When all the observers of a distributed computation do agree

Eddy Fromentin, Michel Raynal

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

When all the observers of a distributed computation do agree

Eddy Fromentin and Michel Raynal

N° 2194

Mars 1994

PROGRAMME 1

Architectures parallèles,

bases de données,

réseaux et systèmes distribués

ISSN 0249-6399

When all the observers of a distributed computation do agree

Eddy Fromentin* and Michel Raynal*

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

Rapport de recherche n°2194 — Mars 1994 — 23 pages

Abstract: A consistent observation of a distributed computation can be seen as a sequence of states and events that might have been produced by executing this computation on a monoprocessor. So a distributed execution generally accepts a lot of consistent observations. This paper concentrates on what all these observations have in common. An abstraction called *inevitable global state* is defined. A necessary and sufficient condition characterizing such states is given and a monitor-based algorithm that detects them is also presented . Possible uses of such states are sketched.

Key-words: distributed computation, relevant event, consistent global state, causality, precedence, concurrency, inevitable global state, unstable property detection, observation, observer-independent property

(Résumé : tsvp)

This work has been supported in part by the Commission of European Communities under ESPRIT Programme BRA 6360 (BROADCAST), by the French CNRS under the grant Parallel Traces and by a French-Israeli grant on distributed computing.

*e-mail:{fromenti, raynal}@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

Quand tous les observateurs d'une exécution répartie sont d'accord

Résumé : On peut considérer qu'une observation cohérente d'une exécution répartie se compose d'une séquence d'états et d'événements qui aurait pu être produite en déroulant cette exécution répartie sur un mono-processeur. Une telle exécution répartie accepte donc généralement un grand nombre d'observations cohérentes. Nous étudions ici une abstraction que nous appellerons *état global inévitable* qui est un point commun à toutes les observations. Nous donnons une condition nécessaire et suffisante qui caractérise cette propriété d'inévitabilité étant donné un état global, présentons un algorithme reposant sur l'emploi d'un contrôleur centralisé qui détecte de tels états globaux et en résumons quelques utilisations possibles.

Mots-clé : exécution répartie, événements observables, état global cohérent, ordre causal, précédence, concurrence, état global inévitable, calcul de propriétés instables, observation, propriété indépendante de l'observateur

1 Introduction

Since Lamport's seminal paper [12], execution of asynchronous distributed programs is modelized by a partial order on the events produced. Due to the asynchronism of the underlying support (no common physical clock, no common shared memory, arbitrary transfer delays) any consistent observation of such a distributed execution can only see a sequence of all these events that respects the partial order. Using this sequence of events an observer can reconstruct, starting from the initial state of the computation, a sequence of global states through which the computation may have progressed [1, 17]. One important question is then: are there global states seen by all observations of a distributed computation ?

The detection of such global states (called *inevitable* in the following) can be very helpful to solve some problems, as an inevitable global state has necessarily been "passed through" by the actual execution. In other words an inevitable global state is independent of observers. According to the application program executed such states can be used for example to define a checkpoint, or to detect properties that must be true for all observations or to solve global state oriented problems whose solution need to be observer-independent [2, 4]. This paper presents a characterization of such global states and an algorithm to detect them.

The paper is structured in the following way. Section 2 introduces a formal model for distributed computations. This model considers such a computation at some observation level and uses notions such as relevant events, local states and causality induced by messages. Moreover, precedence relations on local states of a distributed computation are clearly identified. Section 3 presents a necessary and sufficient condition for a global state to be inevitable. Moreover a generalization is given by introducing the notion of inevitability with respect to a set of processes. Section 4 presents an algorithm to detect all inevitable global states of a distributed computation; its time complexity is $O(n^3k)$ (where n is the number of processes and $k = max_i(number\ of\ local\ states\ of\ P_i)$). Related works are briefly examined in Section 5. The conclusion lists problems that can be solved by using inevitable global states.

2 Distributed computations

2.1 Distributed programs

A distributed program is made of n sequential processes $P_1, \cdots, P_n$ which communicate and synchronize by the only means of message passing.

The underlying system, that executes distributed programs is composed of n processors (one per process) that can exchange messages. Each processor has a local memory. Neither shared memory nor a global clock is available. Messages are exchanged through reliable, non necessarily FIFO, channels. Transmission delays are finite but unpredictable.

2.2 Distributed computations

2.2.1 Basic events and Lamport's level

Execution of a process P_i produces a sequence of *basic events*. A basic event may be either internal (causing only a change to local variables) or it may involve communication (send and receive events) [12]. This sequence is usually called the history H_i of P_i: $H_i = e_i^0 e_i^1 e_i^2 e_i^3 \cdots e_i^x \cdots$ where e_i^x is the x^{th} basic event executed by P_i; e_i^0 a (fictitious) basic event that initializes local variables of P_i. Events are instantaneous.

Let H be the set of all these events and let $\stackrel{e}{\rightarrow}$ be the classical binary relation defined by Lamport on basic events [12] (called *causal precedence*):

$$e_i^x \stackrel{e}{\rightarrow} e_j^y \Leftrightarrow \begin{cases} & i = j \text{ and } x+1 = y \\ \textit{or} & \\ & e_i^x \text{ is the sending of a message and } e_i^y \text{ its reception} \\ \textit{or} & \\ & \exists e_k^z \text{ such that } e_i^x \stackrel{e}{\rightarrow} e_k^z \textit{ and } e_k^z \stackrel{e}{\rightarrow} e_j^y \end{cases}$$

When considering basic events, a distributed execution can be represented by a partially ordered set (poset) $\widehat{H} = (H, \stackrel{e}{\rightarrow})$. This poset defines the computation at Lamport's level; this level is characterized by the fact it comprises all communication events. Figure 1 displays, in the classical space-time diagram, a distributed execution at Lamport's level (events are denoted by black and white circles, messages by arrows).

2.2.2 Relevant events and user's level

According to the problem he has to solve (e.g. detection of a global property) only a subset of basic events are meaningful to the user (e.g. changes of some variables). These events are called *relevant events* (Such an abstraction of events of a distributed computation has already been proposed by several authors, see [5, 14]). Let R be the set of relevant events; these events define the user's level (in the distributed execution of Figure 1, relevant events are noted by black circles). The poset

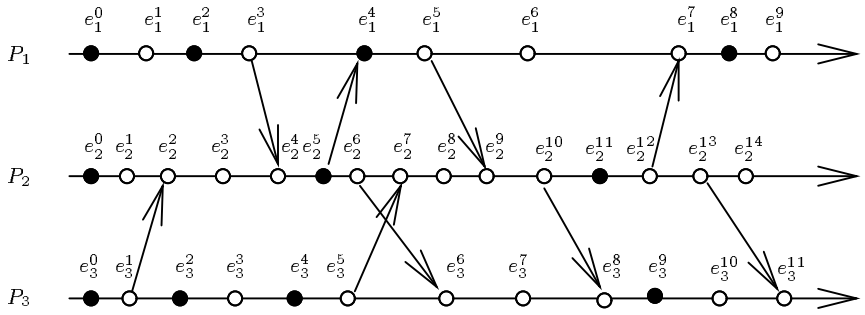

Figure 1: A distributed execution at Lamport's level

$\widehat{R} = (R, \stackrel{e}{\rightarrow})$ defines the distributed computation at the user's level. Interestingly, thanks to transitivity of $\stackrel{e}{\rightarrow}$, $\widehat{R}$ inherits causal precedence induced by communication events, even if communication events are not relevant.

2.3 Local and global states at the user's level

Define R_i as the history of P_i at the user's level: $R_i = r_i^0 r_i^1 r_i^2 \cdots r_i^y \cdots$ (where r_i^y is the y^{th} relevant event of P_i); for $y \geq 1$, r_i^y provokes the local state change from s_i^{y-1} to s_i^y ($next(s_i^{y-1})$ will be used as a synonymous of s_i^y). The initial relevant event r_i^0 is e_i^0: it defines s_i^0, the initial local state of P_i. Contrary to events that are instantaneous, due to synchronization and waiting local states have a duration; a local state of P_i lasts from a relevant event of P_i till the next one (Figure 2 displays, at user's level, local states of the distributed execution of Figure 1 from which irrelevant events having been eliminated).

If process P_i terminates, s_i^{last} will denote its last local state; $next(s_i^{last}) = s_i^\infty$ a local state identical to s_i^{last} as far as local variables of P_i are concerned; $next(s_i^\infty)$ is not defined and r_i^∞ is the fictitious event that produces s_i^∞.

2.3.1 Strong precedence

The set of local states of a distributed computation is partially ordered by a relation called *strong precedence*, denoted $\stackrel{s}{\rightarrow}$. Informally $s_i \stackrel{s}{\rightarrow} s_j$ means that s_i was no more

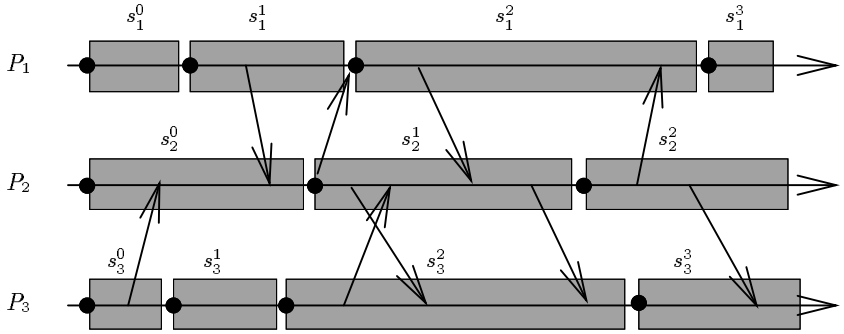

Figure 2: Local states of a distributed execution at user's levels

existing when s_j began to exist (Figure 3 is an enlargement of a part of Figure 2 where we can see $s_1^0 \xrightarrow{s} s_2^1$).

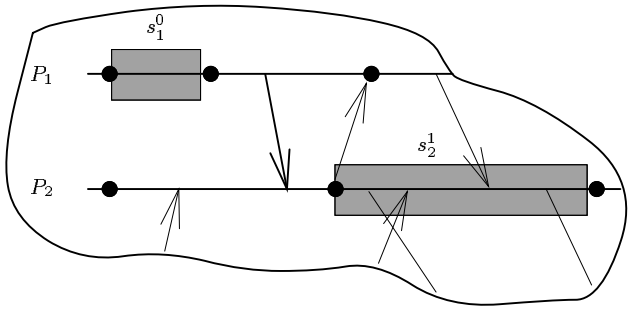

Figure 3: An example of strong precedence

Formally, strong precedence is defined in the following way:

$$s_i^x \xrightarrow{s} s_j^y \Leftrightarrow r_i^{x+1} \xrightarrow{e} r_j^y \ or \ s_j^y = next(s_i^x)$$

Figure 4 shows the relation $\stackrel{s}{\rightarrow}$ (without transitivity edges) associated with the computation of Figure 2. Two local states s_i and s_j are said to be concurrent, denoted $s_i \parallel s_j$, if and only if $\neg(s_i \stackrel{s}{\rightarrow} s_j)$ *and* $\neg(s_j \stackrel{s}{\rightarrow} s_i)$. A consistent global state Σ of a distributed execution is a n-uple of local states $(s_1, s_2, \cdots, s_n)$ such that $\forall i \neq j : s_i \parallel s_j$.

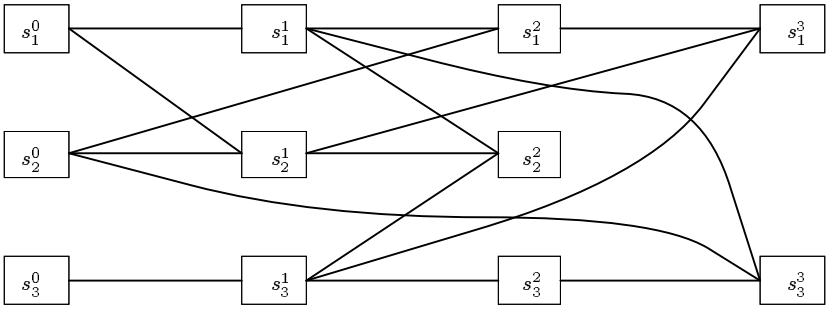

Figure 4: Strong precedence between local states of Figure 2 (arrows are from left to right)

2.3.2 Weak precedence

The set of local states is also structured by a *weak precedence* relation, denoted $\stackrel{w}{\rightarrow}$. Informally $s_i \stackrel{w}{\rightarrow} s_j$ if s_i began before s_j. So it is possible that both of them may exist simultaneously and consequently participate to the same consistent global state. (Figure 5 is an enlargement of a part of Figure 2 where we can see $s_1^1 \stackrel{w}{\rightarrow} s_2^1$).

The relation $\stackrel{w}{\rightarrow}$ is formally defined in the following way:

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

Of course, $s_i \stackrel{s}{\rightarrow} s_j \Rightarrow s_i \stackrel{w}{\rightarrow} s_j$. From the definitions we have also $s_i \stackrel{s}{\rightarrow} s_j \Rightarrow next(s_i) \stackrel{w}{\rightarrow} s_j$. Figure 6 shows the relation $\stackrel{w}{\rightarrow}$ (without transitivity edges) associated with the computation of Figure 2.

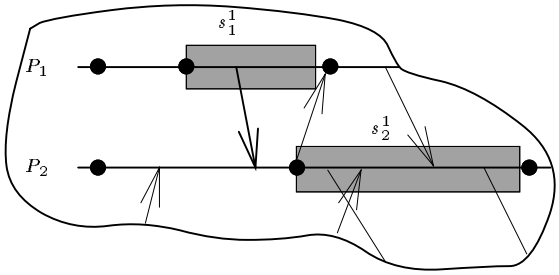

Figure 5: An example of weak precedence

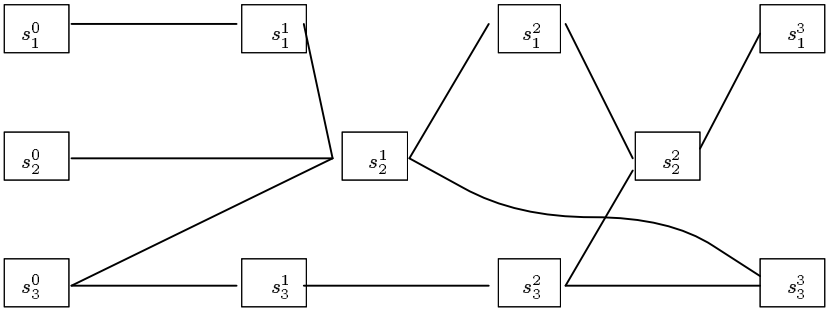

Figure 6: Weak precedence between local states of Figure 2 (arrows are from left to right)

2.3.3 Remark

In [11] Lamport introduced two temporal precedence relations on operation executions. If A and B are two operations $A \longrightarrow B$ (read: A *precedes* B) means all actions composing A are completed before any action of B is begun; $A \dashrightarrow B$ (read: A *can affect* B) means some action of A precedes some action of B.

There is a similarity first between local states at Lamport's level and actions and second between local states at user's level and operations; with such a correspondence relations on local states at user's level $\stackrel{s}{\rightarrow}$ and $\stackrel{w}{\rightarrow}$ correspond to relations on operations $\longrightarrow$ and $\dashrightarrow$. (Remark that in our context a message reception affecting the receiver local state systematically entails a change to a new local state for the receiver process; so in our context $\dashrightarrow$, i.e. $\stackrel{w}{\rightarrow}$, is acyclic).

2.4 Using vector clocks to detect precedence

2.4.1 Vector clock

Introduced simultaneously by Fidge [6] and Mattern [15], vector clocks constitute an operational tool to encode dependency and concurrency of events of a distributed computation. As in [14, 5] we use here such vector clocks in the following way:

- $v_i[1 \cdots n]$ is the vector clock of process P_i; it is initialized to $\mathbb{1}_i$ (a zero vector with 1 in the i^{th} position); this initialization simulates event r_i^0.
- Each time P_i enters a new local state (execution of a relevant event) $v_i[i]$ is incremented by a positive value (e.g. 1) and the new value of the vector clock constitutes the timestamp of this local state.
- All messages carry the current value of the vector clock of their senders.
- When a message m, carrying $v(m)$, is delivered to P_i, v_i is updated to $max(v_i, v(m))$[1].

2.4.2 Formulas to detect local states precedence

Let $v(s_i)$ and $v(s_j)$ be the timestamps respectively associated with local states s_i of P_i and s_j of P_j. From results of [6, 15] on the timestamping of events and results of [1] on the timestamping of local states we can deduce the following relations about the precedence of local states. (Other relations and formulas concerning local states can be found in [7]).

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

Strong precedence $s_i \stackrel{s}{\rightarrow} s_j \Leftrightarrow v(s_i)[i] < v(s_j)[i]$

Weak precedence $s_i \stackrel{w}{\rightarrow} s_j \Leftrightarrow v(s_i)[i] \leq v(s_j)[i]$

3 Inevitable global states

3.1 The lattice of global states

The set S of all consistent global states associated with a distributed computation $\widehat{R} = (R, \stackrel{e}{\rightarrow})$ has a lattice structure whose minimal (respt. maximal) element is the initial (respt. final) global state $\Sigma^0 = (s_1^0, \cdots, s_i^0, \cdots, s_n^0)$ (respt. $\Sigma^{last} = (s_1^{last}, \cdots, s_i^{last}, \cdots, s_n^{last})$[2]. There is an edge from a vertex $\Sigma = (s_1, \cdots, s_i, \cdots, s_n)$ to a vertex $\Sigma' = (s_1, \cdots, next(s_i), \cdots, s_n)$ if and only if there is an event r_i of P_i that can be produced in global state Σ; r_i constitutes the label of this edge [15]. Figure 7 shows the lattice associated with the distributed computation displayed in Figure 2.

Informally a sequential observation of a distributed computation $\widehat{R}$ represents a consistent view of $\widehat{R}$ an external sequential observer could have [1, 17]. More formally an observation O is a sequence: $\Sigma^0 r^1 \Sigma^1 r^2 \Sigma^2 \cdots \Sigma^{i-1} r^i \Sigma^i r^{i+1} \cdots \Sigma^{last}$ of global states and events such that:

- all events appear in an order consistent with $\widehat{R}$ (i.e. the sequence of events in O is a linear extension[3] of $\widehat{R}$);
- Σ^i is the global state obtained from Σ^{i-1} by executing r^i.

An observation is a "path" in the lattice, in which global states correspond to vertices and events to labels of edges. According to the problem we want to solve only events or global states can be considered within an observation. As shown in [17] all possible observations of a distributed computation correspond exactly to all the paths of the lattice (the interested reader will find more details about observations in [1, 17]).

[2] The final global state Σ^{last} exists only if all processes of the distributed computation terminate. We suppose in the following such finite distributed computations but our results also apply to non-terminating computations

[3] A linear extension $\widehat{R'} = (R, \stackrel{e}{\rightarrow}')$ of a partial order $\widehat{R} = (R, \stackrel{e}{\rightarrow})$ is a total order such that $\forall e, f \in R : e \stackrel{e}{\rightarrow} f \Rightarrow e \stackrel{e}{\rightarrow}' f$.

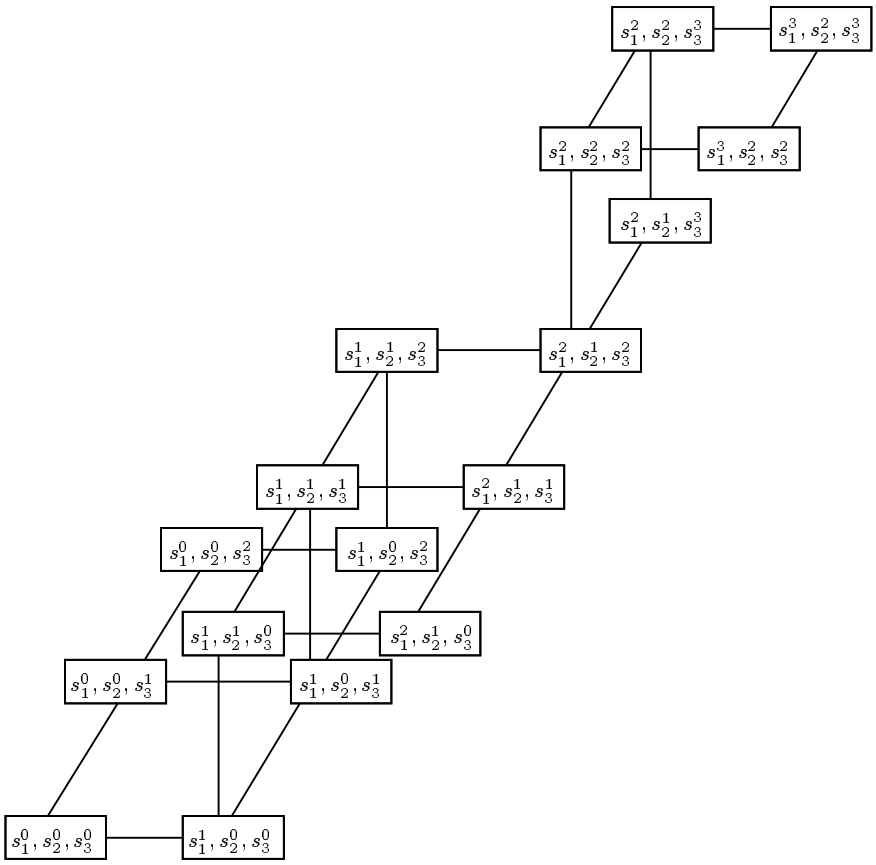

Figure 7: Lattice associated with the distributed computation of Figure 2

3.2 Inevitable global states

3.2.1 Definition

A global state is *inevitable* if it belongs to all the observations of the distributed computation[4]. In Figure 7, $\Sigma = (s_1^2, s_2^1, s_3^2)$ is an inevitable global state.

Relevant events and inevitable global states are the right abstractions to solve problems involving global states seen by all observers. These states characterize the greatest set of global states shared by all possible observations. In addition to their conceptual interest, they present a practical interest as their detection can be done at low cost (see Section 4.4) without building the lattice of global states which can be exponential with respect to the number of processes.

As an example of use consider the detection of predicates such as $DEF\ \Phi$, introduced by Cooper and Marzullo [4]. A distributed computation γ satisfies predicate $DEF\ \Phi$ ($\gamma \models DEF\ \Phi$), where Φ is a predicate on a global state, if and only if each observation O_x of γ includes a global state Σ_x such that $\Sigma_x \models \Phi$.

As Φ can be any predicate on a global state, algorithms to detect $DEF\ \Phi$ are naturally based on a traversal of the entire lattice of global states [1, 2, 4]. Inevitable global states can be seen as a cheap heuristic alternative[5] to limit the search space as we have:

$$(\exists\ \Sigma\ inevitable : \Sigma \models \Phi) \Rightarrow (\gamma \models DEF\ \Phi)$$

If additionally Φ is such that it can only be true in an inevitable global state, the "greedy" detection gives always the right answer as in this case:

$$(\exists\ \Sigma\ inevitable : \Sigma \models \Phi) \Leftrightarrow (\gamma \models DEF\ \Phi)$$

Results of [8, 20] are expressed at Lamport's level. They concern the detection of $DEF\ \Phi$ where $\Phi = \bigwedge_{i=1}^{n} LP_i$ (each LP_i being a predicate local to a P_i[6]). Let $\Sigma = (s_1, \cdots, s_n)$ a consistent global state, $\Sigma \models \Phi$ if and only if $\bigwedge_i (s_i \models LP_i)$. With our level of abstraction (user's level) each P_i, in the distributed computation γ, enters a new local state s_i each time LP_i changes its value[7]; Those are the only relevant

[4] With the initial and the final global states all articulation points of the lattice are inevitable global states.

[5] A detection algorithm visiting only inevitable global states, can be compared to a "greedy" algorithm that, in combinatorial problems, can sometimes miss the solution.

[6] i.e. LP_i is only on local variables of P_i.

[7] So we have $s_i \models LP_i \Leftrightarrow \neg(next(s_i) \models LP_i)$.

events. Expressed with the inevitable global state abstraction, results of [8, 20] can be reformulated as:

$$(\gamma \models DEF\ \Phi) \Leftrightarrow (\exists\ \Sigma\ inevitable : \Sigma \models \Phi)$$

This formulation expresses clearly the exact meaning of $(\gamma \models DEF(\bigwedge_i LP_i))$ with respect to the lattice of global state. To our knowledge this is the first time such an abstract formulation is given.

3.2.2 A necessary and sufficient condition

Let $\Sigma = (s_1, \cdots, s_i, \cdots, s_n)$ be a global state (distinct from Σ^0 and Σ^{last}). We have:

(IGS) $\Sigma = (s_1, \cdots, s_i, \cdots, s_n)$ is an inevitable global state if and only if $\forall i, j : s_i \xrightarrow{w} next(s_j)$

Proof:

$\Rightarrow$

Let r_i be P_i's event that produced s_i, r'_j be P_j's event that produced $next(s_j)$ $(s_j \neq s_j^{\infty})$;
Consider $\Sigma' = (\cdots, next(s_j), \cdots)$. Any observation comprises Σ (by hypothesis) and such a Σ' (because each observation includes all events and so r'_j) and Σ appears before Σ' (as s_j is produced before $next(s_j)$); consequently r_i appears before r'_j within this observation. Moreover this is true for any couple (i, j).
As Σ is inevitable the previous remark is true for all observations. It follows that for all observations: $\forall i, j$: r_i appears before r'_j.
From Szpilrajn's theorem [18] (which states that the intersection of all linear extensions –here the set of all observations– of a partial order –here $\widehat{R} = (R, \xrightarrow{e})$– is precisely this partial order) it follows that $\forall i, j : r_i \xrightarrow{e} r'_j$; that is to say in terms of local states (Section 2.3.2) $\forall i, j : s_i \xrightarrow{w} next(s_j)$.

•

$\Leftarrow$

Let a global state $\Sigma = (s_1, s_2, \cdots, s_n)$ be such that $\forall i, j : s_i \stackrel{w}{\rightarrow} next(s_j)$.
First let us show that Σ is a consistent global state. Suppose that Σ is not consistent. Then $\exists i, j : s_i \stackrel{s}{\rightarrow} s_j$ (See Section 2.3.1 for the definition of a consistent global state); from which we can conclude (see Section 2.3.2) $\exists i, j : next(s_i) \stackrel{w}{\rightarrow} s_j$ which is a contradiction.
Now consider an observation O that comprises two global states $\Sigma' = (\cdots, s_i, \cdots)$ and $\Sigma'' = (\cdots, next(s_j), \cdots)$. We have to show that Σ' precedes Σ''.
Let r'_i be P_i's event that produced s_i, r''_i be P_i's event that produced $next(s_i)$ and r''_j be P_j's event that produced $next(s_j)$. As O includes all events it includes r'_i, r''_i and r''_j; then such Σ' and Σ'' exists.
Σ' appears in O after r'_i and before r''_i; Σ'' appears after r''_j. Moreover $r'_i \stackrel{e}{\rightarrow} r''_j$ (because $s_i \stackrel{w}{\rightarrow} next(s_j)$); it follows that Σ' precedes Σ'' in O.
As by hypothesis $\forall i, j : s_i \stackrel{w}{\rightarrow} next(s_j)$ we have $\forall i, j : r'_i \stackrel{e}{\rightarrow} r''_j$; consequently any observation O is such that:

$$O = \Sigma^0 \cdots r'_{i_1} \Sigma^{i_1} \cdots r'_{i_x} \Sigma^{i_x} \cdots r'_{i_n} \Psi r''_{j_1} \Sigma^{j_1} \cdots r''_{j_x} \Sigma^{j_x} \cdots r''_{j_n} \Sigma^{j_n} \cdots \Sigma^{last}$$

with n-uples $(i_1, \cdots, i_n)$ and $(j_1, \cdots, j_n)$ being permutations of $(1, \cdots, n)$.
It follows that the global state Ψ is the global state Σ.

□

3.3 The case of Σ^0 and Σ^{last}

3.3.1 Conditions to satisfy

Let it be the two following properties:

$$\begin{aligned} H^0 &\equiv \forall i, j : s_i^0 \stackrel{w}{\rightarrow} s_j^1 \\ H^{last} &\equiv \forall i, j : s_i^{last} \stackrel{w}{\rightarrow} s_j^\infty \end{aligned}$$

Σ^0 is not declared inevitable by condition **IGS** if H^0 is not verified. For a given computation, this happens if some P_i has not been informed about initialization of

some P_j before entering s_i^1 (because messages have not been sent or are travelling too "slowly" to it).

Similarly Σ^{last} is not declared inevitable if H^{last} is not verified. For H^{last} to be true each P_i, when in state s_i^{last}, must send a message to each other P_j (and only after reception of all these messages P_j execute r_i^∞ i.e. it terminates).

H^0 (respt. H^{last}) formally expresses in a distributed system context the informal notion "all process started (respt. finished) simultaneously".

3.3.2 Making Σ^0 and Σ^{last} always inevitable

So if one wants the initial global state Σ^0 (or the final one Σ^{last}) of a distributed computation be always recognized as inevitable, conditions H^0 (or H^{last}) must systematically be satisfied. In that case H^0 (or H^{last}) is added to the weak precedence relation[8].

At the operational level this is ensured by adding the following rules:

For H^0: When it enters s_i^1, process P_i $(\forall i)$ does the following updates: $\forall j : v_i[j] := max(v_i[j], 1)$; this ensure $\forall i, j : s_i^0 \xrightarrow{w} s_j^1$ (see relation in Section 2.4.2);

For H^{last}: During local state s_i^{last}, process P_i $(\forall i)$ broadcasts to all P_j a control message carrying only a timestamp. When it has received such a message from each other P_i executes r_i^∞; this ensures $\forall i, j : s_i^{last} \xrightarrow{w} s_j^\infty$ (see relation in Section 2.4.2).

3.3.3 Meaning of inevitability for Σ^0 and Σ^{last}

H^0 states that for any couple (P_i, P_j), P_i was launched (it did its initializations) before P_j executed its first event r_j^1. In other words if H^0 is verified by the computation itself (i.e. without the addition of rules of Section 3.3.2), then all initial local states s_i^0 co-existed at some physical time (i.e. a process P_i was not in state s_i^2 while another one had not yet begun). The same remark applies to Σ^{last}. Consequently, when additional rules of Section 3.3.2 are not used, if condition **IGS** applied to Σ^0 (respt. Σ^{last}) evaluates to true, we can conclude, as H^0 (respt. H^{last}) was satisfied by the computation itself, that Σ^0 (or Σ^{last}) really existed during the computation.

[8] The addition of H^0 (or H^{last}) to the weak precedence relation, eliminates the possibility to execute a distributed program made of n independent processes, one after the other.

3.4 Generalization

In some cases one can be interested only in consistent *partial global states* seen by all observers. A global state is partial if it includes local states from only a subset $Q = \{i_1, \cdots, i_k\}$ of processes: $\Sigma[Q] = (s_{i_1}, s_{i_2}, \cdots, s_{i_x}, \cdots, s_{i_k})$ with $i_x \in Q$.

If Q is the set of all the processes, $\Sigma[Q]$ is a full global state, whereas if Q comprises only one process, $\Sigma[Q]$ reduces to a local state of this process. A consistent partial global state $\Sigma[Q]$ is inevitable if and only if $\forall i, j \in Q : s_i \stackrel{w}{\rightarrow} next(s_j)$. Let $k = card(Q)$. For the case $k = 1$, remember all local states of any process are seen by all observers (obviously $s_i \stackrel{w}{\rightarrow} next(s_i)$). When $k \geq 2$ the previous relation characterizes all consistent k-uples of local states from processes of Q that are seen by all observers. For example in Figure 7, (s_1^1, s_2^1) is inevitable with respect to $Q = \{1, 2\}$. This relation can be used when we concentrate on the detection of properties that do not involve all processes (see for example [19]).

4 A detection algorithm

Thanks to the formulas introduced in Section 2.4.2, it is easy to design an algorithm that detects inevitable global states. A FIFO channel is added between each process and a monitor M. Each time a new local state begins, P_i sends to M a control message composed of the local state and its timestamp. The monitor is equipped with n queues Q_i which store incoming messages from each process P_i. M uses **IGS** to detect inevitable global states.

The protocol executed by the monitor is an adaptation of an algorithm defined by Garg [9] to detect a largest anti-chain (here a n–uple of local states satisfying condition **IGS**) in a partially ordered set given its decomposition into its chains (here the sequences of control messages received from each P_i and stored in queues Q_i). The protocol is described in Sections 4.1 to 4.3 (It can be decentralized using the technique described in [8]). Let k_i be the number of local states (including s_i^0 and $next(s_i^{last})$) of process P_i and $k = max_i(k_i)$. It is shown in Section 4.4 that the number of comparisons of integers of the algorithm is upper-bounded by $O(n^3k)$ to detect all inevitable global states[9].

4.1 Underlying principles

In order to detect inevitable global states, Garg's algorithm [9] is adapted in the following way. Q_i is the sequence of timestamped local states received in order from

[9] $O(n^2k)$ is the time complexity if we search only the first inevitable global state.

P_i; $head(Q_i)$ denotes the first local state of Q_i; $tail(Q_i)$ denotes the sequence Q_i without its first element; $next^*(s_i)$ denotes any successor of s_i including s_i itself.

Tests to decide whether two local states are related by a precedence relation are done on their timestamps, thanks to formulas introduced in Section 2.4.2. To make the algorithm easier to understand we suppose the queues Q_i have been filled up by processes. This version can easily be adapted to work on the fly, with processes filling their queues as they progress (see [9]).

In Garg's algorithm heads of the queues are checked to see if they form a global state (a largest antichain). Its adaptation to detect inevitable global states is based on the two following observations:

1. Let $\Sigma = (\cdots, s_i, \cdots, s_j, \cdots)$ be the global state under consideration candidate to inevitability. if $\neg(s_i \xrightarrow{w} next(s_j))$ we can conclude any global state $\Sigma' = (\cdots, next^*(s_i), \cdots, s_j, \cdots)$ is not inevitable. So in that case s_j is no longer considered and the algorithm considers the global state $\Sigma'' = (\cdots, s_i, \cdots, next(s_j), \cdots)$ as a candidate for inevitability. This observation allows to redefine appropriately the head of the queues (with the auxiliary variable *changed* in the algorithm).

2. After an inevitable global state $\Sigma = (s_1, s_2, \cdots, s_n)$ has been found, the next candidate Σ' for inevitability is defined in the following way (in order not to miss inevitable global states): Σ' is a consistent global state $(s'_1, s'_2, \cdots, s'_n)$ that is an immediate successor of Σ in the lattice, i.e.: $\exists k : (\forall i \neq k : s'_i = s_i \ and \ s'_k = next(s_k))$.

4.2 The algorithm

For any queue Q_i, s_i (respt. $next(s_i)$) is a synonymous of $head(Q_i)$ (respt. $head(tail(Q_i))$). Moreover to simplify the description of the algorithm we suppose $next(s_i^\infty) = s_i^\infty$.

```
    changed:={1, 2, ···, n};
    while ∃i : s_i ≠ s_i^∞ do
        newchanged:=∅;
        % evaluation of IGS on Σ = (s_1, ···, s_i, ···, s_j, ···, s_n)
        ∀i ∈ changed, j ∈ {1, 2, ···, n} do
(1)         if ¬(s_i -w-> next(s_j)) then newchanged:=newchanged ∪ {j} fi;
(2)         if ¬(s_j -w-> next(s_i)) then newchanged:=newchanged ∪ {i} fi;
        od;
        if newchanged = ∅ then
(3)         Σ = (s_1, ···, s_i, ···, s_j, ···, s_n) is inevitable;
(4)         let k such that Σ' = (s'_1, ···, s'_k, ···, s'_n) is a consistent global state
            with ∀i ≠ k : s_i = s'_i and s'_k = next(s_k),
(5)         newchanged:={k};
        fi;
        changed:=newchanged;
        ∀i ∈ changed : Q_i:=tail(Q_i);
    od;
```

4.3 Safety and liveness

The safety property indicates consistency of the detection: if the algorithm claims Σ inevitable then it is. This property follows directly from conditions tested at lines 1 and 2: if in line 3, a global state Σ is declared inevitable it satisfied necessarily condition **IGS**.

The liveness property states that if a global state is inevitable then the algorithm will detect it. The proof of this property is done in two steps.

1. First consider a global state Σ that has been declared inevitable. The search is continued from Σ' which is an immediate successor of Σ; consequently no possibly inevitable global state Σ'' lying in the lattice between Σ and Σ' can be missed.

2. Second consider the algorithm is in its initial state or just after a global state has been declared inevitable. Suppose an inevitable global state $\Sigma' = (s'_1, \cdots, s'_n)$ exists, and it is the first, in the lattice, of the next inevitable global states. Suppose also the algorithm is checking at lines 1 and 2 a global state $\Sigma'' = (s''_1, \cdots, s''_n)$ such that Σ'' is a (not necessarily immediate) successor of Σ' in the lattice (such a Σ'' does exist, at worst it is $(s_1^\infty, s_2^\infty, \cdots, s_n^\infty)$). And

lastly suppose that Σ' has not been found by the algorithm. We show there is a contradiction.

All elements of any queue are examined by the algorithm. So at some time t_1 we have $s'_i = head(Q_i)$ and s'_i is removed Q_i without declaring Σ' inevitable. Consequently it exists a head of some queue Q_j, let it be $\tau_j = head(Q_j)$, such that $\neg(\tau_j \xrightarrow{w} next(s'_i))$. We can conclude that $\tau_j \neq s'_j$ (as Σ' is inevitable, $s'_j \xrightarrow{w} next(s'_i)$) and $s'_j \xrightarrow{s} \tau_j$ (if not we would have $\tau_j \xrightarrow{w} s'_j$ –as these two local states are from the same P_j– and $s'_j \xrightarrow{w} next(s'_i)$ –as Σ' is inevitable– and by transitivity we would have $\tau_j \xrightarrow{w} next(s'_i)$).

As $head(Q_j) = \tau_j$ the algorithm has already eliminated s'_j from Q_j at some time t_2 ($t_2 < t_1$). Consider now the algorithm at t_2: at that time it existed $\tau_h = head(Q_h)$ that provoked the elimination de s'_j from Q_j (same reasoning as before). By induction on the number of queues it follows that at t_{n-1} ($t_{n-1} < t_{n-2} < \cdots < t_1$) we had:

$$\exists k : \begin{cases} head(Q_k) = \tau_k & \text{with } s'_k \xrightarrow{s} \tau_k \\ head(Q_i) = s_i & \text{with } s_i \xrightarrow{s} s'_i \vee s_i = s'_i \text{ (for } i \neq k) \end{cases}$$

so s'_k has been eliminated from its queue Q_k at t_n ($t_n < t_{n-1}$) at lines 1 and 2 by a local state s_j such that $s_j \xrightarrow{s} s'_j \vee s_j = s'_j$ (i.e. we had at t_n:$\neg(s_j \xrightarrow{w} next(s'_k))$). This is impossible as Σ' is inevitable (as we cannot have $(s_j \xrightarrow{s} s'_j \vee s_j = s'_j) \wedge \neg(s_j \xrightarrow{w} next(s'_k))$). This proves the contradiction. It follows Σ' has not been missed.

□

4.4 Time complexity

Let k_i be the number of local states (including s_i^0 and s_i^{last}) of process P_i and $k = max_i(k_i)$.

If we eliminate the statement **if** $newchanged = \emptyset$ **then** $\cdots$ **fi** we obtain an algorithm whose structure is the same as Garg's one. Garg showed, in [9], that the time complexity of this algorithm is $O(n^2k)$ comparisons. Each comparison is here on 2 integers.

Consider now the algorithm without the loop including lines 1 and 2. To advance the appropriate queue Q_k the algorithm has to find a consistent global state Σ' immediate successor of Σ. Obtaining such a global state $\Sigma' = (s_1, \cdots, next(s_k), \cdots, s_n)$

requires at most $O(n^2)$ comparisons of integers ($2(n-1)$ comparisons to test $next(s_k) \parallel s_i$ for $i \neq k$ and, at worst, n such sets of comparisons to find the appropriate k). Such tests are done each time an inevitable global state is found. $k_1 + k_2 + \cdots + k_n$ constitutes an upper bound on the number of inevitable global states (all elements of queues are examined without never backtracking). So the second part of the algorithm is upper bounded by $0(n^2 \Sigma k_i)$.

Consequently $0(n^3 k)$ constitutes an upper bound on the number of comparisons of integers needed by the algorithm.

5 Related Works

In [13] Lee and Davidson solve the generalized rendez-vous problem in a real-time context. Such a rendez-vous involves all the n processes and each of them specifies a deadline for its involvement in the rendez-vous. So for a generalized rendez-vous each process provides a *real-time interval* (which begins at the time he want to participate in the rendez-vous and ends at its deadline). The generalized rendez-vous is possible if the intersection of all these real-time intervals is not empty. Relation **IGS** expresses a similar notion on logical time.

In [8] Garg and Waldecker use notion of *interval* of local states (a sequence of consecutive local states of a process) to detect a particular class of properties. These properties are expressed by a conjunction of local predicates. As before the intersection of appropriate intervals must be not empty for the property to be detected by all observers. In [20] Venkatesan and Dathan use a similar notion called *spectrum* to solve global properties detection problems during the replay of a distributed computation. Both these works consider a distributed computation at Lamport's level. Consequently their definitions and algorithms have to cope with all events (internal, sendings and receptions of messages) whether they are relevant or not for the studied property. Our approach is similar to their but at a more abstract level which allowed us to explicitly introduce the notion of inevitable global state.

6 Conclusion

The abstraction of inevitable global state (full or partial) has been introduced. Such a state has the following characterization: it has been seen by all the observers of the distributed computation. A necessary and sufficient condition to detect these global states has been given and a monitor-based algorithm that detects them has

been presented. Additionally relations on local states of processes of a distributed computation have been clearly identified.

Such global states are interesting for problems whose solutions must be observer independent. Among them there is detection of global predicate with the DEF modality [4] in the context of distributed testing and debugging, and determination of a "which really occurred" checkpoint in the context of fault recovery in distributed systems. Additionally, the number of inevitable global states can be easily computed and used to define an appropriate concurrency measure of a distributed computation [3, 16] (inevitable global states can be bottlenecks from potential parallelism point of view).

This work is part of our current effort in designing and implementing a debugging facility for distributed systems [2, 10].

Acknowledgements

We are grateful to Ö. Babaoğlu and Cl. Jard for interesting discussions about notions of relevant events and observations. We acknowledge F. Schneider whose comments helped to improve the presentation of the paper. This work has been supported in part by the Commission of European Communities under ESPRIT Programme BRA 6360 (BROADCAST), by the French CNRS under the grant Parallel Traces and by a French-Israeli grant on distributed computing.

References

[1] 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.), 1993.

[2] Ö. Babaoğlu and M. Raynal. Specification and detection of behavioral 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.

[3] B. Charron-Bost. Coupling coefficients of a distributed execution. *Theoretical Computer Science*, (110):341–376, 1993.

[4] 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, California, May 1991.

[5] C. Diehl, C. Jard, and J. X. Rampon. Reachability analysis on distributed executions. In *Theory and Practice of Software Development*, pages 629–643, TAPSOFT, Springer Verlag, LNCS 668 (Gaudel and Jouannaud editors), April 1993.

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

[7] 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.

[8] V. K. Garg and B. Waldecker. Detection of unstable predicates in distributed programs. In *Twelfth International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 253–264, Springer Verlag, LNCS 625, New Delhi, India, December 1992.

[9] V.K. Garg. Some optimal algorithms for decomposed partially ordered sets. *Information Processing Letters*, 44:39–43, 1992.

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

[11] L Lamport. On interprocess communication. part 1: basic formalism. *Distributed Computing*, 1,2:77–85, 1986.

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

[13] I. Lee and S. B. Davidson. Adding time to synchronous process communication. *IEEE Trans. on Computers*, C36(8):941–948, 1987.

[14] K. Marzullo and L. Sabel. *Using Consistent Subcuts for Detecting Stable Properties*. Technical Report 91-1205, Dpt. of Computer Science, Cornell Univerity, Ithaca, New York, May 1991. 11 pages.

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

[16] M. Raynal, M. Mizuno, and M.L. Neilsen. Synchronization and concurrency measure for distributed computations. In *Proc. of the 12th IEEE International Conference on Distributed Computing Systems*, pages 700–707, Yokohama, Japan, June 1992.

[17] R. Schwarz and F. Mattern. *Detecting Causal Relationships in Distributed Computations : In Search of the Holy Grail.* To appear in Distributed Computing 7(4), 1994.

[18] E. Szpilrajn. Sur l'extension de l'ordre partiel. *Fund. Math.*, 16:386–389, 1930.

[19] A.I. Tomlinson and V.K. Garg. Detecting relational global predicates in distributed systems. In *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 21–31, San Diego, California, May 1993.

[20] S. Venkatesan and B. Dathan. Testing and debugging distributed programs using global predicates. In *Proc. of the Thirtieth Annual Allerton Conference on Communication, Control and Computing*, pages 137–146, Urbana, Illinois (USA), October 1992.

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