INRIA

UNITE DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP105
78153 Le Chesnay Cedex
France
Tel (1) 39 63 55 11

Rapports de Recherche

N° 945

Programme 1

CONSTRUCTION METHODIQUE D'UN ALGORITHME REPARTI DE DETECTION DE LA TERMINAISON

Jean-Michel HELARY
Michel RAYNAL

Décembre 1988

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042-RENNES CÉDEX
FRANCE
Téléphone: 99 36 20 00
Télex: UNIRISA 950 473 F
Télécopie: 99 38 38 32

Publication Interne n°440 - Décembre 1988 - 18 Pages

Construction méthodique d'un algorithme réparti de détection de la terminaison

Jean-Michel HELARY, Michel RAYNAL
IRISA–Campus de Beaulieu– 35042 Rennes Cédex (France)
E-mail helary@irisa.fr,raynal@irisa.fr

Résumé

La conception des algorithmes répartis doit s'appuyer sur un certain nombre de méthodes, de structures de contrôle et de mécanismes d'abstraction propres au contexte réparti. Après avoir présenté l'un des paradigmes de l'observation répartie: la détection de la terminaison, cet article en propose une solution fondée sur une dérivation méthodique. Cette solution repose sur la définition d'une structure de contrôle répartie abstraite (c'est à dire indépendante d'une mise en œuvre donnée), appelée train de vagues, permettant de réaliser des itérations distribuée. Quelques exemples de mises en œuvre sont proposés. Les outils et la méthode utilisés peuvent être exploités pour résoudre différents problèmes de nature distribuée.

A methodological derivation of a very general distributed algorithm to detect termination

Abstract

Conception of distributed algorithms relies on a number of methods, control structures and abstraction mechanisms peculiar to distributed context. After having set out one of the distributed observation paradigms, namely the termination detection, this paper puts forward a methodically derived solution to this problem. The proposed solution rests on the definition of an abstract (i.e. implementation free) distributed control structure allowing to realize distributed iterations, named wave sequence. Some concrete implementations are shown. Tools and methods presented here can be exploited to solve different distributed problems.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1 I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

Table des matières

1 Introduction

Outre son rôle d'*interpréteur* dans lequel il offre des services aux applications qui lui sont soumises (par exemple services d'exclusion mutuelle, d'accès à des fichiers, de communication, etc), tout système peut également avoir un rôle d'*observateur* de ces mêmes applications. Les observations effectuées peuvent avoir pour but, non seulement l'extraction de mesures destinées à analyser les performances de l'application, mais aussi la détection de propriétés qui en caractérisent les exécutions. L'observation joue ainsi un rôle clé pour l'étude du comportement des applications. Une différence fondamentale entre ces deux rôles tenus par un système réside dans le fait que l'observation ne doit pas perturber le comportement de l'application observée.

On s'intéresse dans cet article à la détection d'une classe particulière de propriétés de stabilité d'une application répartie: les propriétés de repos [CM86]. Une propriété de *stabilité* est définie sur l'état d'une application et est caractérisée par le fait, qu'une fois vérifiée, elle le reste quelle que

soit l'évolution ultérieure de l'application [CL85]. L'application est terminée, telle cellule mémoire est inaccessible, le nombre de messages émis depuis le lancement du système est supérieur à 100, etc, sont des exemples de propriétés stables.

Une propriété de *repos* est une propriété de repos particulère: elle met explicitement en jeu l'état des processus composant l'application [SF86] ; être terminé, être interbloqué sont de telles propriétés : les processus mis en jeu sont alors définitivement inactifs (c'est à dire au repos). Dans cet article, on s'intéresse à la terminaison, car elle constitue le paradigme des propriétés de repos.

La conception de l'algorithme de détection de la terminaison proposé est fondée d'une part sur une dérivation conduisant à utiliser une structure de contrôle itérative répartie et d'autre part sur une méthodologie (analogue à celle qui a prévalu dans la définition des types abstraits de données [LZ74]) en ce qui concerne la définition d'un pas de l'itération. L'algorithme utilise ainsi une structure de contrôle abstraite : le *parcours de réseau* [HR88b, Tel88] indépendamment de ses mises en oeuvre particulières ; selon la mise en oeuvre choisie on obtient alors un algorithme particulier. En un certain sens l'algorithme proposé représente une famille d'algorithmes particuliers (parmi lesquels on retrouve certains algorithmes connus). Indépendamment de l'algorithme général obtenu, cet article présente donc une dérivation et une démarche par niveaux d'abstraction qui peuvent être utilisées pour la conception d'autres algorithmes répartis.

Le §2 présente le modèle auquel obéit l'application observée ; le §3 développe les principes sur lesquels se fondent la solution proposée et la dérivation méthodique de l'algorithme. Celui-ci est présenté au §4. Au §5 diverses mises en oeuvre particulières sont proposées en fonction des choix effectués pour la mise en oeuvre de l'abstraction qu' est un parcours de réseau.

2 Comportement de l'application

L'application dont on veut détecter la terminaison (encore appelée calcul sous-jacent) est composée de n processus (ou sites): P_1, P_2,..., P_n, communiquant à l'aide de messages via des canaux de communication (il n'y a donc pas de mémoire centrale pourvant servir de lieu d'échanges). Les canaux sont supposés fiables (i.e. sans perte, ni altération, ni duplication de messages) ; ils peuvent être ou non fifo, uni- ou bi-directionnels ; les délais de communication sont quelconques mais finis. Le graphe qui modélise l'application est

supposé connexe [Ray87].

A un instant donné un processus P_i est actif ou passif ; cet état est capté par la variable concrète locale $état_i$. Un processus actif peut: a) envoyer des messages à ses voisins, b) devenir passif. Lorsqu'un processus P_i passif reçoit un message il redevient actif. (On suppose, sans perte de généralité, qu'un processus est passif lorsqu'il reçoit un message ; en d'autres termes, tout message ré-active son destinataire).

L'application est dite terminée lorsque tous les processus sont passifs et tous les canaux sont vides.

3 Principes de l'algorithme

3.1 Structure de la solution

Afin de détecter la terminaison introduisons une variable abstraite à valeur booléenne *vide(i,j)* pour chacun des canaux telle que *vide(i,j)* a la valeur *vrai* si, et seulement si, le canal c_{ij} (connectant P_i à son voisin P_j) est vide de messages émis par P_i vers P_j.

Considérons le prédicat suivant, relatif à un processus P_i ($voisins_i$ désigne l'ensemble des processus voisins de P_i):

$$pred_i \equiv \left(état_i = passif \wedge \bigwedge_{j \in voisins_i} vide(i,j) \right)$$

Lorsque $pred_i$ est vrai, P_i est inactif et les messages qu'il a émis vers ses voisins ont été reçus. La propriété de terminaison peut être exprimée par le prédicat *terminé* suivant:

$$terminé \equiv \bigwedge_i pred_i$$

L'analyse précédente suggère donc:

i) d'associer à chaque processus P_i un contrôleur C_i dont le rôle est de calculer la valeur de $pred_i$

ii) puis d'assurer une collecte des valeurs $pred_i$ calculées par les contrôleurs afin de calculer la valeur du prédicat *terminé*.

Le contexte réparti est caractérisé par l'impossibilité, pour un unique observateur, de percevoir simultanément, à un instant donné, les états de plusieurs processus et/ou les états d'un ou de plusieurs canaux [CL85].

En ce qui concerne le point i) cela revient à dire qu'un contrôleur C_i ne peut calculer la valeur exacte de *vide(i,j)* ; à un instant donné quelconque il ne sait pas si les messages émis par P_i ont ou non été reçus (les délais sont arbitraires). Il s'agit donc pour C_i, avec la collaboration de ses voisins C_j, de calculer un prédicat a_pred_i tel que: $a\_pred_i \Rightarrow pred_i$.

En ce qui concerne le point ii) il s'agit de faire coopérer les contrôleurs de manière à réaliser une collecte cohérente c'est à dire telle que, si l'on appelle $collecté(a\_pred_1, \ldots, a\_pred_n)$ le résultat à valeur booléenne d'une collecte, l'on ait:

ii1) propriété de *sureté* (si l'on conclut à la terminaison alors elle est effective):

$$collecté(a\_pred_1, ..., a\_pred_n) \Rightarrow terminé$$

ii2) propriété de *vivacité* (si la terminaison se produit elle sera détectée):

$terminé \Rightarrow$ (il y aura une collecte telle que $collecté(a\_pred_1, ..., a\_pred_n)$)

3.2 Les contrôleurs locaux

Une façon simple pour C_i de savoir si les messages émis par P_i ont été reçus consiste à utiliser un mécanisme d'acquittement systématique des messages et à comptabiliser dans une variable locale $nack_i$ le nombre de messages émis par P_i et non encore acquittés. (D'un emploi très général dans les algorithmes distribués, ces techniques de comptage permettent de capter un état approché (retardé) des canaux, que ceux-ci soient fifo [Awe85a] ou non [DS80, HJPR87, MAT87a, Mat87b]).

On pose donc:

$$a\_pred_i \equiv (état_i = passif) \wedge (nack_i = 0)$$

et l'on a bien: $a\_pred_i \Rightarrow pred_i$.

Remarque.

Le modèle de calcul réparti dit *atomique* [Mat87b] suppose que les temps de calcul sont nuls: seul le transit des messages sur les canaux prend du temps. Dans ce cas a_pred_i se résume à: $nack_i = 0$.

Dans le modèle dit *synchrone* les processus communicants obéissent au protocole du rendez-vous ; il n'y a alors pas de messages en transit. Le prédicat a_pred_i se résume alors à $état_i = passif$ [DFG83].

3.3 La collecte

Collecter les observations effectuées par les contrôleurs locaux nécessite de visiter tous les processus (sites) ; il s'agit donc de réaliser un parcours de réseau; ceci peut être réalisé à l'aide de ce qu'il est convenu d'appeler une *vague* [Sch85, HR88b]. Si l'on ne peut conclure à la terminaison avec les informations recueillies, il est nécessaire de refaire une collecte ; la succession des vagues ainsi engendrées est appelée train de vagues.

Le comportement d'un train de vagues peut facilement être défini à l'aide de 4 primitives indépendamment de toute mise en oeuvre. Les primitives *lancer* et *retour* sont utilisées par le contrôleur C_α, initiateur de la vague. Les primitives *visite* et *faire_suivre* sont utilisées par chacun des autres contrôleurs locaux. Désignons par x_i^p l'événement correspondant à la $p$ième exécution de la primitive x par le contrôleur C_i et par $\rightarrow$ la relation de précédence temporelle [Lam78].

La propriété liée au flot de contrôle supportant la $p$ième vague est spécifiée par:

PF: $\forall p > 0, \forall i \neq \alpha$:
$lancer_\alpha^p \rightarrow visite_i^p \rightarrow faire\_suivre_i^p \rightarrow retour_\alpha^p$

(Si l'on considère un seul contrôleur C_i, ce schéma de contrôle est analogue à un schéma procédural).

La séquentialité entre vagues est spécifiée par:

PS: $\forall p > 0 : retour_\alpha^p \rightarrow lancer_\alpha^{p+1}$

Il s'agit là d'une structure de contrôle répartie abstraite permettant l'expression d'une itération répartie. La logique associée est celle de l'itération séquentielle: le résultat à atteindre peut être exprimé sous forme de la conjonction d'un invariant et d'une condition d'arrêt ; ceux-ci sont des prédicats globaux. L'invariant doit être vrai initialement et à la fin de chaque vague, ainsi peut être établie la correction partielle. Pour l'algorithme de détection qui nous intéresse ici, le problème de l'arrêt (propriété de vivacité) peut être formulé ainsi: lorsque l'application à contrôler est terminée, la condition d'arrêt de l'algorithme de contrôle doit être vérifiée au bout d'un nombre fini de vagues.

Le schéma itératif qu'est le train de vagues peut s'exprimer de la manière suivante:

- comportement de l'initiateur C_α:

répeter
    <calculs locaux ; v:= ... > ;
    lancer *vague (v)* ;
    retour *vague (v')* ;
    <calculs locaux>
jusqua *<condition de terminaison portant sur v'>*

- comportement d'un contrôleur $C_i (i \neq \alpha)$:

lors de la visite *vague (v)*
    <calculs locaux ; v':= ...> ;
    faire_suivre *vague (v')*
    <calculs locaux>

3.4 Dérivation de l'algorithme de détection

Pour toute vague p et tout processus P_i désignons par t_i^p l'instant où l'événement $faire\_suivre_i^p$ se produit, et par t_α^p l'instant où a lieu $retour_\alpha^p$ (on a $t_i^p < t_\alpha^p$). D'autre part, pour toute variable ou expression x_i locale à P_i notons x_i^p sa valeur à l'instant t_i^p. Enfin $terminé^p$ désignera le prédicat global: "à l'instant t_α^p, l'application est terminée". Détecter la terminaison de l'application consiste donc à détecter le passage à vrai du prédicat abstrait et global $terminé^p$. On a alors: $q \geq p \wedge terminé^p \Rightarrow terminé^q$.

Il est clair que l'on a la relation:

$$terminé^p \Rightarrow$$
$$\forall q \left(q > p \Rightarrow \bigwedge_i \left(pred_i^q \wedge P_i \text{ est resté continuement passif entre } t_i^q \text{ et } t_i^{q+1} \right) \right)$$

En effet si l'application est terminée à l'instant t_α^p, elle le sera encore aux instants déterminés par les vagues ultérieures, la terminaison étant une propriété stable. En désignant par $cont\_passif_i^q$ le prédicat local: "entre les 2 événements $faire\_suivre_i^q$ et $faire\_suivre_i^{q+1}$, P_i est resté continûment passif", la relation précédente s'écrit:

$$terminé^p \Rightarrow \forall q \left(q > p \Rightarrow \bigwedge_i (pred_i^q \wedge cont\_passif_i^q) \right)$$

Un algorithme de détection itératif peut alors être fondé sur la propriété "réciproque" suivante:

Proposition P:

$$\left(\bigwedge_i (a\_pred_i^p \wedge cont\_passif_i^p)\right) \Rightarrow terminé^p$$

Démonstration

- 1) Chaque P_i étant resté continûment passif entre t_i^p et t_i^{p+1} et les vagues étant séquentielles (propriété PS) on en déduit que tous les processus étaient passifs à t_α^p (fin de la $p^{ème}$ vague).
- 2) De $a\_pred_i^p$ on déduit qu'à t_i^p l'on a $nack_i = 0$; en d'autres termes, à t_i^p, il n'y a pas de messages en transit émis par P_i ; P_i étant resté continûment passif entre t_i^p et t_i^{p+1} il n'en a pas émis entre t_i^p et t_α^p. Ceci étant vrai pour tout P_i on en conclut qu'à t_α^p il n'y a pas de messages en transit.
- 1+2 $\Rightarrow$ $terminé^p$. □

Cette proposition conduit à prendre les éléments suivants pour construire un algorithme de détection itératif:

Invariant de vague :

$$\bigwedge_i a\_pred_i^p$$

Condition d'arrêt :

$$\bigwedge_i cont\_passif_i^p$$

En effet si une vague p détecte

$$\bigwedge_i cont\_passif_i^p$$

on a d'après l'invariant et la proposition P:

$$\left(\bigwedge_i (a\_pred_i^p \wedge cont\_passif_i^p)\right) \Rightarrow terminé^p.$$

Réciproquement, si l'application se termine, soit p le plus petit entier tel que $terminé^p$; les délais de transmission des acquittements étant finis ou aura au bout d'un temps fini $\bigwedge_i nack_i = 0$, c'est à dire $\bigwedge_i a\_pred_i$. On a donc finalement:

$$terminé^p \Rightarrow \exists q \left(q > p \wedge \left(\bigwedge_i (a\_pred_i^q \wedge cont\_passif_i^q)\right)\right)$$

4 L'algorithme

Chaque contrôleur C_i est muni du contexte local suivant:

var $état_i : (actif, passif)$;
$nack_i$: **naturel**;
$flag_i$: **booléen init vrai ssi** P_i est initialement actif;

Les variables $état_i$ et $nack_i$ servent, comme nous l'avons vu à exprimer a_pred_i; $flag_i$ va servir à exprimer le prédicat $cont\_passif_i$.

Pour maintenir l'invariant de vague chaque contrôleur C_i, lorsque P_i est visité par une vague, attend que a_pred_i soit vrai pour la faire suivre (évènements $faire\_suivre_i^p$.

Pour exprimer $cont\_passif_i$, le contrôleur C_i gère la variable $flag_i$ de la manière suivante: celle-ci est mise à **vrai** après chaque événement *faire_suivre* (on a alors $état_i = passif$) et ne passe à **faux** que si P_i devient actif: en effet, ceci indiquera à la vague suivante que P_i n'est pas resté continuement passif depuis le précédent passage. La valeur de $flag_i$ est collectée par la vague lorsqu'elle quitte P_i.

Le comportement de chaque contrôleur C_i est le suivant en ce qui concerne l'observation de P_i et la collecte nécessaire à la détection.

lorsque P_i **devient passif**
$état_i := passif$

lorsque P_i **envoie** m **à** P_j
% $état_i = actif$ %
$nack_i := nack_i + 1$

lors de la réception de m **depuis** P_j
$état_i := actif$;
$flag_i :=$ **faux**;
envoyer ack **à** C_j

lors de la réception de ack
$nack_i := nack_i - 1$

L'initiateur C_α des vagues effectue:

répéter
  attendre a_pred_α;
  lancer *vague (vrai)*;
  retour *vague (collecté)*;
  $collecté := collecté \wedge flag_\alpha$;
  $flag_\alpha :=$ **vrai**
jusqu'à *collecté*

Les autres contrôleurs C_i effectuent:

lors de la visite *vague (collecté)*
  attendre a_pred_i;
  faire_suivre $vague(collecté \wedge flag_i)$;
  $flag_i :=$ **vrai**

La condition d'arrêt testée par C_α est correcte: celui-ci évalue en effet au retour de la vague p:

$$collecté = \bigwedge_i flag_i^p$$

c'est à dire:

$$\bigwedge_i cont\_passif_i^p.$$

5 Exemples de mises en oeuvre

Selon la mise en oeuvre choisie pour les vagues plusieurs algorithmes "concrets" différents peuvent être obtenus à partir de cet algorithme "abstrait". Nous examinons ici quelques unes de ces mises en oeuvre; toutes doivent bien sûr garantir les propriétés PF et PS du §3.2. Ces structures sur lesquelles sont mises en oeuvre les vagues sont a priori indépendantes du graphe des communications de l'application observée; elles peuvent en utiliser les liaisons ou, au contraire, s'appuyer sur la définition d'un graphe de contrôle qui leur est spécifique.

5.1 Structures à flot de contrôle unidirectionnel

La structure de contrôle uni-directionnelle séquentielle qu'est l'anneau (virtuel) orienté constitue le support le plus simple pour une vague: le flot de

contrôle généré par la vague est matérialisé par un jeton valué (chaque contrôleur C_i est pourvu d'une variable $succ_i$ qui lui indique quel est son successeur sur l'anneau virtuel; voir [HR88b] pour la construction et l'exploitation d'anneaux virtuels).

Les 4 primitives sont alors implémentées par:

pour C_α:

lancer *vague(collecté)* mis en œuvre par:
envoyer *jeton(collecté)* **à** $succ_\alpha$

retour *vague(collecté)* mis en œuvre par:
recevoir *jeton(collecté)*

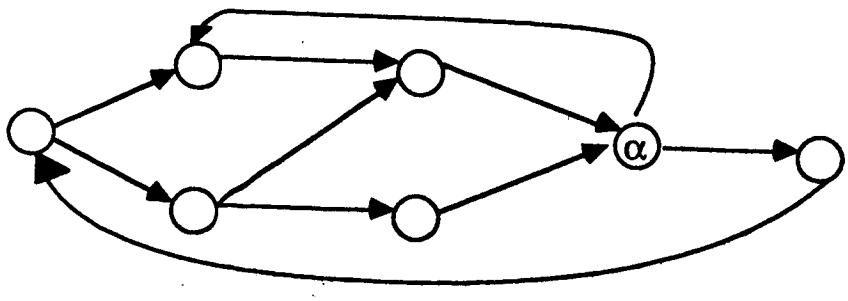
pour C_i:

visite *vague(collecté)* mis en œuvre par:
recevoir *jeton(collecté)*

faire_suivre *vague(collecté)* mis en œuvre par:
envoyer *jeton(collecté)* **à** $succ_i$

Remarque: Dans le cas particulier où les processus P_i de l'application communiquent de façon synchrone et où un anneau sert de support à la vague on retrouve comme mise en œuvre de l'algorithme général présenté, l'algorithme particulier proposé dans [DFG83].

L'anneau unidirectionnel est une structure séquentielle. Il est bien sûr possible d'imaginer des structures de parcours où le flot de contrôle, toujours unidirectionnel, peut progresser en parallèle. Considérons par exemple la structure particulière suivante:



Sur cette structure et avec l'initiateur C_α indiqué, la mise en oeuvre suivante des primitives garantit les propriétés PF et PS:

visite *vague(collecté)* **par** C_i et
retour *vague(collecté)* **par** C_α mis en œuvre par:

soit ci_i = {canaux entrants de C_i};
collecté:= **vrai**;
$\forall x \in ci_i$ **faire:**
attendre *jeton(col)* **sur** x;
collecté:= *collecté* $\wedge$ *col*
fait

lancer *vague(collecté)* **par** C_α et
faire_suivre *vague(collecté)* **par** C_i mis en œuvre par:

soit co_i = {canaux sortants de C_i};
$\forall x \in co_i$: **envoyer** *jeton(collecté)* **sur** x

5.2 Structures à flot de contrôle bidirectionnel

Une vague peut être mise en oeuvre sur une structure où des messages de contrôle circulent dans les 2 sens. L'exemple classique d'un tel support de vagues est uen arborescence de racine C_α. Si les canaux de l'application sont bidirectionnels il est toujours possible de construire une telle structure sur le réseau des processus de l'application observée [HR88b]; chaque contrôleur C_i est alors pourvu d'un ensemble $fils_i$ (éventuellement vide) et d'un père ($père_i$) qui définissent sa position dans l'arborescence de contrôle.

Le lancement de la vague est effectué par C_α, qui envoie un message *aller* vers chacun de ses fils. Lorsque C_i est *visité* par une vague (réception de *aller*) il propage le message *aller* vers chacun de ses fils. *Faire suivre* une vague consiste à la faire remonter vers son père à l'aide d'un message *retour* après avoir reçu un tel message de chacun de ses fils.

Pour C_α:
lancer *vague(collecté)* mis en œuvre par:

$\forall x \in fils_\alpha$ **envoyer** *aller* **à** C_x

retour *vague(collecté)* mis en œuvre par:

collecté:= **vrai**;
$\forall x \in fils_\alpha$ **faire**
recevoir *retour(col)* **de** C_x;
collecté:= *collecté* $\wedge$ *col*
fait

pour C_i:
visite *vague(collecté)* mis en œuvre par:

collecté:= **vrai**;
recevoir *aller* **de** $C_{père_i}$;
$\forall x \in fils_i$ **envoyer** *aller* **à** C_x

faire_suivre *vague(collecté)* mis en œuvre par:

$\forall x \in fils_i$ **faire**
recevoir *retour(col)* **de** C_x
collecté:= *collecté* $\wedge$ *col*
fait;
envoyer *retour(collecté)* **à** $C_{père_i}$

5.3 Autres structures de parcours

D'autres structures sont possibles pour supporter les vagues; (le lecteur intéressé pourra consulter [HR88b, Tel88]).

Nous avons supposé ici que l'initiateur était prédéfini; cela n'est en fait pas nécessaire, la propriété fondamentale sur laquelle s'appuie l'algorithme est la succession séquentielle des vagues. Dans le cas particulier où le support est un anneau et où l'initiateur est défini dynamiquement (un prédicat supplémentaire permet alors à l'un des contrôleurs de se définir comme l'initiateur) on obtient un algorithme semblable à celui de [Nai88].

La particularité du problème de détection étudié permet de considérer une structure support des vagues qui soit évolutive en fonction de l'activité de l'application c'est à dire n'incluant pas les processus passifs et dont l'activité qu'ils ont engendré est terminée. Si l'on considère une structure de parcours arborescente qui évolue de la manière indiquée et une seule

vague on retrouve l'algorithme du calcul diffusant [DS80]. Il est important de remarquer que le cas où les structures de parcours évoluent en fonction du calcul réalisé (ici une détection), se situe hors du cadre abstrait caractérisé par l'indépendance entre utilisation et mises en oeuvre des parcours.

6 Conclusion

Outre la détection de la propriété de terminaison qu'il réalise, l'algorithme distribué obtenu est intéressant par sa construction méthodique. Sa dérivation et sa preuve s'appuient en effet sur la définition d'une structure de parcours de réseau abstraite (le concept de vague) et sur le mécanisme d'itération répartie qu'est le train de vagues. Grâce à la séparation des problèmes ainsi réalisée d'une part la solution obtenue est indépendante d'une structure de communication particulière sur laquelle doivent circuler les messages de contrôle et d'autre part le flot de contrôle nécessaire à la mise en oeuvre d'une vague peut exploiter les particularités du réseau sous-jacent (il s'agit d'un problème de mise en oeuvre de la vague). L'approche utilisée constitue ainsi une forme de composition d'algorithmes par le biais de l'interface que constitue la définition des primitives relatives au concept de vague. Cet article constitue de plus une illustration d'une forme d'itération répartie: la dérivation suivie a montré une façon d'obtenir un invariant et une condition d'arrêt répartis relatifs au problème de détection.

7 Références

[Awe85a] B. AWERBUCH. Complexity of network synchronization. *Journal ACM*, 32(4):804–823, October 1985.

[CL85] K.M CHANDY and L. LAMPORT. Distributed snapshots : determining global states in distributed systems. *ACM TOCS*, 3(1):63–75, Feb. 1985.

[CM86] K.M. CHANDY and J. MIRSA. An example of stepwise refinement of distributed programs : quiescence detection. *ACM Toplas*, 8(3):326–343, July 1986.

[DS80] E.W. DIJKSTRA and C.S. SCHOLTEN. Termination detection for diffusing computations. *Inf. Proccessing Letters*, Vol. 11:1–4, 1980.

[DFG83] E.W. DIJKSTRA, W. H. J. FEIJEN, and A. J. M. VAN GASTEREN. Derivation of termination detection algorithm for distributed computation. *Inf. processing Letters*, Vol. 16:217–219, June 1983.

[HJPR87] J. M HELARY, C. JARD, N. PLOUZEAU, and M. RAYNAL. Detection of stable properties in distributed applications. In *Proc. 6th annual ACM Symposium on Principles of Distributed Computing*, pages 125–136, Vancouver, August 1987.

[HR88b] J.M. HELARY, M. RAYNAL. *Synchronisation et contrôle des Systèmes et Programmes Répartis.* Eyrolles, Paris, septembre 1988, 195p.

[Lam78] L. LAMPORT. Time, clocks and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558-565, July 1978.

[LZ74] B. LISKOV, S.N. ZILLES. Programming with Abstract Data Types. *Proc. ACM Sigplan Conf. on Very High Level Languages*, (April 1974), pp 50-59.

[MAT87a] F. MATTERN. Asynchronous distributed termination: parallel and symmetric solutions with echo algorithms. *Research Report, University of Kaiserlautern*, 1987. A paraitre dans Algorithmica, 1989.

[Mat87b] F. MATTERN. Algorithms for distributed termination detection. *Distributed Computing*, 2:161–175, 1987.

[Nai88] M. NAIMI. Global stability detection in asynchronous distributed computations. *Proc. IEEE Workshop on the future trends of distributed computing systems in the 90's*, Hong-Kong, 1988, pp. 87-92.

[Ray87] M. RAYNAL. *Systèmes Répartis et Réseaux : Concepts, Outils et Algorithmes.* Eyrolles, 1987. 200 p.; (Distributed Computation and Networks, the MIT Press, 1988).

[SF86] N. SHAVIT and N. FRANCEZ. A new approach to detection of locally indicative stability. In *Proc. 13th ICALP*, LNCS, 226, pages 344-358, Springer-Verlag, 1986.

[Sch85] F.P. SCHNEIDER Paradigms for Distributed Programs. In *Distributed Systems*, LNCS 190: 431-480, Springer-Verlag Ed., (1985).

[Tel88] G. TEL. *Total Algorithms. Proc. Int. Workshop on Parallel and Distributed Algorithms*, Bonas, France, oct. 1988, Cosnard, Quinton, Raynal, Robert ed., North Holland, 1989.

LISTE DES DERNIERES PUBLICATIONS INTERNES

PI 437 **EXTENSION OF CHERNIKOVA'S ALGORITHM FOR SOLVING GENERAL MIXED LINEAR PROGRAMMING PROBLEMS**
Felipe FERNANDEZ, Patrice QUINTON,
38 Pages, Octobre 1988.

PI 438 **A PROPOS DE LA RESOLUTION D'UN SYSTEME LINEAIRE DANS UN CORPS FINI : ALGORITHMES ET MACHINES PARALLELES**
Hervé LE VERGE, Patrice QUINTON, Yves ROBERT, Gilles VILLARD
22 Pages, Novembre 1988.

PI 439 **ALPHA DU CENTAUR : A PROTOTYPE ENVIRONMENT FOR THE DESIGN OF PARALLEL REGULAR ALGORITHMS**
Pierrick GACHET, Patrice QUINTON, Christophe MAURAS
Yannick SAOUTER
20 Pages, Novembre 1988.

PI 440 **CONSTRUCTION METHODIQUE D'UN ALGORITHME REPARTI DE DETECTION DE LA TERMINAISON**
Jean-Michel HELARY, Michel RAYNAL
18 Pages, Décembre 1988.

PI 441 **LES GRAPHES A MOTIFS**
Didier CAUCAL
46 Pages, Décembre 1988.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique