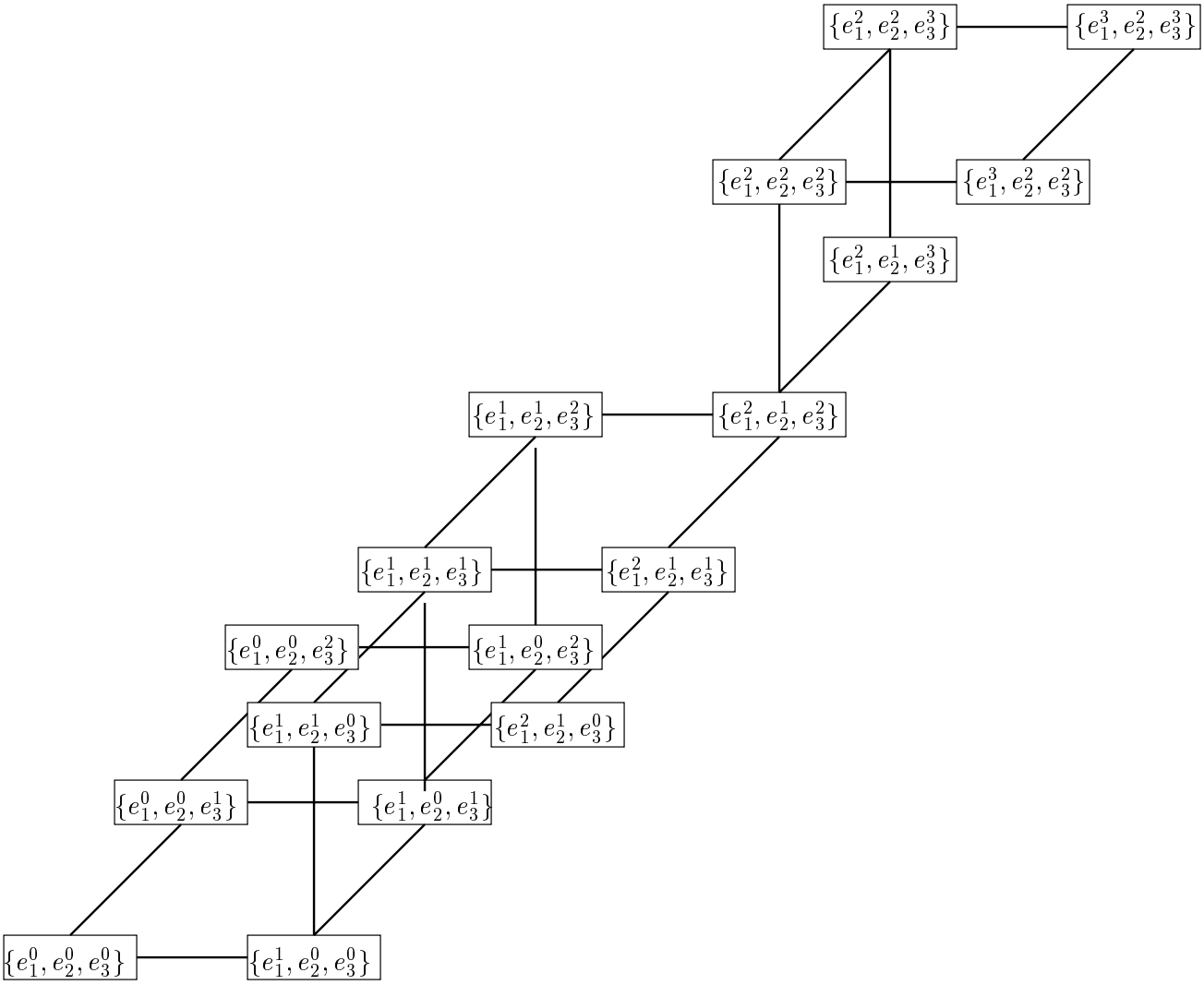


FIG. II.6 - *Treillis des états globaux*

L'ensemble des états globaux cohérents associés à une exécution répartie est structuré en *treillis* [Mat89, JZ90]. La figure II.6 montre le treillis des états globaux cohérents associé à l'exécution de la figure II.4.
Chaque chemin dans le treillis ordonne totalement les événements de l'exécution, de façon compatible avec l'ordre de précédence causale. C'est d'ailleurs ainsi qu'est définie la notion d'observation de l'exécution – on se limite ici aux observations séquentielles ; des observations parallèles d'une exécution répartie sont envisageables, cf. [CBDGF91, GW92, HK90].

Définition 7 *Une* observation *d'une exécution répartie est une extension linéaire de l'ordre de précédence causale sur l'ensemble E des événements de l'exécution (correspondant au niveau d'abstraction choisi).* □

Il y a une correspondance bijective entre les chemins du treillis et les observations de l'exécution [BM93, SM94]. Étudier l'exécution à travers les observations plutôt qu'à travers le treillis des états globaux revient à adopter une sémantique d'entrelacement du parallélisme. Notons que les deux approches se valent, car la connaissance de l'ensemble des observations possibles est équivalente à la connaissance de l'exécution et donc à la connaissance du treillis (c'est le théorème de Szpilrajn [BP82]).
Pour finir, nous définissons la notion de ré-exécution que nous retrouverons un peu plus loin dans ce chapitre (cf. section II.4) :

Définition 8 *Deux exécutions d'un programme réparti sont équivalentes (au niveau d'abstraction choisi) si elles produisent le même ensemble partiellement ordonné d'événements (à ce niveau d'abstraction). On appelle ré-exécution le fait de produire une exécution équivalente à une exécution donnée.*

II.3.2 Analyse des flots de contrôle

Plusieurs types de propriétés d'une exécution répartie peuvent être définis. Parmi les propriétés qui présentent un intérêt pratique, on peut distinguer celles qui portent sur les flots de contrôle et celles qui reposent sur la satisfaction d'un prédicat par les états globaux de l'exécution. On s'intéresse ici à la première catégorie.

Dans de nombreux cas – notamment pour l'analyse des synchronisations dans les systèmes répartis – il suffit de s'intéresser à l'ordre dans lequel les événements se produisent. Par exemple, considérons un système distribué de contrôle des feux tricolores à un carrefour. Soit $vert_i$ l'événement «Le feu i passe au vert». Le système de feux fonctionnera correctement tant que le prédicat $\Phi = (vert_1 \| vert_2)$ ($vert_1$ concurremment à $vert_2$) ne sera pas vérifié.

La détection de tels motifs comportementaux (*behavioral patterns*) dans les flots de contrôle d'une exécution répartie est connue sous le nom d'approche par abstraction comportementale (*behavioral abstraction*) et remonte aux travaux de Bates et Wileden [BW83]. Cette approche consiste à modéliser (abstraire) le comportement du système réparti en termes d'événements, puis à confronter le modèle avec la réalité lors d'une exécution du système.

```
event   feu_trop_court(feu, delta) is
        vert ' rouge
cond
        vert.id == feu;
        rouge.id == feu;
        0 <= rouge.time - vert.time < delta
with
        id = feu
end
```

FIG. II.7 - *Spécification d'événement avec l'EDL de Bates et Wileden*

Bates propose un langage de définition d'événements (EDL «Event Definition Language») [BW83, Bat87]. Un événement est soit un événement primitif de l'exécution, soit un événement composé, défini de manière récursive à l'aide d'opérateurs : opérateur de «ou logique» (noté «|»), opérateur de concaténation (noté «'»), opérateur de mélange (*shuffle*, noté «^»), etc. Les événements composés constituent l'abstraction du comportement attendu du système, que l'on peut ensuite confronter au comportement réel. La figure II.7 donne un exemple de spécification d'événement composé (notons que la spécification des événements de l'EDL suppose l'existence d'une horloge physique globale).

Détection de séquences de prédicats

S'inspirant du travail de Bates, Miller et Choi [MC88b] définissent formellement une classe de prédicats distribués et proposent des algorithmes pour en détecter la satisfaction. Les prédicats définis sont les prédicats simples (*Simple Predicates*, *SP*) qui correspondent à nos événements primitifs; les prédicats dis-

jonctifs (*Disjunctive Predicates*, DP) qui correspondent à un «ou logique» sur des prédicats simples (noté «$\cup$»); et les séquences de prédicats (*Linked Predicates*, LP) qui correspondent à un enchaînement causal de prédicats disjonctifs (noté «$\rightarrow$»). Ces notions sont récapitulées dans le tableau II.1.

SP	Occurrence d'un événement primitif
$DP ::= SP[\cup SP]^*$	Disjonction d'événements primitifs
$LP ::= DP[\rightarrow DP]^*$	Enchaînement causal d'événements disjonctifs

TAB. II.1 - *Définition des séquences de prédicats*

Contrairement à l'EDL, les séquences de prédicats ne supposent pas l'existence d'une horloge physique globale.
Miller et Choi proposent un algorithme pour vérifier au vol la satisfaction des séquences de prédicats. Le principe en est le suivant. Pour fixer les idées, considérons la séquence de prédicats $LP = DP_1 \rightarrow DP_2 \rightarrow DP_3$. Chaque processus possède une copie de LP et tente de détecter la satisfaction du premier prédicat de LP (c'est à dire DP_1). Dès qu'un processus détecte la satisfaction de DP_1, il diffuse causalement (au sens définit en section II.3.1) à tous les processus la queue de LP (c'est à dire $DP_2 \rightarrow DP_3$) et le mécanisme de détection est itéré sur $DP_2 \rightarrow DP_3$. Finalement, la satisfaction éventuelle de LP sera réalisée au moment où un processus détectera la satisfaction du dernier prédicat de la séquence (DP_3).
Notons que l'algorithme de Miller et Choi modifie l'ordre partiel des événements de l'exécution initiale en rajoutant des dépendances causales artificielles, dues à la diffusion de la séquence de prédicats à tous les processus.
Miller et Choi définissent également un prédicat conjonctif correspondant à l'exécution concurrente de prédicats disjonctifs, mais ne proposent pas d'algorithme pour sa reconnaissance au vol.

Détection de motifs réguliers

Le travail de Miller et Choi a donné lieu à deux nombreuses généralisations. Par exemple, Hurfin et al. [HPR93] s'intéressent à la détection de ce qu'ils nomment des séquences atomiques de prédicats locaux. Les prédicats locaux sont l'équivalant des prédicats disjonctifs de Miller. Les séquences atomiques de prédicats locaux sont de la forme $[\theta_1]\varphi_2[\theta_3]\varphi_4[\theta_5]\ldots$, où les θ_i et les φ_i sont des prédicats locaux.
Les séquences atomiques de prédicats locaux sont évaluées le long de chemin causaux de l'exécution, c'est à dire de séquences d'événements dont chacun est le prédécesseur causal immédiat de celui qui le suit dans la séquence (cf. définition 2). Pour fixer les idées, considérons la séquence $\Phi = [\theta_1]\varphi_2[\theta_3]\varphi_4[\theta_5]$. Φ est satisfait lors d'une exécution, s'il existe un chemin causal $P = e_1, e_2, ..., e_{i_2}, ..., e_{i_4}, ...$, tel que e_{i_2} satisfasse φ_2 et e_{i_4} satisfasse φ_4, et tel qu'aucun événement précédant e_{i_2} (sur le chemin causal) ne satisfasse θ_1, qu'aucun événement compris entre e_{i_2} et e_{i_4} ne satisfasse θ_3 et qu'aucun événement suivant e_{i_4} ne satisfasse θ_5.
L'algorithme de détection au vol des séquences atomiques de prédicats locaux dérive de l'algorithme de Miller. Le principe en est le suivant. Premièrement, on

construit un automate fini équivalent à la séquence à reconnaître. Par exemple, l'automate correspondant à la séquence Φ est donné par la figure II.8.

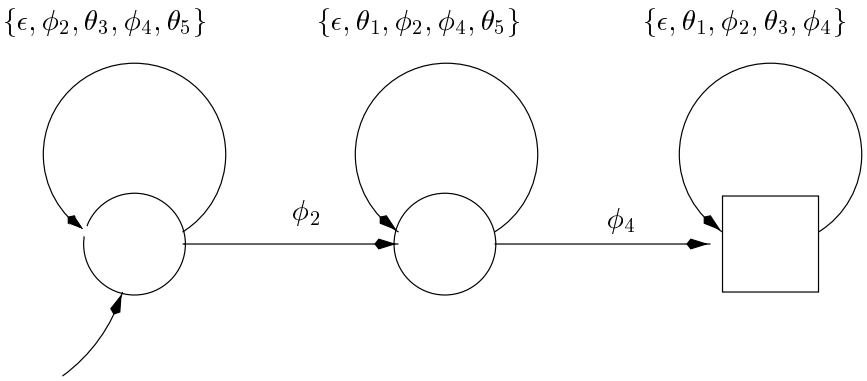


FIG. II.8 - *Automate reconnaissant la séquence atomique* $[\theta_1]\varphi_2[\theta_3]\varphi_4[\theta_5]$

Chaque processus P_i est ensuite muni d'une copie de l'automate et d'un ensemble Q_i d'états de l'automate qui a la signification suivante : Q_i contient un état q de l'automate si et seulement s'il existe un chemin causal C se terminant au dernier événement produit par P_i, et tel que C met l'automate dans l'état q (au début de l'exécution, Q_i contient seulement l'état initial de l'automate).
Les étapes de l'algorithme sont les suivantes :

- Lorsqu'un événement se produit sur un processus P_i, Q_i est mis à jour.
- L'ensemble Q_i dont est muni chaque processus P_i est diffusé causalement aux autres processus de la façon suivante : lorsqu'un processus P_j émet un message, le message est estampillé avec l'ensemble Q_j dont le processus est muni. Lorsqu'un processus P_j reçoit un message, et si ce message ne crée pas d'arc de transitivité [4], alors on ajoute l'ensemble des états estampillés sur le message à l'ensemble Q_j dont le processus est muni. Par rapport à l'algorithme de Miller, cette façon de procéder présente donc l'avantage de ne pas perturber l'ordre partiel des événements, car elle n'ajoute aucune dépendance causale.

La séquence atomique de prédicats locaux est détectée dès que l'ensemble Q_i des états atteints de l'automate, dont est muni un processus P_i, contient au moins un des états finaux de l'automate.
Il est possible de pousser un peu plus loin cette généralisation des séquences de prédicats : on peut s'intéresser aux motifs reconnus par un automate fini quelconque, plutôt que de se restreindre aux seuls automates finis associés à des

4. Il y a arc de transitivité quand l'événement réception du message n'est pas successeur immédiat de son événement émission. Notons que si le système de communication est causal, alors aucun message ne peut créer d'arc de transitivité. En revanche, si le système de communication n'est pas causal, alors il peut y avoir des arcs de transitivité, et le seul moyen pratique de les détecter est d'estampiller les événements de l'exécution avec les valeurs données par une horloge vectorielle [Mat89, Fid88], ce qui alourdit sensiblement l'algorithme.

séquences atomiques de prédicats locaux. C'est en faisant cette remarque que Fromentin et al. [FRGT94] introduisent la notion de motifs réguliers (le terme vient du fait que ces motifs sont définis à partir d'automates finis, i.e. à partir d'expressions régulières) et proposent un algorithme pour la reconnaissance au vol de ces motifs réguliers (il s'agit d'une variante de l'algorithme dont le principe a été présenté plus haut).

Détection d'événements «globaux»

Haban et Weigel [HW88] s'intéressent à la détection d'événements plus sophistiqués. Ils proposent de spécifier des événements globaux à partir d'événements primitifs et d'opérateurs variés. Les opérateurs qu'ils introduisent sont résumés par le tableau II.2.

(G_1, G_2 et G_3 sont des spécifications d'événements)
Spécification disjonctive : $G_1 \vee G_2$
Spécification conjonctive : $G_1 \wedge G_2$
Spécification de précédence causale : $G_1 \rightarrow G_2$
Spécification de concurrence causale : $G1 \| G2$
Spécification de négation : $@G1$
Spécification de contiguïté (*between operator*) : $@G_3(G_1, G_2)$ (vérifiée quand on trouve G_1 et G_2 tels que $G_1 \rightarrow G_2$ et qu'il n'existe pas de G_3 vérifiant $G_1 \rightarrow G_3$ et $G_3 \rightarrow G_2$)

TAB. II.2 - *Événements composés de Haban et Weigel*

Pour déterminer la dépendance et la concurrence causale (ce que ne savaient pas faire Miller et Choi), Haban et Weigel utilisent un système d'horloges vectorielles ([Mat89, Fid88], cf. aussi section V.2).
Le formalisme proposé par Haban et Weigel semble être un puissant outil de spécifications comportementales. Pourtant il est longuement critiqué par Schwarz et Mattern [SM94]. En effet, les règles proposées par Haban et Weigel pour détecter l'apparition de leurs événements sont ambiguës : suivant l'observation faite d'une même exécution (au sens de la définition 7), l'apparition d'un événement sera détectée ou non. Par exemple, considérons l'exécution décrite par la figure II.9 et l'événement composé $(a \| b) \rightarrow c$. L'événement est détecté si l'observation est $< b, a, c >$ car selon les règles de [HW88], l'occurrence de $a \| b$ est alors identifiée à celle de a et $(a \rightarrow c)$ est vrai. Par contre si l'observation est $< a, b, c >$, l'événement n'est pas reconnu, car l'occurrence de $(a \| b)$ est alors identifiée à celle de b et $(b \rightarrow c)$ est faux. Haban et Weigel ont essayé d'améliorer leur règles de détection dans [HZMW91], sans toutefois parvenir à un système tout à fait satisfaisant.

Détection de chemins de données

Hseush et Kaiser [HK88, HK90] proposent un formalisme semblable à celui de Haban et Weigel en ce qui concerne la puissance d'expression, mais qui en évite les points litigieux.

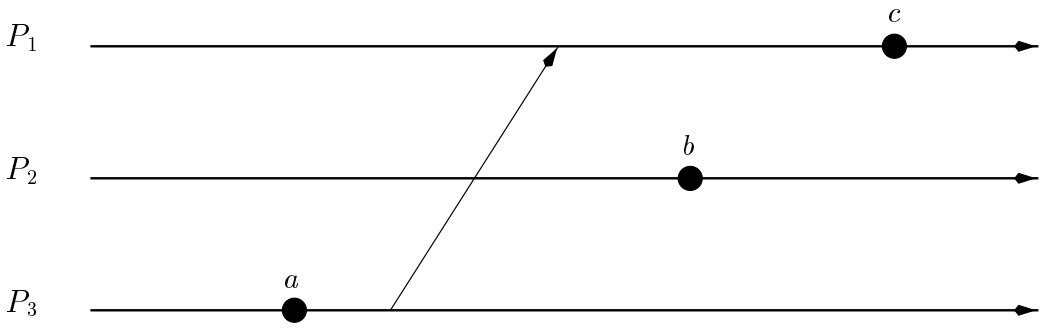


FIG. II.9 - *Est-ce que* $(a \| b) \rightarrow c$ *est satisfait?*

Leur démarche consiste à remarquer (comme le font également Jard et al. dans [JJGVR94]) qu'on peut utiliser le treillis des états globaux d'une exécution comme un automate fini, pour reconnaître les observations de cette exécution. Ceci est illustré par la figure II.10. L'automate fini issu du treillis permet de reconnaître toutes les observations de l'exécution choisie, et uniquement celles-ci.

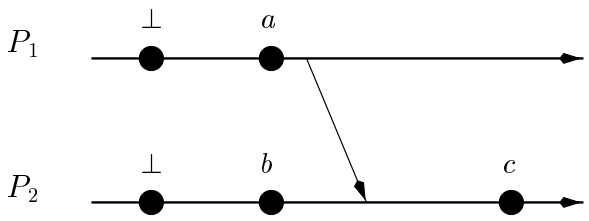
Hseush et Kaiser remarquent que malheureusement, on ne peut pas utiliser une seule observation pour reconnaître une exécution. Par exemple l'observation $< a, b, c >$ peut indifféremment correspondre à chacune des exécutions représentées en figures II.10 et II.11. Si l'on se contente de nourrir directement l'automate avec des observations, il ne sera pas possible de reconnaître à coup sûr une exécution. Cela n'a rien d'étonnant, car de façon générale, une exécution est caractérisée par l'*ensemble* de ses observations, pas par une de ses observations.

Pour résoudre ce problème, Hseush et Kaiser ont l'idée de compléter l'observation de l'exécution par la donnée, pour chaque événement, de ses événements prédécesseurs immédiats (au sens de la définition 2). Notons qu'en complétant ainsi l'observation, il est possible de reconstituer l'ordre partiel des événements, et donc de caractériser l'exécution.

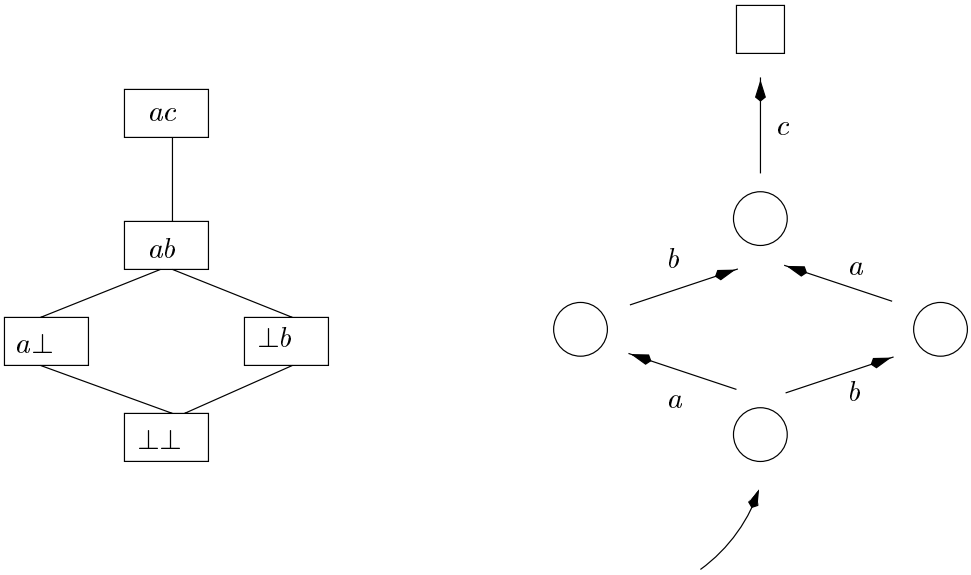
Il suffit ensuite de compléter de la même façon l'automate issu du treillis des états globaux de l'exécution, en étiquetant chaque transition non seulement par l'identificateur de l'événement associé, mais également par l'ensemble des prédécesseurs immédiats de cet événement. Ceci est illustré par la figure II.12. Hseush et Kaiser obtiennent ainsi ce qu'ils appellent un automate à prédécesseur (*predecessor automaton*). L'automate à prédécesseur permet de reconnaître précisément une exécution à partir d'une observation «complétée».

Hseush et Kaiser ont défini un langage d'«expression de chemins de données» (*data path expressions* [HK90]), et des règles qui permettent de construire un automate à prédécesseur à partir d'un chemin de données.

Les expressions de chemins de données sont à base d'événements primitifs, d'un opérateur de précédence causale *immédiate* (noté «;»), d'un opérateur de disjonction (noté «+»), d'un opérateur de répétition (noté «*») et d'un opérateur d'exécution concurrente (noté «&»). Du fait de l'opérateur de répétition, les



Exécution à reconnaître



Treillis associé Automate associé

FIG. II.10 - *Le treillis des états globaux est un automate fini*

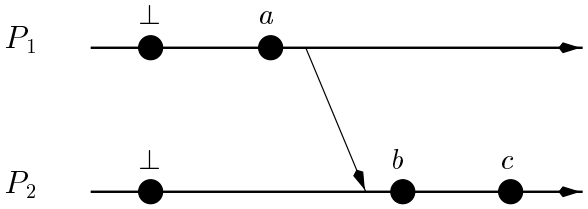


FIG. II.11 - *Une autre exécution reconnue par l'automate*

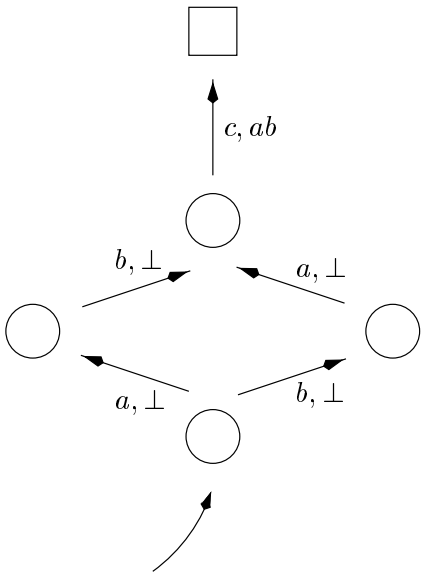


FIG. II.12 - *Un automate à prédécesseur*

automates définis à partir d'une expression de chemin de données peuvent présenter des cycles (contrairement aux automates à prédécesseur obtenus directement à partir d'un treillis d'états globaux). Ceci est illustré par la figure II.13, qui représente l'automate à prédécesseur correspondant au chemin de données : $((a\&b);c)^*$.

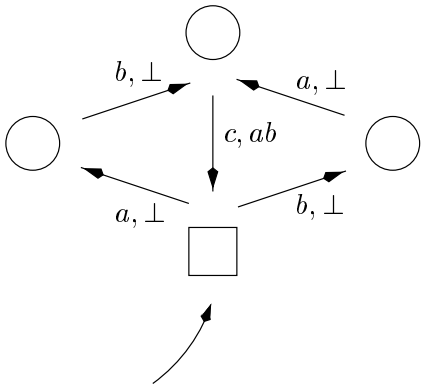


FIG. II.13 - *Un autre automate à prédécesseur*

Ponamgi et al. [PHK91] ont construit un outil de mise au point pour programmes parallèles, fondé sur l'utilisation d'expressions de chemins de données et d'automates à prédécesseur. A l'usage, il apparaît cependant que l'utilisation des automates à prédécesseur n'est pas une panacée :

– l'absence d'un opérateur de précédence causale «$\rightarrow$» (remplacé par l'opérateur de précédence immédiate «;») s'avère assez pénible.

- La taille d'un automate à prédécesseur peut croître de manière exponentielle par rapport à la taille du chemin de données qu'il représente ([BM93, SM94]) ce qui rend la détection coûteuse à implémenter, en terme d'espace mémoire.

- Les automates à prédécesseur engendrés à partir d'expressions de chemins de données peuvent présenter des problèmes d'ambiguïté et d'instabilité (pour plus de détails, se référer à [HK90]).

- L'utilisation (coûteuse) d'horloges vectorielles, même si elle n'apparaît pas explicitement dans la présentation des automates à prédécesseurs, est pratiquement indispensable, car dans bien des cas c'est le seul moyen de déterminer au vol quels sont les événements prédécesseurs immédiats d'un événement donné.

II.3.3 Analyse des états globaux

Dans cette section, on s'intéresse au problème de la satisfaction d'un prédicat par les états globaux d'une exécution.
A titre d'exemple, considérons le prédicat $x_1 + x_2 + \ldots + x_n < k$, où chaque x_i est une variable locale au processus P_i.

Détection de propriétés stables

Si par hasard il se trouve que la valeur des x_i décroit au cours du temps, alors le prédicat $x_1 + x_2 + \ldots + x_n < k$, une fois vérifié, restera vrai. C'est ce qu'on appelle une propriété stable. Le prédicat correspondant à une propriété stable est tel que si ce prédicat est satisfait par une exécution, alors à partir d'un certain moment, tous les états globaux (cohérents) de l'exécution le satisfont. C'est pour cette raison que la détection des propriétés stables est relativement facile.
La technique «classique» de détection des propriétés stables consiste à prendre régulièrement un instantané de l'état global de l'exécution (*snapshot*). On peut pour cela utiliser un algorithme tel que celui proposé par Chandy et Lamport [CL85]. On évalue le prédicat correspondant à la propriété stable sur les instantanés d'état global obtenus, soit jusqu'à ce que l'exécution se termine, soit jusqu'à ce qu'on trouve un état global qui satisfasse la propriété.

Détection de propriétés instables

Certes, la classe des propriétés stables comprend les propriétés de terminaison d'un calcul réparti et d'état d'interblocage. Cependant, de nombreuses propriétés ne sont pas stables ; par exemple : $x_1+x_2+\ldots+x_n < k$ (dans le cas général), ou bien «le nombre de messages en transit dans le système de communication est supérieur à 100», etc.
Il n'est pas possible d'utiliser la stratégie à base de snapshots exposée plus haut pour détecter la satisfaction des propriétés instables. En effet, l'utilisation d'un algorithme d'instantané d'état global tel que celui de Chandy et Lamport ne

permet pas d'obtenir l'ensemble exhaustif des états globaux de l'exécution. Et il est possible que la propriété instable soit satisfaite précisément sur les états globaux qui n'ont pas été trouvés par l'algorithme.
C'est pourquoi (en l'absence d'horloge physique globale), la détection des propriétés instables passe par l'analyse du treillis des états globaux qui contient l'ensemble des états globaux par lesquels l'exécution est passée ou a pu passer.

Modalités POS et DEF

Nous reprenons ici la présentation faite dans [Ray94].
Introduites par Cooper et Marzullo [CM91], les modalités POS et DEF adoptent une sémantique d'entrelacement du parallélisme et constituent deux façons d'envisager la question «l'exécution R satisfait-elle le prédicat global Φ ?»

- R satisfait POS Φ (pour *POSsibly* Φ, noté $R \models \text{POS } \Phi$) s'il existe au moins une observation de l'exécution répartie R qui contient un état global Σ dans lequel Φ est vrai ($\Sigma \models \Phi$). Formellement :
$$R \models \text{POS } \Phi \iff \exists \Sigma \in I(R),\ \Sigma \models \Phi$$
(Où $I(R)$ est l'ensemble des idéaux de R, c'est à dire le treillis des états globaux de R.)

- R satisfait DEF Φ (pour *DEFinitely* Φ, noté $R \models \text{DEF } \Phi$) si toute observation O contient un état global Σ qui satisfait Φ. Formellement :
$R \models \text{DEF } \Phi \iff \forall O$ chemin maximal de $I(R),\ \exists \Sigma \in O,\ \Sigma \models \Phi$

La satisfaction POS est intéressante pour détecter une erreur potentielle exprimée par un prédicat Φ. La satisfaction DEF est plus intéressante pour montrer qu'une propriété est respectée.
L'évaluation des modalités POS et DEF nécessite la construction puis le parcours du treillis des états globaux de l'exécution. Des algorithmes de construction du treillis sont donnés dans [CM91, BM93, Die92]. Ils fonctionnent au vol, c'est à dire en pipe-line avec l'exécution répartie elle-même : celle-ci leur communique les états locaux produits par les processus et ils les assemblent pour former des états globaux cohérents qu'ils placent dans le treillis en cours de construction. [CM91] propose également des algorithmes de parcours du treillis pour détecter les propriétés.
Notons que la taille du treillis (en nombre d'états globaux) peut être exponentielle par rapport au nombre de processus [SM94, BM93], ce qui réduit les applications pratiques d'une telle approche pour la détection de propriétés instables. [5]

Modalité PROP

Le principe de la modalité PROP est de ne s'intéresser qu'au sous-ensembles des états globaux de l'exécution qui ont été perçus par tous les observateurs.

5. La taille du treillis est bornée par $(\#evts)^{(\#procs)}$, borne qui est effectivement atteinte.

Cette modalité à été introduite par Fromentin et Raynal [FR94a] dans le but d'éviter de construire le treillis des états globaux de l'exécution.

- R satisfait PROP Φ (noté $R \models$ PROP Φ) s'il existe un état global Σ, commun à toutes les observations de R, qui satisfait Φ.

En d'autres termes, la propriété exprimée par Φ a été satisfaite dans le même état global pour tous les observateurs. Pour formaliser et rendre opérationnelle la définition précédente, le concept d'inévitabilité d'un état global a été introduit [FR94a]. Un état global est *inévitable* s'il appartient à toutes les observations de l'exécution ; par exemple, l'état $\Sigma = (s_1^2, s_2^1, s_3^2)$ de la figure II.6 est inévitable. Notons que :

$$R \models \text{PROP } \Phi \implies R \models \text{DEF } \Phi \implies R \models \text{POS } \Phi$$

Fromentin et Raynal [FR94c] donnent une caractérisation des états globaux inévitables et proposent un algorithme de complexité $\mathcal{O}(n^3k)$ – où n est le nombre de processus et k le nombre maximal d'états locaux d'un processus durant l'exécution – qui calcule tous les états globaux inévitables de l'exécution. Comme précédemment, l'algorithme fonctionne en *pipe-line* derrière l'exécution qui lui transmet les états locaux des processus.

Détection d'événements «simultanés»

La notion d'événements simultanés est introduite par Spezialetti et Gupta [SG94]. Dans le cadre de la mise au point au vol, ils se posent le problème suivant : peut-on arrêter l'exécution d'un programme réparti sur un prédicat global, et garantir que le prédicat sera effectivement satisfait par l'état global dans lequel le programme se sera arrêté. Le problème est qu'avec des algorithmes de détection du type «pipe-line» tels que ceux décrits plus haut, le prédicat global n'est plus forcément vérifié au moment où il est détecté, et a fortiori encore moins au moment où un point d'arrêt peut être réalisé. Spezialetti et Gupta proposent un algorithme capable, lors de la satisfaction d'un prédicat, de stopper l'exécution dans un état où le prédicat est encore vrai.
Pour cela, ils définissent la notion d'événement simultané :

Définition 9 *Considérons une exécution distribuée et un prédicat global Φ. L'*événement simultané $==(\Phi)$ *est satisfait dans l'état p du processus P si et seulement si les deux conditions suivantes sont vérifiées.*

- Condition de satisfaction *: pour tous les états globaux de l'exécution comprenant l'état p du processus P, le prédicat global Φ est satisfait.*
- Condition de stabilité *: si un point d'arrêt distribué est déclenché depuis P dans l'état p, alors quel que soit l'état global correspondant à la réalisation de ce point d'arrêt, cet état global satisfait Φ.*

Les événements simultanés permettent donc de détecter des propriétés qui, sans être tout à fait stables, présentent une certaine stabilité au cours du temps (elles restent vraies au moins jusqu'à la réalisation du point d'arrêt).

Spezialetti et Gupta utilisent une technique d'analyse statique [SG94] pour déterminer automatiquement les points du programme (et donc les états locaux p de processus P) où la condition de satisfaction est potentiellement vérifiée et où la condition de stabilité est nécessairement vérifiée.

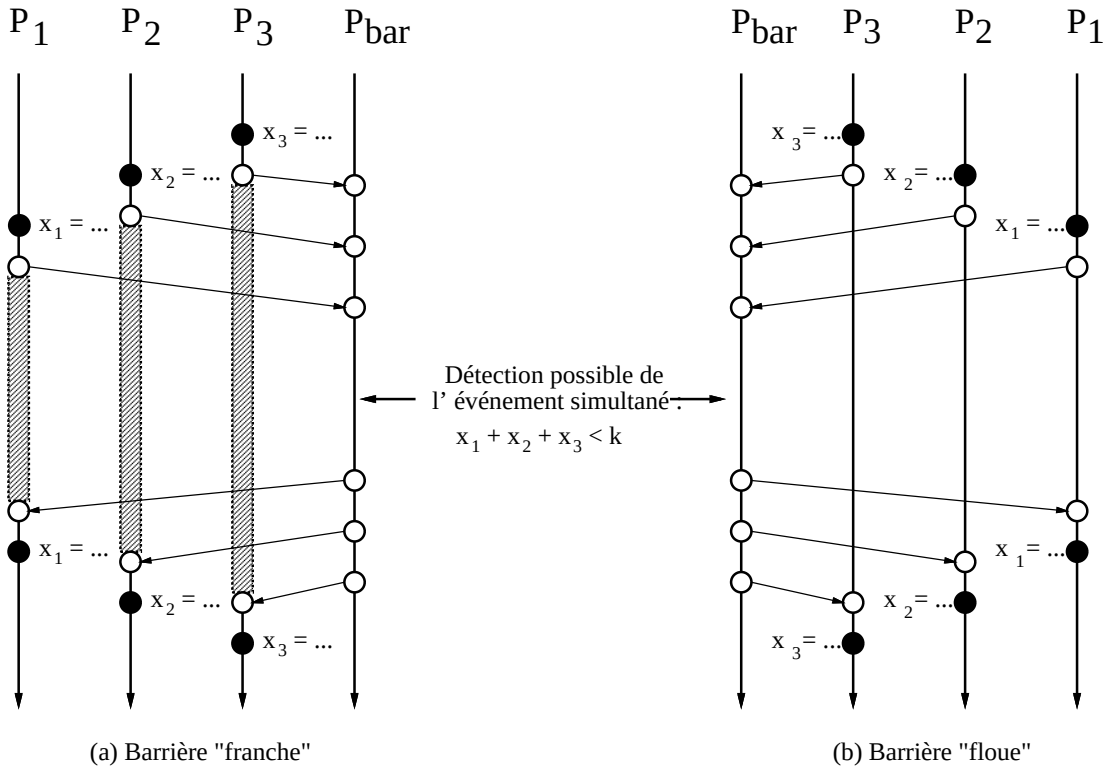


FIG. II.14 - *Détection d'un événement simultané par un processus barrière*

Dans la pratique de tels points correspondent à des barrières de synchronisation naturelles du programme (cf. l'exemple figure II.14). Ils pourraient également correspondre à des points de synchronisation imposés par un synchroniseur [Awe85, Die92].
L'approche de Spezialetti et Gupta partage avec celle de Fromentin et Raynal (modalité PROP) le fait qu'on ne s'intéresse pas à tous les états globaux de l'application, mais seulement à certains états : les états inévitables dans le cas de la modalité PROP, et les états où la condition de stabilité est garantie dans le cas des événements simultanés.

II.4 Analyse post-mortem

Un grand nombre d'outils de mise au point de programmes parallèles ou répartis proposent un fonctionnement en deux phases. Dans un premier temps, l'exécution du programme est observée et enregistrée. Dans un deuxième temps, alors que l'exécution du programme est terminée, les donnée enregistrées sont analysées. Cette approche de la mise au point est qualifiée de *post-mortem*.
Par exemple, les différentes techniques de représentation de l'historique de l'exécution, décrites en section II.2.4 appartiennent à la catégorie post-mortem.
L'avantage des techniques post-mortem est qu'il est possible d'effectuer des traitement coûteux de l'information de trace sans perturber l'exécution. En effet, les seules perturbations de l'exécution sont celles occasionnées durant la phase de collecte et leur importance peut être limitée (cf. section II.2.2).
Une possibilité particulièrement intéressante des techniques post-mortem est la simulation de l'exécution qui a été tracée. Cette simulation est aussi appelée *ré-exécution*. Les ré-exécutions successives permettent de reproduire un comportement identique du programme mis au point, de générer les mêmes flots de données et de contrôle, de produire les mêmes résultats que durant l'exécution initiale.
La méthode de mise au point par ré-exécution s'effectue en deux temps : enregistrement puis ré-exécution proprement dite (*trace-replay*). Lors de la phase de ré-exécution, les informations enregistrées (par exemple l' historique des événements) sont utilisées par un mécanisme de contrôle afin de piloter l'exécution du programme et de la contraindre à reproduire le comportement initial. La phase de ré-exécution peut être répétée autant de fois qu'il est nécessaire, et il est donc possible de mettre en œuvre la méthode «cyclique» de mise au point.
La ré-exécution est bien une technique de simulation : elle permet de reproduire le comportement initial du programme à un niveau d'abstraction donné, qui peut être plus ou moins fin. À la limite, on peut considérer que les techniques d'observation de l'exécution décrites en section II.2.4 sont des techniques de ré-exécution (avec un niveau d'abstraction très grossier). En pratique, il y a toujours une limite à la fidélité avec laquelle on peut reproduire l'exécution d'un programme. Notamment, les outils de ré-exécution ne restituent en général pas les intervalles de temps réels entre les événements (la ré-exécution est plus lente). Et s'ils permettent parfois une reproduction très fidèle des événements du niveau applicatif, ils ne permettent pas la reproduction exacte des événements du niveau système.
La plupart des réalisations d'outils de ré-exécution se différencient de par la nature des informations conservées lors de la phase d'enregistrement (utilisation d'historiques, d'instantanés de l'état global du programme), et la façon d'effectuer la simulation de l'exécution initiale (simulation dirigée par les données, dirigée par le contrôle). Nous en faisons plus loin une présentation un peu plus détaillée.
L'avantage principal des techniques de ré-exécution est de permettre la mise en œuvre de la méthode de mise au point cyclique pour des programmes parallèles et répartis, en résolvant le problème du non-déterminisme de ces programmes. En général les outils de ré-exécution décrits dans la littérature sont capables

de reproduire l'état mémoire des processus du programme cible, ce qui permet au programmeur de disposer de bien plus d'information qu'avec les simples techniques de présentation de l'information décrites en section II.2. Ces techniques restent cependant bien sûr applicables et utilisables en complément à la ré-exécution.

II.4.1 Ré-exécution dirigée par les données

Dans cette approche, la phase d'enregistrement conserve, pour chaque processus de l'exécution, la valeur des données qu'il utilise. La reproduction de l'exécution est effectuée en relançant le programme et en utilisant les données enregistrées pour piloter les processus (Curtis et Wittie [CW82], Jones [JBW87], Pan [PL89]).
Suivant les réalisations, la phase d'enregistrement crée un ou plusieurs historiques (un par processus par exemple). Pour chaque processus, l'historique conserve le résultat des actions potentiellement cause d'un comportement non-déterministe :

- Les valeurs lues par le processus et qui proviennent de l'environnement du programme (par exemple la valeur retournée par une horloge physique, ou la valeur d'une donnée saisie par l'utilisateur du programme).
- Les valeurs des communications inter-processus : le contenu des variables dans le cas d'une communication par variables partagées, ou le contenu des messages dans le cas d'une communication par messages.

Ces données enregistrées sont utilisées lors de la reproduction à la place de celles normalement rendues par les instructions correspondantes.
Pan [PL89] permet de reproduire l'exécution d'un groupe de processus parallèles dans le cas d'une communication par mémoire partagée. Les interactions entre processus s'effectuent soit par l'intermédiaire d'appels au système, soir par accès à la mémoire partagée. Le résultat des appels systèmes est récupéré dans l'historique de chaque processus par le biais d'une librairie particulière. Les accès potentiels à des variables partagées sont détectés à la compilation, par analyse du graphe de dépendances des données, et le résultat des lectures est récupéré par instrumentation du code source du programme. La reproduction de l'exécution d'un processus s'effectue à partir d'un point de reprise (cf. section II.4.4) en réveillant le processus image associé à ce point de reprise et en pilotant son exécution avec les informations de l'historique.
Jones [JBW87] permet de reproduire l'exécution de programmes parallèles dans le cas d'une communication par messages. Les interactions entre processus s'effectuent uniquement par des messages, qui sont interceptés et conservés lors de la phase d'enregistrement. Lors de la reproduction, l'utilisateur peut choisir les processus qu'il désire ré-exécuter normalement. La reproduction s'effectue à partir d'un point de reprise (cf. section II.4.4), en restaurant l'état des processus concernés et en dirigeant leur exécution avec les informations de l'historique.
La technique de visualisation des flots de données aussi bien en avant qu'en arrière (flowback analysis) développée par Choi et Miller [MC88a] peut aussi

être considérée comme une technique de ré-exécution dirigée par les données. Lors de la phase d'enregistrement, l'outil de mise au point conserve les valeurs des variables spécifiées dans les prologues et épilogues des blocs de contrôle, ainsi que la valeur des variables potentiellement partagées. Ces valeurs sont utilisées ultérieurement pour piloter la reproduction de l'exécution. Le lecteur pourra trouver d'autres exemples d'utilisation de la technique de ré-exécution par les données en section IV.7.
L'inconvénient de ce type de ré-exécution est de provoquer une perturbation importante de l'exécution, due à la conservation d'un volume important d'information. Il possède cependant l'avantage de permettre une ré-exécution partielle : le programmeur peut reproduire l'exécution d'une partie seulement des processus qui constituent le programme, voire d'un processus isolé.

II.4.2 Ré-exécution dirigée par le contrôle

L'objectif de la reproduction dirigée par le contrôle [LMC87, LSZ90] est de limiter la perturbation de l'exécution lors de la phase d'enregistrement, en diminuant le volume des informations conservées. L'idée est de n'enregistrer une information de trace que si elle est indispensable à la caractérisation de l'exécution observée. Une information ne doit pas être conservée si elle peut être déduite d'autres informations déjà enregistrées, ou si elle peut être regénérée au moment de la ré-exécution.
Concrètement, lors d'une ré-exécution dirigée par le contrôle, l'historique conserve :

- Les valeurs lues par le processus en provenance de l'environnement du programme (comme dans le cas de la ré-exécution dirigée par les données).
- Une représentation des synchronisations qui caractérisent les communications inter-processus (et non plus la valeur des communications, contrairement à la ré-exécution dirigée par le données). Par exemple, dans le cas d'une communication par messages, on n'enregistre pas le contenu des messages, mais seulement l'ordre des émissions et des réceptions. (En fait, on n'est même pas obligé d'enregistrer l'ordre de toutes les émissions et réceptions [NM92, Net93]). Dans le cas d'une communication par mémoire partagée, on conserve uniquement des numéros de versions des objets manipulés et non plus la valeur de ces objets.

La reproduction de l'exécution est réalisée en pilotant les processus avec les données enregistrées et en reproduisant les synchronisations conservées dans l'historique.
LeBlanc et Mellor-Crummey [LMC87] ont mis en œuvre cette technique dans leur débogueur *Instant Replay*, développé pour un système fortement couplé à base de mémoire partagée. Dans ce système, toutes les communications inter-processus sont modélisées comme des opérations sur des objets partagés. Un numéro de version est associé à chaque objet. Lors d'une communication inter-processus, on enregistre les numéros de version des objets, et on les incrémente de façon atomique. Lors de la ré-exécution, les synchronisations (et les communications) sont reproduites en bloquant un processus qui veut accéder un objet

jusqu'à ce que le numéro de version de l'objet corresponde au numéro enregistré dans l'historique (on peut montrer qu'une telle façon de procéder ne conduit pas à un interblocage [Rug91].)
Leu et Schiper [LSZ90] ont mis en œuvre une généralisation de la technique d'Instant Replay pour des architectures à mémoire répartie avec communication par message.
La ré-exécution dirigée par le contrôle permet 1) de limiter l'importance de l'effet de sonde durant la phase d'enregistrement et 2) d'obtenir des historiques moins volumineux. En revanche, elle implique la ré-exécution complète du programme, car la regénération de certaines données peut nécessiter de connaître l'état de chaque processus, ou du réseau de communication.
Il est possible de réaliser une combinaison de ré-exécution dirigée par les données et par le contrôle. Il suffit pour cela de considérer certains processus du programme comme faisant partie de l'environnement, et d'enregistrer dans l'historique les communications avec ces processus sous forme de données (et non plus de synchronisations). La seule contrainte est de pouvoir bien déterminer quels processus sont considérés comme faisant partie de l'environnement (cf. section IV.4). En procédant ainsi, on peut ré-exécuter l'ensemble des processus ne faisant pas partie de l'environnement indépendamment des autres (voir aussi la discussion à propos des points de reprise en section II.4.4).
Le lecteur pourra trouver d'autres exemples d'utilisation de la technique de ré-exécution par le contrôle en section IV.7.

II.4.3 Ré-exécution d'événements asynchrones

Dans les exemples donnés plus haut, les événements considérés correspondent toujours à des actions synchrones – c'est à dire «volontaires» – des processus du programme : accès à un objet partagé, émission d'un message, récupération d'un message reçu. Il est relativement facile d'instrumenter le code du programme pour réaliser l'enregistrement et la reproduction de ces événements.
Cependant, dans un système réel, il est rare que tous les événements à prendre en compte correspondent soient synchrones. Pour s'en convaincre, il suffit de penser au cas des interruptions matérielles qu'il est impossible de représenter sous forme d'événements volontaires. Dans d'autres cas, la représentation d'un événement sous forme d'une action volontaire serait possible mais à un coût prohibitif, car la façon naturelle de le représenter est de le considérer comme une action asynchrone («involontaire»).
Pour illustrer cette dernière situation, on peut considérer un ensemble de processus qui effectuent des accès à une mémoire partagée avec une granularité très fine (cf. les considérations concernant l'instrumentation matérielle en section II.2.1). Dans un tel cas il serait beaucoup trop coûteux d'instrumenter chaque accès à la mémoire partagée. La bonne façon d'envisager le problème de l'observation et de la ré-exécution de ces processus est

- soit de se placer au niveau du séquenceur (cas où les processus sont exécutés en temps partagé sur un seul processeur). Dans ce cas les événements asynchrones à prendre en compte sont les changements de contexte.

– soit de se placer au niveau des échanges entre mémoire partagée et antémémoires (*caches*) des processeurs (cas où les processus sont réellement exécutés en parallèle sur plusieurs processeurs). Dans ce cas, les événements asynchrones à prendre en compte sont les accès au bus par les différents processus.

En d'autres termes, on est parfois (souvent ?) obligé de considérer les événements asynchrones. Dans ce cas, l'enregistrement et la reproduction sont plus difficile car ils ne peut pas être réalisée par une simple instrumentation des processus cibles.
L'observation d'un événement asynchrone nécessite un certain support du système, de façon à pouvoir enregistrer l'état de l'exécution au moment où l'événement asynchrone se produit. Une technique simple consiste à demander au système d'invoquer une routine spéciale du moniteur, de façon atomique avec l'apparition de chaque événement asynchrone. La routine du moniteur se chargera par exemple de récupérer la valeur d'un compteur d'instructions [MCL89, For89, HP 89].
La reproduction de l'événement asynchrone est plus simple: dans un premier temps, le système de mise au point amène le programme dans l'état correspondant au moment d'apparition de l'événement asynchrone (en utilisant les techniques de ré-exécution «classiques»). Dans une deuxième temps, l'événement asynchrone est simulé (élection d'un nouveau processus pour simuler un changement de contexte; création d'un pseudo-processus pour exécuter une routine d'interruption; etc.)

II.4.4 Points de reprise

L'exécution de certains programmes (par exemple des programmes de calcul scientifique) s'étend sur une très longue durée. Parfois, elle n'est même pas a priori bornée (cas d'un serveur). L'utilisation de la pure technique de ré-exécution comme méthode de mise au point de tels programmes n'est pas praticable. Il n'est en effet pas envisageable de recommencer l'exécution du programme à partir du début.
Pour résoudre ce problème, on utilise la technique des points de reprise. Un point de reprise est une «photographie» (snapshot) partielle ou complète de l'état du programme. Les points de reprise peuvent être utilisés pour reproduire une partie seulement de l'exécution du programme (en l'occurrence la partie qui fait apparaître le comportement erroné) : on reprend la ré-exécution à partir de l'état intermédiaire du programme correspondant au point de reprise.
En général il est préférable de coordonner la création des points de reprises des différents processus [GGLS92, XN93]. À l'extrême, l'utilisation d'un algorithme de détermination d'état global réalise une coordination forte en imposant une contrainte de cohérence aux états correspondant aux points de reprise locaux des différents processus (cf. section II.3.1).
L'utilisation combinée de la ré-exécution et des points de reprise permet de réaliser le «voyage dans le temps» (*time travel*). C'est à dire que le programmeur peut demander au système de mise au point de recréer un état arbitraire de

l'exécution initiale du programme. Pour cela, le système de mise au point recrée un état intermédiaire antérieur à l'état demandé en utilisant les points de reprise les plus récents possible, puis ré-exécute le programme jusqu'à obtenir l'état demandé. Cette technique généralise aux programmes répartis les mécanismes d'exécution arrière développés pour des programmes séquentiels (par exemple [ADS91]).

II.4.5 Exploration de l'espace des exécutions possibles

Les techniques de ré-exécution permettent de reproduire une exécution particulière d'un programme à partir de l'historique des événements de cette exécution. Dans le cadre de la ré-exécution, l'historique est obtenu par enregistrement d'une exécution initiale du programme. On peut cependant imaginer de renverser l'ordre des choses, c'est à dire de piloter l'exécution à partir d'historiques artificiels.
Si on applique ce principe, il devient possible d'explorer l'ensemble des exécutions possibles du programme. On peut par exemple créer un historique qui pilote le programme le long d'un cheminement d'exécution improbable, dans le but de tester son comportement dans des conditions «extrêmes».
Stone [Sto88] réalise une première mise en œuvre cette technique de la façon suivante. Après exécution initiale du programme, l'utilisateur peut effectuer une série de ré-exécutions, pilotées par un tableau de concurrence (concurrency map) qui est une sorte d'historique incomplet. Le fait que l'historique soit incomplet implique que les ré-exécutions engendrées peuvent diverger de l'exécution initiale. Dans ce cas, l'utilisateur doit ajouter des contraintes au tableau de concurrence, jusqu'à ce qu'une ré-exécution fidèle puisse être obtenue.
De façon plus systématique, on trouve dans [AV93] un algorithme qui génère automatiquement des «germes» d'exécutions divergentes à partir de l'historique d'une exécution particulière. L'algorithme consiste à détecter dans l'historique de cette exécution des paires de réceptions de messages qui peuvent être interverties et à créer un nouvel historique en les intervertissant (Smith proposait déjà une idée semblable dans [Smi84] : permettre à l'utilisateur de supprimer, modifier ou ajouter «à la main» des messages afin tester le comportement du programme cible.) L'historique ainsi obtenu permet de reproduire l'exécution initiale jusqu'au point ou les réceptions de messages ont étées échangées, ce qui donne naissance à une nouvelle exécution qui peut être tracée puis fournir à son tour d'autres «germes» d'exécutions différentes.

II.5 Conclusion

Nous venons de présenter un panorama assez large des techniques récentes de mise au point des programmes parallèles et répartis dans le cadre des systèmes distribués asynchrones. Par rapport à ce tableau, le travail de recherche exposé dans cette thèse se positionne de la façon suivante.
Nous avons tenté de construire un outil de mise au point distribuée pour le système CHORUS, en nous inspirant des techniques et des architectures qui nous ont paru les plus prometteuses.
En particulier, nous avons retenu une architecture multi-couches pour le moniteur (moniteur local – moniteur central) et nous avons choisi de placer les sondes d'observation en instrumentant le système plutôt que les processus (l'instrumentation des programmes en vue de leur mise au point est toujours une étape fastidieuse, et de plus, instrumenter le système permet une bonne prise en compte des événements asynchrones).
Pour les fonctions fournies par notre outil de mise au point, nous avons mis l'accent sur le service de ré-exécution, qui nous paraît être une brique de base particulièrement intéressante. Nous avons tenté de définir une interface utilisateur simple et puissante, permettant une mise en œuvre facile de la ré-exécution.
Enfin, nous avons tenté de ne pas oublier les problèmes de performance.
Durant ce travail de recherche, nous n'avons pas essayé de fournir un cadre pratique pour l'évaluation des techniques de détection de propriétés décrites à la section II.3. Cependant, notre outil de mise au point pourrait s'avérer un point de départ adéquat pour une telle évaluation. Il fournit en effet les services d'observation et de traçage qui permettent de représenter une exécution sous forme d'un ordre partiel d'événements.
Enfin, notre travail de recherche a été l'occasion d'une petite contribution à la théorie des horloge logiques, motivée par certains problèmes pratiques rencontrés lors de la réalisation du service de ré-exécution.

Chapitre III

Support du micro-noyau pour la ré-exécution

III.1 Introduction

Dans ce chapitre, nous décrivons un support système pour l'observation et le contrôle d'exécutions réparties. Nous avons défini et implanté ce support dans le cadre de l'architecture répartie à micro-noyau CHORUS [Cho92]. Principalement articulé autour d'un service de notification des événements système et d'un service de redirection des appels système, ce support a été développé pour répondre :

- aux besoins spécifiques du service de ré-exécution offert par le débogueur réparti CDB [Rug94a], qui fait l'objet de cette thèse.
- aux besoins spécifiques du moniteur d'applications «temps réel» PATOC développé par une équipe de recherche de Siemens AG, à Munich [Her91].

Toutefois, nous avons fait en sorte que le support soit suffisamment général et extensible pour être utile dans un cadre plus large. Notamment :

- Le service de notification des événements système pourrait servir de base pour la construction d'outils logiciels dans différents domaines comme **(1)** l'optimisation de performances (*tuning*) d'applications ou de systèmes répartis s'exécutant au dessus du micro-noyau ; **(2)** le contrôle interactif d'applications réparties (*program steering* [GVS94]) ; et **(3)** la mise au point d'exécutions réparties en général (visualisation de l'exécution, détection de propriétés, etc.).
- Le service de ré-exécution pourrait servir de base pour l'implantation d'un certain nombre d'algorithmes (notamment dans le domaine de la tolérance aux fautes).

  Par exemple, il pourrait faciliter l'implantation dans le micro-noyau CHORUS d'algorithmes de recouvrement après défaillance fondés sur les techniques

de retour arrière (*rollback recovery*) et de ré-exécution, tels que ceux décrits dans [JZ87, SBY88, EZ92]. [1]

La réalisation d'une architecture tolérante aux fautes fondée sur le modèle meneur/cohorte (*leader/cohort* [BMST93]) ou principal/copie (*primary/-backup*) serait un autre bénéficiaire potentiel. En effet, le principe d'une telle architecture est la reproduction par la cohorte (resp. par le processus copie) de l'exécution du processus meneur (resp. du processus principal). Il est vrai que dans la pratique, les implémentations du modèle meneur/cohorte font l'hypothèse que tous les processus sont individuellement déterministes, ce qui réduit le problème de la reproduction de l'exécution du meneur par la cohorte à un «simple» problème de diffusion atomique des messages. Cependant, une généralisation à des processus non déterministes est possible, qui nécessiterait une véritable ré-exécution.

Le travail exposé dans ce chapitre s'insère dans le cadre du projet européen ESPRIT N^0 6603 OUVERTURE et a donné lieu à publication [HR93, HR94, Rug95].
Le reste du chapitre est organisé comme suit. La section III.2 présente brièvement le noyau CHORUS et les abstractions nécessaires à la compréhension du chapitre. En section III.3, nous décrivons le service de notification d'événements système, son interface et son architecture. La section III.4 présente le service de redirection des appels système. Les services additionnels que nous avons été amenés à implanter pour le support de PATOC et CDB sont regroupés et brièvement décrits en section III.5. En section III.6, nous faisons une étude assez détaillée de l'impact des services de notification et de redirection sur les performances du système, aussi bien à l'échelle microscopique que macroscopique. Nous concluons en section III.7.

III.2 Le micro-noyau CHORUS

Dans la suite de ce chapitre, et dans les chapitres suivants, nous ferons explicitement référence au système CHORUS [RAA$^+$88, Roz90, Cho92]. Nous avons donc jugé utile d'en faire ici une brève présentation. La présentation suit l'exposé de Rozier [Roz91].
Un système CHORUS se compose d'un certain nombre de *sites* reliés par un réseau de communication. Un site CHORUS est un ensemble de ressources fortement couplées. En général il y a correspondance entre site et machine. Cependant, une machine à architecture faiblement couplée (e.g. un hypercube), peut correspondre à plusieurs sites CHORUS.
Sur chaque site se trouve une copie du micro-noyau CHORUS, qui définit un certain nombre d'abstractions et de services :

– L'*acteur* (*actor*), unité de structuration du système.

1. Par rapport à des algorithmes fondés uniquement sur l'utilisation de points de reprise (*checkpoints*) [KT87], ces algorithmes présentent l'avantage de nécessiter uniquement la reprise des processus défaillants, ce qui peut réduire la durée du recouvrement et le niveau d'indisponibilité du système pendant le recouvrement [EZ92].

L'acteur CHORUS est l'unité d'allocation de ressources. Il fournit un espace d'adressage protégé, ainsi que des accès à des ressources systèmes (objets de communication, objets de mémoire).

– L'*activité* (*thread*), unité d'exécution.

  Une activité CHORUS est un processus séquentiel. Elle est caractérisé par son contexte d'exécution.

  Une activité est attachée à un acteur, qui définit son environnement d'exécution. Plusieurs activités peuvent partager l'environnement offert par un acteur. Elles peuvent librement accéder à toutes les ressources détenues par cet acteur.

  Les activités sont cadencées par le noyau comme des entités indépendantes : les activités d'un même acteur peuvent s'exécuter en parallèle sur les différents processeurs d'un multi-processeur.

  Les activités d'un acteur partagent un même espace d'adressage. Elles peuvent donc se synchroniser et communiquer via un partage de mémoire. Elles peuvent également utiliser le service de communication *IPC* (*Inter-Processus Communication*) offert par le micro-noyau, uniforme et indépendant de la localisation.

– La *porte* (*port*).

  La porte est la ressource de communication point à point bi-directionnelle. Une porte est rattachée à un acteur. Les portes CHORUS permettent aux activités de communiquer par échange de messages. Une activité peut émettre un message vers toute porte dont elle connaît l'identificateur unique (voir plus bas). Une activité peut consommer les messages en attente derrière les portes rattachées à son acteur et seulement celles-ci. À chaque porte est associée la queue des messages non encore consommés.

– Au sein du système CHORUS, les acteurs sont identifiés par des identificateurs uniques globaux, *UI*s (*Unique Identifiers*). Un UI désigne toujours la même entité (un acteur en l'occurrence), où qu'elle soit localisée (en espace et en temps), et où que soit localisée l'entité (activité) qui utilise l'UI.

  Les activités ne possèdent pas de UI. Chaque activité est identifiée par un identificateur local, *LI* (*Local Identifier*), valable au sein de l'acteur auquel elle est attachée. Un LI n'a pas de sens en dehors du contexte d'un acteur. Un LI n'est pas unique dans le temps : si on détruit puis crée une activité dans un même acteur, il est possible que LI de l'activité détruite soit réutilisé pour l'activité crée.

  Les portes sont désignées à la fois par un UI global et par un LI valable au sein de l'acteur auquel la porte est rattachée. Lorsqu'une activité veut émettre un message vers une porte, c'est l'UI qu'elle doit utiliser pour désigner la porte.

– En plus des acteurs décrits plus hauts, dits *utilisateurs* et munis chacun de son propre espace d'adressage, le micro-noyau définit également une catégorie d'acteurs privilégiés nommés *acteurs superviseurs*. Tous les acteurs superviseurs partagent le même espace d'adressage, au sein duquel est implanté le micro-noyau.

III.3 Le service de notification des événements système

Le support que nous avons défini pour l'observation des exécutions réparties consiste en un service de notification des événements du micro-noyau. Pour réaliser cette notification, nous avons choisi d'utiliser un mécanisme d'*upcall*. Il s'agit d'un appel synchrone du noyau vers le client de la notification, qui s'apparente à un appel procédural plus qu'à un mécanisme d'exception, et qui est souvent implémenté comme un simple appel procédural (d'où le terme *up-call*). Le qualificatif de *up*-call lui est attribué pour souligner le fait que c'est la couche système (i.e. le noyau) qui invoque l'upcall et non la couche applicative (i.e. le client de la notification).

III.3.1 Mécanisme d'upcall

Parmi les différentes possibilités envisagées (simple comptage des événements par le noyau, journalisation des événements directement par le noyau, notification des événements de façon asynchrone par le biais du système de communication, etc.), le mécanisme d'upcall nous a semblé le mieux adapté :

– Par rapport à un mécanisme de simple comptage des événements (certes plus efficace), le mécanisme d'upcall est plus flexible et permet d'implanter un plus grand nombre de services.

– Par rapport à un service intégré de journalisation des événements par le noyau lui-même – comme celui décrit par Lehr et Black [LB90] – le mécanisme d'upcall résout les problèmes liés à la gestion d'un tampon d'événements[2] : allocation du tampon, vidage périodique du tampon par le client du service, gestion des débordements, etc.

– En outre, le mécanisme d'upcall, de par son caractère synchrone, présente des avantages déterminants par rapports aux mécanismes asynchrones :

  – il est facile à implanter, efficace et plus fiable.

2. Par exemple, si le client du service de notification ne fait que compter les événements, l'utilisation du mécanisme d'upcall permet de s'affranchir complètement de l'utilisation d'un tampon. Bien sûr, si le client veut journaliser les événements dans un fichier, l'usage d'un tampon est indispensable car l'écriture dans le fichier ne peut pas être réalisée de façon synchrone avec l'upcall. Dans ce cas, le mécanisme d'upcall permet de reporter l'implantation et la gestion du tampon au niveau applicatif.

– il est plus ouvert. En effet, il permet au noyau de ne fournir qu'un petit nombre de paramètres lors de la notification de l'événement, puisque le client de la notification peut lui-même récupérer un certain nombre des paramètres qui lui sont spécifiques (par exemple : la valeur d'une horloge, la répartition des processus actifs sur les processeur et leur contexte d'exécution, la valeur d'un compteur d'instruction, etc.). Ainsi il est possible de n'implanter au niveau du micro-noyau qu'un service de notification minimal, avec un faible encombrement mémoire et de bonnes performances, et qui réussit cependant à satisfaire les besoins de clients variés.

Il faut en revanche mentionner une limitation du mécanisme d'upcall : en général il n'est pas possible de ré-invoquer le noyau depuis un upcall (seuls quelques appels système sont autorisés – les mêmes que ceux qui sont autorisés dans le contexte d'une routine d'interruption.) La raison de cette restriction est que – pour des raisons d'efficacité – l'upcall est effectué sans replacer au préalable le noyau dans un état cohérent. Par exemple, un upcall peut être effectué alors que certains objets internes au noyau n'ont pas été déverrouillées. Lorsqu'on se trouve dans le contexte de cet upcall, il est donc interdit d'invoquer des appels système entraînant le verrouillage de ces objets, sous peine d'interblocage.
Notons que c'est l'état incohérent dans lequel est laissé le noyau, bien plus que le sens de l'invocation (vers le noyau ou vers l'application), qui distingue l'*upcall* de l'appel système standard (*downcall*). Ceci est illustré par la figure III.1.

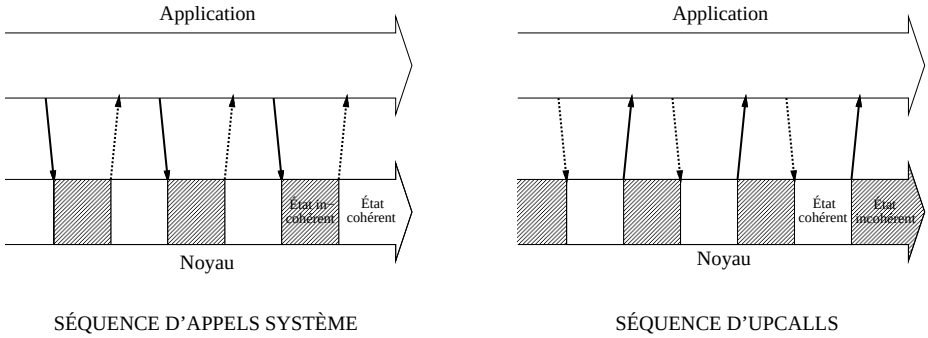


FIG. III.1 - *Downcalls et upcalls*

En pratique, les appels système autorisés dans le contexte d'un upcall sont les mêmes que ceux qui sont autorisés dans le contexte d'une interruption (la situation est en effet très similaire). Dans le cas du système CHORUS, cela inclut la possibilité d'envoyer un signal asynchrone à une activité. Cette activité, possédant un contexte d'exécution normal, pourra effectuer les éventuelles opérations qui n'auraient pas pu être menées à bien dans le contexte de l'upcall.
Une conséquence importante liée à la restriction concernant les appels systèmes autorisés dans le contexte d'un upcall est que le mécanisme d'upcall ne peut être mis en œuvre que dans le cas où les clients du service de notification sont sûrs (*trusted*), i.e. le noyau a la garantie qu'ils vont respecter cette restriction.

Le système CHORUS possède une catégorie d'acteurs privilégiés, les acteurs superviseurs, qui partagent leur espace d'adressage avec le micro-noyau. Les acteurs superviseurs sont supposés sûrs, et pour garantir un certain niveau de sécurité, ils ne peuvent être lancé que par un utilisateur ayant un privilège «super utilisateur». Pour les raisons mentionnées plus haut, nous avons donc choisi de restreindre l'utilisation du service de notification aux seuls clients implantés dans des acteurs superviseurs. Bien entendu, rien n'empêche de tel clients d'utiliser ensuite l'IPC CHORUS pour faire ensuite suivre la notification des événements vers de simple acteurs utilisateurs.
Le fait de restreindre l'usage du service de notification aux acteurs superviseurs a un avantage supplémentaire. Comme les acteurs superviseurs partagent l'espace d'adressage du noyau, les upcalls peuvent s'effectuer sans changement de contexte, ce qui permet d'obtenir de meilleures performances.

III.3.2 Interface du service de notification d'événements

Le service de notification des événements système possède une interface «orientée objet» écrite en C++ [Str86]. L'interface a été réalisée de la façon suivante. Nous avons défini un certain nombre d'événements pertinents dans le cadre de la mise au point : événements de création/destruction d'objets CHORUS, événements liés au cadencement des activités, etc. Nous avons associé à chaque événement un objet CHORUS particulier, qualifié d'objet *principalement impliqué* dans l'événement [HR93] – par exemple, à l'événement «préemption» est associé l'objet «activité».
Pour chaque type d'objet CHORUS (site, acteur, activité et porte), nous avons défini une classe C++ appelée *classe virtuelle de notification* de l'objet. Pour chaque événement associé à l'objet CHORUS, cette classe C++ exporte une méthode virtuelle appelée *méthode virtuelle de notification* de l'événement. Les classes de notification que nous avons ainsi définies sont décrites par les tableaux III.1, III.2, III.3, et III.4. Elles sont exportées par le micro-noyau et accessibles aux clients du service de notification.
Notons que pour l'instant, les seuls événements définis sont :

- les événements matériels : interruptions, dépassements temporels (*time-outs*), déroutements (*traps*), exceptions,
- les événements associés à l'émission/réception de messages IPC,
- les événements associés au cadencement des activités.

Cette liste ne prétend pas être exhaustive et il est tout à fait envisageable de la compléter ultérieurement – par exemple en rajoutant des événements associés aux groupes de communication CHORUS, ou à la mémoire virtuelle.
Pour chaque type d'objet CHORUS qu'il envisage de sonder, un client du service de notification doit produire une *classe concrète de notification*, dérivée de la classe virtuelle de notification. La classe concrète de notification est la représentation chez le client de l'objet à sonder. Elle doit fournir une implémentation concrète des méthodes virtuelles de notification. Ce sont ces méthodes concrètes

`struct KnMonSite: KnMon {`	
`virtual void disConnected();`	Le site a été déconnecté du service de notification.
`virtual void intrBegin(` `u_int intrNo);`	Une interruption s'est produite sur le site (méthode appelée avant la routine de gestion de l'interruption).
`virtual void intrEnd(` `u_int intrNo);`	Une interruption s'est produite sur le site (méthode appelée après la routine de gestion de l'interruption).
`};`	

TAB. III.1 - *Classe de notification exportée par le noyau pour l'objet site*

`struct KnMonActor: KnMon {`	
`virtual void deleted();`	L'acteur a été détruit.
`virtual void disConnected();`	L'acteur a été déconnecté du service de notification.

`virtual void intr(`
`u_int intrNo,`
`KnIntrRoutine intrRout,`
`const KnThreadCtx* thCtx);`
	Une routine de gestion d'interruption va être exécutée dans le contexte de l'acteur.

`virtual void tout(`
`KnToutRoutine toutRout,`
`long toutArg);`
	Un routine de gestion de «time-out» va être exécutée dans le contexte de l'acteur.

`};`	

TAB. III.2 - *Classe de notification exportée par le noyau pour l'objet acteur*

`struct KnMonPort: KnMon {`	
`virtual void deleted();`	La porte a été détruite.
`virtual void disConnected();`	La porte a été déconnectée du service de notification
`virtual void msgEnQueued(` `int msgLi);`	Un message a été ajouté à la queue de la porte.
`virtual void msgDeQueued(` `int msgLi);`	Un message a été retiré de la queue de la porte.
`};`	

TAB. III.3 - *Classe de notification exportée par le noyau pour l'objet porte*

`struct KnMonThread: KnMon {`	
`virtual void deleted();`	L'activité a été détruite.
`virtual void disConnected();`	L'activité a été déconnectée du service de notification.

`virtual KnMonActor* acCreated(`
`const KnUniqueId* acUi,`
`const KnKey* acKey);`
	L'activité a créé un acteur.

`virtual KnMonThread* thCreated(`
`const KnUniqueId* acUi,`
`const KnKey* acKey,`
`int thLi);`
	L'activité a créé une activité.

`virtual KnMonPort* ptCreated(`
`const KnUniqueId* acUi,`
`const KnKey* acKey,`
`int ptLi,`
`const KnUniqueId* ptUi);`
	L'activité a créé une porte.

`virtual void userEvent(`
`int eventNo,`
`const void* dataAddr,`
`u_int dataSize);`
	L'activité a généré un événement «utilisateur».

`virtual void preTrap(`
`int trapNo,`
`const KnThreadCtx* thCtx);`
	L'activité a fait un trap (méthode appelée avant la routine de gestion du trap).

`virtual void postTrap(`
`int trapNo,`
`const KnThreadCtx* thCtx);`
	L'activité a fait un trap (méthode appelée après la routine de gestion du trap).

`virtual void preExc(`
`int excNo,`
`const KnThreadCtx* thCtx);`
	L'activité a fait une exception (méthode appelée avant la routine de gestion de l'exception).

`virtual void postExc(`
`int excNo,`
`const KnThreadCtx* thCtx);`
	L'activité a fait une exception (méthode appelée après la routine de gestion de l'exception).

`virtual void run();`	L'activité va être cadencée.
`virtual void preempted(` `const KnThreadCtx* thCtx);`	L'activité a été préemptée.
`virtual void beReady(` `unsigned thStatus);`	Une condition bloquante a été enlevée de l'état de l'activité.
`virtual void beUnReady(` `unsigned thStatus);`	Une condition bloquante a été rajoutée à l'état de l'activité.
`virtual void msgSent(` `int msgLi);`	L'activité a émis un message.
`};`	

TAB. III.4 - *Classe de notification exportée par le noyau pour l'objet activité*

– que nous appelons *méthodes concrètes de notification* – qui seront ultérieurement invoquées par le mécanisme d'upcall, lors de l'occurrence des événements à notifier.
Le client du service de notification doit ensuite se connecter à l'objet CHORUS qu'il désire sonder. Pour cela, il doit instancier un objet C++ de la classe concrète de notification appropriée. Cet objet C++, appelé *objet sonde* joue un double rôle :

- Il permet d'établir la connexion «objet CHORUS - client» : par le biais d'un appel système spécifique, le client fournit un pointeur sur l'objet sonde au micro-noyau et celui-ci réalise la connexion en mémorisant ce pointeur dans la structure interne qui représente l'objet CHORUS (pour ce faire, nous avons ajouté un champ «sonde» à ces structures internes).

- Il sert au client de représentation de l'objet CHORUS sondé. En particulier, il est le lieu privilégié pour conserver des informations d'état.

Une fois la connexion effectuée, lors de l'occurrence de chaque événement système impliquant l'objet CHORUS, la notification est réalisée par l'invocation directe de la méthode appropriée de l'objet sonde. C'est le mécanisme d'upcall que nous avons décrit à la section précédente.
L'interface que nous proposons permet également de réaliser l'héritage des sondes, lors de la création d'un objet CHORUS par une activité déjà sondée. Pour cela, nous avons ajouté à la classe de notification de l'objet «activité» des méthodes de notification pour les événements de création d'objets. Lors de la création d'un objet CHORUS par une activité sondée, le noyau invoque la méthode de création appropriée chez le client du service de notification. Le client peut alors soit (**1**) retourner un pointeur sur un nouvel objet sonde C++ – que le noyau associera à l'objet CHORUS créé – ou bien (**2**) retourner un pointer nul – indiquant ainsi au noyau que l'objet créé ne doit pas être sondé. Ce mécanisme est illustré par la figure III.2.
Notons que le client n'est pas obligé de fournir une implémentation pour toutes les méthodes virtuelles d'une classe de notification. Par conséquent, le service de notification permet de réaliser un double filtrage des événements :

- Un premier filtrage par objet CHORUS : le micro-noyau ne notifie que les événements relatifs aux objets CHORUS effectivement connectés au service de notification.

- Un deuxième filtrage par type d'événement : le micro-noyau ne notifie que les événements pour lesquels le client de la notification a fourni une implémentation de la méthode de notification.

L'utilisation d'une interface orientée objet présente des avantages certains. En particulier, elle permet de dériver de nouvelles interfaces de notification avec une sémantique plus riche, à partir de l'interface «brute» exportée par le noyau. Ce point est développé dans la section III.3.3.

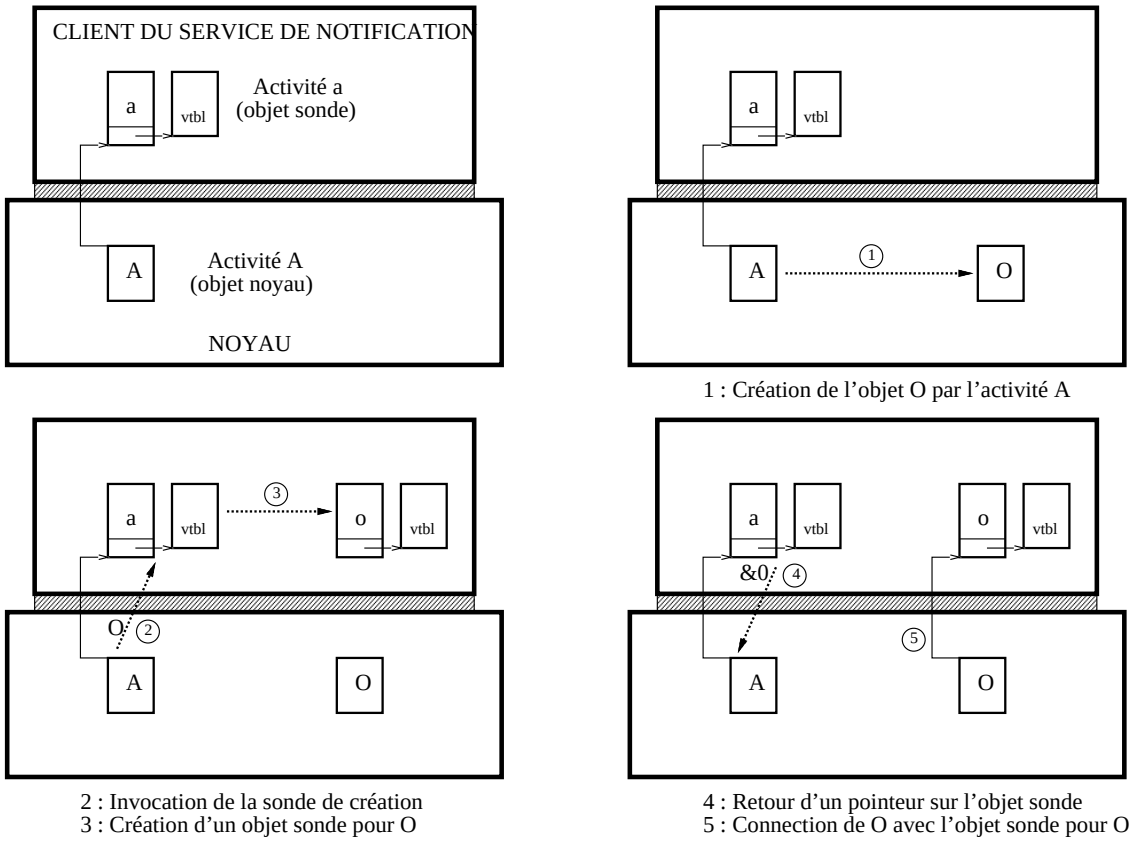


FIG. III.2 - *Mécanisme d'héritage des sondes*

III.3.3 Architecture globale du service de notification

Le service de notification exporté par le noyau implante une fonction «minimale» et ne répond pas directement à tous les besoins des outils de mise au point potentiels :

- Il ne permet pas à plusieurs clients de se connecter simultanément au même objet CHORUS : le noyau enregistre au plus l'adresse d'un seul objet sonde par objet CHORUS.
- Il n'est capable que de notifier des événements produits par le noyau. Or dans certains cas, on voudrait notifier des événements générés en dehors du noyau. [3]

Nous avons donc défini un module additionnel, capable de prendre en compte les besoins non couverts directement par le noyau. Ce module, appelée EXTMON(*EXTended MONitoring*) s'interpose entre le noyau et les clients du service de notification. Il réalise les fonctions suivantes :

- EXTMON réalise un multiplexage des notifications d'événements, ce qui permet à plusieurs clients de sonder le même objet CHORUS simultanément.
- EXTMON intègre le service de nommage ASCII des objets CHORUS et réalise la notification des événements de nommage.

Naturellement, les fonctions du module EXTMON auraient pu être directement intégrées au noyau. Cependant, en les implantant dans un module externe, on réalise plusieurs bénéfices :

- La présence du module EXTMON est naturellement optionnelle. Il est possible de ne pas l'intégrer aux configurations légères du noyau.
- Il est possible d'avoir différentes versions du module EXTMON implantant des politiques différentes – politiques de démultiplexage des notification d'événements par exemple (quel client doit être notifié en premier ? tous les clients doivent-ils être notifiés ? etc.), ou politiques de nommage (doit on garantir l'unicité des noms donnés aux objets CHORUS ?). L'utilisation d'un module externe permet de ne pas imposer une politique donnée.

Du point de vue du noyau, le module EXTMON est un simple acteur superviseur, client du service de notification noyau. Du point de vue de ses clients, le module EXTMON est un service de notification qui exporte une interface étendue par rapport à celle du noyau. En fait, l'interface exportée par EXTMON

3. C'est par exemple le cas des événements de nommage symbolique des objets CHORUS. Pour répondre aux besoins des outils PATOC et CDB, nous avons développé un serveur de noms symboliques pour les objets CHORUS. Le service de bas niveau fourni par le noyau ne permet pas de notifier ces événements de nommages, car le serveur qui les produit est un acteur indépendant du micro-noyau.

`struct MonSite : KnMonSite {`	
`dlink monLink;`	Chaînon reliant les objets «sonde» connectés simultanément au site (réservé à l'usage exclusif du noyau).
`virtual void siteNamed(` `const char* siteName);`	Le site a été nommé.

`virtual void acNamed(`
`const KnUniqueId* acUi,`
`const KnKey* acKey,`
`const char* acName);`
	Un acteur a été nommé.

`virtual void thNamed(`
`const KnUniqueId* acUi,`
`const KnKey* acKey,`
`int thLi,`
`const char* thName);`
	Une activité a été nommée.

`virtual void ptNamed(`
`const KnUniqueId* acUi,`
`const KnKey* acKey,`
`int ptLi,`
`const char* ptName);`
	Une porte été nommée.

`};`	

TAB. III.5 - *Classe de notification exportée par EXTMON pour l'objet site*

`struct MonActor : KnMonActor {`	
`dlink monLink;`	Chaînon réservé à l'usage exclusif du noyau.
`};`	

TAB. III.6 - *Classe de notification exportée par EXTMON pour l'objet acteur*

`struct MonThread : KnMonThread {`	
`dlink monLink;`	Chaînon réservé à l'usage exclusif du noyau.
`};`	

TAB. III.7 - *Classe de notification exportée par EXTMON pour l'objet activité*

`struct MonPort : KnMonPort {`	
`dlink monLink;`	Chaînon réservé à l'usage exclusif du noyau.
`};`	

TAB. III.8 - *Classe de notification exportée par EXTMON pour l'objet porte*

est dérivée (au sens C++ du terme) de l'interface exportée par le noyau. Ceci est illustré par les tableau III.5, III.6, III.7 et III.8 qui décrivent les classes de notification exportée par EXTMON.
L'architecture générale du service de notification d'événements est illustrée par la figure III.3. Lors de la notification d'un événement, les trois premiers niveaux traversés (noyau, module EXTMON, moniteur local) sont situés en espace superviseur et communiquent par upcall. Le dernier niveau (moniteur global ou outil de mise au point proprement dit) est situé en espace utilisateur et communique avec les niveaux précédents via l'IPC CHORUS.

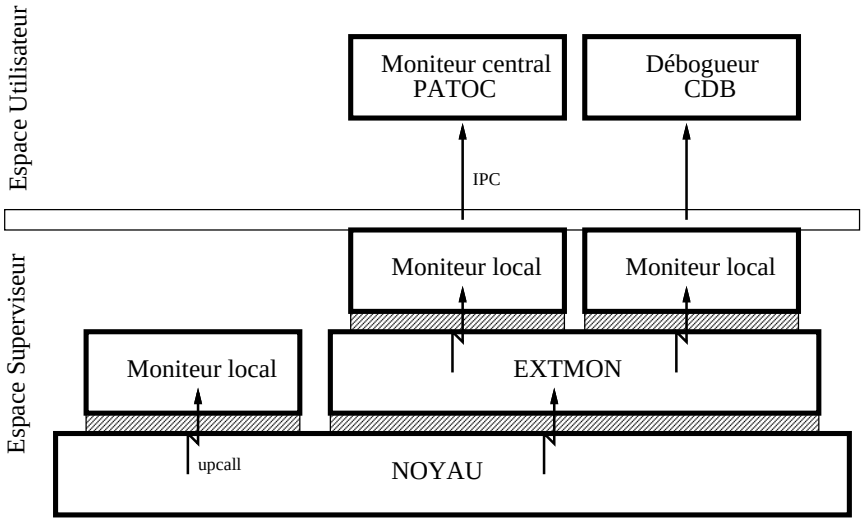


FIG. III.3 - *Architecture du service de notification*

III.4 Redirection des appels système

Normalement, lorsqu'une activité CHORUS effectue un appel système (trap), le noyau CHORUS récupère dans une table la routine à invoquer en fonction du numéro du trap qu'a effectué l'activité. Contrairement à ce qui est réalisé dans Mach [Pat93], CHORUS définit une table de routines unique, partagée par tous les acteurs du site, et non une table par acteur.
Pour permettre à un outil de mise au point de prendre facilement le contrôle d'un acteur CHORUS, nous avons développé un support du micro-noyau qui permet la redirection dynamique des appels système d'un acteur vers un moniteur. Ce support offre la possibilité de définir des tables de traps additionnelles et d'associer à chaque acteur une référence vers une telle table (qu'il peut partager avec d'autres acteurs).
Conceptuellement, le trap transfère l'exécution d'un acteur donné vers un acteur gestionnaire qui prend en compte les appels système du premier acteur (par exemple, l'AM gère les appels système des acteurs mis au point avec CDB ; le PM gère les appels système des processus MiX). À son tour, l'acteur gestionnaire

peut invoquer les services d'un second acteur gestionnaire par un nouveau trap. Au bout de la chaîne, on trouve le micro-noyau, gestionnaire des appels système CHORUS.
À chaque fois que l'exécution est ainsi transférée d'un acteur vers un acteur gestionnaire d'appels, il est nécessaire de changer la table de traps utilisée pour résoudre les appels système. Cette architecture est illustrée par la figure III.4. Elle est décrite avec plus de détails dans [Rug94a].

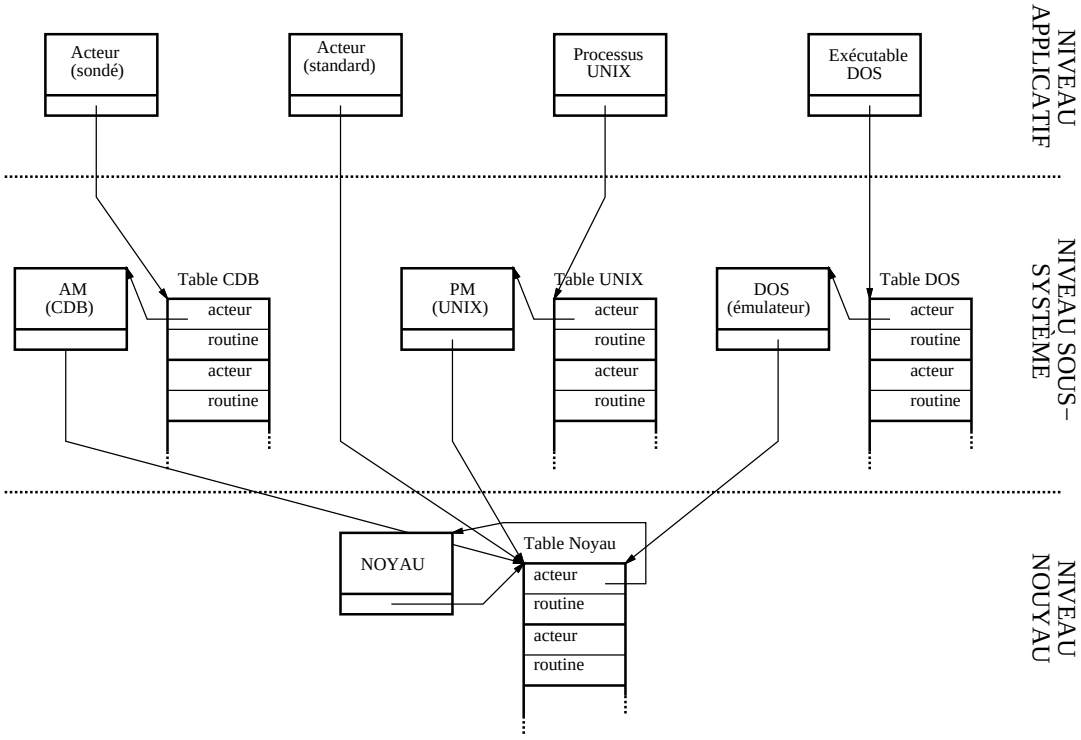


FIG. III.4 - *Structuration des appels système avec les tables de traps*

Nous avons implanté un support pour récupérer, modifier ou remplacer la table de traps utilisée par un acteur. Ce support permet à un moniteur d'intercepter un appel système, d'exécuter un préambule, d'invoquer la véritable routine correspondant à l'appel système, puis d'exécuter un «post-ambule», avant de retourner. On peut ainsi réaliser un service d'interposition complet [Jon92].

III.5 Services divers

En plus du service de notification et du service d'interposition, nous avons dû implanter quelques services additionnels pour supporter CDB et PATOC. Notamment :

– un service pour lister les objets CHORUS présents sur un site et récupérer des informations d'état les concernant (e.g. savoir quelles sont les activités

d'un acteur, puis vérifier lesquelles sont suspendues ou actives).

- un service permettant (**1**) d'espionner le contenu des messages placés dans la queue d'une porte de communication, sans pour autant effectuer un action de réception ; et (**2**) de réorganiser une queue des messages (ce service est utile pour contrôler l'IPC lors d'une ré-exécution).

Ces services sont décrits avec plus de détails dans [HR94].

III.6 Performances

Nous avons effectué deux types de mesures pour évaluer l'influence du support pour la ré-exécution sur les performances du micro-noyau CHORUS : des mesures à un niveau macroscopique (*macro benchmarks*) et des mesures à un niveau microscopique (*micro benchmarks*).

III.6.1 Les tests

GAEDE

GAEDE est un test macroscopique qui calcule le débit du système en mesurant le temps nécessaire à l'exécution en parallèle d'un ensemble de shell scripts UNIX (ces shell scripts simulent l'utilisation du système par des utilisateurs humains). Pour pouvoir exécuter GAEDE, nous avons dû installer le sous-système CHORUS/MiX sur notre versions du micro-noyau. CHORUS/MiX est un système compatible binaire avec UNIXdéveloppé par la société Chorus systèmes (il est disponible en versions 3.2 et 4.0, compatibles respectivement avec UNIX SVR4 r3.2 et r4.0). CHORUS/MiX se compose d'un ensemble de serveurs systèmes (serveur de processus, serveur de fichiers, etc.) implantés sous forme d'acteurs superviseurs s'exécutant sur le micro-noyau.
GAEDE donne une évaluation globale des performances du système. Cette évaluation est relativement précise ($\pm 2.5\%$), comme on peut le constater sur la figure III.5.

KBENCH

KBENCH est un outil développé par Chorus systèmes pour mesurer les performances du micro-noyau à une échelle microscopique. Ce test mesure le coût d'actions spécifiques du micro-noyau (e.g. changement de contexte), à la fois en micro secondes (μs) et en nombre d'instructions.
KBENCH est composé d'un acteur superviseur qui effectue les mesures et d'une application CHORUS/MiX qui réalise l'interface avec l'utilisateur et permet la collecte des résultats. KBENCH répète chaque mesure 20 fois, puis calcule la moyenne et l'écart type des résultats (sur les tableaux, le format employé est : mesure $\pm$ écart type). Pour les mesures en μs, les écart type calculés sont très bons (approximativement $1\mu s$, ce qui correspond à la résolution de l'horloge physique utilisée pour le test. Malheureusement, lorsqu'on exécute GAEDE plusieurs fois consécutives, on obtient des résultats différents, incompatibles

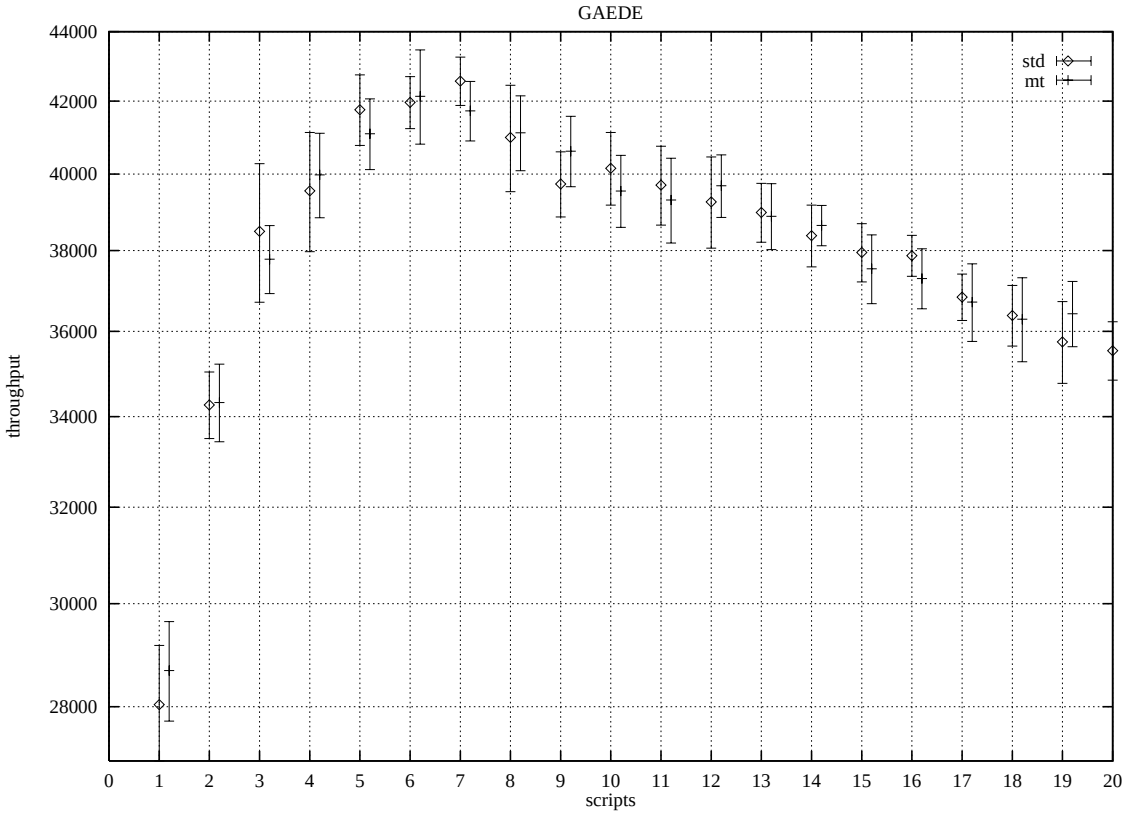


FIG. III.5 - *GAEDE: écarts types (après 20 mesures)*

avec les excellents écarts type calculés pour chaque exécution du test. Cette incohérence est mise en évidence sur le tableau III.9.

	exécution 1	exécution 2	exécution 3	exécution 4
Changement de contexte (μs)	27.5 ± 1.1	24.8 ± 0.8	28.1 ± 1.1	25.9 ± 1.2

TAB. III.9 - *Exécutions consécutives de KBENCH*

Nous attribuons cette instabilité au fait qu'entre deux exécutions successives du test, la «configuration mémoire» du système est modifiée par le lancement d'acteurs CHORUS utilisateur, notamment par le lancement du processus CHORUS/MiX d'interfaçage avec l'acteur superviseur KBENCH. L'influence de la configuration mémoire sur les résultats des tests à échelle microscopique a déjà été soulignée par Bershad et al. [BDF92]. Elle s'explique par des effets plus ou moins obscurs liés à la gestion de l'antémémoire. Durant une exécution de KBENCH, la configuration mémoire reste inchangée et les écarts types sur les mesures sont donc faibles. Entre deux exécutions consécutives de KBENCH, la configuration mémoire est modifiée et les écarts types sont plus importants.
Dans le cadre de nos tests, il ne nous est pas possible d'éviter cette instabilité. D'une part, il nous faudrait garantir que pour chaque série de mesures, le lancement des différents processus MiX destinés à récolter les résultats de KBENCH a eu une influence identique sur la configuration mémoire – cela n'a rien d'évident. D'autre part, nous comparons différentes version du micro-noyau, qui présentent par définition des configurations mémoires différentes.
Pour obtenir malgré tout des résultats relativement stables, nous avons adopté la stratégie suivante. Nous exécutons KBENCH 10 fois consécutivement, puis nous retenons uniquement la meilleure valeur pour chaque type de mesure. Notre intuition est que la meilleure valeur – qui correspond à la configuration mémoire la plus favorable – a une chance d'être une valeur stable (ce que confirment les mesures). Les tableaux que nous présentons un peu plus loin (e.g. le tableau III.10) ont étés obtenus avec cette stratégie. Attention, sur ces tableaux, les écarts type indiqués correspondent à la meilleure des 10 valeurs. Ils sont calculés à partir des 20 mesures prises durant la meilleure exécution de KBENCH, et non à partir de l'ensemble des mesures faites pendant les 10 exécutions du test.
Pour les mesures en nombre d'instruction, les résultats obtenus avec KBENCH sont beaucoup plus stables. Pour la plupart des mesures, le coût en instruction est constant. Pour certaines mesures (émission de message par exemple), le coût d'instructions est légèrement variable (ce qui se traduit par un écart type non nul). Ces légères variations s'expliquent si l'on réalise que les chemins d'exécution dans le noyau ne sont pas toujours déterministes – par exemple dans le cas du parcours d'une table de hachage. Quoi qu'il en soit, la mesure du nombre d'instructions reste la plus significative.
Pour finir, nous voulons mentionner deux fonctions que nous avons ajouté à KBENCH et qui peuvent présenter un intérêt pour l'étude et l'optimisation des performances du noyau. La première fonction permet d'enregistrer le chemin d'exécution correspondant à une action du micro-noyau, puis de le désassembler. La deuxième fonction permet de chronométrer – instruction par instruction –

le temps nécessaire au parcours d'un chemin d'exécution. Ces fonctions sont illustrées par la figure III.6 qui décrit le chemin d'exécution de l'appel noyau `actorSelf()`.

0.0 $\pm$0.7 μs	ScActorSelf	pushl%edi
-0.1 $\pm$0.7 μs	ScActorSelf+0x1	pushl%esi
-0.3 $\pm$0.6 μs	ScActorSelf+0x2	pushl%ebx
0.1 $\pm$0.7 μs	ScActorSelf+0x3	movl0xf0001024,%edi
-0.3 $\pm$0.7 μs	ScActorSelf+0x9	movl0x000000f8 (%edi),%ebx
-0.2 $\pm$0.8 μs	ScActorSelf+0xf	movl0x10 (%esp,1),%eax
-0.3 $\pm$0.8 μs	ScActorSelf+0x13	leal0x000000ac (%ebx),%edx
-0.3 $\pm$0.7 μs	ScActorSelf+0x19	movl (%edx),%ecx
0.6 $\pm$0.9 μs	ScActorSelf+0x1b	movl%ecx, (%eax)
0.4 $\pm$0.7 μs	ScActorSelf+0x1d	movl0x04 (%edx),%ecx
0.7 $\pm$0.6 μs	ScActorSelf+0x20	movl%ecx,0x04 (%eax)
1.0 $\pm$0.7 μs	ScActorSelf+0x23	movl0xf0001024,%edi
0.5 $\pm$1.2 μs	ScActorSelf+0x29	movl0x000000f8 (%edi),%esi
0.5 $\pm$0.6 μs	ScActorSelf+0x2f	leal0x08 (%eax),%eax
0.8 $\pm$1.3 μs	ScActorSelf+0x32	leal0x000000cc (%esi),%edx
0.8 $\pm$1.0 μs	ScActorSelf+0x38	movl (%edx),%ecx
1.5 $\pm$0.7 μs	ScActorSelf+0x3a	movl%ecx, (%eax)
1.6 $\pm$0.7 μs	ScActorSelf+0x3c	movl0x04 (%edx),%ecx
1.9 $\pm$0.8 μs	ScActorSelf+0x3f	movl%ecx,0x04 (%eax)
1.9 $\pm$0.6 μs	ScActorSelf+0x42	xorl%eax,%eax
1.8 $\pm$0.6 μs	ScActorSelf+0x44	popl%ebx
2.0 $\pm$0.7 μs	ScActorSelf+0x45	popl%esi
2.3 $\pm$0.6 μs	ScActorSelf+0x46	popl%edi
2.3 $\pm$0.7 μs	ScActorSelf+0x47	ret

FIG. III.6 - *Chemin d'exécution chronométré de* `actorSelf()`

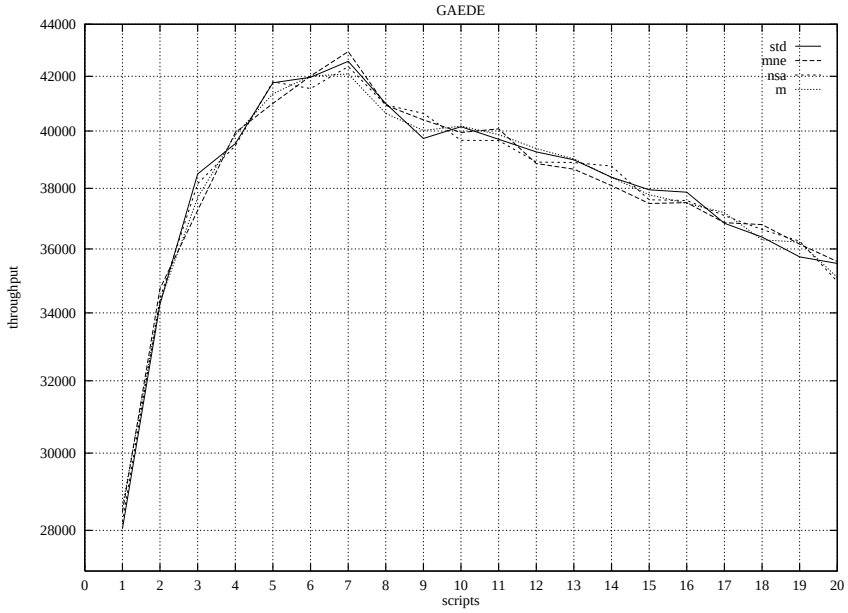


FIG. III.7 - *Coût de la notification (non activée) en unités GAEDE*

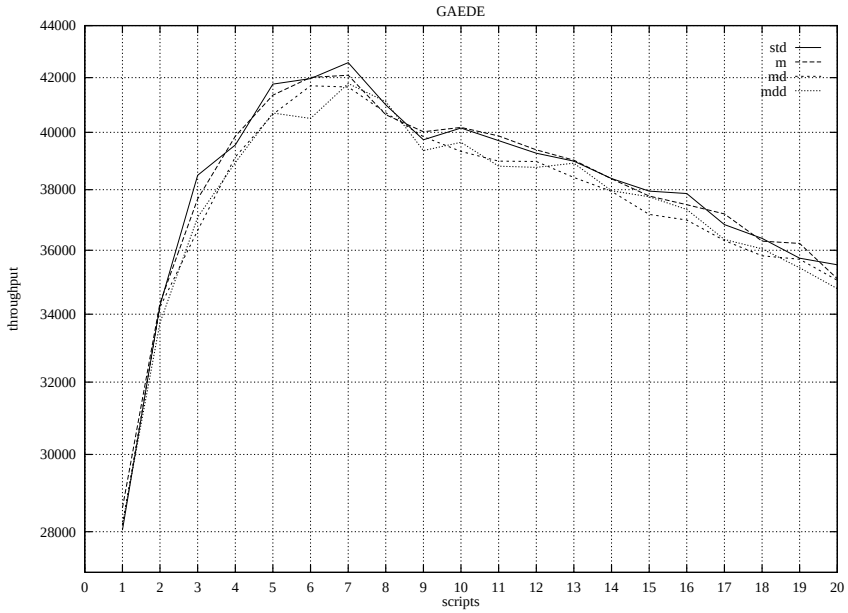


FIG. III.8 - *Coût de la notification (activée) en unités GAEDE*

	std	mne	nsa	m
Appel `threadSelf()`	2.1 ± 0.5	1.2 ± 0.4	1.3 ± 0.5	1.1 ± 0.6
Appel `actorSelf()`	3.1 ± 0.6	2.3 ± 0.6	4.3 ± 0.4	2.2 ± 0.6
Changement de contexte	**24.8 ± 0.8**	23.7 ± 0.6	22.9 ± 1.0	**27.9 ± 0.7**
Routine d'interruption (entrée)	**14.1 ± 0.4**	13.9 ± 0.4	14.4 ± 0.6	**15.9 ± 0.0**
Routine d'interruption (sortie)	**11.6 ± 0.4**	11.7 ± 0.4	12.4 ± 0.6	**12.3 ± 0.5**
Routine de trap (entrée)	4.8 ± 0.4	4.2 ± 0.3	4.2 ± 0.7	3.5 ± 0.4
Routine de trap (sortie)	2.0 ± 0.3	2.0 ± 0.4	1.9 ± 0.3	1.8 ± 0.3
Création d'acteur	76.1 ± 0.7	73.7 ± 0.3	77.8 ± 1.1	82.7 ± 0.6
Création d'activité	91.0 ± 0.5	85.4 ± 0.5	93.9 ± 0.8	91.6 ± 0.6
Destruction d'activité	108.2 ± 1.0	107.0 ± 0.4	111.2 ± 0.9	113.1 ± 1.0
RPC léger (4KO)	39.3 ± 0.5	36.4 ± 0.4	37.6 ± 0.7	35.1 ± 0.9
Émission de message (4KO)	443.3 ± 1.3	467.0 ± 3.8	440.1 ± 0.9	448.2 ± 1.0
Réception de message (4KO)	455.5 ± 1.7	456.3 ± 1.3	454.8 ± 0.8	453.3 ± 1.0

TAB. III.10 - *Coût de la notification (non activée) en micro secondes*

	std	mne	nsa	m
Appel `threadSelf()`	6 ± 0	6 ± 0	6 ± 0	6 ± 0
Appel `actorSelf()`	30 ± 0	30 ± 0	30 ± 0	30 ± 0
Changement de contexte	167 ± 0	167 ± 0	173 ± 0	173 ± 0
Création d'acteur	574 ± 0	574 ± 0	583 ± 0	590 ± 0
Création d'activité	639 ± 0	638 ± 2	653 ± 0	653 ± 0
Destruction d'activité	803 ± 0	803 ± 0	828 ± 0	829 ± 0
RPC léger (4KO)	360 ± 4	360 ± 4	364 ± 4	364 ± 4
Émission de message (4KO)	2924 ± 287	2924 ± 287	2927 ± 287	2927 ± 287
Réception de message (4KO)	2527 ± 0	2527 ± 0	2530 ± 0	2530 ± 0

TAB. III.11 - *Coût de la notification (non activée) en nombre d'instructions*

	std	m	md	mdd
Appel `threadSelf()`	2.1 ± 0.5	1.1 ± 0.6	1.1 ± 0.3‘	2.0 ± 0.4
Appel `actorSelf()`	3.1 ± 0.6	2.2 ± 0.6	3.4 ± 0.6	3.2 ± 0.6
Changement de contexte	24.8 ± 0.8	27.9 ± 0.7	28.3 ± 1.0	33.2 ± 1.3
Routine d'interruption (entrée)	14.1 ± 0.4	15.9 ± 0.0	21.4 ± 0.5	18.2 ± 0.8
Routine d'interruption (sortie)	11.6 ± 0.4	12.3 ± 0.5	15.3 ± 0.6	15.5 ± 0.3
Routine de trap (entrée)	4.8 ± 0.4	3.5 ± 0.4	5.9 ± 0.7	4.4 ± 0.3
Routine de trap (sortie)	2.0 ± 0.3	1.8 ± 0.3	2.0 ± 0.3	2.0 ± 0.3
Création d'acteur	**76.1 ± 0.7**	82.7 ± 0.6	88.0 ± 0.6	**164.3 ± 4.0**
Création d'activité	**91.0 ± 0.5**	91.6 ± 0.6	101.4 ± 0.4	**119.6 ± 0.9**
Destruction d'activité	108.2 ± 1.0	113.1 ± 1.0	127.3 ± 1.0	143.9 ± 1.0
RPC léger (4KO)	**39.3 ± 0.5**	35.1 ± 0.9	**110.2 ± 1.0**	**112.0 ± 0.8**
Émission de message (4KO)	443.3 ± 1.3	448.2 ± 1.0	462.7 ± 1.1	466.7 ± 1.0
Réception de message (4KO)	455.5 ± 1.7	453.3 ± 1.0	493.6 ± 1.3	510.6 ± 1.0

TAB. III.12 - *Coût de la notification (activée) en micro secondes*

	std	m	md	mdd
Appel `threadSelf()`	6 ± 0	6 ± 0	6 ± 0	6 ± 0
Appel `actorSelf()`	30 ± 0	30 ± 0	30 ± 0	30 ± 0
Changement de contexte	167 ± 0	173 ± 0	197 ± 0	221 ± 0
Création d'acteur	**574 ± 0**	590 ± 0	637 ± 0	**1330 ± 22**
Création d'activité	**639 ± 0**	653 ± 0	720 ± 0	**819 ± 8**
Destruction d'activité	803 ± 0	829 ± 0	916 ± 0	1005 ± 0
RPC léger (4KO)	**360 ± 4**	364 ± 4	**814 ± 1**	**843 ± 0**
Émission de message (4KO)	2924 ± 287	2927 ± 287	2991 ± 288	2709 ± 0
Réception de message (4KO)	2527 ± 0	2530 ± 0	2584 ± 0	2596 ± 0

TAB. III.13 - *Coût de la notification (activée) en nombre instructions*

III.6.2 Les résultats

Nous avons effectué les mesures sur un ordinateur mono-processeur compatible IBM PC. La configuration matérielle est décrite par le tableau III.14.

processeur	i486DX cadencé à 33 MHz
mémoire vive	16 MO
antémémoire (*cache memory*)	256 KO

TAB. III.14 - *Configuration matérielle*

Nous avons comparé les performances de différentes configurations du micro-noyau afin de mettre en évidence l'influence du service de redirection des appels système et des différentes modalités du service de notification d'événements. Chaque configuration du micro-noyau est représentée par une mnémonique.
Notre configuration de référence, représentée par la mnémonique `std` (pour *STanDard*) est une configuration standard du noyau CHORUS :

- `std` : configuration standard maximale du noyau CHORUS version v3 r4, comprenant une version étendue du débogueur noyau, l'intégralité du support pour l'IPC et l'intégralité du support pour la mémoire virtuelle.

Influence du service de notification d'événements

La première série de mesures évalue l'influence du service de notification d'événements sur les performances du système, lorsque ce service n'est pas activé. Dans ce but, nous avons comparé `std` avec les configurations suivantes :

- `m` (pour *Monitoring*) : par rapport à `std`, cette configuration inclut le code noyau additionnel correspondant au service de notification d'événements et aux appels systèmes permettant d'obtenir les informations d'état relatives aux objets CHORUS. Le code de notification est activé, mais en revanche aucun objet CHORUS n'est sondé : le noyau n'invoque donc jamais aucune méthode de notification d'événement.

- `mne` (pour *Monitoring Not Enabled*) : cette configuration est identique à `m`, mis à part que nous avons désactivé à la compilation le code de notification d'événements. `mne` est censée être un peu plus performante que `m`, car le noyau n'effectue plus de test pour savoir si un objet est sondé.

- `nsa` (pour *Not Sites not Actors*) : identique à `mne`, mais à la compilation, nous avons seulement désactivé la notification des événements relatifs aux objets site et acteur (c'est à dire principalement les événements relatifs aux interruptions et aux «time-outs»).

Les résultats de GAEDE sont donnés par la figure III.7. Les performances des configurations `std`, `mne`, `nsa` et `m` sont équivalentes.
Les résultats de KBENCH(μs) sont donnés par le tableau III.10. Les performances des configurations `std`, `mne` et `nsa` sont équivalentes. Les performances

de `m` sont légèrement inférieures. En particulier, par rapport à `std`, `m` souffre d'une dégradation de 10% pour les tests relatifs aux routines d'interruption, ce qui est compréhensible car c'est la seule configuration qui active le code de notification des événements d'interruption. `m` souffre également d'une dégradation de 12% pour le test de changement de contexte. Ceci est relativement inexplicable, car le chemin d'exécution du changement de contexte est rigoureusement identique pour les configurations `nsa` et `m` (comme on peut le vérifier sur le tableau III.11).
Les résultats de KBENCH(instr.) sont donnés par le tableau III.11. Les configurations `std` et `mne` ont des performances identiques. `nsa` et `m` exhibent un surcoût de quelques instructions.

La deuxième série de mesures évalue l'influence du service de notification d'événements sur les performances du système, quand ce service est activé. Dans ce but, nous avons comparé `std` avec les configurations suivantes :

- `md` (pour *Monitoring Dummy*) : cette configuration est identique à `m`, mis à part que *tous* les objets CHORUS sont sondés : chaque fois qu'il y a occurrence d'un événement et quelque soit l'objet CHORUS impliqué, le noyau appelle une routine de notification factice, qui retourne immédiatement. La différence principale avec `m` est que le noyau doit à chaque fois préparer les paramètres des routines de notification.

- `mdd` (pour *Monitoring Double Dummy*) : cette configuration est identique à `md`, mais au lieu de retourner immédiatement, les routines de notification traversent complètement la couche EXTMON avant de retourner.

Les résultats de GAEDE sont donnés par la figure III.8. Les performances des configurations `md` et `mdd` sont équivalentes. Ces deux configurations exhibent un surcoût de 1 à 2% environ par rapport à `std` et `m`.
Les résultats de KBENCH(μs) sont donnés par le tableau III.12. En moyenne, le surcoût de la configuration `md` par rapport à `std` est de 20%. Nous notons cependant un pic pour le test de RPC léger : un surcoût de 180%. L'explication en est simple : lors d'un RPC léger non sondé, la requête et la réponse du RPC sont passées directement du serveur au client sans que le noyau les recopie dans un tampon système ; en revanche, lors d'un RPC léger sondé, le noyau effectue une telle recopie afin de rendre le contenu de la requête et de la réponse accessible au client du service de notification. Les performances de la configuration `mdd` sont également en moyenne 20% inférieures à celles de `std`, sauf en ce qui concerne les tests de création/destruction d'objets CHORUS. Pour ces tests les performances sont bien plus mauvaises, ce qui s'explique par le fait que la couche EXTMON doit elle même créer ou détruire les objets EXTMON représentant les objets CHORUS créés ou détruits.
Les résultats de KBENCH(instr) sont donnés par le tableau III.13. En moyenne, la configuration `md` exécute par test 50 instructions de plus que `std` et la configuration `mdd`, 70 instructions de plus. (Naturellement en ce qui concerne `mdd`, le surcoût est beaucoup plus important pour les test de création/destruction).

```
événement:
    ... / traitements
    teste (sondé?)
    sinon_aller cas_non
    ... / notification
cas_non:
    ... / autres traitements
    retour
```

Code produit par le compilateur C

```
événement:
    ... / traitements
    teste (sondé?)
    sioui_aller cas_oui
cas_non:
    ... / autres traitements
    retour

cas_oui:
    ... / notification
    aller cas_non
```

Code optimisé

FIG. III.9 - *Optimisation du code produit par le compilateur C*

Optimisations au niveau assembleur

La troisième série de mesures évalue l'influence d'optimisations locales du code de notification, au niveau du langage machine. Dans ce but, nous avons comparé `std` et `m` avec la configuration suivante:

- `mo` (pour *Monitoring Optimized*) : cette configuration est identique à `m`, mais nous avons modifié le code de notification des événements de la façon décrite par la figure III.9, de façon à optimiser le chemin d'exécution lorsqu'il n'y a pas notification. Comme nous avons dû réaliser cette modification «à la main», nous ne l'avons effectuée que pour un type d'événements (en l'occurrence, les événements correspondant au cadencement des activités).

Les résultats de GAEDE sont donnés par la figure III.10. Les performances des configurations `std`, `m` et `mo` sont équivalentes.
Les résultats de KBENCH sont donnés par le tableau III.15, qui compare les performances du changement de contexte. Les configurations `m` et `mo` utilisent toutes les deux 6 instructions de plus que `std` pour réaliser le changement de contexte. En revanche, le surcoût en micro secondes est beaucoup plus important pour `m` que pour `mo`. Cela s'explique par le fait que `mo` économise une instruction de branchement et interrompt donc une fois de moins le «pipeline» d'instructions du processeur. En généralisant cette optimisation à tous les événements, on pourrait donc vraisemblablement encore rapprocher les performances de `m` de celles de `std`, dans le cas où la notification des événements n'est pas activée.

	`std`	`m`	`mo`
changement de contexte (μs)	24.8 ± 0.8	27.9 ± 0.7	25.1 ± 0.7
changement de contexte (instr.)	167 ± 0	173 ± 0	173 ± 0

TAB. III.15 - *Optimisation du changement de contexte*

Influence du service de redirection des appels système

La dernière série de mesures évalue l'influence du service de redirection des appels système sur les performances du système. Dans ce but, nous avons comparé `std` et `m` avec les configurations suivantes :

- `t` (pour *Trap redirection*) : cette configuration est identique à `std`, mais inclut en plus le support pour la redirection des appels système.
- `mt` (pour *Monitoring + Trap redirection*) : cette configuration est identique à `m`, mais inclut en plus le support pour la redirection des appels système.

Les résultats de GAEDE sont donnés par la figure III.11. `std` et `t` exhibent des performances équivalentes.
Les résultats de KBENCH sont donnés par le tableau III.16. Ce tableau indique le coût de l'appel CHORUS `threadSelf()` et de l'appel UNIX `getpid()` lorsque ces appels sont fait par le biais d'un trap. Nous constatons que pour les deux tests, la redirection coûte 7 instructions. En revanche, en termes de micro-secondes, le surcoût est de 0.5% pour le test `threadSelf()` et de 9% pour le test `getpid()`. Une fois encore, nous attribuons cette incohérence à des problèmes de gestion d'antémémoire.

	std (μs)	t (μs)	std (count)	t (count)
`threadSelf()` (depuis un acteur utilisateur)	149 ± 1	150 ± 1	116 ± 1	123 ± 1
`getpid()` (depuis un processus UNIX)	309 ± 1	336 ± 1	321 ± 1	328 ± 1

TAB. III.16 - *Coût de la redirection des appels systèmes*

Influence sur la taille du noyau

Le tableau III.17 indique la taille de différentes configurations du noyau. Notons que le taille du code ajouté par le service de notification est de 1612 octets, soit 0.6% de la taille du code du noyau ; et la taille du code ajouté par le service de redirection des appels système de 1992 octets, soit 0.8%.

	text	data	bss	total
std	258516	78278	87242	424036
mt, dont :	265868(+7352)	81510(+3232)	106946(+19704)	454324(+30288)
service de notification	(+1612)	(+928)	(+0)	(+2540)
service d'interposition	(+1992)	(+2160)	(+19704)	(+23856)
services divers	(+3748)	(+144)	(+0)	(+3892)
module EXTMON	12156	2924	20292	35372

TAB. III.17 - *Taille des différentes configurations du noyau*

III.7 Conclusion

Dans ce chapitre, nous avons présenté un nouveau support du micro-noyau CHORUS, pour la notification en «temps-réel» d'événements de mise au point,

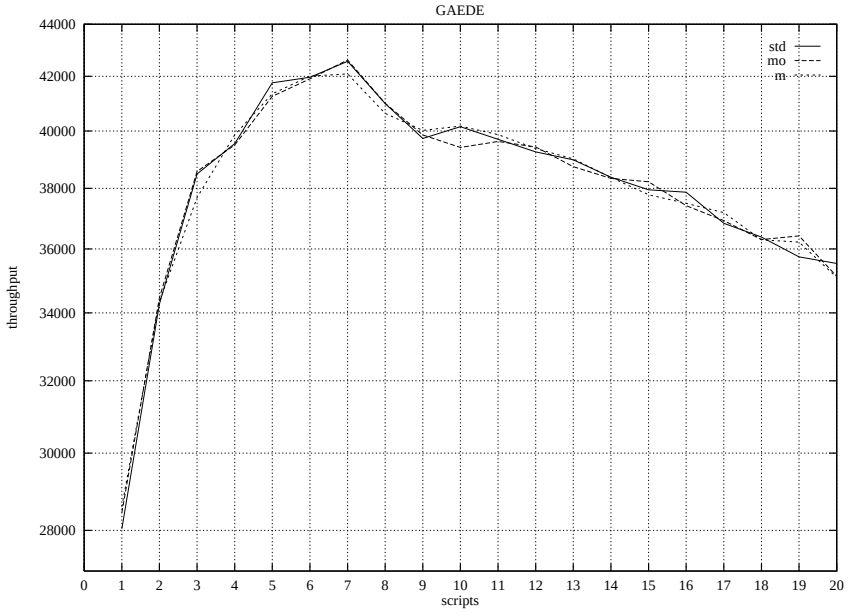


FIG. III.10 - *Optimisations locales*

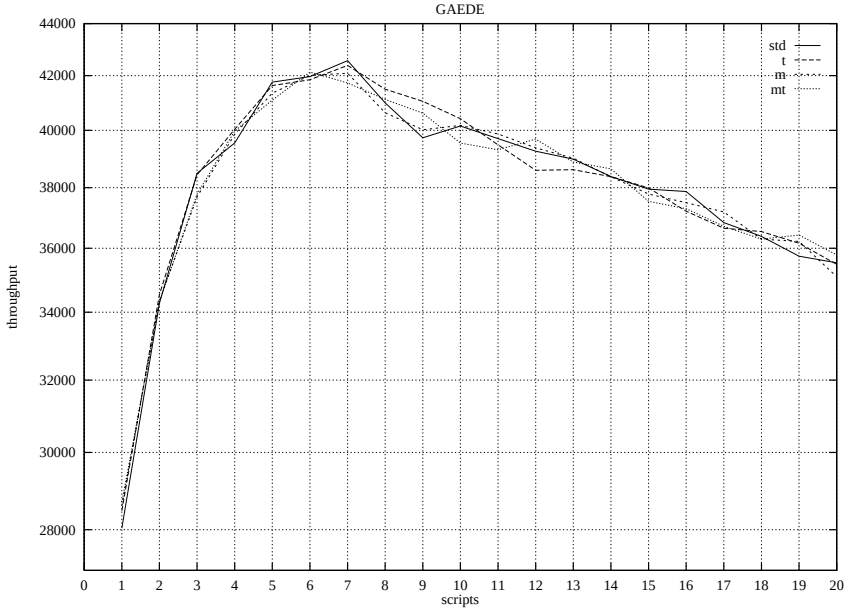


FIG. III.11 - *Coût du service de redirection des appels système*

et la redirection dynamique des appels système.
Le service de notification a une interface orientée objet écrite en C++, qui permet de dériver facilement des supports étendus à partir du support minimal fourni par le micro-noyau. Nous avons actuellement développé un tel support étendu, pour gérer le multiplexage des événements noyau vers plusieurs clients simultanés et pour gérer les événements de nommage symbolique des objets CHORUS.
Les mesures que nous avons effectuées montrent que l'implantation de ces nouveaux services a une influence faible sur les performances du micro-noyau. Pour les applications qui n'y font pas appel, l'impact est négligeable. Pour les applications qui y font appel, le surcoût est évalué à moins de 2% par le test macroscopique GAEDE (dans le cas le plus défavorable où on passe par toutes les couches de notification); et à environ 20% par le test microscopique KBENCH (également dans le cas le plus défavorable). Le surcoût en terme de taille du micro-noyau est d'environ 1.5%.
Ce nouveau support noyau nous a permis d'implanter le service de ré-exécution distribué CDB, qui est décrit au chapitre suivant.

Chapitre IV

Un service de ré-exécution pour CHORUS

IV.1 Introduction

Proposer un environnement de développement de logiciels convivial et puissant est un défi que les constructeurs de systèmes informatiques se doivent de relever. Comme nous l'avons déjà mentionné, cela est aujourd'hui tout particulièrement le cas pour les outils de mise au point. Les systèmes informatiques modernes ont évolué vers le parallélisme et la distribution, et de nouveaux outils de mise au point prenant en compte ces aspects doivent être proposés. Le système d'exploitation distribué à micro-noyau CHORUS [RAA+88, Roz90] supporte actuellement deux outils de mise au point spécifiques : KDB et GDB.

- KDB : le débogueur noyau [Cho91], utilisable uniquement depuis la console, permet de geler l'état du système sur un site et d'interroger ou de modifier les structures du noyau. Il permet notamment de lister et d'exécuter pas à pas le code du noyau. KDB dispose d'une table de symboles pour faciliter la mise au point, mais son interface reste rudimentaire.

- GDB : une version du débogueur de GNU [Sta86], adaptée à CHORUS, permettant de déboguer des processus MiX et des acteurs multi-activités s'exécutant directement au dessus du micro-noyau. GDB est un outil convivial permettant la mise au point au niveau du code source.

En outre, une version du débogueur XRay de Microtec [Mic93] est en cours de portage et permettra une mise au point du micro-noyau de type «hôte – cible». Cependant, CHORUS ne supporte pas encore d'outil pour la mise au point d'applications véritablement distribuées, qui prendrait notamment en compte le problème de non reproductibilité de l'exécution. Dans ce chapitre, nous décrivons un tel outil, CDB, qui permet la mise au point d'applications distribuées sur un réseau de sites CHORUS, et propose notamment un service de ré-exécution. Comme nous avons déjà eu l'occasion de le mentionner, le service de ré-exécution est l'un des outils de base pour la mise au point de programmes parallèles et distribués, et probablement un des plus utiles [LS91], car c'est celui qui

permet de généraliser la méthode de mise au point cyclique à ces programmes habituellement non déterministes. Ce n'est pas un hasard si un grand nombre d'outils de mise au point décrits dans la littérature proposent un tel service de ré-exécution [JBW87, LMC87, SW87, Els88, PL89, TA91, JRS91, HPR92, Leu92, RCM93] (et encore quelques autres, cf. [MH89]).
CDB est lui-même une application distribuée ordinaire s'exécutant sur le noyau CHORUS modifié décrit au chapitre précédent. Rappelons que ce noyau étendu réalise :

- un service permettant la notification de certains événements internes au micro-noyau.
- un service d'interposition, permettant de rediriger dynamiquement les appels systèmes d'une activité.

Le reste du chapitre est organisé comme suit. La section IV.2 est une présentation générale du débogueur CDB, de son interface et des abstractions qu'il définit. L'architecture de CDB est décrite à la section IV.3. À la section IV.4 nous détaillons un composant particulièrement important du débogueur : la «base de données répartie des identificateurs globaux». La section IV.5 présente certains points d'implémentation déterminants, notamment le compteur d'instructions logiciel. Nous donnons quelques résultats de performance et de validation du service de ré-exécution à la section IV.6. La section IV.7 conclut le chapitre.

IV.2 Présentation générale

IV.2.1 Service de ré-exécution

Le service de ré-exécution s'applique à une classe large mais non exhaustive d'applications CHORUS utilisateurs. Les caractéristiques d'une application *ré-exécutable* sont les suivantes :

- Une application ré-exécutable se compose d'un ensemble arbitraire et dynamique d'acteurs CHORUS utilisateurs (cf. III.2), répartis sur un ensemble fixe de sites mono-processeur (nous envisageons le cas des sites multi-processeur à la section IV.7.1).
- Chaque acteur peut comporter un ensemble arbitraire et dynamique de portes de communication.
- Chaque acteur peut comporter un ensemble arbitraire et dynamique d'activités qui sont cadencées de façon préemptive.
- Les activités de l'application peuvent communiquer entre elles et avec les activités de l'environnement de l'application, par le biais du service IPC CHORUS. Elles peuvent invoquer les primitives de communication fiables et non-fiables. De plus, les activités de l'application qui s'exécutent sur un même site peuvent communiquer par le biais des primitives de synchronisation, ou en utilisant la mémoire partagée (cela est notamment le cas pour les activités d'un même acteur).

– En revanche, les activités de l'application ne sont pas autorisées à communiquer par mémoire partagée avec les activités de l'environnement de l'application ou avec d'autres activités de l'application s'exécutant sur un site distant (la possibilité de mémoire partagée répartie est envisagée en section IV.7.1).

Les applications correspondant au profil ci-dessus peuvent être directement prises en compte par le service de ré-exécution. Il n'est pas nécessaire de les recompiler ou de refaire une édition de liens avec des librairies spécifiques (mise à part la question du compteur d'instructions logiciel, cf. IV.5.2.)
Le niveau d'abstraction auquel CDB perçoit l'exécution d'une application ré-exécutable est le niveau utilisateur (par opposition au niveau superviseur). C'est à dire que le service de ré-exécution peut reproduire exactement les chemins d'exécution en espace utilisateur des activités ré-exécutées. En revanche, les chemin d'exécution en espace superviseur (par exemple à l'occasion d'un appel système) ne sont pas observables (i.e. un appel système est considéré comme une action atomique, ou comme une paire d'actions atomiques dans le cas d'appels bloquants).

IV.2.2 Interface utilisateur

Bien que CDB ne possède pas encore d'interface utilisateur graphique, nous avons tenté de le rendre le plus convivial possible. CDB offre à l'utilisateur un shell (ligne de commande) aux fonctions semblables à celles du débogueur GDB de GNU. Le shell conserve l'historique des commandes précédentes (*history browsing*) et effectue automatiquement le complètement des ordres de l'utilisateur (*completion*), aussi bien en ce qui concerne le nom des commandes que leurs arguments.
Le cœur de l'interface utilisateur de CDB est un (mini) interpréteur LISP qui offre à l'utilisateur une certaine puissance et une grande souplesse pour entrer des commandes complexes ou définir des macro-commandes. Cependant, l'utilisateur n'est pas contraint d'utiliser cette syntaxe LISP. Nous avons ajouté un peu du «sucre syntaxique» de sorte que les commandes CDB usuelles ressemblent aux commandes d'un shell ordinaire (cf. la figure IV.1).
Comme nous l'avons indiqué plus haut, les applications que CDB peut prendre en compte se composent d'un ensemble d'acteurs, d'activités et de portes de communications réparties sur un ensemble de sites. Dans le reste du chapitre, nous qualifierons ces entités d'*objets débogués* lorsqu'ils appartiennent à l'application déboguée et d'*objets CHORUS*, lorsqu'ils n'y appartiennent pas.
Le micro-noyau propose un système de nommage des objets CHORUS, à base d'identificateurs uniques (UI) et d'identificateurs locaux (LI) (cf. section III.2). Cependant, ces identificateurs, composés de nombres, restent hermétiques et d'un usage pénible. CDB associe donc des noms symboliques aux objets CHORUS :

– Les noms symboliques peuvent être définis explicitement par l'utilisateur.

– Ils peuvent être spécifiés par l'application déboguée elle-même, si elle s'adresse au serveur de noms EXTMON (présenté en section III.3.3).

– Ils peuvent avoir une valeur par défaut. Par exemple, le nom par défaut d'un acteur, est le nom du fichier exécutable correspondant à l'acteur ; le nom par défaut de l'activité initiale d'un acteur est «`main`».

Ces noms symboliques ne sont pas uniques ; en revanche, l'utilisateur peut les combiner pour former des «chemins d'accès» désignant sans ambiguïté un objet CHORUS donné. Ce moyen de désignation s'apparente à celui utilisé pour les systèmes de fichiers. En effet, CDB organise les objets CHORUS dans une hiérarchie semblable à un système de fichiers. Les sites sont des répertoires d'acteurs ; les acteurs des répertoires d'activités et de portes ; les portes des répertoires de messages. Cette façon de désigner les objets CHORUS s'avère très conviviale à l'usage, d'autant que CDB effectue le complètement automatique des chemins d'accès. Poussant plus loin l'analogie avec les systèmes de fichiers, CDB définit la notion d'objet CHORUS courant, auquel se réfèrent par défaut la plupart des commandes. Les commandes `pwo` (*Print Working Object*), `co` (*Change Object*) et `lo` (*List Objects*) permettent respectivement d'afficher l'objet courant, de changer l'objet courant et de lister les objets rattachés à un objet répertoire donné.
CDB offre un certain nombre de services «traditionnels». Il permet de définir des points d'arrêt, d'effectuer des lectures et écriture dans l'espace mémoire d'un acteur, de récupérer des informations d'état diverses, de désassembler symboliquement le code de l'acteur et de remonter la pile d'appels des activités de l'acteur. En revanche, il ne permet pas de déboguer au niveau du code source. Cette fonction pourrait être implantée dans une future version sous la forme d'une coopération avec le débogueur GDB.
En plus de ces services traditionnels, CDB offre un service de ré-exécution dont l'interface est fondée sur la notion de session de mise au point.

IV.2.3 Sessions de mise au point

Le service de ré-exécution est fondé sur la notion de *session* de mise au point. Une session de mise au point peut être vue comme un objet CHORUS virtuel qui comprend un nombre fixe de sites et définit un réseau CHORUS virtuel avec des propriétés particulières. Il y a deux types de sessions de mise au point : les *sessions d'enregistrement* et les *sessions de ré-exécution.*

– Lorsqu'une application CHORUS est lancée dans une session d'enregistrement, son exécution est automatiquement enregistrée dans un *historique de session.*

– Lorsqu'une application est lancée dans une session de ré-exécution, son comportement est contrôlé de façon à reproduire une exécution préalablement enregistrée. Par exemple, si l'application avait communiqué avec son environnement, ces communications sont simulés par la session de ré-exécution.

Il peut y avoir plusieurs sessions de mise au point actives simultanément dans le système, et si c'est le cas, elles sont totalement indépendantes. Notamment,

`cdb>` *pwo* `/default`	Affiche l'objet courant. (La session spéciale `default` comprend tous les sites connectés.)

`cdb>` *lo*
`chine`
`france`
`japon`
	Liste les objets visibles depuis l'objet courant.
	(Trois sites sont connectés.)

`cdb>` *session ma_session chine france* `/ma_session`	Crée une session avec les sites `chine` et `france`.

`cdb>` *lo /ma_session*
`/ma_session/chine`
`/ma_session/france`
	Liste les sites appartenant à `/ma_session`.

`cdb>` *run site /ma_session/chine ping*	Lance l'acteur `ping` sur le site `/ma_session/chine`.
`cdb>` *co /ma_session/france*	Change l'objet courant.
`cdb>` *run site . pong* ... `cdb>` *flush /ma_session*	Lance l'acteur `pong` sur le site `/ma_session/france`. L'exécution des acteurs est enregistrée par la session. Vide le tampon d'écriture de la session.

`cdb>` *replay ma_session ma_reexecution*
`/ma_reexecution/chine/ping`
`/ma_reexecution/chine/pong`
	Ré-exécute la session dans une nouvelle session.
	Les deux acteurs sont recréés automatiquement (état suspendu).

`cdb>` *lo /*
`/local`
`/default`
`/ma_session`
`/ma_reexecution`
	(L'ensemble des sessions créées.)

`cdb>` *schedule /ma_reexecution all* ...	Reproduit toute l'exécution enregistrée.

FIG. IV.1 - *Session de mise au point et ré-exécution*

il est possible d'avoir plusieurs sessions de ré-exécution de la même exécution actives en même temps ; CDB garantit qu'il n'y aura pas d'interférences entre les différents «clones» – bien que ceux ci utilisent les mêmes identificateurs globaux pour désigner des objets différents.
Du point de vue du nommage, les sessions de mise au point sont partie intégrante de la hiérarchie des objets CHORUS. Elles jouent le rôle de répertoires de sites. En plus des sessions ordinaires décrites plus haut, CDB définit une session spéciale, nommée `default`, qui contient la totalité des sites accessibles au débogueur. La session spéciale `default` ne permet pas d'effectuer l'enregistrement ou la reproduction d'une exécution. Son seul rôle est de permettre à l'utilisateur de nommer les objets CHORUS non encore intégrés à des sessions.
Pour créer une session d'enregistrement, l'utilisateur doit spécifier un ensemble de sites et donner un nom symbolique à la session. CDB ajoute alors un objet session à la hiérarchie des objets CHORUS, et prépare un répertoire de fichiers, dans lequel seront conservés les caractéristiques de la session, ainsi que les historiques d'exécution (pour l'instant, les historiques sont centralisés sur un site unique, qui constitue un goulot d'étranglement. Ceci devra-t-être amélioré dans les prochaines versions). CDB fournit un service de chargement d'acteurs qui permet à l'utilisateur de lancer une application répartie sur l'ensemble des sites de la session d'enregistrement.
La reproduction d'une exécution enregistrée s'effectue dans une section de ré-exécution. Pour ce faire, l'utilisateur spécifie simplement le répertoires associé à l'exécution enregistrée, et donne un nom symbolique à la session de ré-exécution. Ceci est illustré par la figure IV.1.
Comme nous l'avons mentionné plus haut, l'application dont on enregistre ou reproduit l'exécution peut communiquer avec son environnement. En fait, c'est la session de mise au point qui marque la frontière entre l'application et son environnement. Les acteurs lancés à l'intérieur de la session forment ensemble l'application mise au point. Les autres sont considérés comme faisant partie de l'environnement. La session (et donc l'application) peut s'étendre dynamiquement lors de la création de nouveaux objets CHORUS par des activités de la session (e.g. de nouveaux acteurs). Suivant la politique choisie, les objets créés peuvent être intégrés à la session ou bien laissés dans l'environnement.
Durant la phase d'enregistrement, la session produit un historique séquentiel par site (i.e. par site monoprocesseur – sur une machine multi-processeur, il y aurait un historique par processeur.) L'ensemble des historiques séquentiels constitue l'historique de la session. Ces historiques sont des fichiers binaires, mais des outils sont fournis pour en analyser le contenu. Pour l'instant, le seul outil disponible est `ascii`, un petit utilitaire qui traduit de façon symbolique le contenu des historiques. D'autres outils pourrait être portés : outils de détection de propriétés ou de représentation graphique de l'exécution (e.g. PARTAMOS [Sch91] qui trace des diagrammes «espace-temps»).
Durant la phase de ré-exécution, l'utilisateur conserve un certain contrôle sur l'application. Il peut placer des points d'arrêts et inspecter l'état de l'application. Il peut choisir plusieurs modes de ré-exécution : pas à pas (au niveau des instructions machines du processeur), cadencement (d'activité) par cadencement, ou bien sans interruption. Pour les modes pas à pas et cadencement

par cadencement, l'utilisateur peut spécifier une activité particulière parmi l'ensemble des activités dont l'exécution pourrait être continuée (en général, une par site).
Enfin, la session de mise au point serait le lieu indiqué pour implanter un service de point de reprise (un tel service n'est pour l'instant pas fourni par CDB).

IV.3 Architecture générale de CDB

L'architecture générale CDB est classique, avec un moniteur central, un moniteur local par site connecté au système de ré-exécution, et un ensemble de modules de stockage (dans l'implémentation actuelle, les modules de stockage sont tous regroupés au niveau du moniteur central.)
Le moniteur central, également nommé CDB, est un processus MiX composé :

- d'un module d'interface avec l'utilisateur, qui organise le travail des moniteurs locaux,
- d'un service centralisé d'interprétation des historiques, qui pilote les moniteurs locaux lors des phases de ré-exécution.

Les moniteurs locaux, nommés RCDBs, sont des acteurs CHORUS superviseurs. Ils observent et le cas échéant contrôlent le comportement des objets CHORUS présents sur leurs sites respectifs. Ils coopèrent avec les modules de stockage pour transférer les historiques d'exécution sur support stable. Les modules de stockage peuvent résider ou non sur le même site que les moniteurs locaux. Cette architecture est illustrée par la figure IV.2.
Comme le montre la figure IV.3, chaque moniteur local est composé d'un ensemble de gestionnaires que nous allons décrire brièvement :

- Le gestionnaire de données globales : DM (DM pour *global Data Manager*).

  Les DMs des différents moniteurs locaux coopèrent pour former une base de données répartie et répliquée des informations globales relatives aux sessions de mise au point. Ils maintiennent notamment à jour l'ensemble des identificateurs uniques associés aux objets CHORUS appartenant aux différentes sessions de mise au point, ce qui permet à CDB de connaître les «frontières» de chaque session (plus de détails sont donnés en section IV.4).

  Nous envisageons également d'utiliser cette base de données pour rendre l'écriture des historiques d'exécution tolérante aux défaillances de sites (cf. section IV.4).

- Le gestionnaire de temps logique : TM (TM pour *logical Time Manager*).

  On peut considérer le TM comme partie intégrante du DM. Dans l'implémentation actuelle, les TMs des différents moniteurs locaux coopèrent pour implanter une horloge matricielle au sein de chaque session de mise au point. Ces horloges matricielles sont utilisées par le DM pour supprimer les informations «périmées» (cf. section IV.4).

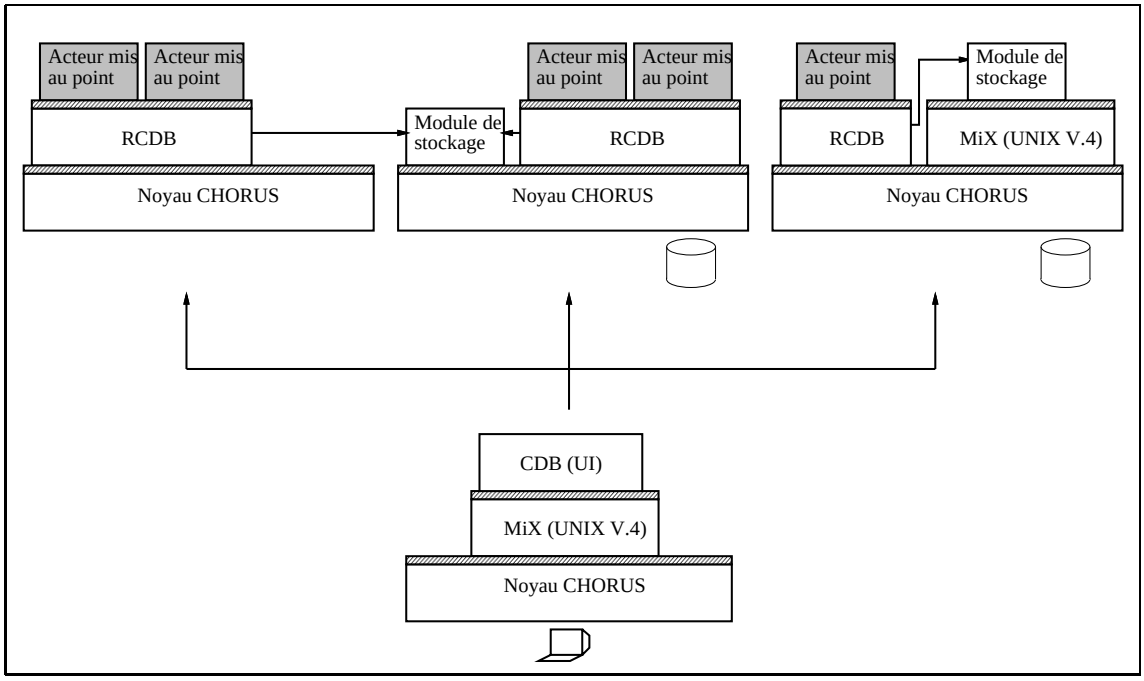


FIG. IV.2 - *Architecture générale de CDB*

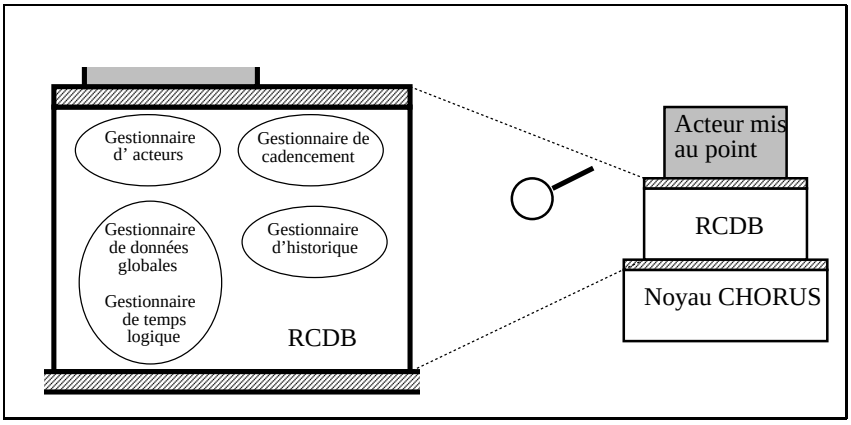


FIG. IV.3 - *Les gestionnaires des modules RCDB*

Cet aspect de CDB sera certainement revu et optimisé dans une future version. En effet, les horloges matricielles ne permettent pas un passage à grande échelle (i.e. sessions de mise au point comprenant de nombreux sites).

– Le gestionnaire de cadencement : SM (SM pour *Scheduling Manager*).

  Le SM est responsable d'enregistrer et de reproduire précisément le cadencement des activités des sessions de mise au point. Il est fondé sur l'utilisation d'un compteur d'instructions (cf. section IV.5.2).

– Le gestionnaire d'acteurs : AM (AM pour *Actor Manager*).

  Le DM interpose son propre code à l'interface des appels système des acteurs appartenant à une session de mise au point [Jon92]. Il procure à ces acteurs l'illusion d'un réseau virtuel CHORUS indépendant (i.e. indépendant des autres sessions de mise au point). En particulier, il rend possible l'activation simultanée de plusieurs sessions de ré-exécution correspondant à la même session d'enregistrement en assurant la correspondance entre les identificateurs (virtuels) connus par les différentes applications «clones» et les identificateurs réels connus par le micro-noyau.

– Le gestionnaire d'historique: LM (LM pour *Log Manager*).

  Le LM est responsable de transférer les historiques d'exécution sur support stable. Pour cela il coopère avec les modules de stockage (cf. section IV.5.3) – et peut-être avec les DM, dans une future version de CDB, afin d'obtenir une écriture des historiques tolérante aux défaillances de sites.

IV.4 La base des données globales

Les gestionnaires de données globales (DMs) des différents modules RCDB coopèrent pour former une base de données répartie des données globales correspondant aux sessions d'exécution. Cette base de donnée enregistre les UIs associés aux différents objets (acteurs et portes) des applications mise au point. Elle joue un double rôle :

– Lors de l'enregistrement.

  Le noyau CHORUS ne fournit pas de support pour identifier de façon unique un message. Par conséquent, il faut estampiller les messages émis par l'application si l'on veut être en mesure d'apparier les réceptions avec les émissions (cela est nécessaire pour la ré-exécution par le contrôle). Cependant, tous les messages ne peuvent pas être estampillés, e.g. les messages destinés à l'environnement de l'application (sur lequel l'outil de mise au point n'a aucun contrôle). Étant donné que CHORUS n'offre pas de support pour estampiller les messages de façon transparente[1], il faut

1. De façon générale, la définition d'un service d'estampillage transparent semble malaisée, surtout si l'on veut donner la possibilité à plusieurs outils de mise au point de poser des

donc être capable de déterminer si un message est destiné à l'application ou à son environnement. Comme la destination d'un message CHORUS est définie par l'UI de la porte réceptrice, le problème revient à savoir si un UI donné appartient ou non à l'application. C'est le premier rôle de la base des données globales

– Lors de la ré-exécution.

  Lors de la phase de ré-exécution, CDB crée des reproductions des objets CHORUS de l'exécution initiale. Malheureusement, ces nouveaux objets ne peuvent pas être totalement identiques aux objets initiaux. En particulier, les identificateurs que le micro-noyau leur attribue sont en général différents des identificateurs initiaux. Dans le cas des UIs par exemple, ils sont même nécessairement différents – par définition, l'UI est un identificateur global unique. Cependant, pour l'application ré-exécutée, ces objets restent identifiés par leurs identificateurs initiaux. En effet, l'état mémoire de ses différents acteurs composants est une reproduction fidèle de l'état correspondant à l'exécution initiale, ce qui implique que toutes les variables ont les mêmes valeurs, en particulier les variables qui mémorisent les identificateurs des objets.

  Il est donc nécessaire de pouvoir traduire les identificateurs (initiaux) connus par l'application en identificateurs réels reconnus par le système au moment de la ré-exécution. Par exemple, dans le cas de la ré-exécution d'une émission de message, il faut pouvoir traduire l'UI de la porte destinataire. C'est le second rôle de la base des données globales.

Pour des versions futures de CDB, nous envisageons d'étendre encore le rôle de la base des données globales. Nous prévoyons en effet de l'utiliser pour enregistrer les queues des historiques d'exécution qui attendent la validation de leur écriture sur support stable (cette technique est utilisée en tolérance des fautes [SBY88, EZ93]). Ainsi, en cas de défaillance (*crash*) d'un ou plusieurs sites, il serait possible de regénérer un ensemble cohérent d'historiques permettant de recouvrer l'état global du système avant la défaillance du (ou des) sites. (Plus précisément, cette technique permet de régénérer l'historique des différents sites jusqu'à l'état global minimum [AHPR91] de l'application associé à l'état des processus restés valides [Rug93].)
De manière remarquable, les rôles de la base des données globales que nous avons décrit plus haut constituent des applications pures de deux problèmes théoriques de bases de données, connus sous les noms de problème du dictionnaire réparti (*replicated dictionary*) et de problème du journal réparti (*replicated log*) [FM82, WB84]. D'ailleurs, nous nous sommes inspirés des algorithmes proposés par Wuu et Bernstein [WB84] pour l'implémentation de nos gestionnaires de données globales.
Prenons l'exemple du dictionnaire réparti. Le problème est le suivant (nous reprenons la définition qu'en donnent Fischer et Michael dans [FM82].) Il s'agit

estampilles différentes sur le même message. En tout état de cause, ce n'est probablement pas un service que l'on veut rendre au niveau du micro-noyau.

de réaliser une base de données contenant un "dictionnaire". Cette base de données est répartie sur un ensemble de sites de façon à ce que le dictionnaire soit toujours disponible, malgré une éventuelle panne de site. A un instant donné, chaque site dispose donc de sa propre vue du dictionnaire, et l'union des vues des différents sites donne l'état du dictionnaire. Maintenant, chaque site peut modifier le contenu du dictionnaire à l'aide de deux opérations : `INSERT`(x) permet d'ajouter le mot x au dictionnaire et `DELETE`(x) permet de retirer le mot x du dictionnaire.

Formellement, le problème consiste à trouver un protocole qui permette d'implanter les opérations `INSERT` et `DELETE` en respectant la contrainte suivante : sur un site donné, le mot x fait partie de la vue du dictionnaire si et seulement si `INSERT`(x) fait partie du passé causal du site et `DELETE`(x) n'en fait pas partie.

Nous notons que le problème du dictionnaire correspond exactement au problème qu'ont à résoudre les gestionnaires de données globales (DMs) de CDB. Plus précisément, les DMs doivent maintenir (entre autres) le dictionnaire des identificateurs uniques des objets CHORUS faisant partie de l'application répartie sous contrôle de CDB. Les opérations `INSERT` et `DELETE` correspondent respectivement à la création d'un nouvel objet CHORUS (qui implique la création d'un nouvel identificateur unique par le noyau) et à la destruction d'un objet CHORUS. Il est effectivement suffisant d'attendre que l'événement de création d'un objet CHORUS soit dans le passé causal d'un DM pour que celui-ci insère l'UI correspondant dans sa vue du dictionnaire. En effet, tant que la création de l'objet n'est pas dans le passé causal du DM, l'UI correspondant ne peut pas être connu de la partie locale de l'application, qui ne peut donc pas invoquer d'opérations mentionnant cet UI.

Pour donner un peu plus de détails, la gestion de la base de données des identificateurs globaux est faite de la façon suivante (on suppose pour plus de clarté qu'il n'y a qu'une seule session d'exécution et que tous les UIs se rapportent à cette session). Chaque DM conserve l'enregistrement des créations et destructions d'UI dont il a eu connaissance. Ces informations sont diffusées par le DM de façon «paresseuse» : elles sont transportées sous forme d'estampille par les messages émis normalement par les activités applicatives contrôlées par le DM. Un algorithme de «ramasse miettes» permet de supprimer de la liste des créations et destructions à diffuser toutes celles dont le DM est sûr que les autres DMs les connaissent déjà.

Pour que chaque DM puisse connaître la liste des événements de création et destructions d'UI dont les autres DMs ont déjà eu connaissance (ou tout au moins une estimation conservative de cette liste), nous utilisons pour l'instant un mécanisme à base d'horloges matricielles, semblable à celui décrit par Wuu et Bernstein [WB84]. L'inconvénient des horloges matricielles est leur coût : elles obligent à estampiller les messages de l'application déboguée avec des étiquettes dont la taille est proportionnelle au carré du nombre de sites. Pour les versions futures de CDB, nous envisageons d'expérimenter avec les algorithmes améliorés décrits au chapitre V de cette thèse, ou bien d'appliquer une technique plus efficace à base d'acquittement site par site, comme celle utilisée par MANETHO [EZ92].

Par rapport à la situation envisagée par Wuu et Bernstein, nous sommes confrontés à un problème supplémentaire: dans le cas d'un message émis à destination de l'environnement de l'application, les DMs n'ont pas la possibilité de diffuser les informations concernant les UIs sous forme d'estampille. Or, par «transitivité», ces messages émis vers l'environnement peuvent être cause d'autres messages à destination de l'application. Chaque DM doit donc s'assurer de la diffusion générale des informations concernant les UIs dont il a connaissance, avant de laisser une activité sous son contrôle émettre un message vers l'environnement de l'application. Ce problème est similaire au problème de validation (*commitment*) rencontré en tolérance aux fautes optimiste, lorsqu'une activité doit émettre d'un message vers l'environnement de l'application tolérante aux fautes [SBY88, EZ93].
Les algorithmes «paresseux» que nous avons utilisés pour l'implémentation de la base des données globales d'une session d'exécution présentent de nombreux attraits :

- Ils tolèrent des situations de défaillance variées : retard ou perte de message, défaillance de site.
- Ils sont relativement peu intrusifs. En particulier, ils ne sont pas bloquants (comme le serait par exemple, un algorithme qui interromprait l'exécution à chaque création d'UI, en attendant que l'ensemble des sites renvoient un acquittement). Il est possible de limiter leur effet au simple estampillage des messages normalement émis par l'application.

Leur faiblesse potentielle est liée à la taille des informations avec lesquelles les messages doivent être estampillés, qui peut dépendre polynomialement du nombre de sites de la session. En pratique, cela ne constitue pas un problème réel pour les applications que vise CDB (quelques sites seulement).

IV.5 Quelques points techniques

Dans cette section, nous détaillons quelques points techniques particulièrement intéressants, et nous montrons comment nous avons tiré parti du support pour la mise au point offert par notre version modifiée du micro-noyau.

IV.5.1 Gestion de mémoire

Le module qui réalise l'interface utilisateur de CDB est implanté au dessus d'un interpréteur LISP. L'interpréteur alloue dynamiquement de la mémoire lorsqu'il a besoin de créer des objets. Un ramasse miettes est activé lorsqu'il n'y a plus d'espace libre. La ramassage des miettes s'effectue selon les deux phases classiques (marquage des cellules active, puis libération des cellules inactives) et peut durer jusqu'à une seconde. Cela n'a pas vraiment d'importance, car l'interface utilisateur de CDB n'est pas soumise à de fortes contraintes temps-réel.
La situation est tout à fait différente pour les modules RCDB. L'exécution du code de ces modules est souvent effectuée dans le contexte d'un upcall du noyau

et il est donc hors de question de mettre en œuvre un mécanisme complexe de ramassage de miettes. Cependant, ces modules allouent et désallouent très fréquemment des objets : objets représentant les objets CHORUS de l'application mise au point (cf. session III.3.2.), listes d'identificateurs uniques, etc. Ils ont donc aussi besoin d'un service de désallocation des objets inutilisés. Pour réaliser un tel service simplement et efficacement, nous avons utilisé un mécanisme de comptage de références. Chaque objet dispose d'un compteur de référence incrémenté quand un pointeur sur l'objet est créé, et décrémenté quand un tel pointeur est détruit. L'objet est désalloué quand le compteur de références devient nul. Bien sûr, il faut s'assurer de l'absence de cycles.
Lorsqu'une activité d'un module RCDB veut accéder à un objet, elle doit également incrémenter le compteur de référence de l'objet de façon à «verrouiller» l'objet en mémoire et à se prémunir contre une désallocation intempestive. Elle décrémente le compteur lorsqu'elle a fini d'accéder à l'objet. Notons que le mécanisme de d'incrémentation/décrémentation des compteurs de référence est mis en œuvre automatiquement par un jeu de constructeurs/destructeurs C++.
Cette technique de verrouillage fonctionne à condition qu'une activité ne puisse pas être détruite alors qu'elle a verrouillé un objet ; en effet, dans un tel cas l'objet concerné ne pourrait plus jamais être désalloué. Malheureusement, il est tout à fait possible qu'une activité soit détruite au moment où elle verrouille un objet. Pour le comprendre, il faut savoir que les activités qui exécutent le code des modules RCDB sont d'une part (1) les activités permanentes de ces modules (jamais détruites), et d'autre part (2) les activités de l'application mise au point, lorsqu'elles effectuent un appel système, après que le service d'interposition du noyau (cf. III.4) les a redirigées vers le code des modules RCDB. Ces dernières activités peuvent être détruites après avoir verrouillé des objets.
Ce problème de destruction d'activités admet deux solutions. La première est d'empêcher la destruction des activités qui exécutent le code RCDB, par une technique de masquage (le micro-noyau CHORUS offre cette possibilité). L'inconvénient de cette technique est que si l'activité en question doit effectuer un appel système potentiellement bloquant, il lui faut nécessairement réactiver (démasquer) au préalable la possibilité de destruction, et par conséquent libérer tous ses verrous. Cette méthode de programmation est relativement pénible à mettre en œuvre.
La deuxième solution, que nous avons choisie pour CDB, est de définir des routines spéciales de gestion de destruction d'activité, qui libèrent automatiquement les verrous encore possédés par une activité au moment de sa destruction. Nous avons implanté ce mécanisme en utilisant les upcalls noyau de notification de destruction d'activité décrits à la section III.3.2. À chaque activité est associé la liste des verrous qu'elle détient. Cette liste est mise à jour automatiquement – par un mécanisme de constructeurs/destructeurs C++ – en même temps que les compteurs de références des objets verrouillés. Au moment où l'activité est détruite, la routine de gestion de la destruction est invoquée par le upcall noyau, et libère les verrous des objets de la liste. L'avantage de cette méthode est qu'elle est complètement transparente au programmeur des modules RCDB. De plus elle est plus efficace, car les objets ne sont déverrouillés que lorsque cela est nécessaire.

`...`	`...`	
`jmp arriere`	`pushfl`	Sauvegarde des registres d'état.
	`incl compteur`	Incrémentation du compteur de branchements.
	`jne ok`	Test à zéro.
	`int $3`	Déclenchement d'une exception si le compteur a bouclé.
	`ok: popfl`	Restauration des registres d'état.
	`jmp arriere`	Restauration des registres d'état.
Code initial	Code instrumenté	

FIG. IV.4 - *Implantation du compteur d'instructions logiciel*

IV.5.2 Compteur d'instructions logiciel

Nous l'avons mentionné un peu plus haut : les activités s'exécutant sur un site CHORUS sont cadencées de façon préemptive. Par conséquent, pour pouvoir enregistrer et reproduire ce cadencement, il est nécessaire que les modules RCDB puissent déterminer avec exactitude le nombre d'instructions (machine) exécutées par une activité entre le moment de son activation et le moment de sa préemption. Cette tâche est rendue difficile par la présence de boucles dans le chemin d'exécution de l'activité (autrement, il suffirait d'utiliser le compteur de programme). Certains processeurs modernes offrent directement un service de comptage d'instruction (cf. sections II.2.1 et II.4.3), mais ce n'est pas le cas du processeur 386DX33 d'Intel, processeur de notre plateforme d'expérimentation. Nous avons donc dû implanter un compteur d'instructions logiciel, inspiré des travaux de Mellor et Crummey [MCL89].

L'idée qui fonde la notion compteur d'instruction logiciel est la suivante : en réalité, on n'a pas besoin de connaître précisément le nombre d'instructions exécutées par une activité ; il suffit de connaître (par exemple) la valeur de son compteur de programme et le nombre de branchements en arrière qu'elle a effectué (au niveau du code machine). Pour déterminer le nombre branchements en arrière effectués par l'activité, il suffit d'instrumenter le code qu'elle exécute (cf. figure IV.4).

Attention, il est important de réaliser que *tout* le code exécuté par l'activité doit être instrumenté, y compris les librairies. Cette instrumentation peut être réalisée automatiquement lors de la compilation. Dans le cas de CDB, elle est réalisée par une règle spéciale de «`makefile`» : par compilation du fichier source C ou C++, un fichier assembleur est créé, au niveau duquel l'instrumentation est réalisée par un programme «`awk`» qui repère les instructions causant potentiellement un branchement. Finalement le fichier assembleur instrumenté est compilé pour produire le code objet désiré.

Mellor et Crummey [MCL89] proposent des implémentations optimisées du compteur d'instruction. Ils suggèrent notamment de réserver un registre du processeur pour conserver la valeur du compteur, plutôt qu'une variable mémoire (comme nous l'avons fait sur la figure IV.4). Ils parviennent ainsi à limiter la dégradation des performance causée par le compteur d'instructions à des valeurs

comprises entre 1% et 10%, suivant le programme instrumenté. Nous n'avons pas exploré cette voie pour plusieurs raisons :

- Nous ne disposions pas d'un compilateur permettant de réserver l'usage de certains registres du processeur à des outils «utilisateur».
- Nous voulions pouvoir appliquer l'instrumentation à des routines écrites en langage machine – qui utilisent potentiellement tous les registres du processeur.
- Nous n'avons pas voulu trop investir dans une voie qui nous paraît être un ersatz imparfait, en comparaison d'un véritable compteur d'instruction matériel. En effet, la valeur du compteur d'instruction, mémorisée en espace utilisateur (que ce soit en mémoire ou dans un registre), peut être modifiée accidentellement par une activité errante. Par conséquent, on ne peut pas avoir de garantie complète que la valeur donnée par le compteur correspond à la réalité.

Pour récupérer la valeur du compteur d'instruction lors d'un changement de contexte, nous avons utilisé le support noyau de notification d'événements décrit en section III.3.2. Au moment du changement de contexte, le noyau effectue l'upcall d'une routine spécifique du module RCDB, qui enregistre la valeur du compteur ainsi que celle du pointeur de programme, puis remet la valeur du compteur à zéro, en préparation au cadencement de la prochaine activité.
Au moment de la ré-exécution, pour reproduire le chemin d'exécution parcouru par l'activité, le module RCDB charge le compteur d'instructions avec la valeur opposée à celle enregistrée. Lors de l'exécution de l'activité, la valeur du compteur est incrémentée jusqu'à atteindre la valeur zéro. À ce moment, une exception est générée et le module RCDB reprend le contrôle. Il installe alors un point d'arrêt au niveau du compteur de programme enregistré, et poursuit l'exécution de l'activité jusqu'à ce point d'arrêt. Ainsi, le chemin d'exécution de l'activité a été fidèlement reproduit.

IV.5.3 Gestion des journaux

Les gestionnaires de journaux de CDB (LM) ne peuvent pas servir les requêtes d'écriture qu'ils reçoivent de façon synchrone. En effet, les requêtes d'écriture que reçoit un LM sont le plus souvent exécutées dans un contexte d'upcall, et il est donc impossible au LM de ré-invoquer le noyau pour demander l'écriture sur fichier. En fait, il n'est même pas sûr qu'une écriture synchrone soit souhaitable, étant donné l'effet la lenteur relative de cette opération – elle génèrerait probablement un effet de sonde important, et aurait une influence néfaste sur les performances du système. Au lieu de cela, les LMs procèdent en deux temps, comme indiqué par Roos et al. [RCM93] : les écritures sont d'abord faites dans un tampon mémoire circulaire avant d'être reportées sur support stable (fichier) par une activité asynchrone. (La mémoire virtuelle de CHORUS permet d'implanter un *véritable* tampon circulaire en «mappant» deux fois consécutivement la même région de mémoire !)

Plus précisément, le fonctionnement des LMs est le suivant. Lorsqu'un client d'un LM veut enregistrer un événement dans le journal, il envoie tout d'abord une requête au LM pour que celui-ci lui alloue une portion du tampon mémoire de taille donnée. Une fois cette portion allouée, le client du LM pourra y écrire directement les informations relatives à l'événement à enregistrer. Notons que l'écriture se fait directement dans le tampon circulaire, il n'est pas nécessaire de préparer au préalable les données à enregistrer dans un buffer intermédiaire. Finalement, une fois l'écriture terminée, le client envoie un acquittement au LM.
L'allocation d'une portion du tampon circulaire par le LM est une opération atomique (effectuée dans un mode du processeur où les interruptions matérielles sont masquées), ce qui permet d'obtenir localement un ordre total sur les événements enregistré dans les journaux. En revanche, l'écriture dans le tampon des informations concernant l'événement à enregistrer n'est pas censé être une opération atomique, et il n'est pas nécessaire de synchroniser les écritures effectuées de façon potentiellement concurremente par les différents clients du LM. Si l'événement b suit l'événement a, il est tout a fait possible que l'écriture des informations concernant b commence – voire se termine – avant que l'écriture des informations concernant a soit terminée. Ce cas se produit d'ailleurs relativement fréquemment : par exemple, si l'écriture de a est interrompue par une interruption matérielle génèrant à son tour un événement b ; ou bien si l'écriture de a est préemptée.
L'activité asynchrone qui transfère le contenu du tampon circulaire sur support stable doit veiller à ne pas transférer les portions du tampon circulaire non encore acquittées. Dans ce but, le LM conserve une liste ordonnée de ces portions, et effectue le transfert de toutes les portions allouées jusqu'à la première portion non acquittée. Notre implémentation du LM garantit de plus que les opérations d'allocation et d'acquitement sont effectuée en temps constant.

IV.5.4 Primitives de communication non fiables

Nous avons rencontré une difficulté particulière pour la ré-exécution des primitives de communication non fiables de CHORUS. Nous ne sommes pas parvenus à imaginer un mécanisme permettant de diriger la ré-exécution de ces primitives par le contrôle. Le problème est le suivant. Considérons la ré-exécution de l'émission d'un message par une primitive non fiable. Dans le cas d'une ré-exécution dirigée par le contrôle, il n'existe aucune copie du message dans les journaux. La seule «copie» est donc celle possédée par RCDB au moment de la ré-exécution de l'émission du message. Notons que localement, RCDB n'a aucun moyen de savoir si le message est parvenu à destination lors de la première exécution, ou bien s'est perdu sur le réseau (à moins d'analyser l'historique complet du site destinataire, ce qui serait trop coûteux).
RCDB est donc confronté au choix suivant :

- Soit supposer que le message c'était perdu, mais ce n'était peut-être pas le cas...

- Soit supposer que le message était arrivé à destination, et en garder une

copie en attendant que son destinataire la «réclame» (la copie pouvant être conservée localement ou au niveau du site destinataire). Malheureusement, si le message initial s'était perdu, la conséquence est la consommation définitive et inutile de resources mémoire, d'où un problème potentiel d'épuisement des resources du système.

Nous avons donc conclu qu'il n'était pas possible de diriger la ré-exécution des primitives IPC non fiables par le contrôle, et nous avons choisi de les diriger par les données. En d'autres termes, pendant la phase d'enregistrement, nous journalisons le contenu complet des messages non fiables au moment de leur réception. Et nous utilisons cette information pour simuler la délivrance du message au moment de la ré-exécution. Notons que si nous n'avions pas un modèle de système strictement asynchrone (par exemple, si le système de communication offrait des garanties supplémentaires comme l'existence un délai au dela duquel un message est certainement perdu), nous aurions pu éviter le recours à la ré-exécution dirigée par les données.

IV.6 Performances et validation

CDB n'est encore qu'un prototype, mais nous l'avons cependant validé à l'aide de petits programmes de test. Nous avons validé le service de ré-exécution sur un réseau de trois PC/AT 386DX33. Les programmes de test étaient les suivants :

- «boucle vide» : une activité qui effectue une boucle vide.
- «appel `threadSelf()`» : une activité qui boucle en effectuant l'appel système `threadSelf()`.
- «fourche - destruction activité mère» : une activité crée une activité fille puis se détruit. L'activité fille crée une autre activité fille puis se détruit, et ainsi de suite.
- «fourche - destruction activité fille» : une activité boucle en créant une activité fille puis en la détruisant.
- «RPC (n KO)» : deux activités dans le même acteur. Elles bouclent en effectuant des appels de procédure à distance fictifs, pour lequels la taille des requêtes et des réponses est n KO.
- «race» : un programme non déterministe comprenant deux activités qui font «la course» pour arriver à franchir la première le 42-ième km d'un marathon.
- «pingpong» : un programme distribué non déterministe qui simule une partie de pingpong entre deux joueurs. «pingpong» peut faire intervenir jusqu'à trois sites: un pour l'affichage, et un pour chaque joueur. La durée d'exécution du programme est très variable suivant les parties.

Nous avons effectué quelques mesures des performances du service d'enregistrement/ré-exécution de CDB en procédant en local sur un seul site. Les résultats son résumés sur les tableaux IV.5 et IV.6.
On constate que le coût lié à la ré-exécution est très variable suivant le programme rejoué. Toutefois, pour des applications ordinaires (comme «race» et «pingpong»), on peut s'attendre à un surcoût en temps de 1 à 2% pour la phase d'enregistrement et de 15 à 50% pour la phase de ré-exécution.
L'étude des performances du service de ré-exécution fait ressortir des invariants:

- la place occupée par l'enregistrement d'un événement dans le fichier historique est comprise entre 12 et 16 octets.
- Le temps passé à enregistrer un événement est compris entre 300 et $1000\mu s$.
- Le temps passé à ré-exécuter un événement est compris entre 2 et $6ms$.

IV.7 Comparaison avec d'autres services de ré-exécution

On trouve dans la littérature un grand nombre de descriptions d'outils de ré-exécution. Dans cette section, nous tentons de comparer ces outils avec CDB. Les comparaisons portent sur trois points: le modèle de système réparti pris en compte par l'outil de ré-exécution, la technique d'implémentation de la ré-exécution, la prise en compte des interactions de l'exécution à reproduire avec son environnement.

IV.7.1 Modèle de système réparti

Instant replay [LMC87] offre un service de ré-exécution pour des systèmes fortement couplés où la communication entre processus a lieu exclusivement par objets partagés[2]. À l'opposé, l'outil Bugnet [JBW87], le débogueur d'Amoeba [Els88], EREBUS [HPR92] et le débogueur des «CAC»'s [RCM93] supposent une architecture à mémoire répartie où les processus communiquent uniquement par message[3]. Ils supposent que le système de communication est fiable et offre des canaux de communication «FIFO» [Els88, HPR92, RCM93].
CDB et Recap [PL89] prennent en compte à la fois la mémoire partagée et la communication par messages (CDB peut même prendre en compte les primitives de communication non fiables). Recap contrôle la communication par mémoire partagée en détectant les accès aux variables partagées. Pour cela, le programme cible est instrumenté à la compilation de façon à ce que la valeur des variables potentiellement partagées soit enregistrée ou reproduite (à la ré-exécution).

2. Cependant la technique utilisée par Instant replay peut être généralisée aux architectures à mémoire répartie où les processus communiquent par message [LSZ90].

3. Dans le cas du débogueur d'Amoeba, les activités d'une même grappe (*cluster*) sont autorisées à communiquer par mémoire partagée car elles partagent l'espace d'adressage de la grappe. Cependant, d'après [Els88], les activités qui appartiennent à la même grappe sont cadencées de façon non préemptive. Il s'agit donc plutôt de coroutines. Le modèle est bien celui de la communication par message uniquement.

Test	Nb de boucles	Exéc. libre (s)	Enregistrement (s)	Ré-exécution (s)	Événements	Historique (octets)
Boucle vide	10^7	7.90	8.10 (+2.5%)	9.00 (+12%)	267	3180
Appel `threadSelf()`	10^6	1.67	1.34 (-20%)	1.43 (-14%)	37	432
fourche – destruction activité mère	10^4	1.70	14.7 (×9)	176 (×100)	30670	488260
fourche – destruction activité fille	10^4	2.18	16.2 (×7)	167 (×80)	50412	725248
RPC (0 KO)	10^3	0.310	1.99 (×6)	24.1 (×80)	8116	133448
RPC (4 KO)	10^3	2.03	5.20 (×2.5)	27.8 (×14)	8240	133492
RPC (40 KO)	10^3	15.5	32.9 (×2)	59.4 (×4)	8722	140648
«race»	NA	1.42	1.44 (+1.5%)	1.70 (+20%)	71	856
«pingpong» - short	NA	NA	16.8	23.9 (+42%)	4028	59224
«pingpong» - medium	NA	NA	27.6	39.8 (+44%)	6516	102392
«pingpong» - long	NA	NA	56.5	80.8 (+43%)	13230	207716

FIG. IV.5 - *Performances de CDB (données brutes)*

Test	Événements par seconde	Historique (octets)		«Coût» d'un événement	
		par seconde	par événement	enregistrement	ré-exécution
Boucle vide	34	400	11.9	$750\mu s$	$4.1ms$
Appel `threadSelf()`	22	260	11.7	-	-
fourche – destruction activité mère	18×10^3	290×10^3	15.9	$420\mu s$	$5.7ms$
fourche – destruction activité fille	23×10^3	330×10^3	14.4	$280\mu s$	$3.3ms$
RPC (0 KO)	26×10^3	430×10^3	16.4	$200\mu s$	$2.9ms$
RPC (4 KO)	4×10^3	66×10^3	16.2	$380\mu s$	$1.7ms$
RPC (40 KO)	0.56×10^3	9×10^3	16.1	$2000\mu s$	$5.0ms$
«race»	50	603	12.1	$280\mu s$	$3.9ms$
«pingpong» - short	240	3.8×10^3	15.7	NA	$1.8ms$
«pingpong» - medium	236	3.7×10^3	15.7	NA	$1.9ms$
«pingpong» - long	234	3.7×10^3	15.7	NA	$1.8ms$

FIG. IV.6 - *Performances de CDB (valeurs calculées)*

CDB contrôle la communication par mémoire partagée en supposant que le site est mono-processeur et en contrôlant précisément le cadencement des activités. Cette technique pourrait même se généraliser au sites multi-processeurs, à condition d'imposer aux processeurs de partager la mémoire uniquement à travers une unité de MMU appropriée qui permettrait de contrôler précisément les échanges se produisant entre les antémémoires (*caches*) des processeurs et la mémoire centrale. (Notons que la technique s'applique également au contrôle de la communication par mémoire partagée distribuée). En procédant ainsi, il serait possible d'être relativement efficace, puisqu'on s'appuie sur le protocole de cohérence d'antémémoire qui est de toute façon activé [MC91].

IV.7.2 Technique d'implémentation

Certains outils de ré-exécution s'appliquent à des programmes écrits dans un langage spécifique (des programmes «ADA» [SW87], des programme «Estelle» dans le cas d'EREBUS [HPR91], des programmes «Guide» dans le cas de THESEE [JRS91] ou des programmes «concurrent ML» [TA91]). L'avantage de ces outils est qu'il est possible d'enregistrer l'exécution sur une machine et de la ré-exécuter sur une autre. Par ailleurs, ils tracent l'exécution du programme cible à un niveau d'abstraction relativement élevé (celui du langage de programmation) et donc avec une granularité d'événement relativement importante [HPR91], ce qui a pour conséquence de rendre la ré-exécution plus facile à implanter. La prise d'instantanés de l'état global du système peut aussi bénéficier d'une granularité plus importante des événements, parce que les états intermédiaires de l'application associés à des états complexes du noyau peuvent alors être évités [LKH92].
En revanche, le débogueur Amoeba et CDB ont une approche de plus bas niveau. Ils s'appliquent à des programmes écrits dans un langage quelconque, mais qui s'exécutent sur un système spécifique (CHORUS ou Amoeba).
Les débogueurs Amoeba et Recap supposent que le programme cible a été lié avec une librairie spéciale d'appels systèmes instrumentés. L'inconvénient de cette façon de procéder est qu'alors une partie du code et des données du débogueur se trouvent dans l'espace d'adressage du programme cible, et peut donc être corrompue par des activités errantes [Els88]. CDB évite ce problème en n'instrumentant pas le programme cible (mise à part la question de l'implémentation du compteur d'instructions logiciel) et en utilisant une technique d'interposition.

IV.7.3 Gestion des interactions avec l'environnement

Le débogueur pour «concurrent ML» et le débogueur «EREBUS» s'appliquent à des programmes qui s'exécutent isolés, sans interaction avec leur environnement. En revanche, le débogueur «THESEE» s'applique à des programmes qui peuvent accéder des objets persistants GUIDE. Ces objets font partie de l'environnement du programme cible, et leur état doit donc être réinitialisé à chaque ré-exécution. Pour ce faire, THESEE crée automatiquement des copies spéciales des objets GUIDE accédés, et les substitue aux objets GUIDE initiaux lors de la

ré-exécution. Le programme cible est ré-exécuté dans un mode spécial *île* (*island mode*) de façon à éviter toute interférence avec les objets GUIDE initiaux (qui font partie de l'environnement). Une restriction importante de cette méthode est qu'il n'est pas possible de ré-exécuter un programme qui communique réellement avec son environnement via un objet partagé [JRS91] (e.g. on ne peut pas ré-exécuter un programme client indépendamment de son programme serveur.) En revanche, les débogueurs «Bugnet», le débogueur des «CAC» et CDB prennent en compte les interactions du programme cible avec son environnement, en simulant l'environnement au moment de la ré-exécution. Par exemple, CDB enregistre le contenu des messages que le programme cible reçoit de l'environnement de façon à pouvoir les recréer au moment de la ré-exécution. Cette technique est appelée *pilotage par les données* (*data driven*). De manière générale, avec les outils de ré-exécution modernes, le ré-exécution des interactions avec l'environnement du programme cible est piloté par les données, et le ré-exécution des interactions entres les activités du programme cible est piloté par le contrôle (*control driven*), dans le but de limiter le volume des fichiers historiques[LMC87, Els88, LSZ90, JRS91, HPR92, RCM93]. Seuls les outils les plus anciens ne proposent que le pilotage par les données [JBW87, PL89].

Chapitre V

Horloges matricielles efficaces

Les horloges matricielles sont un moyen de mesurer le temps «logique» dans un système distribué asynchrone. Elles possèdent des propriétés intéressantes, dont on peut tirer parti pour la réalisation de protocoles de bases de données répliquées, ou de recouvrement après faute. Malheureusement, leur implémentation est coûteuse : elle impose de munir chaque site du système et chaque message échangé d'une estampille dont la taille n'est bornée que par $n \times n$, où n est le nombre de sites du système. Pour cette raison, on considère généralement que les techniques à base d'horloges matricielles ne sont pas applicables quand le nombre de sites est grand.
Pour maintenir la base distribuée et répliquée des identificateurs globaux des objets CHORUS appartenant à une application mise au point, CDB se fonde sur l'utilisation d'une horloge matricielle. Aujourd'hui, cette horloge matricielle est implantée en utilisant un algorithme naïf. Cela ne pose pas de problème car nous n'avons jamais mis en œuvre la ré-exécution avec plus de trois ou quatre sites. Cependant, pour passer à plus grande échelle, il faudra employer des algorithmes moins gourmands en mémoire. Voici donc la motivation de ce chapitre.
Dans une première partie, nous décrivons un algorithme incrémental efficace, pour calculer l'horloge matricielle. Dans le cas favorable où les sites du systèmes sont «bien synchronisés», cet algorithme entraine un estampillage de taille $\mathcal{O}(n)$ seulement. Dans une seconde partie, nous introduisons la notion d'horloge k-matricielle. L'horloge k-matricielle est une approximation de l'horloge matricielle qui entraine un estampillage de taille $\mathcal{O}(kn)$ seulement. Cette taille ne dépend pas des conditions particulières de synchronisation entre les sites. L'horloge k-matricielle peut être utilisée pour implanter des protocoles tolérants aux fautes dans des systèmes distribués à sémantique de défaillance par omission tels que le nombre maximum de défaillances simultanées est borné par $k - 1$.

V.1 Introduction

Comme l'a souligné Lamport dès 1978, pour un système distribué asynchrone, le temps possède une structure logique non équivalente à structure du temps réel

qui pourrait être donné par une horloge «idéale». Un certain nombre d'horloges ont été proposées pour représenter ce temps logique : horloges linéaire [Lam78], intervallaire [DJ92], vectorielle [FM82], matricielle [WB84]. Les applications des horloges logiques sont variées : elles peuvent se substituer de façon transparente à une horloge physique [NT86] ; elles peuvent représenter les relations de causalité entre les événements d'une exécution distribuée [Fid88, Fid91, Mat89] ; elles peuvent aider à découvrir des problèmes d'accès non synchronisés à des variables et des courses entre processus [DS90] ; elles peuvent représenter des formes plus complexes de connaissance globale de l'état du système et aider à la réalisation de protocoles de bases de données distribuées [WB84, KB91] ou de protocoles de recouvrement après défaillance [Rug93].

Les horloges matricielles ont les propriétés les plus intéressantes [WB84, KB91]. Cependant elles sont aussi les plus coûteuses à implanter. Pour un système distribué comprenant n sites, l'algorithme «naïf» de calcul au vol de l'horloge matricielle impose de munir chaque site et chaque message échangé d'une estampille représentant une matrice de $n \times n$ entiers. Quand le nombre de sites est élevé, des algorithmes améliorés moins gourmands en mémoire sont donc nécessaire.

Wuu et Bernstein [WB84] proposent un certain nombre de stratégies pour diminuer la taille des estampilles nécessitées par l'algorithme de calcul de l'horloge matricielle. Ces stratégies ne calculent pas vraiment l'horloge matricielle mais en calculent une approximation. Cependant, en pratique on peut souvent se contenter de l'information véhiculée par l'approximation. Par exemple, les horloges proposées dans [WB84] suffisent pour implanter les protocoles du dictionnaire distribué (*distributed dictionary protocol* [AHU74]) et du journal distribué (*distributed log protocol*).

Ce chapitre apporte (à notre connaissance) deux petites contributions à la théorie des horloges matricielles et a fait l'objet d'une publication [Rug94b, Rug94c]. Dans une première partie, après avoir formellement l'horloge matricielle, nous en proposons un algorithme de calcul exact, incrémental efficace. Cet algorithme est dérivé d'autres travaux réalisés dans le domaine de la tolérance aux fautes (projet MANETHO[EZ92, EZ93]). Il permet de réduire la taille de l'estampillage des sites et des messages à $\mathcal{O}(n)$ lorsque les sites du systèmes sont «bien synchronisés».

Dans une seconde partie, nous introduisons la notion d'horloge k-matricielle. L'horloge k-matricielle est une approximation de l'horloge matricielle qui peut être utilisée pour implanter des mécanismes de tolérance aux défaillances dans des systèmes à sémantique de défaillance par omission [CASD85] tels que le nombre maximum de défaillances simultanées est borné par $k-1$. Nous proposons un algorithme de calcul au vol de l'horloge k-matricielle qui requiert un estampillage des sites et des messages de taille $\mathcal{O}(kn)$. Cette taille est indépendante des conditions particulières de synchronisation entre les sites du système.

Le reste du chapitre est organisé comme suit. Dans la section V.2 nous faisons un bref rappel sur les horloges logiques, leurs propriétés et la façon dont on peut les implanter. Dans la section V.3, nous proposons une taxonomie des approximations de l'horloge matricielle dont nous avons connaissance – cette section

reprend la présentation de [WB84]). La section V.4 présente l'algorithme incrémental de calcul au vol de l'horloge matricielle. La section V.5 présente l'approximation «k-matricielle», décrit ses propriétés et donne des exemples d'application. Par soucis de clarté, nous avons repoussé la plupart des démonstrations mathématiques en annexe.

V.2 Horloges logiques

La notion d'horloge logique a été introduite dans le cadre des systèmes distribués asynchrones. Panangaden et Taylor donnent une bonne définition de ces systèmes [PT88]:

> «Le terme *distribué* signifie que le système est composé d'un ensemble de sites qui communiquent uniquement par échange de messages le long d'un ensemble fixé de canaux de communication. Le terme *asynchrone* signifie que le système ne dispose d'aucune horloge globale, aucune hypothèse n'est faite concernant les vitesses d'exécution relative des différents sites, ni concernant les délais de délivrance des messages par le réseau de communication, et l'émission et la réception d'un message sont deux actions distinctes.»

Il n'est en général pas possible d'ordonner totalement les événements qui se produisent lors de l'exécution d'un système distribué asynchrone. En revanche, ainsi que nous l'avons mentionné en section II.3.1, il est possible de définir un ordre (partiel) de précédence causale entre les événements [Lam78]. Pour mémoire, rappelons qu'un événement e_1 précède causalement un événement e_2 (noté $e_1 \leq e_2$) si et seulement si :

- soit **(1)** e_1 et e_2 se sont produit sur le même site S_i, e_1 avant e_2 (ce que nous notons $e_1 \leq_i e_2$),
- soit **(2)** e_1 est l'émission d'un message et e_2 en est la réception,
- soit encore **(3)** il existe un événement e_3 tel que $e_1 \leq e_3$ et $e_3 \leq e_2$.

Une horloge logique δ associe une date $\delta(x) \in D$ à chaque événement $x \in E$ de l'exécution du système.

$$\begin{array}{c} \delta : E \to D \\ x \mapsto \delta(x) \end{array}$$

L'ensemble D des valeurs que peut donner l'horloge est partiellement ordonné, et toutes les horloges logiques vérifient la condition (CLK) suivante :

$$e_1 < e_2 \implies \delta(e_1) < \delta(e_2) \qquad \text{(CLK)}$$

autrement dit, l'horloge ne «remonte pas le temps» : formellement, elle réalise un morphisme entre l'ensemble partiellement ordonné des événements de l'exécution et l'ensemble partiellement ordonné des dates. La plupart des horloges logiques vérifient en fait la condition plus forte (S-CLK) :

$$e_1 \leq e_2 \iff \delta(e_1) \leq \delta(e_2) \qquad \text{(S-CLK)}$$

autrement dit, l'horloge préserve exactement la structure causale de l'exécution : formellement, elle réalise morphisme *injectif* entre l'ensemble des événements de l'exécution et l'ensemble des dates.
Prenons quelques exemples : l'horloge linéaire de Lamport [Lam78] et l'horloge intervallaire de Diehl et Jard [DJ92] ne vérifient que (CLK), en revanche, l'horloge vectorielle de Fidge [Fid88] et Mattern [Mat89] vérifie (S-CLK).

V.2.1 Horloge linéaire

L'horloge linéaire «δ_{lin}» associe une date entière à chaque événement de l'exécution. Elle est définie comme suit :

$$\begin{array}{c} \delta_{\text{lin}} : E \to \mathbb{N} \\ x \mapsto \text{hauteur}_E(x) \end{array}$$

où $\text{hauteur}_E(x)$ est la hauteur de x dans l'ordre partiel E (des événements de l'exécution), c'est à dire la longueur maximale h d'une suite $e_1 < e_2 < \ldots < e_h < x$ telle que $\forall i, e_i \in E$.
Cette définition est illustrée par la figure V.1. Cette figure représente un diagramme «espace-temps» classique. Pour mémoire, les lignes horizontales représentent les axes temporels associés aux sites et les flèches représentent les messages échangés entre les sites. Les événements sont représentés par des points. À coté de chaque événement, on a indiqué la valeur correspondante de l'horloge.

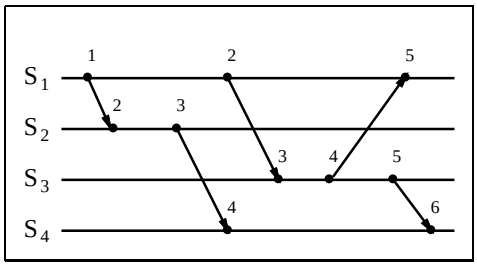


FIG. V.1 - *L'horloge linéaire*

De façon intuitive, $\delta_{\text{lin}}(x)$ mesure le nombre maximum d'événements qui ont été produits en séquence avant l'événement x, quels que soient les sites sur lesquels ces événements se sont produits. Il est donc aisé de voir que δ_{lin} vérifie la condition (CLK).
Rappelons l'algorithme de calcul au vol de l'horloge linéaire. Comme le note Raynal [Ray92], les différents algorithmes de calcul des horloges logiques suivent un schéma unique qui comprend une structure de données et un protocole. La structure de données est une représentation locale de l'horloge logique, dont chaque site est muni et avec laquelle chaque message émis par le site est estampillé. Elle est appelée *vue* (*view*) que le site a de l'horloge logique. Le protocole comprend deux règles notées (INT) et (MSG). La règle (INT) est appliquée

(conceptuellement) juste avant la production d'un événement par un site (émission, réception ou événement interne). Son rôle est d'incrémenter la composante locale de la vue que le site a de l'horloge logique. La règle (MSG) est appliquée (conceptuellement) juste avant la réception d'un message par un site. Son rôle est de mettre à jour la vue du site receveur en la combinant avec la vue avec laquelle le site émetteur a estampillé le message.
Dans le cas de l'algorithme pour l'horloge linéaire, la vue du site S_i consiste simplement en un entier naturel L_i, initialement nul. Les règles (INT-L) et (MSG-L) sont les suivantes :

INT-L : Avant la production d'un événement par S_i :

$$L_i \leftarrow L_i + 1$$

MSG-L : Avant la réception du message (m, L) [1] par S_i :

$$L_i \leftarrow \max(L_i, L)$$

V.2.2 Horloge vectorielle

L'horloge vectorielle «δ_{vec}» associe un vecteur de n entiers à chaque événement de l'exécution, où n est le nombre de sites du système. Elle est définie comme suit :

$$\delta_{\text{vec}} : E \to \mathbb{N}^n$$
$$x \mapsto (\text{hauteur}_{E_i}(x))_{i \in \{1,\ldots,n\}}$$

où $\text{hauteur}_{E_i}(x)$ est la hauteur de x dans l'ensemble E_i des événements qui se sont produits sur le site S_i, c'est à dire la longueur maximale h d'une suite $e_1 < e_2 < \ldots < e_h < x$ telle que $\forall k, e_k \in E_i$.
Cette définition est illustrée par la figure V.2.

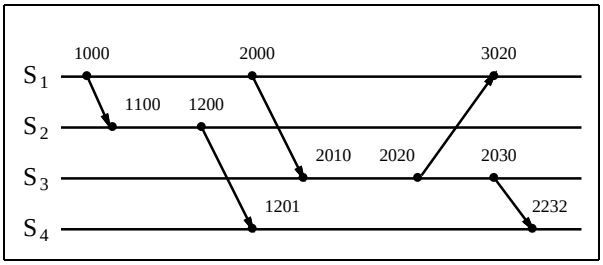


FIG. V.2 - *L'horloge vectorielle*

De façon intuitive, si x est un événement produit par le site S_i, alors la j-ième composante de $\delta_{\text{vec}}(x)$ représente la vue que S_i a de l'avancement du temps local à S_j. L'horloge vectorielle vérifie la condition (S-CLK).
Les horloges vectorielles ont été introduites indépendamment par plusieurs auteurs. En 1982, Fischer et Michael [FM82] ont utilisé un système d'horloges

1. (m, L) est le message reçu, estampillé avec L, la vue que le site émetteur avant de l'horloge logique au moment de l'émission.

vectorielles pour résoudre de façon élégante le problème du dictionnaire distribué [AHU74]. Liskov et Ladin [LL86] ont proposé un système d'horloges vectorielles pour construire des services à haute disponibilité. Finalement, la théorie des horloges vectorielles a été développée indépendamment par Fidge [Fid88, Fid91] et Mattern [Mat89].
Rappelons l'algorithme de calcul au vol de l'horloge vectorielle. Chaque site S_i est muni de sa propre vue V_i de l'horloge vectorielle, et en estampille tous les messages qu'il émet. V_i est un vecteur de n entiers, initialement nuls. Les règles (INT-V) et (MSG-V) sont les suivantes :

INT-V : Avant la production d'un événement par S_i :

$$V_i[i] \leftarrow V_i[i] + 1$$

MSG-V : Avant la réception du message (m, V) par S_i :

$$\forall k, V_i[k] \leftarrow \max(V_i[k], V[k])$$

V.2.3 Horloge matricielle

L'horloge matricielle «δ_{mat}» associe une matrice carrée de $n \times n$ entiers à chaque événement de l'exécution. Elle est définie comme suit :

$$\begin{aligned} \delta_{\text{mat}} : E &\to \mathbb{N}^{n \times n} \\ x &\mapsto \left(\text{card}(\downarrow_{E_j} \downarrow_{E_i} \{x\})\right)_{i,j \in \{1,\ldots,n\}} \end{aligned}$$

où $\downarrow_{E_i} (X)$ est l'ensemble prédécesseur de X dans l'ensemble E_i des événements qui se sont produits sur le site S_i, c'est à dire l'ensemble des événements de E_i qui sont inférieurs à un élément de X [2], et $\text{card}(X)$ représente le nombre d'éléments de X.
Cette définition est illustrée par la figure V.3.

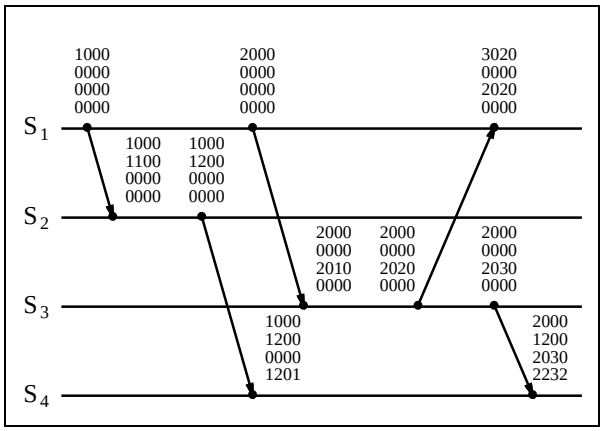


FIG. V.3 - *L'horloge matricielle*

2. Formellement, $x \in \downarrow_B (A) \iff x \in B \wedge \exists y \in A, x \leq y$

Intuitivement, si x est un événement produit sur le site S_i, alors la composante $[j,k]$ de $\delta_{\text{mat}}(x)$ représente la vue que S_i a de la vue que S_j a de l'avancement du temps local à S_k. L'horloge matricielle vérifie la condition (S-CLK).
L'horloge matricielle a été introduite par Wuu et Bernstein [WB84] et par Sarin et Lynch [SL87] comme un moyen pour un site de déterminer si un autre site a eu connaissance d'une information, particulièrement utile pour implanter des protocoles de diffusion.
Rappelons l'algorithme de calcul au vol de l'horloge matricielle. Chaque site S_i est muni de sa propre vue M_i de l'horloge matricielle, et en estampille tous les messages qu'il émet. M_i est une matrice carrée de $n \times n$ entiers, initialement nuls. Les règles (INT-M) et (MSG-M) sont les suivantes.

INT-M : Avant la production d'un événement par S_i :

$$M_i[i,i] \leftarrow M_i[i,i] + 1$$

MSG-M : Avant la réception par S_i du message (m, M) émis par S_j:

$$\forall l, M_i[i,l] \leftarrow \max(M_i[i,l], M[j,l])$$
$$\forall k, l, M_i[k,l] \leftarrow \max(M_i[k,l], M[k,l])$$

V.3 Horloges matricielles approchées

La section V.2.3 décrit un algorithme «naïf» pour calculer l'horloge matricielle au vol. Cet algorithme implique un surcoût de taille $\mathcal{O}(n^2)$ pour chaque site et pour chaque message, i.e. chaque site doit être muni d'une matrice $n \times n$ et chaque message estampillé avec cette même matrice. Dans le cas où le nombre n de sites est élevé, il est nécessaire d'utiliser un algorithme moins gourmand.
Dans [WB84], Wuu et Bernstein décrivent plusieurs stratégies d'optimisation. Ces stratégies ne calculent pas vraiment l'horloge matricielle, mais en calculent une approximation. Cependant, en pratique on peut souvent se contenter de l'information véhiculée par l'approximation. Par exemple, les horloges proposées dans [WB84] suffisent pour réaliser les protocoles du dictionnaire et du journal distribué.
Dans cette section, nous rappelons brièvement les stratégies d'optimisation proposées par Wuu and Bernstein. Pour les illustrations, nous avons considéré l'exécution représentée sur la figure V.4 et nous avons distingué le message représenté par la flèche en pointillé sur la figure. Pour chaque stratégie, nous avons indiqué la partie de l'horloge matricielle qui est effectivement conservée sur le site émetteur du message, et la partie avec laquelle le message est effectivement estampillé. Les résultats sont résumés par la figure V.7.

- Stratégie **NAÏVE**

  C'est la stratégie décrite à la section V.2.3. Le surcoût est $\mathcal{O}(n^2)$ par site et par message.

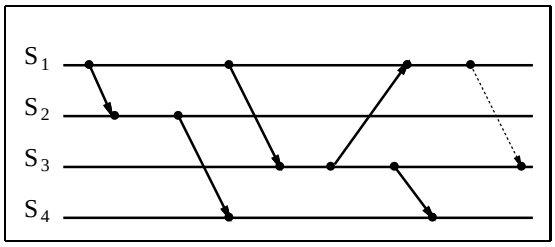


FIG. V.4 - *Exemple d'exécution*

– Stratégie **VECTEUR**

Chaque site est muni d'une matrice $n \times n$ complète, mais les messages émis ne sont estampillés qu'avec la ligne de la matrice qui correspond au site émetteur. Le surcoût pour les messages est donc seulement $\mathcal{O}(n)$. Malheureusement, la stratégie VECTEUR n'assure pas une progression correcte des composantes de la matrice. Ce problème est illustré par la figure V.5: la dérive entre l'horloge matricielle (véritable) et l'horloge obtenue avec la stratégie VECTEUR s'accroît de façon non bornée. Pour résoudre ce problème, il faudrait associer la stratégie VECTEUR à un mécanisme de «commérage» (*gossip* [HHW89]), de sorte que les sites puissent deux à deux mettre régulièrement leurs vues de la matrice à jour.

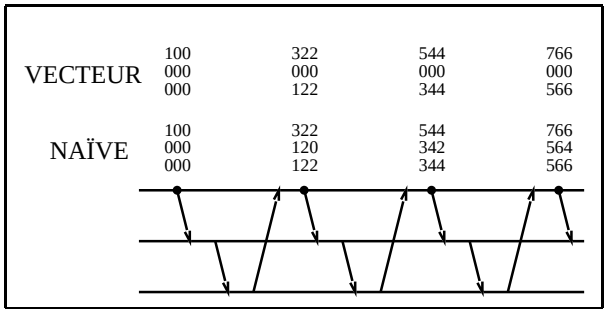


FIG. V.5 - *VECTEUR dérive de façon non bornée*

– stratégie **VOISINS1**

Rappelons que la topologie du réseau de communication est fixée (i.e. la configuration des canaux de communication). La stratégie VOISINS1 est intéressante lorsque le réseau de communication n'est pas complètement connecté (i.e. les sites ne sont pas tous deux à deux connectés). Chaque site conserve seulement sa ligne de la matrice, et une ligne pour chacun de ses voisins (deux sites sont dits *voisins* s'ils sont directement connectés par un canal de communication). Les messages ne sont estampillés qu'avec les lignes qui correspondent aux voisins du site destinataire. Le surcoût est $\mathcal{O}(kn)$ par site et par message, où k est un majorant du nombre de voisins qu'un site peut avoir.

– stratégie **VOISINS2**

Chaque site conserve seulement les composantes de l'horloge matricielle qui correspondent aux canaux appartenant à son aire de communication (i.e. au sous-réseau maximum complètement connecté auquel le site appartient). Autrement dit, le site S_i conserve la composante $[j,k]$ de la matrice si et seulement si les sites S_i, S_j et S_k appartiennent à la même aire de communication. Les messages ne sont estampillés qu'avec les composantes de la matrice qui appartiennent à la même aire que le site destinataire. En ce qui concerne les messages, le surcoût est $\mathcal{O}(k^2)$, où k est un majorant du nombre de voisins qu'un site peut avoir. En ce qui concerne les sites, il est de $\mathcal{O}(k^2)$ par aire de communication à laquelle le site appartient.

La figure V.7 illustre les différentes stratégies d'optimisation de l'horloge matricielle (pour les stratégies VOISINS1 et VOISINS2, nous avons considéré la topologie du réseau de communication donnée par la figure V.6).

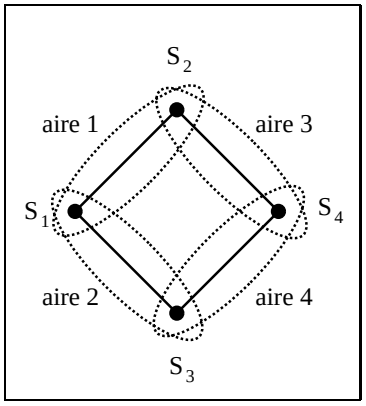


FIG. V.6 - *Exemple de topologie de réseau*

V.4 Horloge matricielle incrémentale

Dans cette section, nous décrivons un algorithme incrémental efficace pour calculer l'horloge matricielle au vol. Cet algorithme est dérivé de l'algorithme du *graphe d'antécédence* d'Elnozahy et Zwaenepoel [EZ92, EZ93]. Nous commençons par un rappel du travail d'Elnozahy et Zwaepenoel, qui est lui-même une amélioration de [SBY88], [JZ87] et [SY85].
Le cadre général est le recouvrement après défaillance. Elnozahy et Zwaenepoel modélisent leur système d'exécution par un ensemble d'unité de recouvrement (URs [SY85]) qui communiquent uniquement par messages à travers un réseau de communication asynchrone. L'exécution d'une UR consiste en une suite d'«états intervallaires» (*state intervals* [SY85]), qui correspondent à des segments d'exécution déterministes. Chaque état intervallaire est initié par un événement non-déterministe (e.g. une réception de message). Le i-ième état intervallaire de l'UR p est noté σ_i^p. Elnozahy et Zwaenepoel définissent le *graphe d'antécédence* (GA) de l'état intervallaire σ_i^p comme l'ensemble des états intervallaires qui précèdent σ_i^p pour la relation de précédence causale.

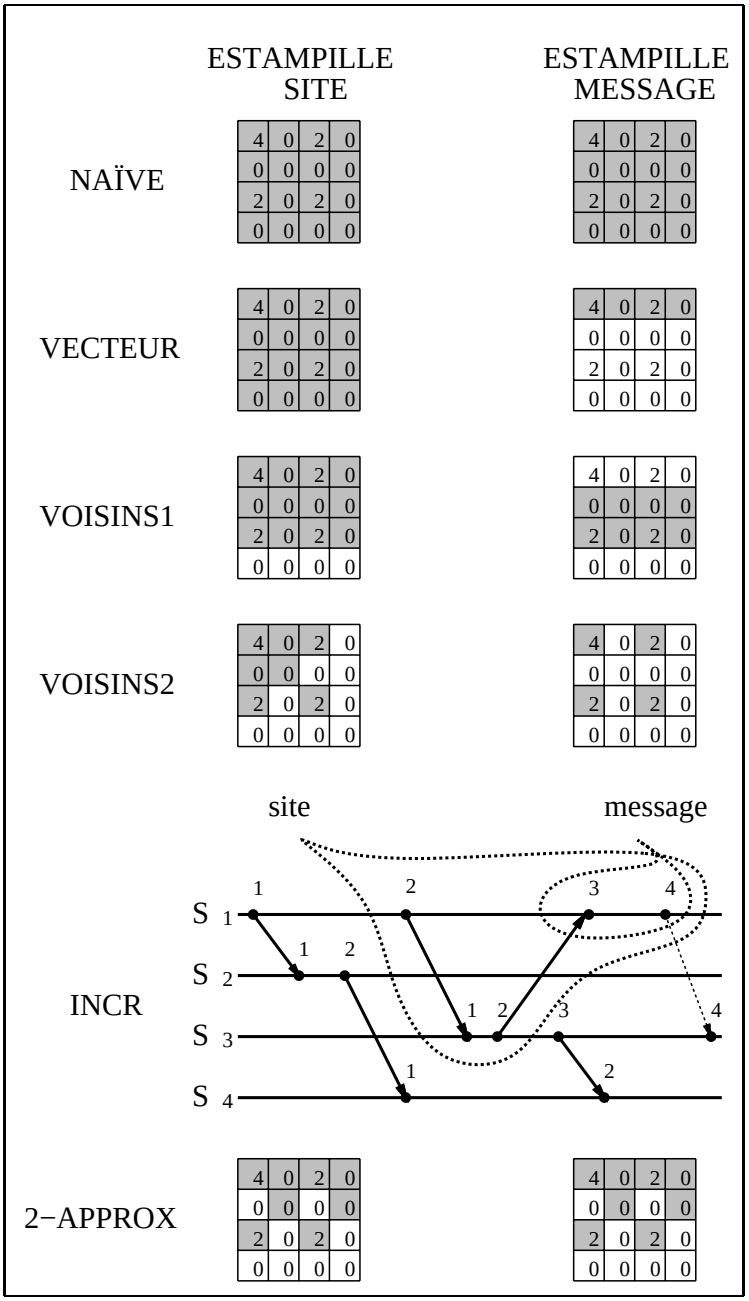


FIG. V.7 - *Stratégies d'optimisation*

Elnozahy et Zwaenepoel proposent un algorithme pour calculer au vol le GA de l'état intervallaire courant de chaque UR. Le principe de l'algorithme est de transporter (conceptuellement) le GA avec les messages : chaque fois qu'une UR émet un message, celui est (conceptuellement) estampillé avec le GA de l'état intervallaire courant de l'UR émettrice. Au moment de la réception du message, un nouvel état intervallaire est initié chez l'UR réceptrice, et le GA de ce nouvel état est construit à partir du GA de l'état intervallaire précédent et du GA transporté par le message.

L'algorithme est dit incrémental parce que seule une (petite) partie du GA est effectivement transportée avec le message : celui-ci n'est estampillé qu'avec le sous-graphe du GA comprenant les états intervallaires dont l'UR émettrice ne sait pas s'ils ont déjà été portés à la connaissance de l'UR réceptrice.

Pour traduire les résultats d'Elnozahy et Zwaenepoel dans notre modèle d'exécution, nous devons interpréter les URs comme des sites et les états intervallaires comme des événements (en associant à chaque état intervallaire l'événement qui l'a initié). Une fois cette traduction faite, nous obtenons un algorithme de calcul au vol du GA de l'événement courant de chaque site (i.e. l'ensemble des événements qui le précèdent causalement, son passé causal, ou encore sa section initiale pour l'ordre causal).

Il est clair que le GA permet de déduire l'horloge matricielle. En effet, il contient tous les événements pouvant intervenir dans le calcul de l'horloge matricielle. Paradoxalement, il peut être moins coûteux (en termes de taille d'estampilles) de calculer le GA plutôt que directement l'horloge matricielle. Il y a deux raisons à cela. Premièrement, le GA peut être calculé incrémentalement de la façon décrite dans [EZ92, EZ93]. Deuxièmement, il n'est pas nécessaire que les sites conservent l'intégralité du GA : un mécanisme de ramasse miettes inspiré de [WB84] ou [SL87] peut être utilisé pour supprimer les événements obsolètes du GA.

V.4.1 L'algorithme

Nous décrivons maintenant l'algorithme incrémental de calcul au vol de l'horloge matricielle. Nous faisons l'hypothèse que chaque événement est estampillé avec **(1)** l'identificateur du site sur lequel il s'est produit et **(2)** son numéro de séquence sur ce site. Nous noterons e_l^i le l-ième événement à s'être produit sur le site S_i. Nous notons également ϵ_i l'événement courant du site S_i (i.e. le dernier événement à s'être produit sur S_i).

Chaque site S_i est muni d'une structure de données notée GA_i et permettant de représenter un graphe dirigé. GA_i est initialement vide.

Tout au long du déroulement de l'algorithme, GA_i va représenter un sous-graphe du GA de l'événement courant du site S_i. Par ailleurs, une contrainte supplémentaire est respectée : quels que soient les sites S_j et S_k, et si l'on considère e, le dernier événement de S_k à précéder causalement le dernier événement de S_j à précéder causalement l'événement courant ϵ_i de S_i, alors deux choses l'une : soit cet événement n'existe pas, soit il est contenu dans GA_i. Formellement, l'algorithme fait respecter la contrainte (GA) suivante :

$$\forall j,k, \max(\downarrow_{E_k}\downarrow_{E_j} \{\epsilon_i\}) \in \mathrm{GA}_i \qquad \text{(GA)}$$

Le but de la contrainte (GA) est de garantir qu'on va pouvoir calculer l'horloge matricielle à partir des événements contenus dans le sous-graphe GA_i. Rappelons la définition de l'horloge matricielle donnée en section V.2.3:

$$\delta_{\mathrm{mat}}(\epsilon_i)[j,k] = \mathrm{card}(\downarrow_{E_k}\downarrow_{E_j} \{\epsilon_i\})$$

En supposant (GA), cette définition se ré-écrit :[3]

$$\delta_{\mathrm{mat}}(\epsilon_i)[j,k] = \text{seq-num}\Big(\max \downarrow_{E_k\cap \mathrm{GA}_i} \Big\{\max \downarrow_{E_j\cap \mathrm{GA}_i} \{\epsilon_i\}\Big\}\Big)$$

Par conséquent, GA_i permet bien de calculer l'horloge matricielle.
Chaque site S_i estampille les messages qu'il émet avec son GA_i. Les règles (INT-I) et (MSG-I) sont les suivantes :

INT-I : Avant la production d'un événement par S_i :

$$\mathrm{GA}_i \leftarrow \mathrm{GA}_i \cup \{e_l^i\}$$

MSG-I : Avant la réception du message (m, GA) par S_i :

$$\mathrm{GA}_i \leftarrow \mathrm{GA}_i \cup \mathrm{GA}$$

Nous introduisons une règle supplémentaire :

GC-I : À tout moment, le site S_i peut supprimer les événements «obsolètes» du sous-graphe GA_i :

$$\forall j,k,\ M_i[j,k] \leftarrow \text{seq-num}\Big(\max \downarrow_{E_k\cap \mathrm{GA}_i} \Big\{\max \downarrow_{E_j\cap \mathrm{GA}_i} \{c\}\Big\}\Big)$$

$$\mathrm{GA}_i \leftarrow \mathrm{GA}_i - \Big\{e_l^j \in \mathrm{GA}_i |\ \forall k, M_i[k,j] > l\Big\}$$

La règle (MSG-I) combine le GA local à S_i avec celui transporté par le message, de la façon décrite dans [EZ92, EZ93]. La règle (GC-I) effectue un ramassage de miettes de façon similaire à [WB84, SL87].

Proposition 1 *Les règles (INT-I), (MSG-I) and (GC-I) préservent effectivement la contrainte (GA).*

La preuve de cette proposition est donnée en annexe A.1.
L'algorithme incrémental est illustré sur la figure V.7, en tant que stratégie «INCR».

3. Dans la formule, l'opérateur seq-num retourne le numéro de séquence de l'événement maximum considéré, ou bien 0 si cet événement n'existe pas.

V.4.2 Coût de l'algorithme incrémental

L'algorithme entraine un coût peu élevé quand les sites du systèmes sont «bien synchronisés». En effet, grâce au mécanisme de ramassage de miettes, la taille des graphes GA_i estampillés reste alors faible : typiquement $\mathcal{O}(n)$ (voir l'exemple un peu plus bas). Un tel surcoût de $\mathcal{O}(n)$ par site et par message est bien meilleur que le surcoût de $\mathcal{O}(n^2)$ obtenu avec l'algorithme naïf. Il y a néanmoins un prix à payer :

- Premièrement, comme chaque site n'est muni que du sous-graphe A_i, il est nécessaire d'effectuer des calculs pour produire les composantes de l'horloge matricielle. Ceci se traduit par un surcoût en terme de temps d'exécution.

- Deuxièmement, si l'exécution du système n'est pas bien synchronisée, la taille des sous-graphes GA_i peut s'accroître de façon non bornée. Pour résoudre ce problème, une solution simple consiste à «synchroniser» l'exécution en utilisant des vagues de messages de «commérage» (*gossip mechanism* [HHW89]) qui visitent régulièrement l'ensemble des sites selon un parcours en anneau ou en arbre de recouvrement (*spanning tree*). Cette technique permet de maintenir la taille des GA_i à une faible valeur au prix de messages de contrôle additionnels.

Nous donnons maintenant un exemple de ce que nous entendons par «exécution bien synchronisée». Considérons un système de n sites où les communications ont lieu selon le principe de l'anneau à jeton (*token ring*). Cette situation est illustrée par la figure V.8 : nous y avons indiqué les GA_i's correspondant aux événements de réception de message. Nous constatons que la taille des GA_i's est $4n + 3$ ($2n + 2$ sommets et $2n + 1$ arêtes – il est nécessaire de prendre en compte les arêtes car un graphe de n sommets peut comprendre jusqu'à n^2 arêtes). Autrement dit, pour cet exemple l'algorithme incrémental engendre effectivement un surcoût de $\mathcal{O}(n)$.

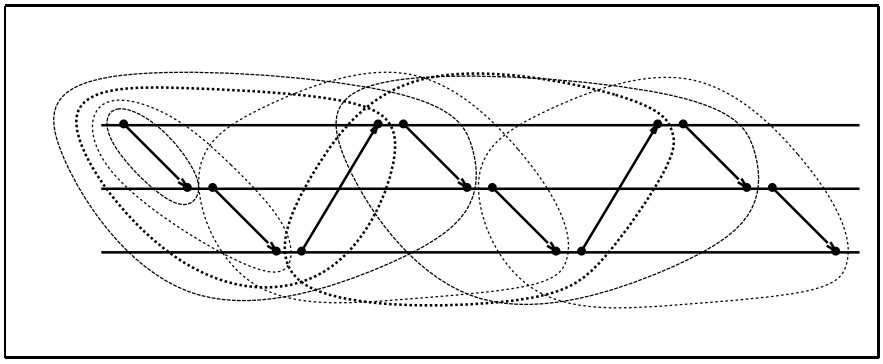


FIG. V.8 - *Taille des GA_i's*

V.5 Horloge k-matricielle

Dans cette section, nous introduisons notre approximation «k-matricielle» de l'horloge matricielle. Nous proposons un algorithme pour calculer l'horloge k-matricielle au vol avec un surcoût de $\mathcal{O}(kn)$ par site et par message. Ce surcoût ne dépend pas de l'état de synchronisation particulier de l'exécution. Nous donnons brièvement deux exemples d'application des horloges k-matricielles.
Nous commençons par introduire la notion de k-approximation d'un vecteur et d'une matrice.

V.5.1 k-approximation d'un vecteur

Définition 10 *L'ensemble des entiers non négatifs est noté* $\mathbb{N}$. *Considérons* $a, b \in \mathbb{N}^n$ *deux vecteurs d'entiers de dimension* n. *Considérons un entier* $k \leq n$. *Nous disons que* b *est une* k*-approximation de* a *(noté* $b \preceq_k a$*) le fait que* b *contienne les* k *plus grandes composantes de* a. *Formellement :*

$$b \preceq_k a \stackrel{\text{def}}{=} \exists I \subseteq \{1,..,n\}, \begin{cases} \text{card}(I) = k & \\ \forall i \notin I, b_i \leq a_i & (\text{I}\leq) \\ \forall i \in I, b_i = a_i & (\text{I}=) \\ \forall i \notin I, \forall j \in I, a_i \leq a_j & (\text{I}) \end{cases}$$

Intuitivement, (I) affirme que I contient les indices des k plus grandes composantes du vecteur a. (I=) affirme que pour chaque indice appartenant à I, les composantes correspondantes de a et b sont égales. (I$\leq$) affirme que pour chaque indice n'appartenant pas à I, la composante correspondante de a est plus grande que celle de b.
Par exemple :

$$\begin{bmatrix} 0 \\ 5 \\ 6 \end{bmatrix} \preceq_2 \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} \quad ; \quad \begin{bmatrix} 0 \\ 5 \\ 6 \end{bmatrix} \preceq_1 \begin{bmatrix} 0 \\ 6 \\ 6 \end{bmatrix} \quad ; \quad \begin{bmatrix} 0 \\ 4 \\ 5 \end{bmatrix} \not\preceq_1 \begin{bmatrix} 1 \\ 5 \\ 6 \end{bmatrix}$$

Alors qu'un vecteur de dimension n a en général une représentation de taille $\mathcal{O}(n)$, il est toujours possible de trouver une k-approximation du vecteur dont la taille soit seulement $\mathcal{O}(k)$ (en conservant uniquement les k composantes les plus grandes du vecteur, et en fixant les autres à zéro).

Proposition 2 *L'ensemble des vecteurs d'entiers de dimension* n *est partiellement ordonné par la relation de* k*-approximation «*$\preceq_k$*» (*$k \leq n$*).*

La preuve de la proposition est donnée en annexe A.2.

Proposition 3 *Notons* max *l'opérateur qui prend composante par composante le maximum de deux vecteurs. Alors* max *et* $\preceq_k$ *«commutent».*

Plus précisément, considérons A et B deux vecteurs d'entiers de dimension n. Soit M, le maximum composante par composante de A et B. Considérons ensuite a et b, des k-approximations respectives de A et B. Soit enfin m, le maximum composante par composante de a et b. Alors m est une k-approximation de M. Formellement :

$$\forall A, B, M, a, b, m \in \mathbb{N}^n, \begin{cases} M = \max(A, B) \\ a \preceq_k A \\ b \preceq_k B \\ m = \max(a, b) \end{cases} \implies m \preceq_k M$$

La proposition est démontrée en annexe A.2. Elle est illustrée par les diagrammes qui suivent.

$$\begin{array}{ccccc} \begin{bmatrix} 0 \\ 5 \\ 6 \end{bmatrix}, & \begin{bmatrix} 2 \\ 7 \\ 3 \end{bmatrix} & \overset{\max}{\rightarrow} & \begin{bmatrix} 2 \\ 7 \\ 6 \end{bmatrix} & \qquad\qquad \begin{bmatrix} 0 \\ 5 \\ 6 \end{bmatrix} \quad \begin{bmatrix} 4 \\ 7 \\ 2 \end{bmatrix} \quad \overset{\max}{\rightarrow} \quad \begin{bmatrix} 4 \\ 7 \\ 6 \end{bmatrix} \\ \downarrow\preceq_2 & \downarrow\preceq_2 & & \downarrow\preceq_2 & \qquad\qquad \downarrow\preceq_2 \quad \downarrow\preceq_2 \qquad \downarrow\preceq_2 \\ \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}, & \begin{bmatrix} 4 \\ 7 \\ 3 \end{bmatrix} & \overset{\max}{\rightarrow} & \begin{bmatrix} 4 \\ 7 \\ 6 \end{bmatrix} & \qquad\qquad \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} \quad \begin{bmatrix} 4 \\ 7 \\ 3 \end{bmatrix} \quad \overset{\max}{\rightarrow} \quad \begin{bmatrix} 4 \\ 7 \\ 6 \end{bmatrix} \end{array}$$

V.5.2 k-approximation d'une matrice

Définition 11 *Soient A et B deux matrices carrées de $n \times n$ entiers. Nous disons que B est une k-approximation de A (noté $B \preceq_k A$) quand les colonnes de B sont des k-approximations des colonnes de A. Formellement :*

$$\forall A, B \in \mathbb{N}^{n \times n},$$
$$B \preceq_k A \overset{\text{def}}{=} \forall 1 \leq j \leq n, B[\star, j] \preceq_k A[\star, j]$$

($A[\star, j]$ représente la j-ième colonne de la matrice A.)
Par exemple:

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 2 & 0 & 3 \end{bmatrix} \preceq_2 \begin{bmatrix} 2 & 0 & 0 \\ 1 & 2 & 0 \\ 2 & 0 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 5 & 3 & 3 \\ 0 & 5 & 0 \\ 5 & 0 & 6 \end{bmatrix} \preceq_2 \begin{bmatrix} 5 & 3 & 3 \\ 4 & 5 & 3 \\ 5 & 3 & 6 \end{bmatrix}$$

Alors qu'une matrice de dimension $n \times n$ a en général un représentation de taille $\mathcal{O}(n^2)$, il est toujours possible de trouver une k-approximation de la matrice de taille seulement $\mathcal{O}(kn)$.

V.5.3 Horloge k-matricielle

Définition 12 *Une horloge k-matricielle (notée δ_k) est une horloge qui associe à chaque événement d'une exécution une k-approximation de l'horloge matricielle (véritable). Formellement :*

$$\delta_k : E \to \mathbb{N}^n$$
$$\forall e \in E,\ \delta_k(e) \preceq_k \delta_{mat}(e)$$

□

Nous donnons maintenant un algorithme qui calcule au vol une horloge k-matricielle.
Chaque site S_i est muni d'une k approximation de matrice, notée A_i et en estampille tous les messages qu'il émet. A_i est initialement nulle. Les règles (INT-A) et (MSG-A) sont les suivantes :

INT-A : Avant la production d'un événement par S_i :

$$A_i[i,i] \leftarrow A_i[i,i] + 1$$

MSG-A : Avant la réception par S_i du message (m, A) [4] émis par S_j :

$$\forall l, A_i[i,l] \leftarrow \max(A_i[i,l], A[j,l])$$
$$\forall k,l, T_i[k,l] \leftarrow \max(A_i[k,l], A[k,l])$$
$$A_i \leftarrow \text{apr}(A_i), \text{ où } \text{apr}(A_i) \preceq_k A_i$$

Le fonctionnement de l'algorithme est illustré par la figure V.9. Le surcoût engendré par l'algorithme est illustré par la figure V.7 (stratégie «2-APPROX»).

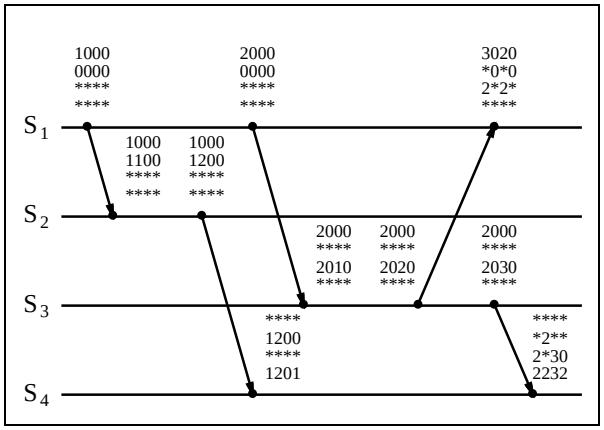


FIG. V.9 - *L'horloge k-matricielle ($k = 2$)*

4. (m, A) est le message reçu, estampillé avec A, la vue que le site émetteur avait de l'horloge k-matricielle au moment de l'émission.

Proposition 4 *L'horloge k-matricielle A produite par l'algorithme ci-dessus est effectivement une k-approximation de l'horloge matricielle M. Formellement, si $M(e)$ et $A(e)$ sont les dates associées à l'événement e respectivement par l'horloge matricielle et par l'algorithme ci-dessus, alors :*

$$\forall e \in E, A(e) \preceq_k M(e)$$

La démonstration de la proposition s'appuie sur les propositions 2 et 3. Elle est donnée en annexe A.2.

V.5.4 k-approximations et conditions d'horloge

Il est possible de définir un ordre partiel sur $\mathbb{N}^{n \times n}$ tel que les horloges k-matricielles vérifient la condition forte (S-CLK). Ceci est expliqué en annexe A.3.

V.5.5 Applications de l'horloge k-matricielle

Rappelons qu'intuitivement :

- l'horloge matricielle M_i est telle que la composante $M_i[j,k]$ représente la vue que le site S_i a de la vue que S_j a du progrès du temps local à S_k.
- l'horloge k-matricielle A_i conserve uniquement les k plus grandes composantes de chaque colonne de M_i

Par conséquent, $A_i[\star,k]$ donne la vue que le site S_i a des vues que les k sites les plus à jour ont du progrès du temps local à S_k.
Une application des horloges k-matricielles concerne les protocoles optimistes de tolérance aux défaillances. Avec ces protocoles, chaque unité de recouvrement estampille les messages qu'elle émet seulement avec les états intervallaires du graphe d'antécédence pour lesquels elle ne sait pas s'ils ont été enregistrés par un certain quorum d'unités de recouvrement. Si le système obéit à une sémantique de défaillance des site par omission (*crash failures*) telle que le nombre maximum de défaillances simultanées soit borné par $k-1$, alors le quorum est fixé à k URs. Un état intervallaire doit donc être estampillé sur un message si et seulement si l'UR émettrice ne sait pas si au moins k URs (y compris elle-même) ont enregistré l'état intervallaire. Pour déterminer cela, il lui suffit de contrôler dans l'horloge k-matricielle la valeur des k plus grandes entrées de la colonne correspondant au processus auquel appartient l'état intervallaire.
Une autre application des horloges k-matricielles concerne l'implantation d'un service de journalisation tolérant aux fautes dans un système obéissant à une sémantique de défaillance de sites par omission, tel que le nombre maximum de défaillance simultanées soit borné par $k-1$. Nous envisageons par exemple d'utiliser les horloges k-matricielles pour rendre le service de journalisation d'événements de CDB tolérant aux défaillances de sites (*crash*).
Le lecteur averti aura remarqué les similitudes entre les horloges k-matricielles et la notion de k-ignorance (*k-bounded ignorance* [KB91]). Toutefois, alors que la k-ignorance est une technique de bases de données distribuées qui permet de garantir qu'un transaction donnée ne peut pas être ignorante de plus de

k transactions la précédant causalement, les horloges k-matricielles sont une technique qui garantit qu'au maximum $n - k$ sites du système peuvent être ignorants d'un événement causalement précédant leur état courant.

Chapitre VI

Conclusion

La mise au point des programmes parallèles et répartis est un art qui n'est pas encore arrivé à maturité. Les voies de recherche sont nombreuses : vérification formelle de propriétés, choix de représentations adéquates du programme et de son exécution, détection au vol de propriétés au cours d'une exécution particulière du programme, outils d'analyse post-mortem. Aucune ne constitue une panacée et les promesses de telle ou telle technique ne sont pas toujours tenues dans la pratique. Les approches les plus pragmatiques, faciles à mettre en œuvre, et qui constituent des évolutions plutôt que des révolutions par rapport aux outils connus des programmeurs non spécialistes ont le plus de chances de se développer.

Nous pensons que la technique post-mortem de ré-exécution fait partie de ces approches, et qu'elle pourraient bientôt déboucher sur des outils largement répandus. Elle étend en effet naturellement le concept de mise au point cyclique aux programmes répartis, et réalise une symbiose avec les outils de mise au point traditionnels : l'outil de ré-exécution contrôle le déroulement du programme réparti et permet de s'affranchir des contraintes de temps réel et de non déterminisme ; il co-existe avec les outils de mise au point traditionnels qui permettent quant à eux une analyse microscopique de l'état du programme.

C'est pourquoi nous avons choisi d'orienter notre recherche vers la réalisation d'un outil de ré-exécution. Il s'agit à l'origine d'un travail appliqué, ce qui (nous l'espérons) n'a pas exclu la rigueur. Notre thèse est qu'on peut implanter la fonction de ré-exécution à l'aide de techniques et matériels aujourd'hui largement distribués. Concrètement, nos développements ont eu comme support un réseau d'ordinateurs PC-x86DX connecté par ethernet et «tournant» le système d'exploitation à micro-noyau CHORUS de la société Chorus systèmes. Nous n'avons malheureusement pas pu expérimenter avec des machines multi-processeurs – cependant nous pensons que les difficultés additionnelles liées au multi-processeurs sont plus d'ordre pratique que conceptuel (ce point est discuté dans la thèse).

Les résultats que nous avons obtenus se résument ainsi.

Premièrement nous avons réalisé une extension du système à micro-noyau CHORUS pour le rendre apte à supporter des outils de ré-exécution. L'objectif de cette tâche était de fournir un support simple à utiliser et suffisamment souple, utili-

sable dans un cadre débordant la seule ré-exécution (e.g. pour des outils d'observation, de statistiques, de validation de protocoles de communication par injection de fautes, etc.). La contrainte principale que nous avions à respecter était de minimiser le coût de ces extensions, en termes de performance, de taille et de maintenance du micro-noyau CHORUS (au final, une augmentation de taille de 1.5% et une dégradation des performances négligeable). Ce travail a été effectué dans le contexte du projet européen ESPRIT N^0 6603 «OUVERTURE» et les développements réalisés seront intégrés dans la version standard du micro-noyau CHORUS.

Deuxièmement nous avons conçu et réalisé l'outil de ré-exécution CDB, supporté par le micro-noyau étendu. CDB peut être considéré comme un prototype servant à valider les extensions du micro-noyau (dans une optique restreinte), ou comme la version alpha d'un futur produit CHORUS, qui viendrait enrichir l'environnement de développement proposé aux programmeurs de systèmes et d'applications réparties (dans une optique plus large). Lors de la réalisation de CDB, le but était de réaliser un outil aussi puissant que possible s'appliquant à une catégorie d'applications la plus large et la plus réaliste possible. Au final, CDB est capable d'enregistrer et de reproduire l'exécution d'applications distribuées sur un réseau CHORUS de machines monoprocesseur, et prend en compte les appels système non déterministes, la mémoire virtuelle partagée, le cadencement préemptif des activités, certaines défaillances du système de communication, ainsi que les interactions entre l'application sous contrôle et son environnement. Nous ne nous étions pas fixé d'objectif précis en ce qui concerne les performances. Cependant, les mesures effectuées apparaissent satisfaisantes (un surcoût en temps de 3% à l'enregistrement et de 30% à la ré-exécution pour des applications distribuées typiques.)

Troisièmement, nous avons développé pour l'usage de CDB des algorithmes d'horloges logiques originaux et efficaces (ces développements relativement tardifs nont cependant pas encore été intégrés). Ces algorithmes permettent d'implanter une horloge matricielle pour un système de n sites avec un coût qui peut être réduit à $O(n)$ (taille des structures de données conservées sur chaque site et estampillées sur chaque message.) Ceci présente un progrès important par rapport à l'algorithme naïf qui implique un coût en $O(n^2)$.

L'utilisation d'un système à micro-noyau (CHORUS) s'est révélée particulièrement adéquate pour la réalisation de l'outil de ré-exécution, car elle a permis de manipuler des abstractions simples et de minimiser les informations contextuelles.

Cette recherche dans le domaine de la mise au point répartie a été particulièrement motivante car elle nous a amené à des incursions dans des domaines variés : le système, la causalité, les horloges logiques, les bases de données distribuées, la tolérance aux fautes, etc.

Pour conclure, nous voudrions mentionner quelques perspectives et suites possibles au «projet CDB».

En termes de fonctions, on pourrait :

- généraliser CDB à des machines multi-processeurs,
- implanter la notion de point de reprise pour donner à CDB une véritable

fonction de «voyage temporel»,

- associer à CDB un service de détection de propriétés d'exécution (cette recherche pourrait donner lieu à une coopération avec le projet INRIA ADP de Michel Raynal.)

En termes d'ingénierie, il serait très intéressant de tenter réaliser un produit à partir du prototype CDB. Cela impliquerait certainement :

- de réaliser une bonne intégration de CDB avec un débogueur traditionnel comme GDB,
- de fournir une interface graphique conviviale à CDB, et de s'intéresser de plus près au problème de la représentation des exécutions réparties,
- d'optimiser encore les performances, par exemple en utilisant des protocoles d'horloges logiques plus efficaces.

Finalement, l'expérience acquise avec la ré-exécution dans le cadre de CDB pourrait servir de base pour développer une boîte à outils facilitant la mise en place d'applications tolérantes aux pannes sur le principe du «leader-suiveur», mais c'est une autre histoire.

Annexe A

Démonstrations du chapitre V

A.1 Algorithme incrémental

Dans cette annexe, nous donnons les démonstrations promises au chapitre V. Pour identifier les propositions que nous démontrons, nous avons repris la numérotation du chapitre V.

Proposition 1 *Les règles (INT-I), (MSG-I) and (GC-I) préservent la contrainte (GA).*

Preuve : Rappelons la contrainte (GA) concernant le sous-graphe GA_i dont est muni le site S_i :

$$\forall j,k, \max(\downarrow_{E_k}\downarrow_{E_j} \{\epsilon_i\}) \in \mathrm{GA}_i \qquad \text{(GA)}$$

où ϵ_i est l'événement courant du site S_i et $\mathrm{GA}(\epsilon_i)$ est le graphe d'antécédence de ϵ_i.
Il est facile de voir que la contrainte (GA) est préservée par les règles (INT-I) et (MSG-I) définies en section V.4.1. Nous montrons qu'elle est également préservée par la règle (GC-I). Nous notons par GA_i et GA'_i les valeurs de GA_i respectivement avant et après le ramassage des miettes par la règle (GC-I). Nous avons :

$$\mathrm{GA}'_i = \mathrm{GA}_i - \left\{ e_l^j \in \mathrm{GA}_i | \ \forall k, M_i[k,j] > l \right\}$$

Nous supposons que GA_i vérifie la contrainte (GA). Nous voulons prouver que c'est aussi le cas de GA'_i. Considérons des indices j et k, et l'événement $e_l^k = \max(\downarrow_{E_k}\downarrow_{E_j} \{\epsilon_i\}) \in \mathrm{GA}_i$. e_l^k est le l-ième événement du site S_k. Par définition de l'horloge matricielle, $M_i[j,k] = \mathrm{card}(\downarrow_{E_k}\downarrow_{E_j} \{le\}) = l$. Donc, $M_i[j,k] \not> l$ et par conséquence, le ramassage des miettes ne peut pas ôter e_l^k du graphe GA_i. Autrement dit : $e_l^k = \max(\downarrow_{E_k}\downarrow_{E_j} \{\epsilon_i\}) \in \mathrm{GA}'_i$. □

A.2 k-approximations

Rappelons la définition formelle de l'opérateur «$\preceq_k$» de k-approximation :

$$b \preceq_k a \stackrel{\text{def}}{=} \exists I \subseteq \{1,..,n\}, \begin{cases} \text{card}(I) = k & \\ \forall i \notin I, b_i \leq a_i & (\text{I}\leq) \\ \forall i \in I, b_i = a_i & (\text{I}=) \\ \forall i \notin I, \forall j \in I, a_i \leq a_j & (\text{I}) \end{cases}$$

Proposition 2 *L'ensemble des vecteurs d'entiers de dimension n est partiellement ordonné par* $\preceq_k$ *(*$k \leq n$*).*

Preuve : $\preceq_k$ est visiblement réflexive et antisymétrique. Nous allons prouver qu'elle est aussi transitive (la preuve est illustrée par la figure A.1). Considérons $c \preceq_k b \preceq_k a$. Par définition de $\preceq_k$, nous savons qu'il existe deux ensembles d'indices $S, T \subseteq \{1,..,n\}$ de cardinal k tels que :

$$\begin{aligned} \forall i \notin S, c_i \leq b_i & \quad (\text{S}\leq) \\ \forall i \in S, c_i = b_i & \quad (\text{S}=) \\ \forall i \notin S, \forall j \in S, b_i \leq b_j & \quad (\text{S}) \\ \forall i \notin T, b_i \leq a_i & \quad (\text{T}\leq) \\ \forall i \in T, b_i = a_i & \quad (\text{T}=) \\ \forall i \notin T, \forall j \in T, a_i \leq a_j & \quad (\text{T}) \end{aligned}$$

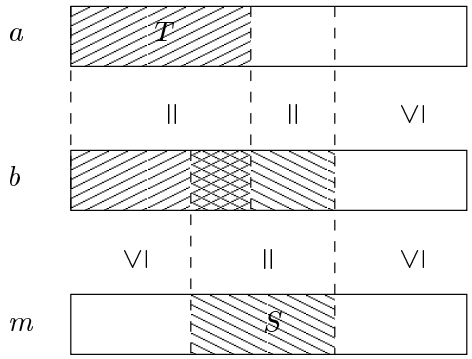


FIG. A.1 - $\preceq_k$ *est transitive*

De (S$\leq$), (T$\leq$) et (T=), nous déduisons que

$$\forall i \notin S, c_i \leq a_i \quad (\text{U}\leq)$$

Maintenant considérons $i \in S$. Nous allons prouver par l'absurde que $c_i = a_i$. Supposons donc que $c_i \neq a_i$. (S=) et (T=) impliquent que $i \notin T$, puis grâce à (T$\leq$) nous déduisons que $b_i < a_i$. Rapproché de (T=) et (T), ceci nous donne $\forall j \in T, b_i < b_j$. En contradiction avec (S). Par conséquent

$$\forall i \in S, c_i = b_i = a_i \quad (\text{U}=)$$

Considérons $j \in S$. De deux choses l'une. Soit **(a)** $S = T$ et alors $\exists i \in T, b_i \leq b_j$, ou bien **(b)** $S \neq T$ ce qui implique également $\exists i \in T, b_i \leq b_j$ à cause de (S). Rapproché de (U=) et (T=) ceci nous donne:

$$\forall j \in S, \exists i \in T, a_i \leq a_j \tag{1}$$

Nous prouvons maintenant que:

$$\forall i \notin S, \forall j \in S, a_i \leq a_j \tag{U}$$

Considérons $i \notin S$ et $j \in S$. De deux choses l'une. Soit **(a)** $i \in T$, alors $a_i = b_i$ (d'après (T=)), $b_i \leq b_j$ (d'après (S)), $b_j = a_j$ (d'après (U=)). Ou bien **(b)** $i \notin T$, et alors (1) nous permet d'exhiber un $j' \in T$ tel que $a_{j'} \leq a_j$, donc (d'après (T)) $a_i \leq a_{j'}$. Dans les deux cas (U) est vérifiée.
Finalement (U≤),(U=) et (U) prouvent que $c \preceq_k a$. □

Proposition 3 *Soit* max *l'opérateur qui prend le maximum de deux vecteurs composante par composante.* max *and* $\preceq_k$ *«commutent».*

Plus précisément, considérons A et B deux vecteurs d'entiers de dimension n. Soit M, le maximum composante par composante de A et B. Considérons ensuite a et b, des k-approximations respectives de A et B. Soit enfin m, le maximum composante par composante de a et b. Alors m est une k-approximation de M. Formellement:

$$\forall A, B, M, a, b, m \in \mathbb{N}^n,$$
$$\begin{cases} M = \max(A, B) \\ a \preceq_k A \\ b \preceq_k B \\ m = \max(a, b) \end{cases} \implies m \preceq_k M$$

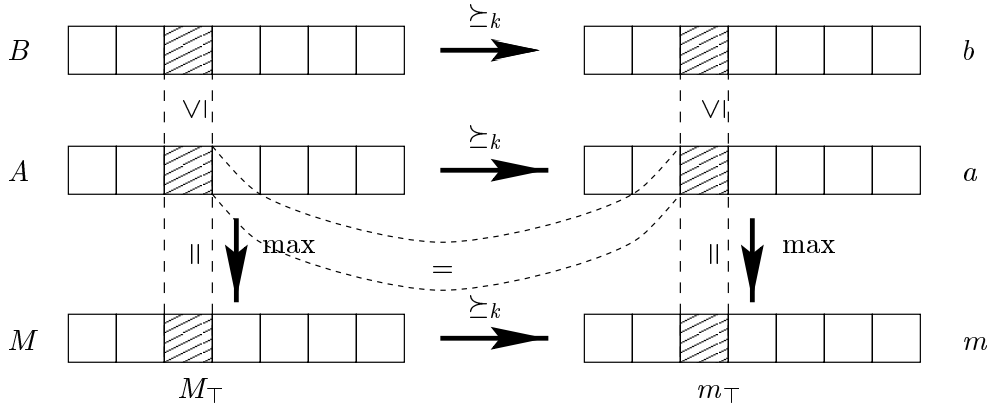


FIG. A.2 - max *et* $\preceq_k$ *«commutent»*

Preuve: La proposition se démontre par induction sur le nombre de sites n du système (la preuve est illustrée par la figure A.2[1]). La proposition est vraie

1. Sur la figure, nous avons fait l'hypothèse que $A_\top = M_\top = a_\top = m_\top$

pour $n = 1$. Supposons qu'elle soit vraie pour $(n-1)$, et démontrons la pour n. Considérons $k \leq n$ and A, B, M, a, b, m définis par la proposition. Le cas $k = 0$ est trivial. Si $k \neq 0$, alors notons $M_\top = m_\top$ la composante la plus grande de M qui soit aussi dans m. Ôtons ensuite la composante d'indice $\top$ des vecteurs A, B, M, a, b and m, de façon à obtenir des vecteurs A', B', M', a', b' and m' de dimension $(n-1)$. Ces vecteurs vérifient les hypothèses d'induction pour $(n-1)$ et $(k-1)$, par conséquent $m' \preceq_{(k-1)} M'$. Et donc $m \preceq_k M$. □

Proposition 4 *L'horloge approchée A est effectivement une k-approximation de la véritable matrice matricielle M:*

$$\forall e \in E, A(e) \preceq_k M(e)$$

Preuve : Nous ne considérons pas le cas $k = 0$ qui est trivial. Donc supposons $k > 0$. La preuve est par induction sur le cardinal de $\downarrow_E \{e\}$ [2]. Si $\text{card}(\downarrow_E \{e\}) = 0$, alors à la fois $A(e)$ et $M(e)$ sont remplis de zéros, et la proposition est vérifiée. Soit maintenant un entier x, et supposons que la proposition soit vérifiée pour tous les événements e tels que $\text{card}(\downarrow_E \{e\}) < x$. Considérons un événement e du site S_i, tel que $\text{card}(\downarrow_E \{e\}) = x$.

(a) Si e n'est pas une réception de message, notons p le prédécesseur immédiat de e (sur le site S_i). Par hypothèse d'induction, $A(p) \preceq_k M(p)$. Notons que $M(p)[i,i]$ est strictement supérieur à tous les autres $M(p)[h,i]$. Du fait que $A(p) \preceq_k M(p)$ et que $k > 0$, nous devons donc avoir $A(p)[i,i] = M(p)[i,i]$. Par conséquent $A(e) \preceq_k M(e)$ (parce que $A(e)$ et $M(e)$ sont construits en ajoutant 1 à la composante $[i,i]$ de $A(p)$ et $M(p)$ respectivement).

(b) Si e est la réception du message m, notons p l'événement prédécesseur immédiat de e sur le site S_i, et notons s l'événement émissions du message m. Pour fixer les idées, supposons que s se produit sur le site S_j. Par hypothèse d'induction, $A(p) \preceq_k M(p)$ et $A(s) \preceq_k M(s)$. Notons A_1 la matrice obtenue en ajoutant 1 à la composante $[i,i]$ de A(p), et notons A_2 la matrice obtenue en prenant le maximum de A_1 et $A(s)$, composante par composante. Définissons M_1 et M_2 de la même façon. Encore une fois, $A(p)[i,i] = M(p)[i,i]$ est strictement supérieur à tous les autres $M(p)[h,i]$, et par conséquent, $A_1 \preceq_k M_1$. La proposition 3 ($\preceq_k$ et max «commutent») démontre alors que $A_2 \preceq_k M_2$. Pour finir, $A(e) \preceq_k M(e)$ parce que $A(e)$ et $M(e)$ sont obtenues en remplaçant la composante d'indice i de chaque colonne de A_2 et M_2 par le maximum des composantes d'indice i et j de ces colonnes respectives.

Pour montrer ce dernier point, considérons a et b, deux vecteurs d'entiers de dimension n tels que $b \preceq_k a$. Considérons A et B obtenus en remplaçant la composante d'indice i de chaque colonne de a et b par le maximum des composantes d'indice i et j de la colonne. Nous pouvons supposer sans perte de généralité que $i = 1$ and $j = 2$. Il y a donc quatre cas possibles, suivant la position de l'ensemble S des k indices correspondant aux composantes les plus élevées, relativement à i et j. Les quatre cas sont illustrés par la figure A.3 (sur la figure, l'ensemble S est indiqué en grisé). Le lecteur peut vérifier que pour chacun des cas, on a bien $B \preceq_k A$. □

2. Rappelons que $\downarrow_E \{e\}$ est l'ensemble des événements prédécesseurs de e parmi l'ensemble E de tous les événements de l'exécution.

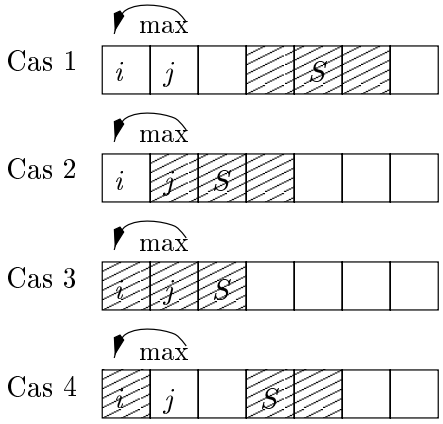


FIG. A.3 - *Positions relatives de i, j, and S*

A.3 k-approximations et condition d'horloge

Dans cette section, nous supposons que $k > 0$. Tout d'abord, nous remarquons que l'ordre composante par composante n'est pas adéquat pour comparer les valeurs données par une horloge k-matricielle. En effet, les conditions d'horloge (CLK) or (S-CLK) (cf. section V.2) ne sont pas vérifiées en utilisant cet ordre. Pour le voir, il suffit de considérer les événements e_2 et e_3 et les dates $M(e_2)$ et $M(e_3)$ que leur associe respectivement l'horloge. Le lecteur peut vérifier que $M(e_2) \not\leq M(e_3)$ est en contradiction avec (CLK) et (S-CLK). La question est donc de savoir s'existe un ordre partiel sur les matrices carrées de taille $n \times n$, tel que (CLK) ou (S-CLK) soient vérifiée. C'est le but du reste de cette section.

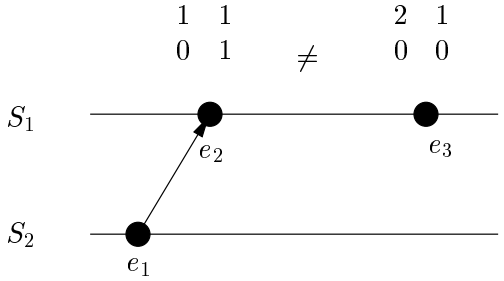


FIG. A.4 - *Inadéquation de l'ordre composante par composante*

Définition 13 *Considérons $a, b \in \mathbb{N}^n$, et deux vecteurs d'entiers, de dimension n. Nous définissons la relation $\leq_k$ comme suit.*

$$b \leq_k a \stackrel{\text{def}}{=} \exists i_1, j_1, i_2, j_2, \ldots, i_n, j_n, \begin{cases} \{i_1, \ldots, i_n\} = \{j_1, \ldots, j_n\} = \{1, \ldots, n\} \\ a[i_1] \geq a[i_2] \geq \ldots \geq a[i_n] \\ b[j_1] \geq b[j_2] \geq \ldots \geq b[j_n] \\ \forall l \leq k, b[i_l] \leq a[i_l] \end{cases}$$

De façon informelle, $b \leq_k a$ si et seulement si les k plus grandes composantes de a sont supérieures aux k-plus grandes de b. Nous disons que a est k-supérieur à n, ou de façon équivalente, que b est k-inférieur à a.
Par exemple :

$$\begin{bmatrix} 0 \\ 5 \\ 6 \end{bmatrix} \leq_2 \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} ; \begin{bmatrix} 1 \\ 5 \\ 6 \end{bmatrix} \leq_2 \begin{bmatrix} 6 \\ 6 \\ 0 \end{bmatrix} ; \begin{bmatrix} 0 \\ 4 \\ 5 \end{bmatrix} \not\leq_2 \begin{bmatrix} 1 \\ 3 \\ 6 \end{bmatrix}$$

Bien sûr, si $a \preceq_k b$, alors $a \leq_k b$.

Définition 14 *Considérons $A, B \in \mathbb{N}^{n \times n}$, deux matrices carrées d'entiers de dimension $n \times n$. Nous notons $B \leq_k A$ le fait que chaque colonne de B soit k-inférieure à la colonne correspondant deA. Formellement :*

$$\forall A, B \in \mathbb{N}^{n \times n},$$
$$B \leq_k A \stackrel{\text{def}}{=} \forall 1 \leq j \leq n, B[\star, j] \leq_k A[\star, j]$$

(Où $A[\star, j]$ représente la j-ième colonne de A.)
Par exemple :

$$\begin{bmatrix} 5 & 3 & 3 \\ 2 & 5 & 0 \\ 4 & 0 & 6 \end{bmatrix} \leq_2 \begin{bmatrix} 5 & 3 & 3 \\ 1 & 5 & 3 \\ 5 & 3 & 6 \end{bmatrix}$$

Proposition 5 *Si l'ensemble de matrices carrées d'entiers de taille $n \times n$ est muni de la relation $\leq_k$, alors l'horloge k-matricielle A vérifie la condition forte d'horloge(S-CLK) :*

$$\forall e_1, e_2 \in E, e_1 \leq e_2 \iff A(e_1) \leq_k A(e_2)$$

Preuve : « $\Longrightarrow$ » est facile, nous allons démontrer « $\Longleftarrow$ ».
Considérons deux événements de l'exécution du système : $e_l^i \in E_i$ et $e_m^j \in E_j$, tels que $A(e_l^i) \leq_k A(e_m^j)$. Nous montrons que $e_l^i \leq e_m^j$.
Le cas $i = j$ est facile. Nous supposons donc $i \neq j$. Premièrement, notons que pour tout événement $e \in E$, et pour tout indice c, la valeur de la plus grande composante de la c-ième colonne de $A(e)$ est égale à la valeur de la composante $M(e)[c, c]$ de l'horloge matricielle véritable (nous supposons en effet $k > 0$). Alors, de trois choses l'une : soit **(a)** $e_l^i > e_l^j$: dans ce cas, $M(e_l^i)[j, j] > M(e_m^j)[j, j]$. En contradiction avec $A(e_l^i) \leq_k A(e_m^j)$ (considérer la colonne j) ; soit **(b)** e_l^i et e_l^j sont concurrent concurrent: dans ce cas, $M(e_l^i)[j, j] < M(e_m^j)[j, j]$ et $M(e_l^i)[i, i] > M(e_m^j)[i, i]$. En contradiction avec $A(e_l^i) \leq_k A(e_m^j)$ (considérer soit la colonne i, soit la colonne j). Il ne reste donc plus que le cas **(c)** $e_l^i < e_l^j$. □
Le lecteur peut vérifier que l'exécution représentée par la figure A.4 satisfait bien la condition forte (S-CLK) lorsqu'on muni $\mathbb{N}^{n \times n}$ de la relation $\leq_1$.
Bien sûr, $\leq_k$ n'est pas tout à fait un ordre partiel sur $\mathbb{N}^{n \times n}$. Par exemple :

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \leq_1 \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \leq_1 \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

En réalité, $\leq_k$ satisfait seulement les propriétés de réflexivité et de transitivité ; c'est donc un *pré-ordre*. Cependant, comme la condition (S-CLK) est vérifiée, $\leq_k$ est effectivement un ordre partiel sur le sous-ensemble de $\mathbb{N}^{n\times n}$ composé de l'ensemble des dates associées aux événements de l'exécution par l'horloge k-matricielle.

Bibliographie

[ADS91] H. Agrawal, R. A. DeMillo, and E. H. Spafford. An execution-backtracking approach to debugging. *IEEE Software*, pages 21–26, May 1991.

[AG89] Z. Aral and I. Gertner. High level debugging of distributed systems. *SIGPLAN Notices*, 24(1), January 1989.

[AHPR91] M. Adam, M. Hurfin, N. Plouzeau, and M. Raynal. Distributed debugging techniques. Technical note, IRISA, 1991.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Mass., 1974.

[AM86] E. Adams and S. S. Muchnick. Dbxtool: A window-based symbolic debugger for sun workstations. *Software Practice and Experience*, 16(7), July 1986.

[AM88] W. Appelbe and C. McDowell. Developing multitasking programs. In *Proc. of Hawaii International Conf. on System Sciences*, pages 94 – 102, January 1988.

[AV93] S. Alagar and S. Venkatesan. Hierarchy in testing distributed prorams. In *Proc. Int. Workshop AADEBUG 93*, pages 101–116. Springer-Verlag LNCS 749, May 1993.

[Awe85] B. Awerbuch. Complexity of network synchronization. *Journal of ACM*, 32(4):801–823, October 1985.

[Bat87] P. Bates. The EBBA modelling tool, a.k.a. event definition language. Technical Report 87–35, Univ. of Massachusetts, April 1987.

[BDF92] B. N. Bershad, R. P. Draves, and A. Forin. Using microbenchmarks to evaluate system performance. In *Proceedings of the Third Workshop on Workstation Operating Systems (WWOS-3)*, April 1992.

[BG91] D. F. Bacon and S. C. Goldstein. Hardware-assisted replay of multiprocessor programs. In *Proc. ACM/ONR workshop on parallel and distributed debugging*, 1991.

[BHL89] T. Bemmerl, O. Hansen, and T. Ludwig. Ein leistungsmessystem fuer multiprozessoren. In *PARS-Workshop*, Muenchen, 1989.

[BHS92] G. Brooks, G. J. Hansen, and S. Simmons. A new approach to debugging optimized code. *ACM SIGPLAN'92 Conf. on Programming Language Design and Implementation*, 27(7):1–11, July 1992.

[BM93] Ö. Babaŏglu and K. Marzullo. Consistent global states. In Sape Mullender, editor, *Distributed Systems*, chapter 4. ACM press, 1993.

[BMST93] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In Sape Mullender, editor, *Distributed Systems*, chapter 8. ACM press, 1993.

[BP82] R. Bonnet and M. Pouzet. Linear extensions of ordered sets. In I. Rival (ed.), editor, *Ordered Sets*, pages 125–170. D. Reidel Publishing Company, 1982.

[BW83] P. Bates and J. Wileden. High-level debugging of distributed systems: The behavioral abstraction approach. *Journal of System Software*, 3:235–244, 1983.

[CASD85] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. In *Proc. 15th Int. Symp. on Fault-tolerant Computing*, June 1985.

[CBDGF91] B. Charron-Bost, C. Delporte-Gallet, and H. Fauconnier. Local and temporal predicates in distributed systems. Technical report, LITP, IBP, Université de Paris 7, April 1991.

[CBM90] W. H. Cheung, J. P. Black, and E. Manning. A framework for distributed debugging. *IEEE Software*, 7(1):106–115, January 1990.

[CC80] P. Cousot and R. Cousot. Semantic analysis of communicating sequential process. *Lecture Notes in Computer Science No 85*, 1980.

[Cho91] Chorus Team. CHORUS kernel v3 r4.0 – debugger user's manual for COMPAQ deskpro386. Technical Report CS/TR-91-72, Chorus Systèmes, 1991.

[Cho92] Chorus Team. Overview of the Chorus distributed operating system. In *USENIX Workshop on Micro Kernels and Other Kernel Architecthres*, Seattle (USA), 1992.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[CM91] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 167–174, Santa Cruz, CA, May 1991.

[Coo87] R. Cooper. Pilgrim: A debugger for distributed systems. In *Proc. of the 3rd Int. Conf. on Distributed Computing Systems*, pages 458–465. IEEE, 1987.

[Cou90] P. Cousot. Methods and logics for proving programs. In *Handbook of theoretical computer science*, chapter 15, pages 843–993. Elsevier Science Publishers B. V., 1990.

[Cri89] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–158, 1989.

[CS91] J.-D. Choi and J. M. Stone. Balancing runtime and replay costs in a trace-and-replay system. In Barton P. Miller and Charles McDowell, editors, *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 26–35, Santa Cruz, California, May 1991.

[CW82] R. Curtis and L. Wittie. BugNet: A debugging system for parallel programming environments. In *Proc. of the 3rd Int. Conf. on distributed Computing Systems*, Hollywood, October 1982.

[Die92] C. Diehl. *Analyse de la relation de causalité dans les exécutions réparties.* PhD thesis, University of Rennes 1, Sept 1992.

[DJ92] C. Diehl and C. Jard. Interval approximations of messages causality in distributed executions. In Finkel and editors Jantzen, editors, *STACS, LNCS 577*, pages 363–374. Springer-Verlag, Cachan, February 1992.

[DS90] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorighms for access anomaly detection. In *Proceedings of PPOPP'90*, 1990.

[EA88] S. Eichholz and F. Abstreiter. Runtime observation of parallel programs. In *CONPAR'88*, Manchester, September 1988.

[Els88] I. J. P. Elshoff. A distributed debugger for amoeba. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 1–10. ACM, 1988.

[EZ92] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast ouput commit. *IEEE Transactions on Computers, Special Issue on Fault-Tolerant Computing*, 41(5):526–531, May 1992.

[EZ93] E. N. Elnozahy and W. Zwaenepoel. Fault tolerance for a workstation cluster. In *Proc. of the Workshop on Hardware and Software Architectures for Fault Tolerance*, May 1993.

[Fid88] J. Fidge. Timestamps in message passing systems that preserve the partial ordering. In *Proc. 11th Australian Computer Science Conference*, pages 55–66, February 1988.

[Fid91] C. J. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, August 1991.

[FL89] R. Fowler and T. LeBlanc. An integrated approach to parallel program debugging and performance analysis on large-scale multiprocessors. *SIGPLAN Notices (proc. of ACM/ONR Workshop on Parallel and Distributed Debugging)*, 24(1), January 1989.

[FM82] M. J. Fischer and A. Michael. Sacrifying serializability to attain high availability of data in an unreliable network. In *Proc. ACM SIGACT-SIGMOD Symp. on principles of database systems*, pages 70–75, Los Angeles, March 1982.

[For89] A. Forin. Debugging of heterogeneous parallel programs. In *Proc. of the ACM Workshop on Parallel and Distributed Debugging*, pages 130–139, 1989.

[FR94a] E. Fromentin and M. Raynal. Inevitable global states: a new concept to detect unstable properties of distributed computations in an observer independent way. In *6th IEEE Symp. on Par. and Dist. Processing (submitted)*, Dallas, 1994.

[FR94b] E. Fromentin and M. Raynal. Local states in distributed computations: a few relations and formulas. *ACM Operating Systems Review*, 28(2):65–72, April 1994.

[FR94c] E. Fromentin and M. Raynal. When all the observers of a distributed computation do agree. Research Report 794, IRISA, January 1994.

[FRGT94] E. Fromentin, M. Raynal, V. Garg, and A. Tomlimson. On the fly testing of regular patterns in distributed computations. In *23rd Annual Int. Conf. on Parallel Processing (ICPP94)*, Pennsylvania, August 1994.

[Gai85] J. Gait. A debugger for concurrent programs. *Software Practice and Experience*, 15(6):539–554, June 1985.

[GGLS92] A. P. Goldberg, A. Gopal, A. Lowry, and R. Strom. Restoring consistent global states of distributed computations. *ACM SIGPLAN Notices*, 26(12), Decembre 1992.

[GMGK84] H. Garcia-Molina, F. Germano, and W. H. Kohler. Debugging a distributed computing system. *IEEE Transactions on Software Engineering*, 10(2):210–219, March 1984.

[Gri87] J. Griffin. Parallel debugging system user's guide. Technical report, Los Alamos National Laboratory, 1987.

[GVS94] W. Gu, J. Vetter, and K. Schwan. An annotated bibliography of interactive program steering. Technical Report GIT-CC-94-15, College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332-0280, February 1994.

[GW92] V. K. Garg and B. P. Waldecker. Detection of unstable predicates in distributed programs. In *Proc. 12th Int. Conf. on Foundations of Soft. Technology*, pages 253–264, December 1992.

[Hab87] D. Haban. DTM-A method for testing distributed systems. In *6th Symp. on Reliability in Distributed Software and Database Systems*, Williamsburg, Virginia, March 1987.

[HCU92] U. Hoelzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic debugger. *ACM SIGPLAN'92 Conf. on Programming Language Design and Implementation*, 27(7):32–43, July 1992.

[Her91] M. Herdieckerhoff. Implementation of PATOC on the EDS testbed. Technical Report EDS.DD.8F.0027, ESPRIT II, January 1991.

[HHW89] A. Heddaya, M. Hsu, and W. E. Weihl. Two phase gossip: Managing distributed event histories. *Information Sciences*, 49(1):35–57, 1989.

[HK88] W. Hseush and G. E. Kaiser. Data path debugging: Data-oriented debugging for a concurrent programming language. In *ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 236–246, Madison (WI), May 1988.

[HK90] W. Hseush and G. E. Kaiser. Modeling concurrency in parallel debugging. *ACM SIGPLAN Notices*, pages 11–20, March 1990.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International Series in Computer Science, 1985.

[HP 89] Precision architecture and instruction set reference manual. Hewlett Packard, September 1989.

[HPR91] M. Hurfin, N. Plouzeau, and M. Raynal. Implementation of a distributed debugger for estelle programs. In *ERCIM workshop*, 1991.

[HPR92] M. Hurfin, N. Plouzeau, and M. Raynal. Erebus: A debugger for asynchronous distributed computing systems. In *Proc. of 3rd IEEE Work. on Future Trends in Distribued Computing Systems*, Taipei, Taiwan, April 1992.

[HPR93] M. Hurfin, N. Plouzeau, and M. Raynal. Detecting atomic sequences of predicates in distributed computations. In *Proceedings*

of the ACM/ONR Workshop on Parallel and Distributed Debugging, pages 32–42, San Diego, California, May 1993.

[HR93] M. Herdieckerhoff and F. Ruget. CHORUS support for monitoring and debugging. OUVERTURE Technical Report (D2.5/2) OU/TR-93-12, Chorus Systems, 1993.

[HR94] M. Herdieckerhoff and F. Ruget. Matching operating systems to application needs – a case study. In *Proc. of SIGOPS'94*, pages 160–165, Germany, September 1994.

[HW88] D. Haban and W. Weigel. Global events and global breakpoints in distributed systems. In *Proc. of 21th Int. Conf. on System Sciences*, pages 166–175, Hawaii, January 1988.

[HZMW91] D. Haban, S. Zhou, D. Maurer, and R. Wilhelm. Specification and detection of global breakpoints in distributed systems. Technical Report SFB124 - 08/1991, Universitaet des Saarlandes, Saarbruecken, Germany, April 1991.

[Int92] Intel. *Intel486 Microprocessor Family - Programmer's Reference Manual*. Intel, 1992.

[Jam93] H. Jamrozik. *Aide à la Mise au Point des Applications Parallèles et Réparties à base d'Objets Persistants*. Ph.D. thesis, Université Joseph Fourier, Grenoble (France), May 1993.

[JBW87] S. H. Jones, R. H. Barkan, and L. D. Wittie. Bugnet: a real time distributed debugging system. In *Proc. of the 6th Symp. on Reliability in Distributed Software and Database Systems*, pages 56–65, March 1987.

[Jéz89] J.-M. Jézèquel. Building a global time on parallel machines. In *Proc. 3rd Inter. Workshop on Distributed Algorithms*, Nice, Septembre 1989. Springer Verlag.

[JJGVR94] C. Jard, T. Jéron, J. Guy-Vincent, and J.-X. Rampon. A general approach to trace-checking in distributed computing systems. Technical Report 811, IRISA, 1994.

[JLSU87] J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring distributed systems. *ACM Transactions on computer System*, 5(2):121–150, March 1987.

[Jon92] M. B. Jones. Transparently interposing user code at the system interface. In *Proc. of the 2nd Work. on Workstation Operating Systems*, April 1992.

[JRS91] H. J. Jamrozik, C. Roisin, and M. Santana. A graphical debugger for object-oriented distributed programs. In *Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 117–128, July, 1991.

[JZ87] D. B. Johnson and W. Zwaenepoel. Sender-based message logging. In *The 17th Int. Symp. on Fault-Tolerant Computing*, pages 14–19. IEEE Computer Society, June 1987.

[JZ90] D. B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11(3):462–491, September 1990.

[KB91] Krishnakumar and Bernstein. Bounded ignorance in replicated systems. In *Proc. ACM Symp. on Principles of Database Systems*, 1991.

[KT87] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, se–13(1):23–31, January 1987.

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

[LB90] T. Lehr and D. L. Black. Mach kernel monitor (with applications using the pie environment). Available on host mach.cs.cmu.edu in /usr/mach/public/doc/unpublished/monmanual.ps through anonymous FTP, February 1990.

[LCSM90] J. E. Lumpp, T. L. Casavant, H. J. Siegle, and D. C. Marinescu. Specification and identification of events for debugging and performance monitoring of distributed multiprocessor systems. In *Proc. of the 10th Int. Conf. on Distributed Systems*, pages 476–483, June 1990.

[Leu92] E. Leu. *La ré-exécution, pierre angulaire de la mise au point des programmes parallèles.* PhD thesis, École Polytechnique Fédérale de Lausanne, 1992.

[Lin86] H. R. Lingg. Test machine. a tool for the development of efficient errorfree realtime systems. *Revue Landis & Gyr No 1*, pages 20–27, 1986.

[LKH92] W. Lux, W. E. Kuhnhauser, and H. Hartig. The birlix migration mechanism. In *Workshop on Dynamic Object Placement and Load Balancing in Parallel and Distributed Systems Programs*, pages 83–90, June 1992.

[LL86] B. Liskov and R. Ladin. Highly available distributed services and fault-tolerant distributed garbage collection. In *Proc. 5th ACM Symp. on PODC*, pages 29–39, 1986.

[LMC87] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Computers*, C-36(4):471–481, April 1987.

[LP85] C. H. LeDoux and D. S. Parker. Saving traces for ada debugging. In *Ada in use, Proc. of the Ada Int. Conf*, pages 87–108, 1985.

[LR89] P. C. Lewis and D. Reed. Debugging shared memory parallel programs. *SIGPLAN Notices*, 24(1), January 1989.

[LS91] E. Leu and A. Schiper. Techniques de déverminage pour programmes parallèles. *Technique et Science Informatiques*, 10(1):5–21, 1991.

[LSZ90] E. Leu, A. Schiper, and A. Zramdini. Execution replay on distributed memory architectures. In *Proc. of 2nd IEEE Symp. on Parallel and Distributed Processing*, Dallas, December 1990.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In *Proc. of Int. Workshop on Parallel and Distributed Algorithms, Bonas (France)*, pages 215–226. Cosnard, Quinton, Raynal and Robert editors, 1989.

[MC88a] B. Miller and J.-D. Choi. Techniques for debugging parallel programs with flowback analysis. Technical Report 786, University of Wisconsin-Madison, August 1988.

[MC88b] B. P. Miller and J.-D. Choi. Breakoints and halting in distributed systems. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 316–323. IEEE, 1988.

[MC88c] B. P. Miller and J.-D. Choi. A mechanism for efficient debugging of parallel programs. In *SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 135–144, Atlanta, Georgia, June 1988. ACM.

[MC91] S. L. Min and J.-D. Choi. An efficient cache-based access anomaly detection scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 235–244, Santa Clara, CA, April 1991.

[MC93] J. Mellor-Crummey. Compile-time support for efficient data race detection in shared-memory parallel programs. In *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 129–139, San Diego, California, May 1993. ACM.

[MCL89] J. M. Mellor-Crummey and T. J. LeBlanc. A software instruction counter. In John L. Hennessy, editor, *Proc. of Third Int. Conf. on Architectural Support for Programming Languages and Operating Systems, Boston, MA*, pages 78–86. ACM/IEEE, 1989.

[McR93] A. McRae. Hardware profiling of kernels. In *Proc. of USENIX Winter 1993 Technical Conference*, pages 375–386, San Diego, California, January 1993.

[MCWB91] K. Marzullo, R. Cooper, M. D. Wood, and K. P. Birman. Tools for distributed application management. *IEEE Computer*, pages 42–51, August 1991.

[MH89] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[Mic93] NewBits: A quarterly newsletter from microtec research, inc., 10(3), 1993.

[Mon93] B. Monsuez. *Typage par interprétation abstraite*. PhD thesis, LIENS, 1993.

[Mou90] P. Moukeli. Les principales approches de conception des débogueurs pour les langages parallèles. Technical Report 90–05, Laboratoire LIP Ecole Normale Supérieure de Lyon, 69364 Lyon Cedex 07, FRANCE, Janvier 1990.

[MRA+89] A. D. Malony, D. A. Reed, J. W. Arendt, D. Grabas, R. A. Aydt, and B. K. Totty. An integrated performance data collection analysis and visualisation system. In *Proc. of the 4th Conf. on Hypercube Concurrent Computers and Applications*, pages 229–236, 1989.

[Net93] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 129–139, San Diego, California, May 1993.

[NM92] R. H. B. Netzer and B. P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Supercomputing '92*, Minneapolis, MN, November 1992.

[NR88] I. Nudler and L. Rudolph. Tools for the efficient development of efficient parallel programs. In *1st Israeli Conference on Computer System Enginneering*, 1988.

[NT86] G. Neiger and S. Toueg. Substituting for real time and common knowledge in distributed systems. Technical Report 86-790, Cornell University, Ithaca, New York 14853-7501, 1986.

[OSS93] D. M. Ogle, K. Schwan, and R. Snodgrass. Application-dependent dynamic monitoring of distributed and parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):762–778, July 1993.

[Pat93] S. Patience. Redirecting system calls in mach 3.0 - an alternative to the emulator. In *Mach III Symposium*, pages 57–73. Usenix association, 1993.

[Pdb86] Dynix pdbx parallel debugger user's manual. Sequent Corp., 1986.

[PHK91] M. K. Ponamgi, W. Hseush, and G. E. Kaiser. Debugging multithreaded programs with mpd. *IEEE Software*, 8(3):37–43, May 1991.

[PL89] D. Pan and M. Linton. Supporting reverse execution for parallel programs. *SIGPLAN Notices*, 24, January 1989.

[PT88] P. Panangaden and K. Taylor. Concurrent common knowledge: A new definition of agreement for asynchronous systems. In *Proc. of 7th. ACM Symp. on Principles of Distributed Computing*, 1988.

[RAA+88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, and M. Gien. CHORUS distributed operating system. *Computing Systems Journal, The Usenix Association*, pages 305–370, 1988.

[Ray92] M. Raynal. About logical clocks for distributed systems. *Operating Systems Review*, 26(1):41–48, January 1992.

[Ray94] M. Raynal. Ré-exécution et analyse de calculs répartis. Publication interne 814, IRISA, 1994. Invited paper, JISI 94, May 1994.

[RCM93] J. F. Roos, L. Courtrai, and J. F. Mehaut. Execution replay of parallel programs. In *Proc. of the Euromicro Work. on Parallel and Distributed Processing*, pages 429–434, 1993.

[Roz90] M. Rozier. Overview of the CHORUS distributed operating system. Technical report, Chorus Systems, 1990.

[Roz91] M. Rozier. Technologie des noyaux système. In R. Balter, J.-P. Banâtre, and S. Krakowiak, editors, *Construction des systèmes répartis*, chapter 10. INRIA, 1991.

[RR89] R. Rubin and L. Rudolph. Debugging parallel programs in parallel. *SIGPLAN Notices*, 24(1), January 1989.

[Ruf92] M. Ruffin. Kitlog–a generic logging service. In *Proc. of the 11th Symp. on Reliable Dist. Syst.*, Houston (TX, USA), Oct., 5–7 1992.

[Rug91] F. Ruget. Distributed debugging–application to CHORUS. Master's thesis, Ecole Normale Supérieure–Ecole Polytechnique, 1991.

[Rug93] F. Ruget. états globaux d'un système réparti–reprise de l'exécution après une faute. In *Journées des Jeunes Chercheurs en Systèmes Informatiques Répartis*, pages 81–88, Grenoble (France), 1993. IMAG.

[Rug94a] F. Ruget. Actor-wide trap tables for CHORUS. Technical note, Chorus Systems, 1994.

[Rug94b] F. Ruget. Cheaper matrix clocks. In *Proc. of the 8th Int. Workshop on Distributed Algorithms*, Terschelling, the Netherlands, September 1994. Springer Verlag.

[Rug94c] F. Ruget. Cheaper matrix clocks. In *Unifying Theory and Practice in Distributed Systems*, Schloss Dagstuhl, Germany, September 1994.

[Rug95] F. Ruget. Micro-kernel support for trace-replay. In *Proc. of ER-SADS workshop*, L'Alpe d'Huez, April 1995. Presented at OSDI'94 Mach/Chorus Workshop, Monterey, CA.

[SBN89] D. Socha, M. L. Bailey, and D. Notkin. Voyeur: Graphical views of parallel programs. *Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1):206–215, January 1989.

[SBY88] R. E. Strom, D. F. Bacon, and S. A. Yemini. Volatile logging in n-fault-tolerant distributed systems. In *The Eighteenth Annual International Symposium on Fault-Tolerant Computing: Digest of Papers*, pages 44–49, June 1988.

[Sch91] M. Schiefert. PARTAMOS: Parallel real-time application monitoring system. Product sheet, Alcatel Austria–ELIN Research Center, Ruthnergasse 1–7, A1210 Vienna, 1991.

[SG86] R. W. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(1), April 1986.

[SG94] M. Spezialetti and R. Gupta. Debugging distributed programs through the detection of simultaneous events. In *Proc. of Int. Conf. on Distributed Computing Systems ICDCS*, 1994.

[SL87] S. K. Sarin and L. Lynch. Discarding obsolete information in a replicated database system. *IEEE Trans. on Soft. Eng.*, SE 13(1):39–46, January 1987.

[SM94] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3), March 1994.

[Smi84] E. Smith. Debugging tools for message-based communicating processes. In *4th Internat. conf. on Distributed Computing Systems*, pages 303–310, April 1984.

[Sno87] R. Snodgrass. The temporal query language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, June 1987.

[Sno88] R. Snodgrass. A relational approach to monitoring complex systems. *ACM Transactions on Computer Systems*, 6(2):157–196, May 1988.

[Sta86] R. Stallman. The gnu debugger. Technical report, Free Software Foundation, Inc, 675 Mass. Avenue, Cambridge, MA, 02139, USA, 1986.

[Sto88] J. M. Stone. Debugging concurrent processes: A case study. *SIGPLAN Notices (proc. of Conf. on Programming Language Design and Implementation, Atlanta, GA USA)*, 23(7):145–153, 1988.

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Mass., 1986.

[Sun86] NeWs preliminary technical overview. Sun MicroSystems, 1986.

[SW87] D. Snowden and A. Wellings. Debugging distributed real-time applications in ada. Technical report, University of York, UK, April 1987.

[SY85] R. E. Strom and S. A. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[TA91] A. P. Tolmach and A. W. Appel. Debuggable concurrency extensions for standard ML. In *Proc. ACM/ONR workshop on parallel and distributed debugging*, pages 120–131, 1991.

[Tok89] H. Tokuda. A real-time monitor for a distributed real-time operating system. *Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1):68–77, January 1989.

[Val90] C. Valot. A survey of distributed debugging techniques. Note technique SOR–94, INRIA Rocquencourt, Septembre 1990.

[WB84] G. T. J. Wuu and A. J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proc. 3rd ACM Symp. on PODC*, pages 232–242, 1984.

[XN93] J. Xu and R. H. B. Netzer. Adaptive independent checkpointing for reducing rollback propagation. In *Proc. 5th IEEE Symposium on Parallel and Distributed Processing*, pages 754–761, Dallas, December 1993.

[YT88] M. Young and R. Taylor. Combining static concurrency analysis with symbolic execution. *IEEE Transactions on Software Engineering*, 14(10), October 1988.

[ZPS88] M. Zimmermann, F. Perrenoud, and A. Schiper. Understanding concurrent programming through program animation. In *Proc. of the 19th SIGCSE Technical Symposium on Computer Science Education*, Atlanta, Georgia, February 1988.