# Ré-éxécution et analyse de calculs répartis

Michel Raynal

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Ré-exécution et analyse de calculs répartis*

Michel Raynal

**N° 2257**

Avril 1994

PROGRAMME 1

Architectures parallèles,

bases de données,

réseaux et systèmes distribués


ISSN 0249-6399

INRIA
RENNES

# Ré-exécution et analyse de calculs répartis

Michel Raynal

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet Adp

Rapport de recherche n° 2257 — Avril 1994 — 26 pages

**Résumé :** La mise au point (*debugging*) de programmes répartis est une activité délicate et difficile. Elle s'apparente à la fois à la mise au point des programmes séquentiels (que l'on ré-exécute pour trouver des erreurs et traquer les comportements incorrects) et au test de protocoles (qui mettent en jeu des entités communiquant par messages).
On examine ici deux points fondamentaux de la mise au point des programmes répartis. Le premier concerne les techniques de ré-exécution dont le but est d'obtenir, pour un programme donné, des exécutions réparties identiques à une exécution initiale. Le second point abordé est la détection de prédicats globaux (stables ou instables) dans les exécutions réparties (un prédicat sur un état global servant à décrire une propriété souhaitée ou non du calcul).
En plus des techniques présentées pour résoudre les problèmes posés, cet article peut être vu comme une introduction à des concepts de base des systèmes répartis (pris au sens *distributed computing systems*).

**Mots-clé :** Détection de propriétés, Exécution répartie, Mise au point, Non-déterminisme, Précédence causale, Programme réparti, Ré-exécution

*(Abstract: pto)*

Article invité à JISI 94, Tunis, Mai 1994.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

# Replay and Analysis of Distributed Executions

**Abstract:** Distributed programs are much more difficult to design, understand and implement than sequential or parallel ones. This is mainly due to the uncertainty created by the asynchrony inherent to distributed machines. So appropriate concepts and tools have to be devised to help the programmer of distributed applications in his task.
This paper is motivated by the practical problem called distributed debugging. It presents concepts and tools that help the programmer to analyze distributed executions. Two basic problems are addressed: replay of a distributed execution (how to reproduce an equivalent execution despite of asynchrony) and the detection of a stable or unstable property of a distributed execution. Concepts and tools presented are fundamental when designing an environment for distributed program development.

**Key-words:** Causal precedence, Debugging, Distributed execution, Distributed program, Non determinism, Properties detection, Replay.

# 1 Introduction

Les outils de validation de logiciels peuvent être classés en deux catégories. Dans la première, on trouve les techniques de vérification 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.

Comme on peut le voir, la deuxième catégorie de ces techniques ne s'intéresse qu'à une exécution particulière. Ceci est justifié dans un certain nombre de situations : soit l'on n'a pas le programme à sa disposition (approche dite "boite noire" du test de protocoles) ; on ne dispose alors que de sa trace d'exécution. Soit le nombre des états dans lesquels peut passer le programme est potentiellement infini et la construction du graphe des états potentiellement accessibles (ou une de ses projections, fonction de la propriété étudiée) n'est pas réalisable. Soit, plus simplement, les techniques actuelles de vérification ne permettent pas d'affirmer la correction de toutes les exécutions possibles d'un programme donné.

Les outils et techniques permettant d'étudier une exécution particulière d'un programme rentrent donc dans la catégorie des outils d'analyse d'exécution. On s'intéresse ici à l'analyse de la dynamique des exécutions réparties. Une telle exécution résulte de l'évolution de processus ne communiquant que par messages, ces processus et leur coopération étant définis par un programme[1] qualifié de réparti. Analyser une exécution demande généralement d'être capable de la reproduire. Or si cela est trivial pour un programme séquentiel déterministe, il n'en est pas de même pour un programme réparti ; les temps de transit des messages peuvent être différents d'une exécution à l'autre, et en conséquence les processus peuvent ne pas produire les mêmes séquences de traitement lors de chaque exécution. Ceci est appelé l'effet "relativiste" des exécutions réparties. Un premier problème à résoudre, si l'on désire analyser une exécution répartie, consiste donc à être capable de la reproduire, la seule connaissance du programme dont elle est issue n'étant pas suffisante. Ce problème est appelé *problème de la ré-exécution répartie*.

Le second problème fondamental est celui de la détection de propriétés de calculs répartis. Contrairement à un calcul séquentiel ou parallèle où il y a "unité de temps" (il existe une horloge globale qui peut servir de référence) et "unité de lieu" (il existe

1. Pour simplifier l'exposé, et sans en altérer la généralité, on suppose que le programme inclut ses données d'entrée, s'il en a.

une mémoire centrale accédée par le ou les différents processus) une exécution répartie est caractérisée par l'absence de référentiel temporel ou spatial unique qui pourrait servir de lieu d'observation privilégié de cette exécution, lieu à partir duquel on pourrait déclarer que telle propriété a été ou non vérifiée lors de l'exécution. La répartition du calcul sur plusieurs sites et l'incertitude sur les délais de communication peuvent en permettre plusieurs observations également cohérentes mais distinctes les unes des autres. Quelle signification donner, dès lors, à la question : "cette exécution répartie satisfait-elle cette propriété ?". En effet tel observateur peut avoir vu le calcul dans tel état satisfaisant la propriété alors que tel autre peut ne pas avoir vu d'état la satisfaisant ! Répondre à la question demande donc d'une part de définir précisément ce qu'est une observation d'un calcul réparti et d'autre part de définir des règles de satisfaction, d'une propriété par un calcul, qui soient "réalistes" au sens où elles peuvent être facilement implémentées et ont un sens "pratique" pour l'utilisateur, c'est-à-dire qui permettent d'analyser qualitativement le calcul et, si tel est le but recherché, d'en réaliser la mise au point.

Cet article s'intéresse donc à la ré-exécution de calculs répartis et à la détection de leurs propriétés lorsque celles-ci sont exprimées par des prédicats sur leurs états globaux. Il se décompose en 3 parties principales. La partie 2 présente le modèle de calculs répartis. Différents niveaux d'abstraction associés à un calcul sont introduits ; le niveau de Lamport met l'accent sur les événements de communication et est fondamental pour la ré-exécution ; le niveau de l'utilisateur met l'accent sur les événements qui changent les valeurs de certaines variables et est fondamental pour la détection de propriétés. La partie 3 présente les techniques de base pour ré-exécuter, de façon identique à une première exécution, un programme réparti. La partie 4 s'intéresse à la détection de propriétés ; le treillis des états globaux possibles associés à une exécution répartie est introduit ainsi que le concept d'observation ; plusieurs significations sont ensuite proposées pour la question énoncée précédemment ("telle éxécution satisfait-elle cette propriété?"). Une notion originale, le concept d'état global inévitable, est introduite. Des algorithmes de détection de propriétés sont également présentés.

Au travers cet article de synthèse, des notions fondamentales d'algorithmique répartie, destinées aux systèmes distribués sont présentées. Ce travail s'inscrit dans nos activités de conception et de mise en oeuvre d'un environnement pour la programmation répartie [17].

# 2 Les calculs répartis

## 2.1 Programmes répartis

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. L'ensemble de ces processus coopèrent à un but commun ("unité d'action") qui définit la sémantique du programme réparti.
Un processus peut exécuter 3 types d'actions. Une action interne ne met pas en jeu de messages. Une action d'émission consiste à envoyer un message à un processus donné ; elle est non bloquante et donc toujours possible. Une action de réception consiste à recevoir un message en provenance de n'importe quel processus ; elle est bloquante jusqu'à l'arrivée d'un message. Ce schéma de communication peut être exprimé, au niveau d'un langage par le concept de port non typé. Tout processus est doté d'un port d'entrée via lequel il reçoit les messages qui lui sont envoyés ; tout processus peut envoyer des messages à n'importe quel autre processus en les adressant à son port d'entrée. En d'autres termes l'envoi est sélectif (on nomme le destinataire), la réception ne l'est pas (on attend un message). Le modèle de programmation est donc le modèle réparti asynchrone à communication bi-point, le graphe des communications étant supposé complet.

## 2.2 Le système sous-jacent

Le système sous-jacent, qui exécute les programmes répartis, est composé de processeurs, dotés de mémoires locales, qui peuvent échanger des messages. On suppose dans un souci de simplicité, qu'il y a un processus par processeur. Il n'y a ni horloge globale, ni mémoire centrale partagée par les processeurs. Les messages sont échangés via un réseau de communication, modélisé par des canaux logiques. Ceux-ci sont fiables (pas de perte, ni d'altération) ; ils peuvent être ou non FIFO ; les délais de transmission sont finis mais a priori arbitraires.

## 2.3 Evénements primitifs et niveau de Lamport

L'exécution d'un processus $P_i$ produit une séquence d'événements *primitifs*. Un tel événement est soit un événement interne (exécution d'une action interne), soit un événement de communication (exécution d'un envoi ou d'une réception de messages). Cette séquence est appelée l'histoire $H_i$ de $P_i$ :

$$H_i = e_i^o \; e_i^1 \; e_i^2 \; ... \; e_i^x \; e_i^{x+1} ...$$

où $e_i^x$ est le $x$-ième événement primitif exécuté par $P_i$ ; $e_i^0$ est un événement fictif qui initialise les variables locales de $P_i$. Les événements sont instantanés.
Soit $H$ l'ensemble de tous les événements et $\stackrel{e}{\rightarrow}$ la relation binaire sur les événements, définie par Lamport, et appelée *précédence causale* [19] [2] :

$$e_i^x \stackrel{e}{\rightarrow} e_j^y \Leftrightarrow \begin{cases} i = j \textbf{ et } y = x+1 \\ \textbf{ou } e_i^x \text{ est l'émission d'un message et } e_j^y \text{ sa réception} \\ \textbf{ou } \exists\ e_k^z\ :\ e_i^x \stackrel{e}{\rightarrow} e_k^z \textbf{ et } e_k^z \stackrel{e}{\rightarrow} e_j^y \end{cases}$$

Quand on considère les événements primitifs, une exécution répartie peut être représentée par un ensemble partiellement ordonnée (poset) $\widehat{H} = (H, \stackrel{e}{\rightarrow})$. Ce poset définit le calcul à un niveau d'abstraction que nous appelons *niveau de Lamport* (la caractéristique fondamentale de ce niveau est d'inclure tous les événements de communication). La figure 1 montre une exécution répartie au niveau de Lamport, dans le diagramme classique espace-temps (les événements sont dénotés par les points noirs et blancs sur les axes qui représentent l'évolution des processus ; les messages sont dénotés par les flèches qui créent des relations de causalité entre événements produits par des processus différents). Ce niveau d'abstraction est fondamental pour résoudre le problème de la ré-exécution des programmes répartis.

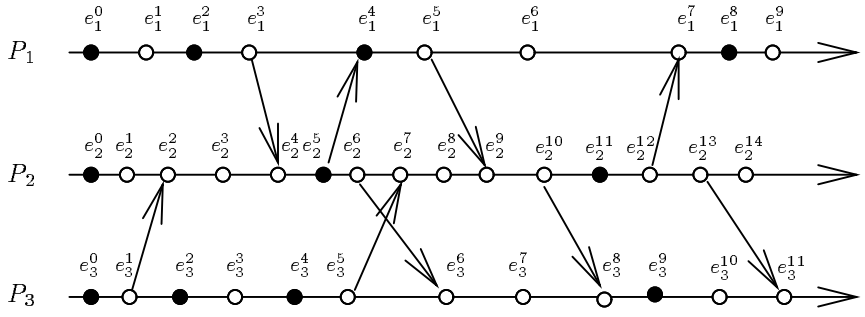


FIG. 1 - *Une exécution vue au niveau de Lamport.*

2. Cette relation est parfois appellée *happened before*; cette terminologie fait référence au temps logique, le seul a être pertinent dans un contexte asynchrone.

## 2.4 Evénements significatifs et niveau de l'utilisateur

Selon le problème qu'il a à résoudre, seulement un sous-ensemble des événements primitifs sont *significatifs* pour l'utilisateur. Si l'on considère par exemple la détection d'un prédicat global portant des variables de plusieurs processus[3] seules les modifications de ces variables constituent des événements significatifs, ce sont en effet les seuls à pouvoir modifier la valeur de vérité du prédicat. On appelle $R$ l'ensemble des événements significatifs (dans la figure 1, $R$ est constitué des événements dénotés par les points noirs).

L'ensemble partiellement ordonné $\widehat{R} = (R, \stackrel{e}{\rightarrow})$ définit l'exécution répartie au *niveau de l'utilisateur* (ou de façon équivalente au niveau du prédicat concerné [13]). Il est important de constater que grâce à la transitivité de $\stackrel{e}{\rightarrow}, \widehat{R}$ hérite de la précédence causale induite par les événements de communication, même si ceux-ci ne sont pas considérés comme significatifs et à ce titre n'appartiennent pas à $\widehat{R}$. La figure 2 décrit, dans un diagramme espace-temps, l'ordre $\widehat{R} = (R, \stackrel{e}{\rightarrow})$ associé à l'exécution de la figure 1.

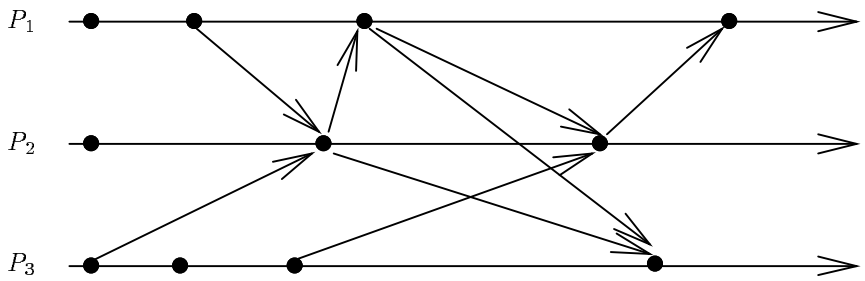


FIG. 2 - *Une exécution vue au "niveau d'un prédicat".*

3. Par exemple $x_1+x_2+...+x_n < k$ où $x_i$ est une variable locale au processus $P_i$. Les modifications de $x_i$ constituent les événements significatifs de $P_i$.

### 2.5 Quelques définitions

Les définitions suivantes s'appuient sur la relation de précédence causale $\stackrel{e}{\rightarrow}$, et s'appliquent donc à $\widehat{H}$ et à $\widehat{R}$.

- deux événements $e_i$ et $e_j$ sont *indépendants* ($e_i \parallel e_j$) si, et seulement si, chacun ne précède pas causalement l'autre :

$$e_i \parallel e_j \Leftrightarrow \neg(e_i \stackrel{e}{\rightarrow} e_j) \textbf{ et } \neg(e_j \stackrel{e}{\rightarrow} e_i)$$

- le passé d'un événement est constitué de tous les événements qui le précèdent causalement :

$$PASSE(e_i) = \{e_j \mid e_j \stackrel{e}{\rightarrow} e_j\}$$

## 3 Ré-exécutions equivalentes

### 3.1 Le problème à résoudre

Comme indiqué dans l'introduction, on suppose que les données d'entrée de chaque processus font partie de son code. Considérons le programme suivant composé de 3 processus :

$$\begin{array}{lcl} P_1 & : & \ldots envoyer\ (m_1) \text{ à } P_2 \ldots \\ P_2 & : & \ldots recevoir\ (x)\ ;\ recevoir\ (y)\ ;\ \ldots \\ P_3 & : & \ldots envoyer\ (m_3) \text{ à } P_2\ \ldots \end{array}$$

Les deux exécutions décrites à la figure 3 sont possibles pour ce programme ; les temps de transfert des messages n'y sont pas les mêmes d'une exécution à l'autre. On dit qu'il y a un *conflit de messages* [24].

**Définition :** *messages conflictuels*
Deux messages sont en conflit (*racing messages*) s'ils ont même destinataire et si, lors d'une autre exécution du programme, ils auraient pu être reçus dans un ordre différent (en d'autres termes leur ordre de réception ne dépend pas seulement du programme mais également des vitesses d'exécution des processeurs et des temps de transfert sur les canaux).

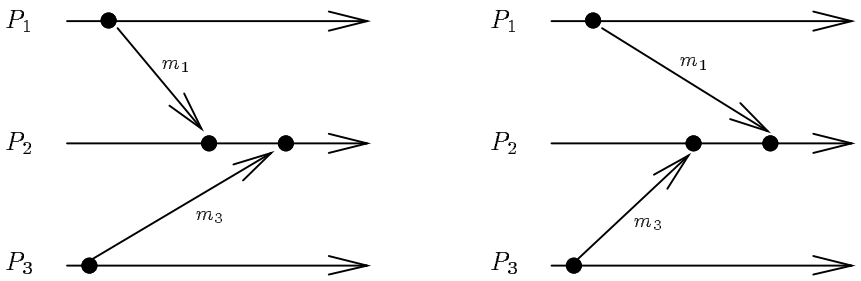


FIG. 3 - *Deux exécutions d'un même programme.*

**Définition :** *exécutions équivalentes*
Deux exécutions d'un même programme réparti sont équivalentes si elles produisent au niveau d'abstraction concernée (niveau de Lamport ou niveau de l'utilisateur) le même ensemble partiellement ordonné d'événements.

On voit donc que, si obtenir une exécution équivalente à une exécution donnée est un problème trivial pour un programme séquentiel déterministe, il n'en est plus de même pour un programme réparti. Obtenir une telle exécution nécessite la prise d'informations, lors de l'exécution de référence, qui serviront à piloter les nouvelles exécutions de façon à ce qu'elles reproduisent le même ordre partiel, c'est-à-dire ce qu'il est convenu d'appeler le même *comportement*.

## 3.2 La méthode de Curtis et Wittie

**Les informations mémorisées**
Proposée dès 1982, cette méthode consiste à associer à tout processus $P_i$, un journal dans lequel est mémorisée la séquence des événements produits par $P_i$. Bien sûr seuls les événements de cette séquence liés aux choix non-déterministes du processus ou

à l'arrivée des messages ont à être mémorisés. Ce journal est ensuite utilisé pour piloter les ré-exécutions [7] . Il est composé de deux parties :

**i)** $jn\_nondet_i$ : mémorise une séquence d'informations relatives aux actions internes à $P_i$ dont le résultat n'est pas déterministe. Ceci se produit, par exemple, dans les cas suivants :

- $P_i$ lit la valeur de l'horloge du processeur : celle-ci est mémorisée.
- $P_i$ choisit telle branche dans une alternative non-déterministe : l'identité de la branche (numéro) est mémorisée.

**ii)** $jn\_msg_i$ : mémorise la séquence des identités des messages reçus par $P_i$. L'identité d'un message est constituée d'un couple $(j, s_j^i)$ ; $j$ désigne l'émetteur du message et $s_j^i$ désigne son numéro d'ordre parmi les messages envoyés par $P_j$ à $P_i$.

**La constitution des journaux**
Afin de pouvoir constituer les journaux, chaque processus $P_i$ est enrichi, à la compilation, d'instructions supplémentaires qui rempliront $jn\_nondet_i$. Chaque $P_i$ est de plus pourvu d'un tableau $ns_i[1..n]$ ; $ns_i[j]$ lui sert de générateur de numéros de séquence pour les messages qu'il envoie à $P_j$. Tout message transporte son numéro de séquence, qui joint à l'identité de l'émetteur constitue sa propre identité. Comme indiqué précédemment l'identité $(j, s_j^i)$ d'un message reçu par $P_i$ est placée dans $jn\_msg_i$.

**Le pilotage de la ré-exécution**
Durant une ré-exécution, chaque processus $P_i$ lit dans $jn\_nondet_i$ les valeurs dépendantes de son environnement et les résultats des choix non-déterministes qu'il exécute. Les messages reçus sont stockés dans un tampon. Lorsqu'il exécute une réception de message, $P_i$ lit dans $jn\_msg_i$ l'identité du prochain message qu'il doit consommer, et reste bloqué jusqu'à ce que ce message soit dans le tampon. Ceci nécessite, bien sûr, que les messages, lors de ré-exécutions, transportent leurs identités comme lors de l'exécution initiale.

## 3.3 Ré-exécuter tout ou partie du programme

Le schéma précédent mémorise les identités de tous les messages, mais non leurs contenus. En effet, cela n'est pas nécessaire puisque les contenus des messages sont

à nouveau calculés lors de la ré-exécution. Il n'en est plus de même si l'on désire ne ré-exécuter qu'un sous-ensemble $S$ des processus. Dans ce cas, on étend le schéma précédent en mémorisant l'identité et le contenu de tout message émis par un processus $P_j \notin S$. Ceci permet de ré-exécuter les processus de $S$ comme dans l'exécution d'origine ; la seule contrainte est la connaissance préalable de $S$. Il est ainsi possible de ne ré-exécuter qu'un seul processus, toutes les interactions avec son environnement étant répertoriées dans ses deux journaux. Par ailleurs, si un processus $P_i$ a un comportement interne déterministe, seul le journal $jn\_msg_i$ est nécessaire. On constatera de plus que la ré-exécution peut se faire sur le même système (réseau local, machine répartie) que l'exécution initiale ou sur un système différent (sur une seule station de travail par exemple) ; en effet la ré-exécution n'a besoin de reproduire que le même ordre partiel. Le lecteur intéressé trouvera dans [7,17,20,21] la description de systèmes de ré-exécution fondés sur le principe que nous venons de voir.

## 3.4 Le traçage optimal de Netzer et Miller

**Quelle optimalité ?**

Reprenons l'exemple de la figure 3 qui montre deux exécutions différentes d'un même programme. Cet exemple montre qu'il n'est pas nécessaire de tracer et de mémoriser les identités des deux messages conflictuels $m1$ et $m3$ ; seul l'un des deux a besoin d'être tracé, $m3$ par exemple. Si lors de la ré-exécution $P_2$ n'accepte le message tracé $m3$ que lors de sa deuxième réception, la seconde exécution de la figure 3 ne peut se produire et la ré-exécution produira nécessairement le même ensemble d'événements partiellement ordonné que celui qui a été produit par la première exécution. Considérons maintenant la figure 4 : l'exécution décrite ne présente pas de messages conflictuels ; $m1$ et $m3$, bien que tous deux reçus par $P_2$, ne peuvent être reçus dans l'ordre inverse : $r1 \in PASSE(s3)$. Aucun des messages n'a besoin d'être tracé pour reproduire cette exécution.

Ce sont ces constatations qui ont amené Netzer et Miller à définir un traçage optimal pour la ré-exécution [24], l'optimalité recherchée étant la réduction maximale de la taille des journaux qui serviront à piloter les ré-exécutions de façon à ce qu'elles soient identiques à l'originale. On va présenter cette technique. Elle réduit considérablement le nombre de messages tracés (qui ne représente alors que 1% à 15% du nombre total des messages [24]) au prix d'un traitement supplémentaire sur les informations de contrôle associé aux messages tracés).

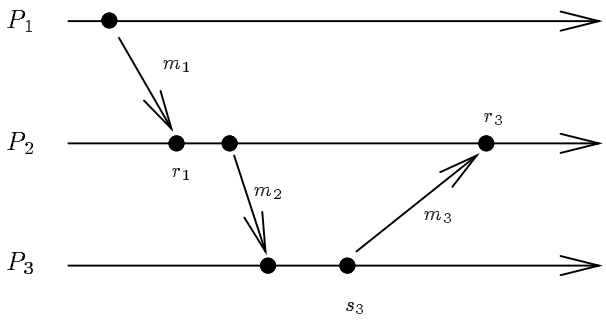


FIG. 4 - *Exécution sans messages conflictuels*

**Contexte**

On suppose, dans un souci de simplification et sans perte de généralité, que chaque processus $P_i$ a un comportement interne déterministe (pas de tirage de nombre aléatoire avec un germe extérieur au programme, pas de lecture d'horloge locale physique, pas d'instruction de choix non-déterministe, etc) ; ceci permet d'éliminer tout ce qui a trait aux journaux $jn\_nondet_i$ et donc de se consacrer au non-déterminisme dû aux seuls messages échangés.

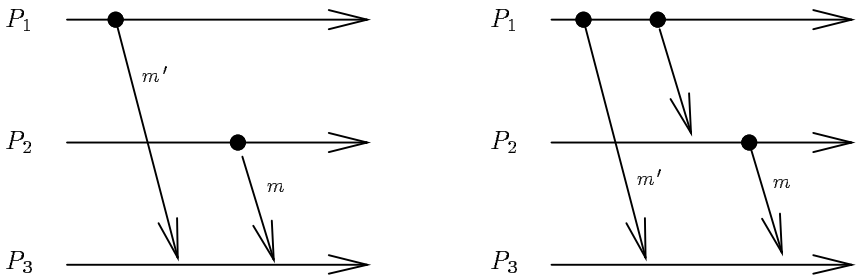


FIG. 5 - *Messages conflictuels*

**La technique de traçage**

Seuls certains messages conflictuels ont besoin d'être tracés afin d'être traités dans le même ordre lors d'une ré-exécution [24]. C'est au récepteur $P_i$ d'un message $m$, de tester, lors d'une exécution initiale, si $m$ est en conflit avec le message $m'$ précédemment reçu. La figure 5 illustre les deux situations possibles de conflit. Dans la première les deux messages, indépendants l'un de l'autre[4], sont conflictuels. Dans la seconde, bien que $m$ dépende de $m'$ (car *émission(m')*$\in$*PASSE(émission(m))*, les deux messages sont également conflictuels. On en conclut que les messages indépendants qui ont même destinataire sont toujours conflictuels, par contre les messages non indépendants peuvent être ou non conflictuels. Appelons :

- *prev_rec* : l'événement "dernière réception de message par $P_i$" (celle-ci concernait le message $m'$).
- *émission*$(m)$ : l'événement "émission de $m$".

A la réception de $m$, le processus $P_i$ destinataire sait que les messages $m$ et $m'$ ne peuvent pas être conflictuels si :

$$prev\_rec \xrightarrow{e} émission(m)$$

(c'est le cas de la figure 4 dans laquelle $r_1 \xrightarrow{e} s_3$) ; $P_i$ effectue donc le test suivant[5] pour savoir s'il trace ou non le message reçu :

$$\textbf{si } \neg(prev\_rec \xrightarrow{e} émission(m)) \textbf{ alors } tracer\ m \textbf{ fsi}$$

En effet ce test garantit que si deux messages conflictuels sont reçus par $P_i$, le deuxième reçu sera toujours tracé, ce qui permettra de règler les conflits lors des ré-exécutions ultérieures. Le traçage d'un message par $P_i$ consiste à ajouter son identité dans un journal local $jn\_aux_i$. Une identité de message $m$ est maintenant composée de 4 champs $(j, i, s_j, r_i)$ :

- $j$ est l'identité de l'émetteur, $i$ celle du destinataire.
- $s_j$ est le numéro de séquence de $m$ dans la séquence des messages émis par $P_j$.
- $r_i$ est le numéro de séquence de $m$ dans la séquence des messages reçus par $P_i$.

4. Deux messages sont dits indépendants si leurs émissions constituent deux événements indépendants : $m \parallel m' \Leftrightarrow$ émission$(m) \parallel$ émission$(m')$.

5. La mise en oeuvre opérationnelle de ce test est explicitée en annexe.

Ces identités vont permettre de générer des journaux d'entrée $jn\_sent_i$ que chaque processus émetteur $P_i$ exploitera, lors des futures ré-exécutions, pour marquer un sous-ensemble des messages conflictuels qu'il émettra (ces messages marqués sont donc les messages conflictuels qui étaient arrivés en "deuxième" position lors d'un conflit dans l'exécution initiale, cf. le test de traçage).

**Piloter la ré-exécution**
Les journaux construits lors de l'exécution initiale, $jn\_aux_i$ sont tout d'abord fusionnés et pour tout processus $P_i$ on construit le journal $jn\_sent_i$ qui inclut les identités des messages qu'il a émis, triées dans l'ordre croissant de leur numéro d'ordre d'émission (troisième champ de l'identité).
Tout processus $P_i$ gère, comme lors de l'exécution initiale, deux compteurs : $sent_i$ et $rec_i$ qui comptabilisent le nombre des messages qu'il a respectivement émis et reçus. La ré-exécution est alors assujettie aux 2 règles suivantes :

- Emission par $P_i$ d'un message $m$ :

  $sent_i := sent_i + 1$ ;
  **soit** $(i, j, s_i, r_j)$ *la première identité de* $jn\_sent_i$ ;
  **si** $sent_i = s_i$ **alors**
  % *il s'agit d'un message conflictuel reçu par son destinataire*
  % *en "deuxième" position (cf. le test de traçage)*
  $envoyer(m, r_j)$ à $P_j$ ;
  $supprimer(i, j, s_i, r_j)$ **de** $jn\_sent_i$
  **sinon** % *le message, non conflictuel ou conflictuel et arrivé en première position,*
  % *n'a pas à transporter son numéro de séquence en réception*
  *envoyer* $m$ à $P_j$
  **fsi**

- Consommation par $P_j$ d'un message par $recevoir(x)$ :

  $rec_j := rec_j + 1$ ;
  *les messages sans numéro de séquence sont consommés dans leur ordre d'arrivée ; un message transportant un numéro de séquence* $r_j$ *est consommé lorsque* $rec_j = r_j$.

Seuls certains des messages conflictuels transportent un numéro de séquence $r_j$ (qui indique leur ordre de consommation par le récepteur $P_j$). Ils sont stockés dans un tampon s'ils arrivent alors que $rec_j < r_j$. Les autres messages ne transportent pas de

numéro de séquence, il s'agit de tous les messages non conflictuels et des messages conflictuels non tracés lors de l'exécution initiale[6] ; ces messages sont consommés normalement dans leur ordre d'arrivée. La preuve de correction de cette stratégie est donnée dans [24] (on remarquera qu'un message conflictuel et un message non conflictuel ne peuvent se trouver simultanément dans le tampon : si tel était le cas, ils seraient tous deux conflictuels !).
Il est également montré dans [24] que cette stratégie de traçage est optimale (en nombre minimal de messages tracés) lorsque la relation de conflit entre messages est transitive. Dans les autres cas, le problème du traçage optimal est NP-difficile. La stratégie présentée est donc une bonne stratégie gloutonne (comme indiqué moins de 20% des messages sont généralement tracés). La figure 6 présente une exécution particulière dans laquelle la relation de conflit entre messages n'est pas transitive :

$m1$ et $m2$ sont conflictuels
$m2$ et $m3$ sont conflictuels
$m1$ et $m3$ ne sont pas conflictuels

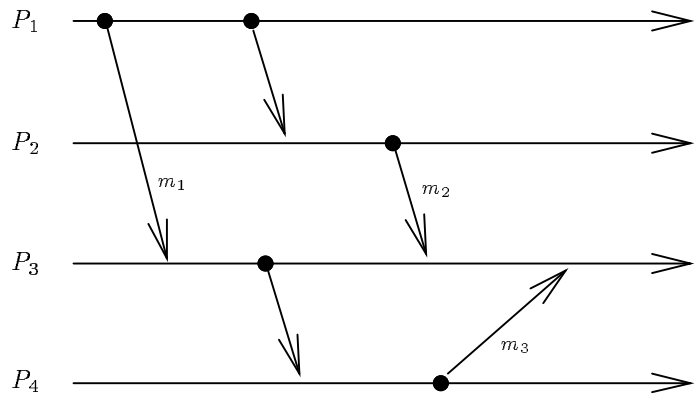


FIG. 6 - *La relation du conflit n'est pas toujours transitive*

Dans cette exécution $m2$ et $m3$ sont tracés par l'algorithme précédent. On peut constater que le traçage de seulement $m2$ peut permettre une ré-exécution identique

6. Rappelons qu'il s'agit des messages conflictuels arrivés en "premier" lors des conflits.

($m3$ n'est pas en conflit avec $m1$). Le lecteur pourra constater que dans une exécution initiale où $m3$ serait reçu avant $m2$, l'algorithme ne traçerait que $m2$ pour piloter les ré-exécutions ultérieures.

### 3.5 Générer des ré-exécutions différentes

Considérons un programme réparti composé de processus au comportement interne déterministe, connectés en anneau unidirectionnel par des canaux FIFO. Il est facile de voir que, pour un jeu d'entrées donné, ce programme ne peut présenter qu'une seule exécution possible.
Il est montré dans [1] que, pour un programme réparti composé de processus au comportement interne déterministe, il existe plusieurs exécutions différentes si, et seulement si, l'une d'entre elles comprend des messages conflictuels. Ce résultat permet d'écrire un algorithme qui reproduit une exécution jusqu'à un point donné puis intervertit deux réceptions de messages conflictuels pour donner naissance à une nouvelle exécution. Des détails sur un tel algorithme seront trouvés dans [1].
Le lecteur intéressé trouvera de plus dans [8] un ensemble de conditions qui garantissent des ré-exécutions identiques à l'originale, dans le cas de primitives de communication très diverses (bloquante ou non, avec ou sans rendez-vous, contrainte par la taille des tampons ou non, etc).

### 3.6 Ré-exécution partielle

Les techniques présentées permettent de reproduire une exécution depuis son état initial. Dans certains cas, on veut reproduire une exécution à partir d'un certain état intermédiaire, sans avoir à tout ré-exécuter depuis le début. Pour cela il est nécessaire, lors de l'exécution initiale, de définir des points de reprise (ou états globaux intermédiaires) à partir desquels la ré-exécution peut être menée [16,26]. La définition de ces points de reprise est un problème de calcul d'un état global cohérent ; cet aspect est abordé dans la partie suivante.

## 4 Détection de propriétés

### 4.1 Type des propriétés étudiées

De nombreux types de propriétés sur une exécution répartie peuvent être définis et étudiés. Parmi celles qui présentent un intérêt pratique pour l'analyse et la mise au point, on peut distinguer celles qui portent sur les flots de contrôle qui composent

l'exécution répartie et celles qui reposent sur la satisfaction d'un prédicat par un de ses états globaux. Dans la première de ces catégories, on trouve notamment les séquences des prédicats [23], les séquences atomiques de prédicats locaux [18] et leur généralisation aux modèles réguliers [12].

On s'intéresse ici aux propriétés qui peuvent être formulées par un prédicat sur un état global d'une exécution répartie. Idéalement, une exécution répartie satisfait un prédicat si elle est passée dans un état global le satisfaisant. Le problème de la détection d'une propriété, durant l'exécution même du programme, provient du fait que l'on ne peut connaître avec exactitude la séquence des états globaux dans lesquels celle-ci est passée. Ceci est dû à l'absence d' "unités de temps et de lieu" comme indiqué dans l'introduction. On ne peut connaître qu'un ensemble des états globaux dans lequel l'exécution a pu passer. Dès lors, répondre à la question "telle exécution satisfait-elle cette propriété?" requiert des interprétations particulières.

Cette partie définit tout d'abord la notion d'état global (partie 4.2). Ensuite sont introduites 2 notions fondamentales pour la détection de propriétés: le treillis des états globaux et le concept d'observation d'une exécution répartie (partie 4.3). Enfin, trois interprétations sont fournies pour répondre à la question précédente (partie 4.4).

## 4.2 Etats locaux et états globaux

### Etats locaux

On se place au niveau d'abstraction défini par le prédicat qui nous intéresse (cf. partie 3.2): les seuls événements observés, dits *significatifs*, sont ceux qui modifient les variables locales de chaque processus intervenant dans le prédicat.

L'événement $r_i^x$ ($x$-ième événement significatif de $P_i$) provoque le changement d'état local de $P_i$ de $s_i^{x-1}$ à $s_i^x$. Les événements sont instantanés. Par contre, un état local $s_i^x$ dure de $r_i^x$ jusqu'à la production par $P_i$ de $r_i^{x+1}$ ; $suivant(s_i^{x-1})$ est un synonyme de $s_i^x$.

Considérons à titre d'exemple l'observation du prédicat $x_1 + x_2 + ... + x_n < k$ où chaque $x_i$ est une variable locale du processus $P_i$ .Les événements significatifs de chaque processus $P_i$ sont les modifications de sa variable locale $x_i$. Un état local de $P_i$ est défini, à ce niveau d'observation, par la valeur de la seule variable $x_i$ et celles des variables d'état internes de $P_i$ au moment de la modification de $x_i$ ; cet état local ne varie pas entre deux événements significatifs de $P_i$.

**Relations entre états locaux**
Deux relations de précédence peuvent être définies sur l'ensemble des états locaux produits par une exécution répartie $\widehat{R}$ [11] :

- précédence faible, notée $\stackrel{w}{\rightarrow}$ :
  $s_i \stackrel{w}{\rightarrow} s_j$ si $s_i$ a débuté avant $s_j$ dans le temps logique défini par la relation de précédence causale entre événements $\stackrel{e}{\rightarrow}$. De manière formelle :

$$s_i^x \stackrel{w}{\rightarrow} s_j^y \Leftrightarrow r_i^x \stackrel{e}{\rightarrow} r_j^y$$

- précédence forte, notée $\stackrel{s}{\rightarrow}$ :
  $s_i \stackrel{s}{\rightarrow} s_j$ si $s_i$ n'existait plus lorsque $s_j$ a commencé à exister (par rapport au temps logique défini par $\stackrel{e}{\rightarrow}$). De façon formelle :

$$s_i^x \stackrel{s}{\rightarrow} \Leftrightarrow r_i^{x+1} \stackrel{e}{\rightarrow} r_j^y \textbf{ ou } s_j^y = next(s_i^x)$$

Dans le cas de la précédence faible, les deux états locaux peuvent co-exister à un instant donné ; ce ne peut être le cas avec la précédence forte. La figure 7 montre les états locaux significatifs issus de l'exécution donnée à la figure 1 ; on a :

$$\begin{array}{c} s_1^1 \stackrel{w}{\rightarrow} s_2^1 \\ s_1^1 \stackrel{s}{\rightarrow} s_2^2 \\ \neg(s_2^1 \stackrel{s}{\rightarrow} s_3^3) \end{array}$$

**Etat global cohérent**
Intuitivement, un état global cohérent est un état global, comprenant un état local de chaque processus, qui a pu exister au cours de l'exécution. Formellement, un état global cohérent $\Sigma = (s_1, ..., s_n)$, où $s_i$ est un état local de $P_i$, est caractérisé par la relation suivante :

$$\forall\ i \neq\ j\ :\ \neg\ (s_i \stackrel{s}{\rightarrow} s_j)$$

## 4.3 Treillis et observations

L'ensemble des états globaux cohérents qui peuvent être associés à une exécution répartie est structuré en *treillis* [22]. La figure 8 montre le treillis des états globaux cohérents associée à l'exécution décrite par la figure 7.

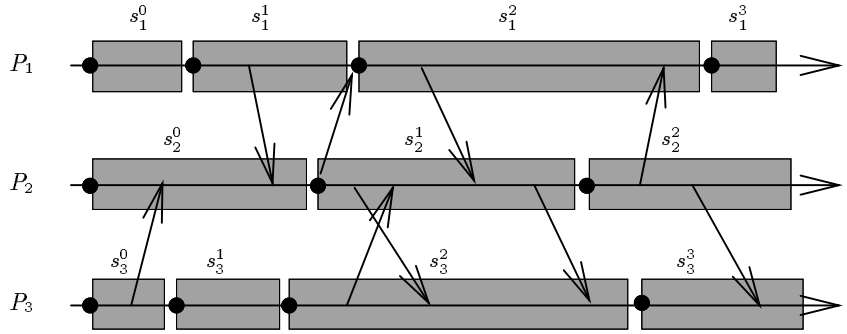


FIG. 7 - *Etats locaux au niveau de l'utilisateur*

Une *observation*[7] d'une exécution répartie $\widehat{R}$ peut être vue comme une séquence d'états globaux et d'événements (significatifs) qui auraient été produits en exécutant séquentiellement $\widehat{R}$. Il s'agit de ce qu'aurait pu voir de l'exécution un observateur qui n'en percevrait qu'un événement à la fois. Plus formellement, une observation $O$ est une séquence :

$$O = \Sigma^0 \; r^1 \; \Sigma^1 \; r^2 \; \Sigma^2 ... r^{i1} \; \Sigma^i \; r^{i+1} ...$$

telle que :

- tous les événements significatifs (et eux seuls) apparaissent dans un ordre cohérent[8] avec $\widehat{R}$.
- $\Sigma^i$ est l'état global obtenu à partir de $\Sigma^{i-1}$ par l'exécution de $r^i$ ($\Sigma^0$ étant l'état initial).

7. On se limite ici aux observations séquentielles. Des observations parallèles d'une exécution répartie sont envisageables [5,14,15,25].

8. En d'autres termes, la séquence $r^1 \; r^2 ... r^i ...$ est une extension linéaire de $\widehat{R}$.

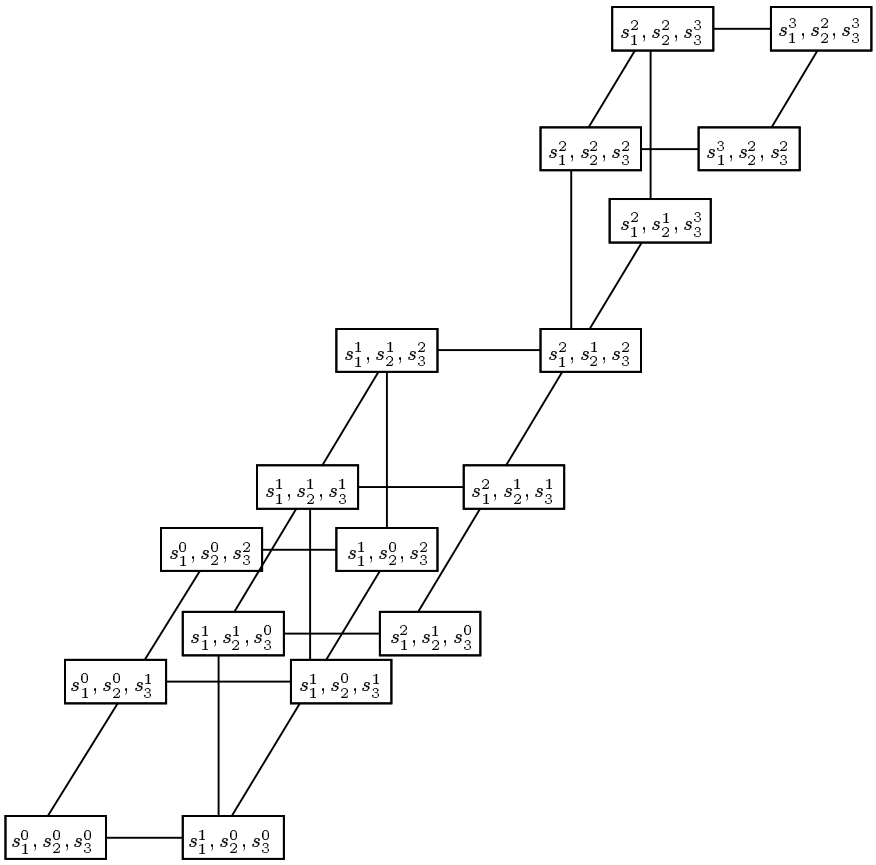


FIG. 8 - *Treillis des états globaux*

[25] et [2] ont montré que tout chemin dans le treillis correspond à une observation et réciproquement. Ceci va permettre d'exprimer la satisfaction de propriétés à l'aide du concept d'observation, le treillis correspondant pouvant aider à calculer la satisfaction de la propriété.

## 4.4 Satisfactions d'une propriété

**Le cas des propriétés stables**
Une propriété stable est une propriété qui, une fois vérifiée, reste vraie. C'est le cas par exemple de la terminaison et de l'interblocage. Une telle propriété est donc indépendante d'une observation particulière : si elle est vraie dans un état d'une observation, elle est ou sera vraie pour toutes les observations.
La détection d'une propriété stable peut donc être réalisée à l'aide d'un algorithme de calcul d'instantanés d'états globaux (*snapshot*), tel que celui de Chandy et Lamport [4]. La détection consiste à calculer des états globaux jusqu'à ce que soit l'exécution termine, soit on trouve un état global qui satisfait la propriété.

**Le cas des propriétés instables**
Une propriété instable peut ne pas rester vraie une fois vérifiée. C'est par exemple le cas des propriétés suivantes : "il y a plusieurs processus en section critique" ou "la charge du sytème est supérieure à 75%". Il n'est pas possible d'utiliser un algorithme de calcul d'états globaux tels que celui de Chandy et Lamport pour détecter les propriétés instables [15]. En effet, cet algorithme délivre une séquence d'états du treillis qui ne sont pas nécessairement consécutifs ; il est donc possible que la propriété ne soit vérifiée que dans les états globaux que l'algorithme de Chandy et Lamport ne donne pas en résultat.
C'est ce problème qui a motivé l'analyse des propriétés au moyen du treillis qui contient tous les états globaux dans lequel l'exécution est passée ou a pu passer.

**Les modalités $POS$ et $DEF$**
Introduites par Cooper et Marzullo [6], il s'agit de deux réponses à la question "l'exécution $\widehat{R}$ satisfait-elle le prédicat global $\Phi$".

- $\widehat{R}$ satisfait $POS\ \Phi$ (noté $\widehat{R} \models POS\ \Phi$) s'il existe au moins une observation $O$ de l'exécution répartie $\widehat{R}$ qui contient un état global $\Sigma$ dans lequel $\Phi$ est vrai ($\Sigma \models \Phi$). En d'autres termes :

$$\widehat{R} \models POS\ \Phi \Leftrightarrow \exists\, \Sigma \in treillis : \Sigma \models \Phi$$

- $\widehat{R}$ satisfait $DEF\ \Phi$ (noté $\widehat{R} \models DEF\ \Phi$) si toute observation $O_x$ contient un état global $\Sigma_x$ qui satisfait $\Phi$. En d'autres termes :

$$\widehat{R} \models DEF\,\Phi \Leftrightarrow (\forall\, O_x\, \textit{chemin maximal du treillis} : (\exists\, \Sigma_x \in O_x : \Sigma_x \models \Phi))$$

La satisfaction $POS$ est intéressante pour détecter une erreur potentielle exprimée par un prédicat $\Phi$. La satisfaction $DEF$ est plus intéressante pour montrer qu'une propriété est respectée.
L'inconvénient des deux interprétations précédentes de la question "$\widehat{R}$ satisfait-elle $\Phi$ ?" réside dans le fait qu'il faille construire le treillis des états globaux, puis le parcourir pour détecter $\widehat{R} \models POS\ \Phi$ ou $\widehat{R} \models DEF\ \Phi$. Des algorithmes de construction du treillis sont donnés dans [6,2,9] ; 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. [6] et [3] présentent des algorithmes de parcours du treillis pour détecter des propriétés.
La taille du treillis (en nombre d'états globaux) peut être exponentielle par rapport au nombre de processus ([2,25]). Ceci réduit considérablement l'intérêt pratique d'une telle approche pour la détection de propriétés instables.

**La modalité** *PROP*
Introduite par Fromentin et Raynal [14], le but de cette règle de satisfaction est d'éviter de construire le treillis tout en ayant un intérêt pratique.

- $\widehat{R}$ satisfait $PROP\ \Phi$ (noté $\widehat{R} \models PROP\ \Phi$) s'il existe un état global $\Sigma$, commun à toutes les observations, qui satisfait $\Phi$.

En d'autres termes la propriété, exprimée par $\Phi$, a été satisfaite, pour tous les observateurs, dans le même état global. Pour formaliser et rendre opérationnelle la définition précédente, le concept d'inévitabilité pour les états globaux a été introduit [14] : un état global est *inévitable* s'il appartient à toutes les observations (il s'agit des points d'articulation du treillis) ; l'état $\Sigma = (s_1^2, s_2^1, s_3^2)$ de la figure 8 est inévitable.
Le théorème suivant est démontré dans [14] (Il est également montré que tout état global inévitable est cohérent) :

$$\Sigma = (s_1, ..., s_n) \text{ inévitable} \Leftrightarrow \forall\, i, j : s_i \xrightarrow{w} suivant(s_j)$$

Grâce à ce théorème, il est possible de construire un algorithme, de complexité $O(n^3)$ où $n$ est le nombre de processus) qui calcule tous les états globaux inévitables.

Comme précédemment, il fonctionne en *pipe-line* derrière l'exécution qui lui transmet les états locaux des processus. Le lecteur intéressé trouvera l'algorithme et sa preuve dans [13].

# 5 Conclusion

Cet article a présenté un état de l'art sur des problèmes et des techniques liés à la ré-exécution répartie et à la détection de propriétés (exprimées par un prédicat sur un état global de l'exécution répartie). Le point de vue adopté a été essentiellement pratique, l'objectif visé étant la mise au point des programmes répartis.
Les concepts et techniques introduits ont également permis d'illustrer l'essence et la difficulté du réparti : absence de référentiel commun à l'ensemble des processus, asynchronisme et interprétations multiples de la question : "cette exécution satisfait-elle cette propriété (instable)?".

**Remerciements**
Je tiens à remercier G. Lelann qui m'a incité à écrire cet article ainsi que toutes les personnes dont les discussions et échanges m'ont permis de mieux préciser les idées ici exposées : O. Babaoğlu, E. Fromentin, V. Garg, M. Hurfin, Cl. Jard, E. Leu, F. Mattern, C. Maziero, K. Marzullo, A. Mostefaoui, N. Plouzeau, A. Schiper et G. Viho.
Par ailleurs, cette synthèse n'aurait pas vu le jour si je n'avais pas travaillé sur l'analyse des exécutions réparties ; ces travaux de recherche ont été effectués dans le cadre du projet Esprit BRA BROADCAST (Basic Research on Advanced Distributed Computing : from Algorithms to Systems) et du projet "Traces" financé par la CNRS.

**Annexe**
On a utilisé, dans cet article, les relations $\xrightarrow{e}$ et $\|$ sur les événements et $\xrightarrow{w}$ et $\xrightarrow{s}$ sur les états locaux. Lorsqu'elles doivent être utilisées dans un algorithme, ces relations doivent trouver une forme opérationnelle afin de pouvoir être exploitées. Cette annexe présente de telles formules fondées sur une adaptation des vecteurs d'horloge de Fidge[10] ou Mattern[22].

- Tout processus $P_i$ est doté d'un vecteur horloge $v_i[1..n]$ initialisé à un vecteur de 0, avec la valeur 1 en position $i$.
- Chaque fois que $P_i$ exécute un événement significatif $r_i$ il incrémente $v_i[i]$ d'une valeur positive (e.g.1). La valeur de $v_i$ sert alors à estampiller l'événement $r_i$ et

l'état local $s_i$ produit. Ces estampilles sont dénotées respectivement par $v(r_i)$ et $v(s_i)$.

- Tout message $m$ transporte la valeur du vecteur d'horloge de son émetteur ; soit $v(m)$ cette valeur.
- Lorsqu'un message $m$ est reçu par $P_i$, son vecteur horloge $v_i$ est mis à jour de la façon suivante :

$$\forall\ k\ \in\ 1..n\ :\ v_i[k] := max(v_i[k], v(m)[k])$$

Avec ce mécanisme d'horlogerie logique, on obtient la "traduction" des relations précédentes dans les formules suivantes (le lecteur trouvera les démonstrations dans [10,22] en ce qui concerne les événements, et dans [11] en ce qui concerne les états locaux). L'indice indique le processus origine de l'événement ou de l'état local.

$$\begin{array}{rcl} r_i \stackrel{r}{\rightarrow} r_j & \Leftrightarrow & v(r_i)[i] \leq v(r_j)[i] \\ r_i \parallel r_j & \Leftrightarrow & v(r_i)[i] > v(r_j)[i] \textbf{ et } v(r_j)[j] > v(r_i)[j] \\ s_i \stackrel{s}{\rightarrow} s_j & \Leftrightarrow & v(s_i)[i] < v(s_j)[i] \\ s_i \stackrel{w}{\rightarrow} s_j & \Leftrightarrow & v(s_i)[i] \leq v(s_j)[i] \end{array}$$

## Références


1 S. Alagar and S. Venkatesan. *Hierarchy in testing distributed programs.* Proc. Int. Workshop AADEBUG 93, May 1993, Springer-Verlag LNCS 749, pp. 101-116.

2 Ö. Babaoğlu and K. Marzullo. *Consistent global states of distributed systems : fundamental concepts and mechanisms.* in Distributed Systems Chapter 4. ACM Press, Frontier Series, (S.J. Mullender Ed.), pp. 55-93, 1993.

3 Ö. Babaoğlu and M. Raynal. *Specification and detection of behavorial patterns in distributed computations.* In Proc. of 4th IFIP WG 10.4 Int. Conference on Dependable Computing for Critical Applications. Springer Verlag Series in Dependable Computing, San Diego, January 1994.

4 K. M. Chandy and L. Lamport. *Distributed snapshots : determining global states of distributed systems.* ACM Transactions on Computer Systems, 3(1):63-75, February 1985.

5 B. Charron-Bost, C. Delporte-Gallet, and H. Fauconnier. *Local and Temporal Predicates in Distributed Systems.* Technical Report, LITP, IBP, Université Paris 7, April 1991, 40 pages.

6 R. Cooper and K. Marzullo. *Consistent detection of global predicates.* In Proc. ACM/ONR Workshop on Parallel and Distributed Debugging, pp. 167-174, Santa Cruz, California, May 1991.

7 R. Curtis and L. Wittie. *Bugnet: a debugging system for parallel programming environments.* Proc. 3rd IEEE Int. Conf. on Dist. Comp. Systems, (1982), pp. 394-399.

8 R. Cypher and E. Leu. *Message-passing semantics and portable parallel programs.* Research Report RJ 9654, IBM San José, (Jan. 1994), 57 p.

9 C. Diehl, Cl. Jard, J.X. Rampon. *Reachability analysis on distributed executions.* Proc. TAPSOFT Conf. 93, Springer-Verlag LNCS 668, (1993), pp.629-643.

10 J. Fidge. *Timestamps in message passing systems that preserve the partial ordering.* In Proc. 11th Australian Computer Science Conference, pp. 55-66, February 1988.

11 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.

12 E. Fromentin, M. Raynal, V. Garg and A. Tomlimson. *On the fly testing of regular patterns in distributed computations.* To appear in 23rd Annual Int. Conf. on Parallel Processing (ICPP 94), Pennsylvania, August 1994.

13 E. Fromentin and M. Raynal. *When all the observers of a distributed computation do agree.* Research Report 794, IRISA, Janvier 1994, 20 pages.

14 E. Fromentin and M. Raynal. *Inevitable global states: a new concept to detect unstable properties of distributed computations in an observer independent way.* Soumis à publication au 6th Symposium IEEE on Par. and Dist. Processing, Dallas, (1994).

15 V.K. Garg and B.P. Waldecker. *Detection of unstable predicates in distributed programs.* In Proc. 12th Int. Conf. on Foundations of Soft. Technology, Springer-Verlag, LNCS 625, (Dec. 1992), pp. 253-264.

16 A. Goldberg, A. Gopal, A. Lowry, R. Strom. *Restoring consistent global states of distributed computations.* ACM Sigplan Notices, Vol. 26, 12, (Dec. 1992).

17 M. Hurfin, N. Plouzeau and M. Raynal. *A debugging tool for distributed Estelle programs.* Journal of Computer Communications, 16(5):328-333, May 1993.

18 M. Hurfin, N. Plouzeau and M. Raynal. *Detecting atomic sequences of predicates in distributed computations.* In Proc. ACM workshop on Parallel and Distributed Debugging, pages 32-42, San Diego, May 1993, (Reprinted in SIGPLAN Notices, Dec. 1993).

19 L. Lamport. *Time, clocks and the ordering of events in a distributed system.* Communications of the ACM, 21(7):558-565, July 1978.

20 T. Le Blanc and J.M. Mellor-Crummey. *Debugging parallel programs with instant replay.* IEEE Trans. on Computers, Vol. C36,4, (April 1987), pp. 471-482.

21 E. Leu, A. Schiper, A. Zramdini. *Efficient execution replay techniques for distributed memory architectures.* Proc. 2d European Dist. Memory Comp. Conf., Springer-Verlag LNCS 487, (1991), pp. 315-324.

22 F. Mattern. *Virtual time and global states of distributed systems.* In Cosnard, Quinton, Raynal, and Robert, editors, Parallel and Distributed Algorithms, pp. 215-226, North-Holland, October 1988.

23 B.P. Miller and J.D. Choi. *Breakpoints and halting in distributed programs.* Proc. 8th IEEE Int. Conf. on Dist. Comp. Systems, (1988), pp. 316-323.

24 R.H.B. Netzer and B.P. Miller. *Optimal tracing and replay for debugging message-passing parallel programs.* Proc. Supercomputing 92, Nov. 1992.

25 Schwartz and F. Mattern. *Detecting Causal Relationships in Distributed Computations : in Search of the Holy Grail.* To appear in Distributed Computing, 7(4),1994.

26 J. Xu and R.H.B. Netzer. *Adaptative independent checkpointing for reducing rollback propagation.* In Proc. 5th IEEE Symposium on Parallel and Distributed Processing, pp. 754-761, Dallas, December 1993.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399