Detecting atomic sequences of predicates in distributed computations

Michel Hurfin, Noël Plouzeau, Michel Raynal

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Detecting atomic sequences of predicates in distributed computations

Michel HURFIN
Noël PLOUZEAU
Michel RAYNAL

N° 1872
Mars 1993

PROGRAMME 1

Architectures parallèles,
Bases de données,
Réseaux et Systèmes distribués

IRISA

INSTITUT DE RECHERCHE
EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 RENNES CEDEX FRANCE
Tél: 99 84 71 00 - Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Detecting Atomic Sequences of Predicates in Distributed Computations *

Michel Hurfin, Noël Plouzeau et Michel Raynal
Equipe ADP - Programme 1
e-mail: <Name>@irisa.fr

Abstract

This paper deals with a class of unstable non-monotonic global predicates, called herein *atomic sequences of predicates*. Such global predicates are defined for distributed programs built with message-passing communication only (no shared memory) and they describe global properties by causal composition of local predicates augmented with atomicity constraints. These constraints specify forbidden properties, whose occurrence invalidate causal sequences. This paper defines formally these atomic sequences of predicates, proposes a distributed algorithm to detect their occurrences and gives a sketch of a proof of correctness of this algorithm.

Détection de séquences atomiques de prédicats dans les exécutions réparties

Résumé

Cet article traite d'une classe de propriétés globales instables et non monotones appelées *séquences atomiques de prédicats*. De tels prédicats globaux sont utilisés pour analyser les programmes répartis où les communications se font par le biais de messages (pas de mémoire partagée). Ils permettent de décrire des propriétés globales sous forme de séquences causales de prédicats locaux. Ces séquences sont enrichies par des contraintes d'atomicité qui stipulent des événements qui ne doivent pas apparaître (i.e. dont l'occurence invalide une séquence causale).
Cet article donne une définition formelle de ces prédicats, montre leurs intérêts, présente un algorithme réparti capable de les détecter et dévoile dans ses grandes lignes la démarche suivie pour prouver l'algorithme.

* This work has been partly funded by a CNRS grant on the study of parallel traces and by the Basic Research Action number 6360 (BROADCAST) of the ESPRIT programme of the Commission of European Communities.

This paper will appear in the proceedings of the ACM Conference on Parallel and Distributed Debugging, San Diego, May 1993.

Detecting Atomic Sequences of Predicates in Distributed Computations *†

Michel HURFIN Noël PLOUZEAU Michel RAYNAL

IRISA – Campus de Beaulieu – 35042 RENNES Cedex – FRANCE
{name}@irisa.fr – Fax: +33 99 38 38 32

Abstract

This paper deals with a class of unstable non-monotonic global predicates, called herein *atomic sequences of predicates*. Such global predicates are defined for distributed programs built with message-passing communication only (no shared memory) and they describe global properties by causal composition of local predicates augmented with atomicity constraints. These constraints specify forbidden properties, whose occurrence invalidate causal sequences. This paper defines formally these atomic sequences of predicates, proposes a distributed algorithm to detect their occurrences and gives a sketch of a proof of correctness of this algorithm.

1 Introduction

Analyzing a distributed program and checking it against behavioral properties are two difficult topics [11]. Such an analysis may be done statically, i.e. by structural analysis [11] or dynamically by examining a set of behaviors exhibited during executions. The current paper deals with this second kind of analysis, and focus on detecting unstable non-monotonic properties specified as atomic sequences.

Most properties useful to the computer scientist interested in distributed program analysis refer to global states of distributed computations. But evaluating global predicates (*i.e.* predicates on global states) is notoriously a difficult task in a distributed context, because there is no real global state but only a set of local states whose evaluation cannot be done instantaneously . Research efforts in the distributed program analysis and debug field have produced interesting results for evaluating stable properties [2, 6].

While detecting unstable properties is notably more difficult than in the case of stable ones, since their occurrences are transient, interesting results have been obtained, for instance by Haban and Weigel [5], Miller and Choi [12], Garg and Waldecker [4], and Cooper and Marzullo [3]. The current paper exposes results belonging to a context similar to the first three ones, but focuses on a new class of global predicates, named hereafter atomic sequences. Informally speaking, such sequences are defined by a pair of sequences of local predicates: expected properties and unwanted properties, which should not occur during a computation. Miller and Choi [12], as well as Garg and Waldecker [4] have published a solution for detecting sequences when the unwanted properties sequence is omitted, in other words these works focus on detecting occurrences of expected sequences of local predicates. Haban and Weigel in [5] give an implementation for detecting atomic sequences of length two.

When atomicity constraints are omitted, sequences exhibit a kind of monotonicity property with respect to prefix occurrence detection [15]: if a sequence is made of three predicates then it is sufficient to detect each predicate in turn, with no need to discard solutions later; the length of predicates already satisfied is a non-decreasing function of time. Atomic sequences do not have this monotonicity property [15]; a prefix of predicate sequence may have been found satisfied by a computation at some time and is then invalidated by occurrence of a forbidden event.

The current paper answers this problem: detecting a sequence of m local predicates, while other predicates continuously evaluate to *false*. Moreover, our algorithm is able to count how many times the atomic sequence

*This work has been partly funded by a CNRS grant on the study of parallel traces and by a Basic Research Action #6360 (Broadcast) of the Esprit Programme of European Communities Commission.

†This paper will appear in the proceedings of the ACM Conference on Parallel and Distributed Debugging, San Diego, May 1993.

was satisfied by the computation. Before this new algorithm is exposed in Section 4, formal definitions of the computation model and of the predicate semantics are given in Sections 2 and 3; a sketch of proof of the detection algorithm is given in Section 5.

2 A model for distributed computations

2.1 Structure of distributed programs

In this paper we consider programs made of n processes $(P_1, P_2, \ldots, P_n)$, which communicate and synchronize by the only means of message passing. These messages are exchanged through communication channels, whose transmission delays are arbitrary but finite. Communication is loss and error-free, but FIFO delivery of message on a channel is not required. Neither shared memory nor a global clock are assumed available. In other words, we consider a *distributed asynchronous* computation model.

2.2 Distributed executions

Events occurring during a process execution are of three types: message emission, message receipt and internal event. Initial state of process P_i is noted s_i^0. Process P_i's p^{th} event occurrence is noted e_i^p; when this event occurs, P_i's state changes from s_i^{p-1} to s_i^p. Hereafter process P_i's local history during a given computation is defined by sequence $h_i = s_i^0 s_i^1 s_i^2 \ldots$. Thus the whole computation history is noted by a tuple made of the n local histories:

$$H = (h_1, h_2, \ldots, h_i, \ldots, h_n)$$

The set of local states reached by a computation is structured by a partial order relation noted $\longrightarrow$, defined as follows:

$$\forall s_i^p, \forall s_j^q \quad s_i^p \longrightarrow s_j^q \Longleftrightarrow \begin{cases} (i = j) \wedge (p < q) \\ \text{or} \\ \text{There exists a message } a \text{ such that} \\ \quad e_i^{p+1} \text{ is emission of } a \text{ to } P_j \\ \quad e_j^q \text{ is receipt of } a \text{ from } P_i \\ \text{or} \\ \text{There exists a local state } s_k^r \text{ such that:} \\ \quad s_i^p \longrightarrow s_k^r \longrightarrow s_j^q \end{cases}$$

This definition is nothing else than Lamport's relation [9] applied to local states instead of events. In order to ease definitions in the rest of this paper, we also define an abstract state s^{-1} defined by $\forall i, s^{-1} \longrightarrow s_i^0$. A distributed execution is thus defined by a partially ordered set $(H, \longrightarrow)$.

2.3 Causal past and causal future

For every local state s_i^p two sets are defined as follows:

$$\begin{aligned} \text{past } s_i^p &= \{s_j^q \mid s_j^q \longrightarrow s_i^p\} \\ \text{future } s_i^p &= \{s_j^q \mid s_i^p \longrightarrow s_j^q\} \end{aligned}$$

3 Sequence of predicates with atomicity constraints

3.1 Local and global predicates

A predicate LP_i is *local* to process P_i if it only mentions variables local to P_i. We write $s_i^p \models LP_i$ when LP_i evaluates to true in state s_i^p. Let LP^0 be an ancillary and abstract predicate which evaluates to true in state s^{-1} only. Every event occurrence may be designated by an *ad hoc* local predicate. We will use such predicates to pin-point relevant events.

3.2 Global predicates

Several definitions are possible for the global predicate concept. A first possible definition refers to comparing values of variables taken in different processes; for instance, if variables x_1 and x_2 belong to processes P_1 and P_2 respectively then predicate $x_1 > x_2+4$ is global, because it involves two different processes. Detecting such predicates with particular modalities (*definitely* or *possibly*) has been studied by Cooper and Marzullo in [3] (see also [1] for a nice presentation), and implies the construction of the lattice of all possible global states. This construction is very expensive but seems unavoidable when dealing with this kind of global predicates.

A different type of global predicates relies on composing local predicates in several ways. For instance, Miller and Choi define sequences of local predicates in [12], Garg and Waldecker propose sequences, conjunctions and disjunctions of local predicates in [4]. In [5] Haban and Weigel design a language for local predicate composition; although no formal semantics is given for this language, it introduces an interesting negation operator in local predicates pairs. In the current paper this negation operator is used for sequences of arbitrary length.

3.3 Simple sequence of local predicates

Let LP_i be a predicate local to process P_i. Detecting whether there exists local states s_i^p where LP_i is satisfied (*i.e.* such that $s_i^p \models LP_i$) is a trivial problem, as is counting these states.

More generally, a sequence of m local predicates $LP_{i_1}^1; LP_{i_2}^2; \ldots; LP_{i_m}^m$ defines a predicate $SEQ^{(1,\ldots,m)}$ whose solutions are sequences of local states $(s_{i_1}^{p_1}; s_{i_2}^{p_2}; \ldots; s_{i_m}^{p_m})$ such that:

$$(s_{i_1}^{p_1}; s_{i_2}^{p_2}; \ldots; s_{i_m}^{p_m}) \models SEQ^{(1,\ldots,m)} \Longleftrightarrow (\bigwedge_{u=1}^{m} (s_{i_u}^{p_u} \models LP_{i_u}^u)) \wedge (\bigwedge_{u=2}^{m} (s_{i_u}^{p_u} \in \text{future } s_{i_{u-1}}^{p_{u-1}}))$$

An interesting problem is then detecting whether this predicate has solutions, and how many times was this predicate satisfied by a given computation. Predicate $SEQ^{(1,\ldots,m)}$ is a formalization of the following question: did some given event occur on process P_{i_1} (*i.e.* does a local state $s_{i_1}^{p_1}$ exist, such that $s_{i_1}^{p_1} \models LP_{i_1}^1$), and was this event occurrence causally followed by another given event occurrence on process P_{i_2} (expressed by $s_{i_2}^{p_2} \models LP_{i_2}^2$), and so on?

Two sequences of local states are distinct if and only if they differ by at least one local state. Figure 1 displays an example where squares stand for local states; white squares represent local states which do not satisfy any local predicate, while grey squares indicate that the corresponding state satisfies at least one local predicate; in this last case the rank within sequence $SEQ^{(1,\ldots,m)}$ of the predicate satisfied is written above the square. For instance, the eighth local state of process P_i satisfies predicate LP_i^4 as indicated by number 4 above the corresponding square.

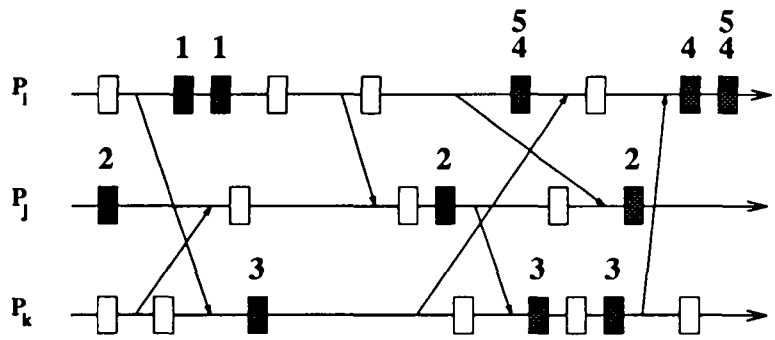

Figure 1: Simple sequence

Moreover, Figure 1 illustrates the following facts:

- Predicate $SEQ^{(1,\ldots,4)}$ corresponding to sequence $LP_i^1; LP_j^2; LP_k^3; LP_i^4$ has eight solutions,
- Sequence $LP_i^1; LP_j^2; LP_k^3; LP_i^4; LP_i^5$ (noted $SEQ^{(1,\ldots,5)}$) has only four solutions.

3.4 Sequence with atomicity constraints

In some situations, a sequence of predicates satisfied by a sequence of local states is a valid solution only if some events did not occur between the first and last events satisfying the sequence. In this case the property to be detected is not monotonic, because when a forbidden event occurs any sequence prefix already satisfied can no longer be prefix of a solution. Hence acceptable solutions are free of occurrence of forbidden events; for this reason we name them *atomic sequences.*

A forbidden event may occur on any process, so atomicity constraints are defined by a conjunction of local predicates in the following way:

$$CA = \neg NP_1 \wedge \neg NP_2 \wedge \ldots \wedge \neg NP_n$$

Hence atomicity is broken (*i.e.* CA becomes false) as soon as there exists a local state s_i^p of some process P_i such that $s_i^p \models NP_i$. If process P_i is not involved in constraint CA then NP_i always evaluates to *false.*

Let $SEQ^{(1,2)}$ be a sequence $LP_{i_1}^1 ; LP_{i_2}^2$, and let $CSEQ^{(1,2)}$ be a new sequence defined from $SEQ^{(1,2)}$ and CA; solutions to $CSEQ^{(1,2)}$ are defined as follows:

$$(s_{i_1}^{p_1} ; s_{i_2}^{p_2}) \models CSEQ^{(1,2)} \Longleftrightarrow \begin{cases} (s_{i_1}^{p_1} ; s_{i_2}^{p_2}) \models SEQ^{(1,2)} \\ \text{and} \\ \forall i, 1 \leq i \leq n,\ \forall s, s \in h_i \cap \text{future } s_{i_1}^{p_1} \cap \text{past } s_{i_2}^{p_2} : \\ \quad (s \models \neg NP_i) \end{cases}$$

Figure 2 illustrates this definition. Sequence $CSEQ^{(1,2)}$ is verified only if none of the NP_i is satisfied by local states within the grayed out area.

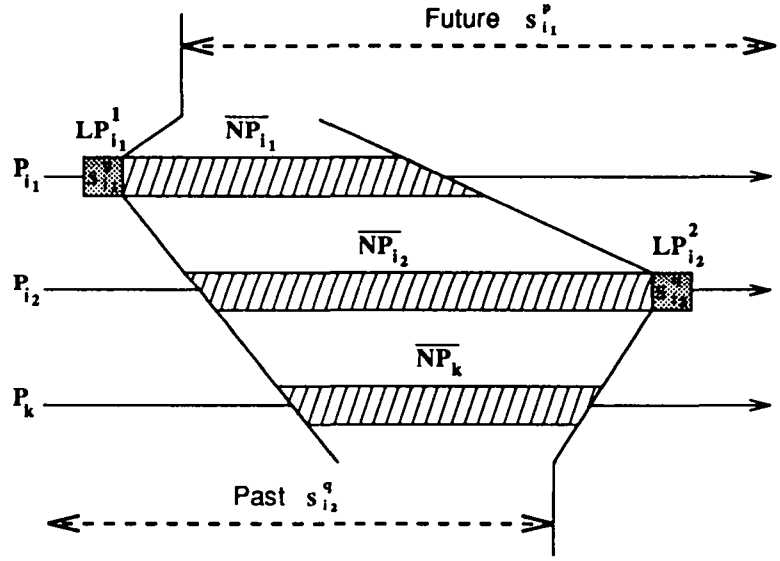

Figure 2: Sequence of length 2 with an atomicity constraint

Atomic sequences may be extended to lengths greater than 2. Sequence $CSEQ^{(1,\ldots,m)}$ is then defined by the composition of a sequence of m local predicates $LP_{i_1}^1 ; LP_{i_2}^2 ; \ldots ; LP_{i_m}^m$ and of a sequence of $m-1$ atomicity constraints $CA^2 ; CA^3 ; \ldots ; CA^m$. Each atomicity constraint CA^u specifies events whose occurrences are forbidden between detection of $LP_{i_{u-1}}^{u-1}$ (at depth $u-1$) and detection of $LP_{i_u}^u$ (at depth u):

$$CA^u = \neg NP_1^u \wedge \neg NP_2^u \wedge \ldots \wedge \neg NP_n^u$$

A set of local states $(s_{i_1}^{p_1}; s_{i_2}^{p_2}; \ldots; s_{i_m}^{p_m})$ satisfies $CSEQ^{(1,\ldots,m)}$ if and only if

$$\begin{cases} & (s_{i_1}^{p_1}; s_{i_2}^{p_2}; \ldots; s_{i_m}^{p_m}) \models SEQ^{(1,\ldots,m)} \\ \text{and} & \\ & \forall u, 2 \leq u \leq m, \ \forall i, 1 \leq i \leq n, \\ & \quad \forall s, \quad s \in h_i \cap \text{future } s_{i_{u-1}}^{p_{u-1}} \cap \text{past } s_{i_u}^{p_u} \ : (s \models \neg NP_i^u) \end{cases}$$

In this form predicate $CSEQ^{(1,\ldots,m)}$ cannot specify that some events occurring before detection of $LP_{i_1}^1$ are forbidden. To lift this restriction, sequence $LP_{i_1}^1; LP_{i_2}^2; \ldots; LP_{i_m}^m$ is extended to $LP^0; LP_{i_1}^1; \ldots; LP_{i_m}^m$ and an initial atomicity constraint CA^1, describing unwanted initial events, is added to the sequence of atomicity constraints, giving $CA^1; CA^2; \ldots; CA^m$. These new forms together define predicate $CSEQ^{(0,\ldots,m)}$. As by definition predicate LP^0 is satisfied in state s^{-1}, solutions to $CSEQ^{(0,\ldots,m)}$ have the form $(s^{-1}; s_{i_1}^{p_1}; s_{i_2}^{p_2}; \ldots; s_{i_m}^{p_m})$. Note that if $CA^1 = CA^2 = \ldots = CA^m = true$ then $CSEQ^{(0,\ldots,m)} = SEQ^{(1,\ldots,m)}$. The following notation is used to represent $CSEQ^{(0,\ldots,m)}$:

$$CSEQ^{(0,\ldots,m)} =_{\text{def}} LP^0; [CA^1] LP_{i_1}^1; [CA^2] LP_{i_2}^2; \ldots; [CA^m] LP_{i_m}^m$$

Figure 3 shows a simple example, and Figure 4 indicates for five different global predicates the count of solutions detected by the algorithm of Section 4.

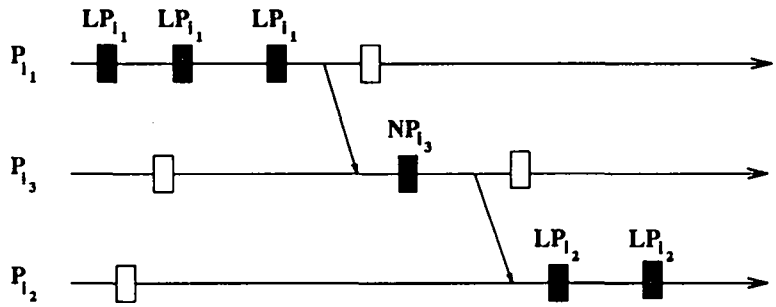

Figure 3: Sequence with atomicity constraints

Predicates	Number of solutions
$LP_{i_1}\ ; LP_{i_2}$	6
$LP_{i_1}\ ;\ [\neg(\ LP_{i_1}\ )]\ LP_{i_2}$	2
$LP_{i_1}\ ;\ [\neg(\ LP_{i_2}\ )]\ LP_{i_2}$	3
$LP_{i_1}\ ;\ [\neg(\ LP_{i_1}\ ) \wedge \neg(\ LP_{i_2}\ )]\ LP_{i_2}$	1
$LP_{i_1}\ ;\ [\neg(\ NP_{i_3}\ )]\ LP_{i_2}$	0

Figure 4: Solutions detected

3.5 Related work

Algorithms for detecting existence of solutions for sequence of predicates without atomicity constraints are given in [12, 4]; these algorithms do not give the count of solutions. In [5] an algorithm for detecting atomicity constraints is presented; a predicate noted@(A, B, C) specifies that predicate A and predicate B should be satisfied, with a causal dependency between them, and that predicate C should not be satisfied in between. This algorithm is limited to atomic sequences of length two, and no formal definition of the predicate detected is given (see [14] for a critique of this solution).

3.6 Use of these global predicates

When used to analyze or debug distributed computations, simple sequences and sequences with atomicity constraints basically address the two fundamental properties a programmer is interested in, namely safety and liveness. More specifically, simple sequences allow to describe execution paths and consequently express liveness properties ("will something – here this path – happen?"). On the other hand sequences with atomicity constraints are more suited to invalidate an execution path that occurred and thus allow expression of safety properties in a negative form ("do bad things happen?").

Counting the number of solutions (as does algorithm of Section 4) is of course meaningless if the number of solution is huge. In that case only boolean answers are interesting. But the number of solutions depends on the form of the global predicate, and so the user defining $LP^u_{i_*}$ predicates (liveness side of the searched for property) and CA^u predicates (safety side) must be careful to avoid specifying trivial properties.

Lastly, counting solutions of sequences of predicates has an interesting property. The interested reader can verify the following result. Let $\#(GP)$ the number of solutions of global predicate GP. If A and B are two predicates local to processes P_1 and P_2 then we have : $\#(A \wedge B) = (\#(A) * \#(B))$ - $(\#(A;B) + \#(B;A))$. The left member of the equality is the number of consistent global states whose local states of P_1 and P_2 concurrently satisfy A and B. As shown by the equation, this number can be computed by the algorithm by the calculation of the four values : $\#(A)$,$\#(A;B)$,$\#(B)$ and $\#(B;A)$.

4 The detection algorithm

The algorithm presented in this Section detects unstable non-monotonic properties expressed by a predicate of the form $CSEQ^{(0,\ldots,m)}$ and computes the number of solutions that satisfy this predicate.

4.1 Local observers

Each main computation process P_i is associated with an observer process OBS_i whose task is first to detect P_i's states satisfying relevant local predicates ($LP^u_{i_*}$ and NP^u_i) and second to cooperate with the other observers in order to evaluate $CSEQ^{(0,\ldots,m)}$. To this aim, each observer uses the following local constants and variables:

- `LP : array[1..m] of predicate;`

  `LP[u]` is initialized to LP^u_i if $CSEQ^{(0,\ldots,m)}$ includes such a predicate, or to `false` otherwise. Let us recall that for a given depth u there is one and only one process whose LP^u_i is not trivially false.

- `NP : array[1..m] of predicate;`

  `NP[u]` is initialized to NP^u_i if such a predicate exists in $CSEQ^{(0,\ldots,m)}$, or to `false` otherwise.

- `NB_SOL_CSEQ : array[0..m] of integer;`

  `NB_SOL_CSEQ[u]` contains the current count of distinct solutions to $CSEQ^{(0,\ldots,u)}$, to the knowledge of observer OBS_i. In the initial state, `NB_SOL_CSEQ[0]=1` holds, by definition of LP^0, and in this state $\forall u, 1 \leq u \leq m$: `NB_SOL_CSEQ[u] = 0`.

- `NB_SOL_INV_CSEQ : array[0..m-1] of integer initialized to 0;`

NB_SOL_INV_CSEQ[u-1] is OBS_i's local count of solutions to prefix $LP^0; [CA^1]LP^1_{i_1}; \ldots; [CA^{u-1}]LP^{u-1}_{i_{u-1}}$ which have been invalidated because CA^u evaluated to **false**. The set of solutions that are counted in NB_SOL_INV_CSEQ[u-1] are not valid prefixes of solutions to $CSEQ^{(0,\ldots,u)}$.

- DEPTH : integer initialized to 1;

  This variable indicates OBS_i's local knowledge of the current depth in $CSEQ^{(0,\ldots,m)}$ of the search for solutions. OBS_i participates to the observation of P_i's local predicates and to the detection of new solutions to $CSEQ^{(0,\ldots,u)}$ for $u <$ DEPTH and to the detection of the first solution to $CSEQ^{(0,\ldots,\text{DEPTH})}$. Initial value of DEPTH is 1.

4.2 Cooperation between observers

The causality relation defined between local states of processes depends on message exchanges (see Section 2.2). Any pair of observer(OBS_i, OBS_j) needs to know this causality relation and thus they have to exchange some information about message exchanged between P_i and P_j. The classical technique of piggybacking is used to transmit control information between OBS_i and OBS_j, as follows. Suppose that process P_i sends a message m to process P_j. OBS_i appends to m information about its local knowledge of solutions to $CSEQ^{(0,\ldots,m)}$, more precisely the following data is appended to m: OBS_i's DEPTH value, vectors NB_SOL_CSEQ[0..DEPTH-1] and NB_SOL_INV_CSEQ[0..DEPTH-1]. Upon receiving this information, OBS_j updates its own variables.

4.3 The algorithm

An observer OBS_i handles three kinds of events when they occur in process P_i:

- emission of a message by P_i: OBS_i appends control data to the message, as explained above;
- receipt of a message by P_i: OBS_i extracts the control part of the message and updates its local variables;
- some local predicate of P_i (*i.e.* LP_i^u or NP_i^u) that belongs to $CSEQ^{(0,\ldots,\text{DEPTH})}$ (*i.e.* $1 \leq u \leq$ DEPTH) evaluates to true in the current local state s_i^p of P_i: OBS_i examines predicate $CSEQ^{(0,\ldots,\text{DEPTH})}$, then predicate $CSEQ^{(0,\ldots,\text{DEPTH-1})}$, etc till predicate $CSEQ^{(0,\ldots,1)}$.
  For each of these predicates $CSEQ^{(0,\ldots,u)}$, $1 \leq u \leq$ DEPTH, OBS_i checks:
  - whether new solutions exist (this is the case when local predicate LP_i^u evaluates to true in the current local state s_i^p).
  - whether some solutions to $CSEQ^{(0,\ldots,u-1)}$ are invalidated (this is the case when predicate NP_i^u evaluates to true in s_i^p).

  OBS_i updates its context by incrementing NB_SOL_CSEQ[u] in the first case, and NB_SOL_INV_CSEQ[u-1] in the second case. OBS_i updates NB_SOL_CSEQ[u] only if valid solutions to $CSEQ^{(0,\ldots,u-1)}$ still exist.

  More formally, OBS_i's behavior is defined by the next three statements.

S1 : When P_i is in a local state s_i that verifies LP_i^u or NP_i^u for $1 \leq u \leq$ DEPTH

```
begin
for u := DEPTH downto 1 do
    begin
    if (NB_SOL_CSEQ[u-1] > NB_SOL_INV_CSEQ[u-1]) then
      begin
      if LP[u] then
        begin
        NB_SOL_CSEQ[u] := NB_SOL_CSEQ[u] + (NB_SOL_CSEQ[u-1] - NB_SOL_INV_CSEQ[u-1]);
        if ((u = DEPTH) and (DEPTH < m)) then DEPTH := DEPTH + 1; fi; (σ)
        end;
      fi;
```

```
        if NP[u] then
          NB_SOL_INV_CSEQ[u-1] := NB_SOL_CSEQ[u-1];
        fi;
        end;
      fi;
      end;
    end;
```

S2 : When P_i sends a message a

```
    begin
    Add (DEPTH,NB_SOL_CSEQ[0..DEPTH-1],NB_SOL_INV_CSEQ[0..DEPTH-1]) to message a;
    end;
```

S3 : When P_i receives a message (a,d,T1,T2)

```
    begin
    Deliver a to P_i;
    DEPTH := max(DEPTH,d);
    for u := 0 to (d-1) do
        begin
        NB_SOL_CSEQ[u] := max(NB_SOL_CSEQ[u],T1[u]);
        NB_SOL_INV_CSEQ[u] := max(NB_SOL_INV_CSEQ[u],T2[u]);
        end;
    end;
```

4.4 Remarks

Every observer OBS_i manages two vectors LP and NP to store predicates LP_i^u and NP_i^u. Arrays are used here for ease of exposition, but they are replaceable for efficiency by two lists of (depth of predicate , predicate) pairs (predicates trivially constantly evaluating to false are not included in these lists).

Checking for satisfaction of local predicates does not necessarily involve repeated evaluations of these predicates; classical techniques for process reactivation in conditional critical regions can be used [13].

If one is not interested in counting solutions to $CSEQ^{(0,\ldots,m)}$ a simple modification halts the observer set upon the first occurrence of a solution: the following line should be inserted at the place marked with σ (*i.e.* as soon as a solution is found).

```
if u = m then
  broadcast STOP to OBS_j, ∀j, 1 ≤ j ≤ n
endif
```

This broadcast may be implemented by piggybacking also. In this simplified case of first occurrence detection, better modifications of the algorithm are also possible. When predicate $LP_{i_u}^u$'s value changes from false to true, no new evaluation of this predicate is needed until a message is sent (Figure 5) or predicate NP_i^{u+1} evaluates to true (Figure 6).

At the end of the computation, for the only observer OBS_i such that LP[u] has not been initialized to false, the variable NB_SOL_SEQ[u] indicates how many prefix solutions of length u were found; consequently, it is possible to find out depth v such that $CSEQ^{(0,\ldots,v)}$ is satisfied at least once and $CSEQ^{(0,\ldots,v+1)}$ is never satisfied (if $v < m$): v is the length of the longuest prefix for which partial solutions exist. Moreover, an algorithm for breakpoints computation such as those given in [10, 12] can be synchronized with the present algorithm and be triggered by detection of solutions to $CSEQ^{(0,\ldots,u)}$ sequences. Moreover in an interactive context (for instance in a distributed debugger), sequence $CSEQ^{(0,\ldots,m)}$ may be defined dynamically.

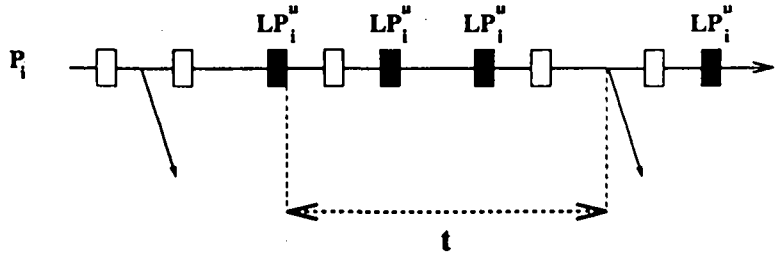

Figure 5: No new evaluation of LP_i^u is needed during interval t

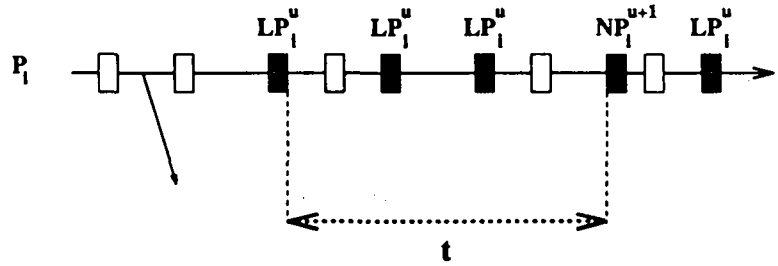

Figure 6: No new evaluation of LP_i^u is needed during interval t

5 Sketch of the proof

5.1 Notations

Before P_i produces its first event e_i^1, the associated local observer OBS_i calls procedure **test** (that encapsulates the statement S1 of the algorithm). This first invocation is denoted $\mathbf{test}_i^0$. OBS_i then executes this procedure after each event e_i^p produced by P_i. The invocation which follows e_i^p is denoted $\mathbf{test}_i^p$.

Execution of the procedure **test** has no effect on the current local state of P_i. Only control variables (DEPTH,NB_SOL_CSEQ,NB_SOL_INV_CSEQ) can be modified. Let c_i^{2p} (respt c_i^{2p+1}) the state of all these control variables just before (respt after) the execution of $\mathbf{test}_i^p$. Event e_i^p can modify local variables of P_i and local control variables of OBS_i provoking a transition from the pair of states (s_i^{p-1}, c_i^{2p-1}) to the pair of states (s_i^p, c_i^{2p}). The sequence of all these modifications is represented in Figure 7. Let us note that $c_i^{2p-1} \neq c_i^{2p}$ only if e_i^p is a message reception (see statement S3).

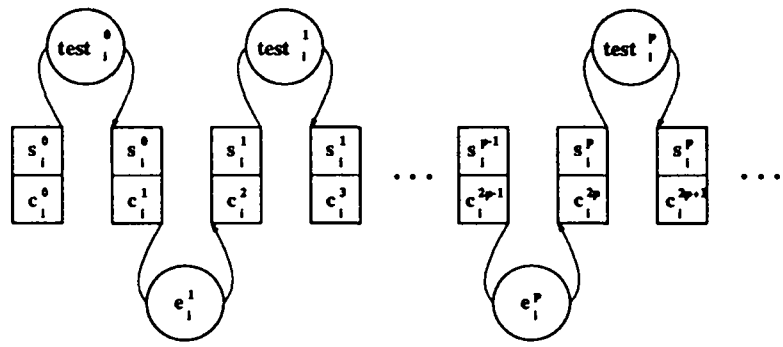

Figure 7: The local states (s) and the states of the control variables (c) of P_i

The successive values of a control variable V of OBS_i are denoted:

$$V_i^0, V_i^1, \cdots, V_i^{2p}, V_i^{2p+1}, \cdots$$

5.2 Statement of safety and liveness properties

Recall that at any depth u there is only one local predicate LP[u] distinct from false; so let P_{i_u} the only process to which $LP^u_{i_u}$ is associated.

Let $\chi(u,p)$ be the number of solutions of the predicate $CSEQ^{(0,\ldots,u)}$ when s^p_i is the last local state of process P_{i_u}. More formally:

$$\chi(u,p) = Cardinal(\{ (s^{-1} ; s^{p_1}_{i_1} ; \ldots ; s^{p_u}_{i_u}) \mid (0 \leq p_u \leq p) \wedge (s^{-1} ; s^{p_1}_{i_1} ; \ldots ; s^{p_u}_{i_u}) \models CSEQ^{(0\ldots u)} \})$$

The safety property expresses the detection is consistent (*i.e.* the number of solutions found don't exceed the actual number of solutions). This is formally stated by:

$$\left.\begin{array}{l} \forall u,\ 1 \leq u \leq m \\ \forall s^p_{i_u},\ s^p_{i_u} \in h_{i_u} \end{array}\right\} \quad \texttt{NB\_SOL\_CSEQ}^{2p+1}_{i_u}\texttt{[u]} \leq \chi(u,p)$$

The liveness property expresses the detection is effectively done (*i.e.* each actual solution is found). This is formally stated by:

$$\left.\begin{array}{l} \forall u,\ 1 \leq u \leq m \\ \forall s^p_{i_u},\ s^p_{i_u} \in h_{i_u} \end{array}\right\} \quad \texttt{NB\_SOL\_CSEQ}^{2p+1}_{i_u}\texttt{[u]} \geq \chi(u,p)$$

The proof of these properties is done by recurrence on u (the depth of the predicate) and on p (the rank of the local state considered). Due to space limitations, the reader is referred to [7] that displays this (technical and long) proof.

6 Conclusion

In this paper a class of unstable non-monotonic global properties has been defined and studied. These unstable properties are global since they refer to local predicates from distinct distributed processes, and non-monotonic because partial solutions to these properties may be rejected by occurrence of invalidating predicates, named atomicity constraints. An algorithm detecting occurrence of atomic sequences has been given and a sketch of its proof presented. These sequences are useful in analyzing and debugging distributed programs. This algorithm is implemented in the distributed debugger Erebus [8].

Acknowledgments

The authors wish to thank C. Jard and C. Maziero for interesting discussions on distributed program analysis.

References

[1] O. Babaoglu and K. Marzullo. Consistent global states of distributed systems: fundamental concepts and mechanisms. In S. Mullender, editor, *Distributed Systems 2nd edition, ACM Press, Frontier Series*, 1993.

[2] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM TOCS*, 3(1):63–75, 1985.

[3] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, California, May 1991.

[4] V.K. Garg and B. Waldecker. Detection of unstable predicates in distributed programs. In *Proc. of the 12th conf. Foundations of Software Technology and Theoretical Computer Science*, pages 253–264, Lecture Notes in Computer Science 652, Springer-Verlag, New Delhi, India, December 1992.

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

[6] J.-M. Hélary, C. Jard, N. Plouzeau, and M. Raynal. Detection of stable properties in distributed applications. 6^{th} *ACM SIGACT-SIGOPS, Symp. Principles of Distributed Computing, Vancouver, Canada*, 125–136, 1987.

[7] M. Hurfin. Réexécution et analyse de la dynamique des programmes répartis (Replaying and analysing the dynamics of distributed executions). Thèse, Université de Rennes I, (to appear), 1993.

[8] M. Hurfin, N. Plouzeau, and M. Raynal. A debugging tool for Estelle distributed programs. *Journal of Computer Communications*, May 1993.

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

[10] Y. Manabe and M. Imase. Global conditions in debugging distributed programs. *Journal of Parallel and Distributed Computing*, 15:62–69, 1992.

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

[12] B.P. Miller and J. Choi. Breakpoints and halting in distributed programs. In *Proc.* 8^{th} *IEEE Int. Conf. on Distributed Computing Systems, San Jose*, pages 316–323, July 1988.

[13] H.A. Schmid. On the efficient implementation of conditionnal critical regions and the construction of monitors. *Acta Informatica*, 6:227–249, 1976.

[14] R. Schwarz and F. Mattern. *Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail.* Technical Report 215/91 (36 pp), University of Kaiserslautern, 1991.

[15] M. Spezialetti and J.P. Kearns. A general approach to recognizing event occurrences in distributed computations. In 8^{th} *IEEE Int. Conf. on Distributed Computing Systems, San Jose*, pages 300–307, July 1988.

Unité de Recherche INRIA Rennes
IRISA, Campus Universitaire de Beaulieu 35042 RENNES Cedex (France)

Unité de Recherche INRIA Lorraine Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 VILLERS LES NANCY Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 GRENOBLE Cedex (France)
Unité de Recherche INRIA Rocquencourt Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)
Unité de Recherche INRIA Sophia Antipolis 2004, route des Lucioles - B.P. 93 - 06902 SOPHIA ANTIPOLIS Cedex (France)

EDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

ISSN 0249 - 6399