ELSEVIER

Information Processing Letters 54 (1995) 267-274

Information Processing Letters

On-the-fly analysis of distributed computations

Eddy Fromentin [1], Claude Jard [1], Guy-Vincent Jourdan [1], Michel Raynal *

IRISA - Campus de Beaulieu, 35042 Rennes cedex, France

Communicated by S. Zaks; received 14 October 1994; revised 23 February 1995

Abstract

At some abstraction level a distributed computation can be modeled as a partial order on a set of observable events. This paper presents an analysis technique which can be superimposed on distributed computations to analyze the structure of control flows terminating at observable events. A general algorithm working on the longest control flows of distributed computations is introduced. Moreover it is shown how this algorithm can be simplified according to the position of observable events with respect to communication events.

Keywords: Distributed computation; Observable event; Control flows; Causal precedence; Sequence analysis

1. Introduction

One important topic addressed by computer science is the analysis of sequences of symbols or words (e.g. syntactical analysis, pattern recognition, etc.). Well-suited formalizations have been designed and specialized tools have been implemented to make feasible such analyses in specific domains (formal languages and automata theory are the most famous example of these works).

Computations performed by distributed computing systems do not yield a linear sequence of events. The relationship between events inherently define a partial ordering. For a particular computation, events produced by each process are totally ordered and communications creates dependencies among events belonging to distinct processes. Since Lamport's seminal paper [8], this partial order relation on events is generally called *happened before* (with respect to a logical time frame) or *causal precedence*. It formally expresses control flows and their mutual dependencies which organize the distributed execution.

In this paper we are interested in analyzing "on-the-fly" the set of "words" produced by a distributed computation; a word being defined From sequences of observable events produced by an execution of a distributed program. The practical motivation of our work comes from debugging, testing and monitoring of distributed systems [5,7]. In this context we choose to explore analysis techniques which must be run on-the-fly and without delay (this is particularly important in the context of reactive monitoring). These constraints eliminate the possibility to log events produced by each process (as the analysis cannot be done off-line) or to use an additional process (monitor) that would receive notification messages sent by processes of the computation in order to analyze their traces (in that case notification messages would add some delay between event occurrences and their knowledge by the monitor).

* Corresponding author. Email: raynal@irisa.fr.
[1] Email: {fromentin,jard,jourdan}@irisa.fr.

0020-0190/95/$09.50 © 1995 Elsevier Science B.V. All rights reserved
SSDI 0020-0190(95)00048-8

In other words we constraint our analysis mechanism to be superimposed on the computation and to use only a piggybacking technique to convey analysis related information from one process to another.

According to the aim of the analysis (detection of a property, for example) only a subset of all the events generated by a distributed execution are meaningful to the user: these events are called "observable events". From this point of view, the other events are ignored at the abstraction level considered; they participate only in the establishment of causal dependencies between observable events. Sequences of observable events are defined by the chains of the partial order relation associated with the computation. Although each observable event of the computation is unique, several events can be occurrences of the same action. So a labeling function is introduced and the sequences of observable events are associated with words (concatenation of labels). The properties considered in this paper are described as finite state automata. The analysis consists of checking whether the pattern of actions defined by the automaton occurs in the words defined by the considered computation. The analysis is carried out on-the-fly without delay on these words. Other kinds of automata could be used but finite state automata are sufficient to illustrate our analysis technique and to solve efficiently practical problems (algorithms presented in [3,4,10] are particular instances of our general algorithm as properties they detect can be formalized by finite state automata).

The paper is divided into four main sections.

Section 2 presents our model of distributed computations.

Section 3 presents the kind of specification or properties we are interested in, and gives their formal interpretation: the definition of languages (set of words) associated with distributed computations and the basic question (satisfaction rule) which can be answered by the analysis.

Section 4 presents a general distributed algorithm which, in this context, analyzes on the fly and without delay a distributed computation.

Section 5 examines particular cases according to the position of observable events with respect to communication events.

2. Model of distributed computations

2.1. Causal precedence

Distributed computations result from the execution of distributed programs. A distributed program is made of n sequential processes $P_1, \ldots, P_n$ which synchronize and communicate by the only mean of message passing. A distributed program can be directly written by a programmer or can be the result of the compilation of a parallel or sequential program for a distributed memory parallel machine. Processes that realize the distributed computation execute actions which are either communication actions (sending of a message, reception of a message) or internal actions (all the other actions). Each execution of an action constitutes an event.

The activity of a process P_i is perceived as a set of local atomic events E_i, totally ordered by a local precedence relation $<_i$. This set E_i can be partitioned into two subsets:

- I_i: the set of internal events of P_i (resulting from internal actions); $\perp_i$ is a fictitious initial internal event of P_i which initializes P_i's context;
- X_i: the set of communication events of P_i (send and receive events).

In the following we will note $\perp = \bigcup_i \{\perp_i\}$

The set $E = \bigcup_i E_i$ of all the events produced by the distributed execution is partially ordered by Lamport's relation called *happened before* or *causal precedence* [8], denoted by $\leqslant_E$. For all $x \in E_i$, $y \in E_j$:

$$x \leqslant_E y \stackrel{\text{def}}{\equiv} \begin{cases} x = y \\ \text{or} \\ i = j \text{ and } x <_i y \\ \text{or} \\ x \text{ is the sending of a message and } y \text{ its reception} \\ \text{or} \\ \exists z \text{ such that } x \leqslant_E z \text{ and } z \leqslant_E y. \end{cases}$$

2.2. Abstraction level and observable events

Analyzing a (distributed) computation requires to define precisely the *abstraction level* we consider (usually language, system or hardware level). In our approach, this level is defined by the set of events that must be observed in order to check the property

defined by the user. So we consider here that, for a given abstraction level, only a subset of internal events are relevant, and result from execution of specific actions (for example, modifications of some process variables). These events are called *observable*. Communication events create causal dependencies between observable events but are supposed to be not observable. This observation notion defines a screen that filters out all irrelevant events while keeping all causal dependencies between relevant events.

Let $O_i \subseteq I_i$ be the set of observable events of P_i and $O = \bigcup_{i \in 1..n} O_i$. At the considered abstraction level, the distributed computation is characterized by the poset $(O, \leqslant_o)$ with $\leqslant_o$ defined by[2]:

$$\forall x, y \in O: \quad x \leqslant_o y \Leftrightarrow x \leqslant_E y.$$

Distinct events can be executions of a same action. So each event is labeled by the name (label) of the corresponding action. These labels constitute the atoms of the language used to specify properties. Let us denote Σ the finite set of labels and λ the labeling function from O to Σ.

In the sequel, posets $(O, \leqslant_o)$, labeled with λ, are represented by sets of words of Σ^*. As there is no satisfactory method for coding a labeled poset as a set of strings when vertices with the same label are not totally ordered [11], we assume the labeling function λ is not auto-concurrent; this means that two concurrent events (i.e. not related by $\leqslant_o$) cannot have the same label[3]:

$$\forall x, y \in O, \quad \lambda(x) = \lambda(y) \Leftrightarrow x \leqslant_o y \text{ or } y \leqslant_o x.$$

Definition 1. A *labeled computation* C is thus defined as the labeled partial order (lpo for short):

$$C = \langle O, \leqslant_o, \Sigma, \lambda \rangle.$$

When necessary, we will point out the observable events by a couple of integers: the process number on which it occurs, and its rank on this process. Fig. 1(a) displays a distributed computation in the classical space-time diagram. Observable events are denoted by black dots; exchanges of messages are represented by arrows going from one process line to another one. Lpos can be usually represented by a covering graph (or Hasse diagram)[4] oriented bottom-up (canonical representation). In such a graph $x <_o y$ iff there exists a path from x to y. Fig. 1(b) displays the lpo C associated with the distributed computation depicted in Fig. 1(a).

3. Specification of properties

3.1. Language associated with observable events

With each observable event x can be associated the set of its causal predecessors; this set is denoted $\downarrow_O x$. It is formally defined as

$$\downarrow_O x = \{y \in O \mid y <_o x\}.$$

Some of the causal predecessors of x constitute the set of its *immediate* predecessors (the predecessor nodes in the covering graph); this set is denoted by

$$\downarrow_O^{\text{im}} x = \max_{\leqslant_o}(\downarrow_O x),$$

where $y \in \max_{\leqslant_o}(X)$ iff $y \in X$ and $\nexists z \in X, y <_o z$ (such an y is called a *maximal event* of X). For example, in Fig. 1(a), the second observable event on process P_4 (event (4,2)), labeled f has two immediate predecessors, one on P_2 (event (2,2), labeled by c), and one on $P3$ (event (3,1), labeled by e). Note that event (4,1) precedes the event (4,2) but not immediately.

More generally, the set of all maximal paths (in the covering graph) ending at an observable event x can be associated with it. These paths form words on Σ^* when considering the labeling function λ; let us denote by $\mathcal{L}(x)$ this set of words.

Definition 2. $\mathcal{L}(x)$ constitutes the language associated with the observable event x. Formally

[2] We will use $x <_o y$ as a shorthand for $x \leqslant_o y and x \neq y$.

[3] This is easily achieved in this model by associating with each observable event, the process number on which it occurred. Assuming a finite number of processes this restriction does not change the power of the specification language.

[4] In a covering graph only non-reflexive and non-transitive edges are drawn, i.e. there is an edge between x and y iff $x <_o y$ and / $\exists z \in O$: $x <_o z$ and $z <_o y$.

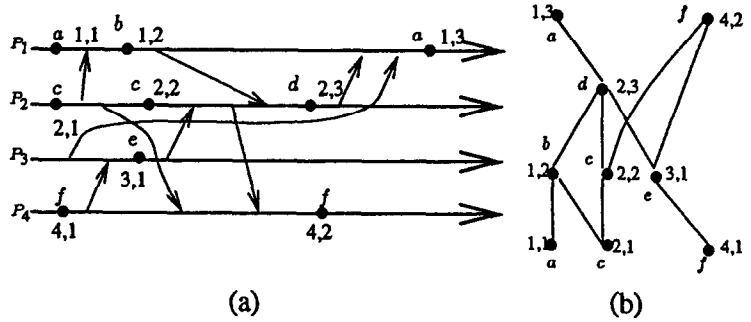

Fig. 1. (a) A distributed computation C. (b) Its associated lpo.

$$\mathcal{L}(x) = \textbf{if } \downarrow_O^{\text{im}} x \neq \{\} \textbf{ then } \bigcup_{y \in \downarrow_O^{\text{im}} x} (\mathcal{L}(y).\lambda(x)) \textbf{ else } \{\varepsilon\}$$

In Fig. 1(b), we have $\mathcal{L}((1,3)) = \{abda, cbda, ccda, feda\}$. The elements of $\mathcal{L}(x)$ correspond to the "longest control flows" that are needed to produce x. We can see that, in the previous example (Fig. 1(a)), the path aba, including only events produced by P_1 is not a "longest control flow". The set of all these maximal sequences is sufficient to include all the observable events that causally occurred before a given observable event x.

3.2. Specifying properties

Specifying properties by set of words is commonplace. We consider here regular languages, i.e. properties which can be specified by finite automata[5]. So a property is described by an automaton (possibly nondeterministic) $\Phi = (\Sigma, Q, q_0, \delta, F)$, where

- Σ is the alphabet (labels of observable events),
- Q is the set of states,
- q_0 is the initial state,
- $\delta : Q \times A \to 2^Q$ is the transition function,
- $F \subseteq Q$ is the set of final states.

$\mathcal{L}(\Phi)$ will denote the language recognized by Φ.

[5] Other kinds of specification languages could be considered. The important thing is that the concept of state can be defined (given by an operational semantics of the specification language for example). One can think to use "real" formal description techniques like SDL, State-charts, and so on.

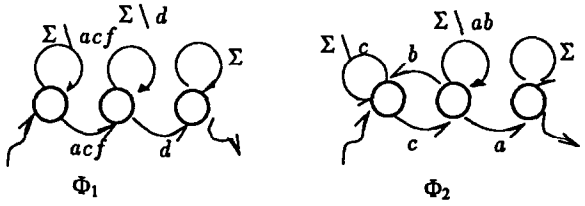

Fig. 2. Automata Φ_1 and Φ_2.

Considering an observable event x the basic question we want to answer concerns the validity of a maximal sequence terminating at x: "Does a word of $\mathcal{L}(x)$ belong to $\mathcal{L}(\Phi)$?" More formally, the satisfaction rule is defined by:

Definition 3 (*Satisfaction rule*). [6]

$$\forall x \in O, \quad x \models \Phi \Leftrightarrow \mathcal{L}(x) \cap \mathcal{L}(\Phi) \neq \emptyset.$$

Among the properties that can described by such automata we find the analogs (obtained by replacing predicates by labels) of the "linked predicates" introduced in [10] and of the "atomic sequences of local predicates" introduced in [6]. As examples let us consider $\Sigma = \{a, b, c, d, e, f\}$ and the two properties Φ_1 and Φ_2: $\mathcal{L}(\Phi_1)$ is the language composed of all words which comprise at least letter a or c or f and letter d in such a way that first occurrence of an a or c or f is followed (not necessarily immediately) by a d. $\mathcal{L}(\Phi_2)$ is the language composed of all words which include at least a and c in such a way that there is a sub-word with no b between a and c. (See the au-

[6] The question "Do all the words of $\mathcal{L}(x)$ belongs to $\mathcal{L}(\Phi)$?" is easily answered by: $x \not\models \neg\Phi$.

tomata described in Fig. 2). We have (see Fig. 1(b)) $(1,3) \models \Phi_1$, $(1,3) \models \Phi_2$ and $\neg((1,3) \models \neg\Phi_1)$.

4. An on-the-fly analysis algorithm

To answer the question "$x \models \Phi$" on the fly and without delay we have to confront a partial order, incrementally defined, to an automaton. We first show that this question can be answered by associating a set of states of the automaton with each observable event. Then we show that, for each observable event x, such an association can be done on-the-fly and without delay if we know the states of the automaton associated with the observable events preceding immediately x (the set of these events is $\downarrow_O^{\text{im}} x$). Consequently, the algorithm proceeds inductively along causal paths; its most subtle part resides in detecting whether an observable event is an immediate predecessor of another observable event.

4.1. An incremental analysis

Consider an observable event x. Let $\Phi(x)$ be the set of states reached by the automaton by analyzing all words of $\mathcal{L}(x)$:

$$\Phi(x) = \bigcup_{w \in \mathcal{L}(x)} \delta^\star(q_o, w),$$

where $\delta^\star$ is the transition function of Φ extended to words and sets of states:

$$\delta^\star(\{q_o\}, u.x) = \bigcup_{q \in \delta^\star(\{q_o\},u)} \delta(q,x),$$

$$\delta^\star(\{q_o\}, \varepsilon) = \{q_o\} \quad \text{where } \varepsilon \text{ is the empty word.}$$

We have [7]

$$x \models \Phi \quad \Leftrightarrow \quad \Phi(x) \cap F \neq \emptyset.$$

The important point is that $\mathcal{L}(x)$, and thus $\Phi(x)$, can be incrementally computed, as shown by the following proposition.

[7] For question "Do all the words of $\mathcal{L}(x)$ belongs to $\mathcal{L}(\Phi)$?" we have to consider the automaton $\neg\Phi$ resulting from negation of $\Phi(x)$ and to check: $(\neg\Phi)(x) \cap \neg F = \emptyset$ where $\neg F$ is the set of final states of the automaton $\neg\Phi$.

Proposition 4.

$$\forall x \in O \setminus \bot, \quad \Phi(x) = \bigcup_{y \in \downarrow_O^{\text{im}} x,\ q \in \Phi(y)} \delta(q, \lambda(x)),$$

$$\forall \bot \in \bot, \qquad \Phi(\bot) = \{q_0\}.$$

Proof. $\forall \bot \in \bot$, $\lambda(\bot) = \varepsilon$ so $\forall \bot \in \bot$, $\mathcal{L}(\bot) = \{\varepsilon\}$. $\forall x \in O \setminus \bot$, $\mathcal{L}(x) = \mathcal{L}(\downarrow_O^{\text{im}} x).\lambda(x)$ (where $\mathcal{L}(\downarrow_O^{\text{im}} x)$ denotes the set of words $\bigcup_{y \in \downarrow_O^{\text{im}} x} \mathcal{L}(y)$) as $\forall \bot \in \bot$, $\mathcal{L}(\bot) = \{\varepsilon\}$. Thus $\forall \bot$, $\Phi(\bot) \bigcup_{w \in \mathcal{L}(\bot)} \delta^\star(\{q_0\}, w) = \delta^\star(\text{e } sq_0, \varepsilon) = \{q_o\}$ by definition of $\delta^\star$. Second,

$$\Phi(x) = \bigcup_{y \in \downarrow_O^{\text{im}} x,\ w \in \mathcal{L}(y)} \delta^\star(\{q_o\}, w.\lambda(x)),$$

and by definition of $\delta^\star$,

$$\Phi(x) = \bigcup_{y \in \downarrow_O^{\text{im}} x,\ w \in \mathcal{L}(y),\ q \in \delta^\star(\{q_o\},w)} \delta(q, \lambda(x)).$$

This leads to the proposition. □

4.2. Principle of the algorithm

The computation of $\Phi(x)$ for each observable event x is distributed among the processes. The non-local information is acquired by piggybacking control information on messages. The automaton Φ is known by all the processes. Each process P_i is endowed with two arrays: $LO_i[1..n]$ and $SLO_i[1..n]$. The idea is that at every moment $LO_i[j]$ contains the last observable event on P_j in the causal past of the current event $t \in E_i$. $SLO_i[j]$ contains the state information $\Phi(LO_i[j])$ if $LO_i[j]$ is a maximal event in the set of the last observable events $\bigcup_{k \in 1..n} LO_i[k]$. $SLO_i[j]$ takes the value $\emptyset$ otherwise. More precisely:

Definition 5. For all $t \in E_i$ and $j \in 1..n$

$$LO_i[j](t) = \max_{\leqslant_o}(\downarrow_E t \cap O_j),$$

$$SLO_i[j](t)$$

$$= \textbf{if } LO_i[j](t) \in \max_{\leqslant_o}\Big(\bigcup_{k \in 1..n} LO_i[k](t)\Big)$$

$$\textbf{then } \Phi(LO_i[j](t))$$

$$\textbf{else } \emptyset$$

$$\textbf{fi}$$

For a process P_i, and for the next observable event x that will occur on P_i, LO_i gives the n potential candidates to be immediate predecessors of x. When x occurs, the information about states of the automaton is present in SLO_i. According to the previous definition, $\Phi(x)$ can be computed by using the following rule:

$$\forall x \in O_i, \quad \Phi(x) = \bigcup_{j \in 1..n,\ q \in SLO_i[j](x)} \delta(q, \lambda(x)).$$

4.3. Construction of the algorithm

The algorithm may be designed inductively. Supposing that $\Phi(y)$ is correctly computed for all the observable events in the past of a current event $t \in E_i$, the question is to derive for each event t occurring on P_i, the code that correctly computes LO_i and SLO_i according to their definitions.

Initially. $\forall j \in 1..n$, $LO_i[j] = \perp_j$ and $SLO_i[j] = \Phi(\perp_j) = \{q_o\}$.

Upon observation of x on P_i. x becomes the last observable event on P_i. The last observable event on P_j ($j \neq i$) known by P_i, remains unchanged. Thus $LO_i[i]$ becomes $\{x\}$. x is also the only maximal element of $\bigcup_{j \in 1..n} LO_i[j]$ since, by definition of LO_i, all the $LO_i[j]$ are in the past of x. Thus $SLO_i[i]$ becomes $\bigcup_{j \in 1..n, q \in SLO_i[j]} \delta(q, \lambda(x))$. The other components of LO_i are not maximal and then, by definition, $\forall j \neq i$, $SLO_i[j]$ becomes $\emptyset$.

Upon sending a message from P_i. LO_i and SLO_i do not change. They are piggybacked on the message in order to propagate control information.

Upon receiving a message from P_k, transporting LO_k and SLO_k. This is the most subtle part of the algorithm. We proceed componentwise by comparing the relative sequencing of the event $LO_i[j]$ (the last observable event on P_j, known by P_i before reception) with respect to the event $LO_k[j]$ (the last observable event on P_j known by P_k when it sent the message).

Note that $LO_i[j]$ and $LO_k[j]$ are necessarily ordered since they occurred on the same process P_j. Thus we have to consider three cases: $LO_i[j] <_j LO_k[j]$, $LO_k[j] <_j LO_i[j]$ and $LO_k[j] = LO_i[j]$ (same event). These cases are illustrated in Fig. 3.

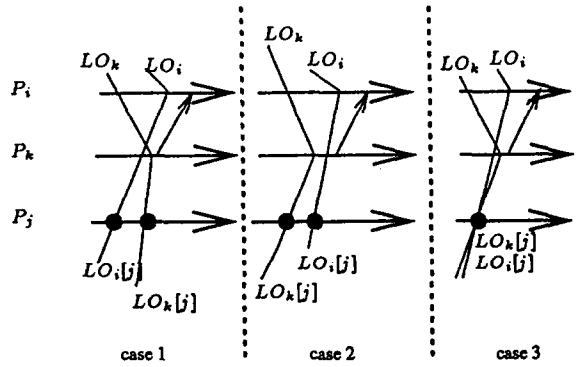

Fig. 3. Relative sequencing of $LO_i[j]$ and $LO_k[j]$.

$SLO_i[j]$	$\emptyset$	X	$\emptyset$	X
$SLO_k[j]$	$\emptyset$	X	X	$\emptyset$
new $SLO_i[j]$	$\emptyset$	X	$\emptyset$	$\emptyset$

Fig. 4. Values of $SLO_i[j]$ and $SLO_k[j]$, and expected result.

Case 1: $LO_i[j] <_j LO_k[j]$. In that case the last observable event on P_j known by P_k occurred after the one known by P_i. The last observable event on P_j known by P_i is now $LO_k[j]$, and the corresponding value of $SLO_k[j]$ has to be registered. $LO_i[j]$ is updated to $LO_k[j]$ and $SLO_i[j]$ is updated to $SLO_k[j]$.

Case 2: $LO_k[j] <_j LO_i[j]$. This is the symmetrical situation. The values known by P_i must be kept. There is nothing to do.

Case 3: $LO_k[j] = LO_i[j]$. Here, P_i and P_k refer to the same event on P_j. Let us remark that if $SLO_i[j]$ and $SLO_k[j]$ are not empty, they have the same value: this is because the function Φ has been computed for this event by process P_j and remained unchanged until it is set to empty. The situation is summarized in Fig. 4, where we consider the respective values of $SLO_i[j]$ and $SLO_k[j]$, which are $\emptyset$ or a non-empty set X. The event $LO_i[j]$ is not maximal if it is already known as non-maximal by P_i or P_k. This is why the new value of $SLO_i[j]$ is not empty if and only if its previous value on P_i and $SLO_k[j]$ are both non-empty. In that case, the value of $SLO_i[j]$ remains the same. In other words (see Fig. 4), $SLO_i[j]$ is updated by setting it

to empty only when $SLO_i[j] \neq \emptyset$ and $SLO_k[j] = \emptyset$.

In the algorithm an observable event is represented by its rank on the process that produced it. In others words each $LO_i[j]$ can be coded by an integer. Moreover, the local sequencing $<_i$ is then just the usual order on integers[8]. The algorithm is defined by the four following statements S1 to S4 executed by each process P_i.

Algorithm

Control Variables:
$LO_i[1..n]$ of integer;
$SLO_i[1..n]$ of set of states (2^Q);

S1: initialization:
$\forall j \in 1..n$: $LO_i[j] := 0$;
$\forall j \in 1..n$: $SLO_i[j] := \{q_0\}$;

S2: When P_i produces an observable event x:
$LO_i[i] := LO_i[i] + 1$;
let *previous* $= \bigcup_{k \in 1..n} SLO_i[k]$;
$SLO_i[i] := \bigcup_{q \in previous} \{\delta(q, \lambda(x))\}$;
$\forall j \in 1..n, j \neq i$: $SLO_i[j] := \emptyset$;
$\% \ \Phi(x) = SLO_i[i] \ \%$

S3: When P_i sends a message to P_k:
$LO_i[1..n]$ and $SLO_i[1..n]$
are added to the message;

S4: When P_i receives from P_k a message piggybacking LO_k and SLO_k:
$\forall j \in 1..n$: **do**
case
$LO_i[j] < LO_k[j]$ **then**
$SLO_i[j] := SLO_k[j]$;
$LO_i[j] := LO_k[j]$
$LO_i[j] > LO_k[j]$ **then** skip
$LO_i[j] = LO_k[j]$ **then**
if $SLO_i[j] \neq \emptyset$ and $SLO_k[j] = \emptyset$
then $SLO_i[j] := \emptyset$
fi
end-case

[8] The array LO_i is implemented by using the vector clocks proposed by Fidge and Mattern [2,9].

5. Particular cases

According to the position of observable events with respect to communication events, two particular cases can be defined. In these cases the analysis algorithm simplifies.

5.1. Non-invisible process participation

In this case we assume there is at least one observable event during any interval of a process beginning with a receive event and ending with a send event. This assumption, called "non-invisible process participation" has the following immediate consequence: when a process P_i sends a message we have always:

$SLO_i[k] := \emptyset, \quad \forall k \in 1..n, \ k \neq i.$

It follows that only the vector $LO_i[1..n]$ and the set $SLO_i[i]$ have to be piggybacked. Statement S3 and S4 can be simplified accordingly.

5.2. Non-invisible communications

Here it is assumed all communication events are observed in the following way: there is always an observable event just before every send event and just after every receive event. (In that case each observable event x occurring on P_i has only one or two immediate predecessors: in the case of two predecessors, one on P_i and another on the process which sent the last received message.) This is called "non-invisible communications". In that case the array of sets $SLO_i[1..n]$ can be represented by just a single set, since there is only one component which is not empty.

6. Concluding remarks

A general algorithm working on the fly and without delay has been introduced to analyze distributed computations. It associates with each observable event x of the computation the set of the longest control flows (sequences of observable events) that terminate at this event. A labeling function allows the user to consider these sequences as words on some alphabet and the algorithm checks whether these words belongs to some language (defined by a finite state automaton). It has been shown that according to the constraints on the

position of observable events with respect to communication events, the analysis algorithm can be simplified.

This work is in continuation of previous works on on-the-fly distributed detection of predicates. It generalizes some of them and provides a good description power for the specification of properties while keeping an efficient and simple distributed detection algorithm. For example, the algorithms introduced in [4,10] to detect linked predicates and the algorithm which detects regular patterns in distributed computations introduced in [3] are two particular cases which consider all control flows ending at observable events (and not only the maximal ones). Moreover, it appears that the algorithm described in [1] that computes the immediate predecessors of an observable event x constitutes the core of the verification algorithm that has been introduced.

Acknowledgements

We are grateful to S. Lorcy and N. Plouzeau for interesting discussions related to this 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. Moreover, referees are gratefully acknowledged for their comments which help improve the presentation of the paper.

References

[1] C. Diehl, C. Jard and J.X. Rampon, Reachability analysis on distributed executions, in: *Theory and Practice of Software Development*, Lecture Notes in Computer Science **668** (Springer, Berlin, 1993).

[2] J. Fidge, Timestamps in message passing systems that preserve the partial ordering, in: *Proc. 11th Australian Computer Science Conf.* (1988) 55–66.

[3] E. Fromentin, M. Raynal, V.K. Garg and A.I. Tomlinson, On the fly testing of regular patterns in distributed computations, in: *Proc. of the 23rd Internat. Conf. on Parallel Processing*, St. Charles, IL (1994) 73–76.

[4] V.K. Garg and B. Waldecker, Detection of unstable predicates in distributed programs, in: Lecture Notes in Computer Science **625** (Springer, Berlin, 1992) 253–264.

[5] M. Hurfin, N. Plouzeau and M. Raynal, A debugging tool for distributed Estelle programs, *J. Comput. Comm.* **16** (5) (1993) 328–333.

[6] M. Hurfin, N. Plouzeau and M. Raynal, Detecting atomic sequences of predicates in distributed computations, in: *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, San Diego, CA (1993) 32–42.

[7] C. Jard, T. Jeron, G.V. Jourdan and J.X. Rampon, A general approach to trace-checking in distributed computing systems, in: *Proc. 14th IEEE Internat. Conf. on DCS*, Poznań, Poland (1994) 396–403.

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

[9] F. Mattern, Virtual time and global states of distributed systems, in: M. Cosnard, Y. Robert, P. Quinton and M. Raynal, eds., *Parallel and Distributed Algorithms* (North-Holland, Amsterdam, 1988) 215–226.

[10] B.P. Miller and J. Choi, Breakpoints and halting in distributed programs, in: *Proc. 8th IEEE Internat. Conf. on Distributed Computing Systems*, San Jose (1988) 316–323.

[11] V. Pratt, Modelling concurrency with partial orders, *Internat. J. Parallel Programming* **15** (1) 1986.