On-the-fly Analysis of Distributed Computations

Eddy Fromentin, Claude Jard,
Guy-Vincent Jourdan, Michel Raynal

IRISA, Campus de Beaulieu,
F-35042 Rennes, FRANCE.
email: <name>@irisa.fr

May 13, 1994

Abstract

At some abstraction level a distributed computation can be modelled 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 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 definition of observable events or to the set of control flows we want to analyze.

Key Words: Distributed computation, observable event, longest control flows, causal precedence, sequences analysis.

1 Introduction

Since Lamport's seminal paper [8], distributed computations are modelled as sets of events structured by partial order relations. For a particular computation, events produced by each process are totally ordered and communications creates dependencies among events belonging to distinct processes. These partial order relations on events are generally called *happened before* (with respect to a logical time frame) or *causal precedence*. They formally express control flows and their mutual dependencies which organize distributed executions.

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). 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 relevant events produced by an execution of a distributed program. The practical motivation of our work comes from debugging, testing and monitoring of distributed computations [5, 7]. In this context we choose to explore analysis techniques of distributed computations which must be done on-the-fly anf 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 (as in that case notification messages would add some delay between event occurences and their knowledge by the monitor). 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 informations 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 partial order relation associated with the computation. Although each observable event of the computation is unique, several events can be execution occurences of the same action for example. So a labelling function is introduced and the sequences of observable events are associated with words (concatenation of labels). The analysis is then carried out on the fly and without delay on these words. The analysis considered in this paper is based on finite state automata. Other kinds of automata could be used but finite state automata are sufficient to illustrate our analysis technique; moreover finite state automata can solve interesting practical problems (see Section 5) and allow an efficient analysis, as far as the automaton is concerned, as they require only the piggybacking of a bounded number of bits –one per state of the automaton– to do the analysis.

The paper is divided into 4 main sections. Section 2 presents the model of distributed computation; Section 3 presents the kind of analysis we are interested in, the definition of languages (set of words) associated with distributed computations and the two question (satisfaction rules) 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 Distributed Computations

2.1 Distributed programs

We are interested here by distributed computations. Such computations result from the execution of distributed programs. A distributed program is made of n sequential processes $P_1, .., P_n$ which synchronize and communicate by the only means of message passing. A distributed program can be directly produced 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). Execution of an action is called an event.

2.2 Model of distributed computations

2.2.1 Lamport's precedence relation

When executed, each sequential process P_i produces a set of 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);
- X_i: the set of communication events of P_i (send and receive events).

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]. The resulting poset is noted $\widehat{E} = (E, \leq_E)$:

$$\forall x \in E_i, y \in E_j : x \leq_E y \stackrel{def}{\equiv} \begin{cases} & x = y \\ or & \\ & i = j \text{ and } x <_i y \\ or & \\ & x \text{ is the sending of a message and } y \text{ its reception} \\ or & \\ & \exists z \text{ such that } x \leq_E z \text{ and } z \leq_E y \end{cases}$$

2.2.2 Abstraction level and observable events

Analysis of a (distributed) computation is always done at some *abstraction level* (usually language, system or hardware level). For an abstraction level, a distributed computation is characterized by the events that must be observed in order to analyze it. So we consider here that, for a given abstraction level, only a subset of internal events are relevant; these events are called *observable* events and result from execution of specific actions (for example, modifications of some processes variables); these actions will be identified by labels as described in Section 3.2. Communication events create causal dependencies

between observable events but are not supposed to be observable (if necessary a communication event can be made observable by generating an additional internal event just before it –in case of a send– or just after it –in case of a receive–; see Section 5.2).

So an abstraction level is a screen that filters out all irrelevant events and keeps all causal dependencies between relevant events. Let $O_i \subseteq I_i$ be the set of observable events of P_i and $O = \bigcup_i O_i$. At the abstraction level considered the distributed computation is characterized by the poset $\widehat{O} = (O, \leq_O)$ with $\leq_O$ defined by[1]:

$$\forall x, y \in O : x \leq_O y \Leftrightarrow x \leq_E y$$

Figure 1 displays a distributed computation in the classical space-time diagram. Observable events are denoted by black points; exchanges of messages are represented by arrows going from one process line to another one.

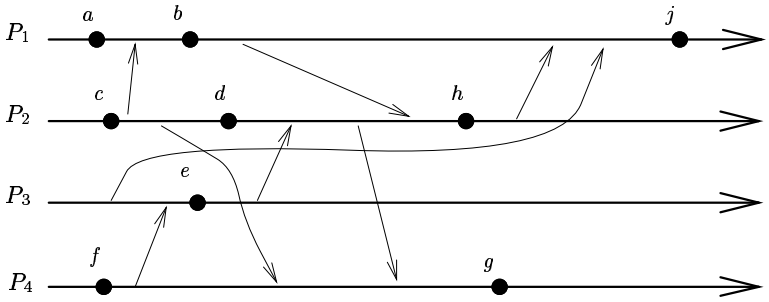


Figure 1: A distributed computation.

Figure 2 displays the poset $\widehat{O} = (O, \leq_O)$ associated with the distributed computation of Figure 1 (only non reflexive and non-transitive edges are represented).

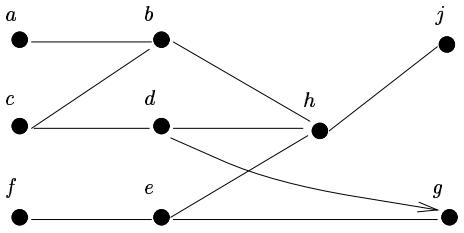


Figure 2: A poset $\widehat{O} = (O, \leq_O)$

[1] We use $x <_O y$ as a shorthand for $x \leq_O y$ *and* $x \neq y$.

2.3 Deciding about order of events

By using a vector clock mechanism [2, 9] a timestamp can be associated with each event and used to decide about order of events. Such a vector timestamp $V_x[1..n]$ associated with event x is such that:

$$\forall k \in 1..n : V_x[k] = |\{z \in O_k : z \leq_o x\}|$$

and we have

$$\forall x \in O_i, y \in O : x \leq_o \Leftrightarrow V_x[i] \leq V_y[i]$$

These timestamps are obtained by using the classical vector clock mechanism [2, 9]:

- each process P_i maintains an integer vector $V_i[1..n]$ initialised to 0;
- each time P_i produces an observable event $x \in O_i$, it executes: $V_i[i] := V_i[i] + 1; V_x := V_i$;
- each time P_i sends a message, it adds V_i to the message;
- each time P_i receives a message from P_j, piggybacking V_j it executes: $\forall x \in 1..n : V_i[k] := max(V_i[k], V_j[k])$.

3 On the Fly Analysis of a Distributed Computation

3.1 Covering graph of causal dependencies

A way to perform analyses of distributed computations on the fly and without delay is to carry them out incrementally each time an observable event is produced. So the analysis done when such an event x is produced is on the set of its causal predecessors, denoted $pred\, x$:

$$pred\, x = \{y \in O : y \leq_o x\}$$

Among all the causal predecessors of x some are its *immediate* predecessors. If it exists the immediate predecessor of x on some process P_j is unique. It constitutes the singleton or empty set $im\_pred_j\ x$. Formally:

$$(im\_pred_j\ x = \{y\}) \stackrel{def}{\equiv} \left\{ \begin{array}{ll} & y <_o x \\ and & \\ & y \in O_j \\ and & \\ & \nexists z \in O : (y <_o z\ and\ z <_o x) \end{array} \right.$$

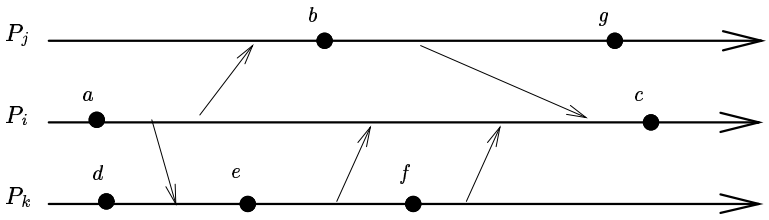


Figure 3: b is an immediate predecessor of c.

For example, in Figure 3, the observable event c has an immediate predecessor on P_j ($im\_pred_j\ c = \{b\}$) and there is no immediate predecessor of x on process P_i ($im\_pred_i\ c = \{\}$).

Let $\widehat{C}$ be the covering graph of $\widehat{O}$ (i.e. $\widehat{C}$ is $\widehat{O}$ from which all transitivity edges have been suppressed); we have:

$$(y, x) \in \widehat{C} \Leftrightarrow \exists j \in 1..n : im\_pred_j\ x = \{y\}$$

Considering an observable event x, the analysis is carried out on the paths of $\widehat{C}$ beginning with an observable event without predecessor and ending at x; let $C(x)$ be the set of all these paths. In Figure 3 we have $C(c) = \{abc, aefc, defc\}$. Paths of $C(x)$ include all events of $pred\ x$ and correspond to the "longest control flows" that are needed to produce x. We can see that, in the previous example, the path ac is not a "longest control flow".

It would be possible to consider a set of paths ending at x larger than $C(x)$ to perform the analysis. We choose the previous set $C(x)$ essentially for the generality of the associated analysis algorithm described in Section 4. The resulting algorithm is very general and can be simplified to work on a larger set of control flows ending at some observable event x. Such an algorithm is sketched at Section 5.3. When considering the observable event c in Figure 3, this simplified algorithm works on the set of paths: $\{abc, ac, aec, aefc, dec, defc\}$ which represents all possible control flows ending at c and not only the "longest" ones.

3.2 Language associated with observable events

Observable events are execution of relevant actions at the abstraction level considered. These actions define an alphabet A and a labelling function $\lambda : O \longrightarrow A$ associates with each observable event an element of the alphabet A.

A language $\mathcal{L}(x)$ (set of words of $A^\star$) is associated with each observable event x in the following way:

$$\mathcal{L}(x) = \{\lambda(\sigma) : \sigma \in C(x)\}$$

where $\lambda(x_1 x_2 \ldots x_k) = \lambda(x_1)\lambda(x_2)\ldots\lambda(x_k)$.

3.3 Specifying a pattern

The analysis consists in answering wether the computation meets some pattern. The patterns we are here interested in are described with a finite state automaton $\Phi = (A, Q, q_0, \delta, F)$ where:

- A is the alphabet (labels of observable events);
- Q is the set of states;
- q_0 is the initial state;
- $\delta \subseteq Q \times A \times Q$ is the transition function;
- $F \subseteq Q$ is the set of final states.

Other kinds of automata could be used to specify patterns. But as claimed in the introduction, finite state automata are sufficient to describe a lot of practical properties (whose linked predicates [4] and atomic sequences of predicates [6] are special cases).

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

3.4 Satisfaction rules

Consider an observable event x. Two kinds of question can be answered according to kind of analysis we are interested in: either only one or all paths of $C(x)$ match the pattern described by the automaton Φ. More formally the two following satisfaction rules are defined for $x \in O$:

$$\begin{aligned} x \models_{\exists} \Phi &\Leftrightarrow \mathcal{L}(x) \cap \mathcal{L}(\Phi) \neq \{\} \\ x \models_{\forall} \Phi &\Leftrightarrow \mathcal{L}(x) \subseteq \mathcal{L}(\Phi) \end{aligned}$$

The first satisfaction rules is useful in the context of debugging (with Φ describing a pattern revealing am error). The second one is more interesting in the field of on-line testing to verify an execution did not exhibit an anomalous behavior.

4 An On the Fly Analysis Algorithm

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 $\mathcal{L}(x)$:

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

where $\delta^\star$ is the transition function of Φ extended to words. We have:

$$\begin{aligned} x \models_\exists \Phi &\Leftrightarrow \Phi(x) \cap F \neq \{\} \\ x \models_\forall \Phi &\Leftrightarrow \Phi(x) \subseteq F \end{aligned}$$

As the analysis is on $C(x)$, the set of "longest control flows" ending at x, $\Phi(x)$ can be computed in an incremental way by using the immediate predecessors of x:

$$\Phi(x) = \left\{ \delta(q, \lambda(x)), \forall q \in \bigcup_j \Phi(im\_pred_j\ x) \right\}$$

with $\Phi(\{\}) = \{\}$ and $\Phi(im\_pred_j\ x) = \{q_0\}$ if x is a minimum of O.

4.2 Description of the algorithm

Each process P_i is endowed with a vector clock $V_i[1..n]$ managed as described in Section 2.3. It is also associated with an array $\Phi pred_i[1..n]$ of sets of elements of Q.

$\Phi pred_i[j]$ is managed in such a way that supposing the next event x produced by P_i is an observable event, we have: $\Phi pred_i[j] = \Phi(im\_pred_j\ x)$ (the proof in Section 4.3 will establish this invariant).

The algorithm is defined by the 4 following statements S1 to S4 executed by each process P_i.

S1: initialization:

$\forall j \in 1..n : V_i[j] := 0;$
$\forall j \in 1..n, \Phi pred_i[j] := \{q_0\}\,;$

S2: When P_i produces an observable event x:

$V_i[i] := V_i[i] + 1;$
$\Phi pred_i[i] := \{\delta(q, \lambda(x)), \forall q \in \bigcup_k \Phi pred_i[k]\}\,;$
$\forall j \in 1..n, j \neq i : \Phi pred_i[j] := \{\}\,;$
% $\Phi(x) = \Phi pred_i[i]$ %

S3: When P_i sends a message to P_k:

$V_i[1..n]$ and $\Phi pred_i[1..n]$ are added to the message;

S4: When P_i receives from P_k a message piggybacking V_k and $\Phi pred_k$:

$\forall j \in 1..n$: **do**
case
$V_i[j] < V_k[j]$ **then** $\Phi pred_i[j] := \Phi pred_k[j]$;
$V_i[j] := V_k[j]$
$V_i[j] > V_k[j]$ **then** skip
$V_i[j] = V_k[j]$ **then if** $\Phi pred_i[j] \neq \{\}$
then $\Phi pred_i[j] := \Phi pred_k[j]$
fi
end-case

S2 acts as a reset: when it produces an observable event x, P_i computes $\Phi(x)$ according to the states in which the automaton was when the immediate predecessors of x were produced. Moreover if the next event produced by P_i is observable it will have x as the only immediate predecessor.

S4 updates the array $\Phi pred_i$ in order, $\forall j$, the invariant relying the concrete variable $\Phi pred_i[j]$ and the abstract variables $\Phi(im\_pred_j\ x)$ be maintained. Vector clocks play an essential role in this management by permitting to know which events are immediate predecessors of each observable event of the distributed computation. Section 4.3 proves the correctness of this updating strategy.

4.3 Proof

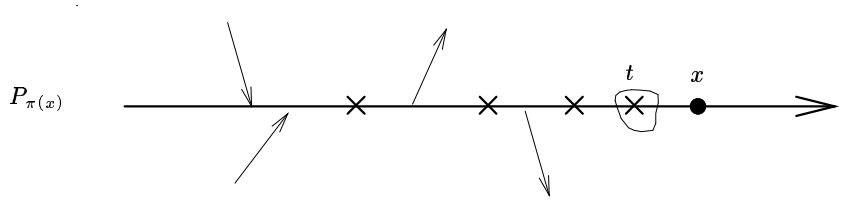
Notation:

- For the sake of simplicity we will note $\pi(x) = i$ if $x \in E_i$, $\forall i \in 1..n$.
- In the following $\Phi pred[k](t)$ denotes the value of $\Phi pred_{\pi(t)}[k]$ just after the event t has been produced.
- We assume that each process P_i begins with an initial fictitious internal event $\perp_i \notin O$ such that $\forall j, \Phi pred[j](\perp_i) = \{q_0\}$ ($\perp_i$ represents the initialization of P_i).

As explained in Section 4.2, we have to show the following proposition in order to prove the correctness of the algorithm (illustraded in Figure 4).

Proposition
Let $x \in O$ and $t \in E$ such that $t <_{\pi(x)} x$ and $\nexists y \in E : t <_{\pi(x)} y \wedge y <_{\pi(x)} x$ then we have:

$$\forall j \in 1..n : \Phi pred[j](t) = \Phi(im\_pred_j\ x)$$



The value of $\Phi pred_{\pi(x)}[k], \forall k$ just before x is equal to
the value produced just after t, the last event on $P_{\pi(x)}$ just before x.

Figure 4: The proposition.

$\forall t \in E$, let $last\_modifier(t)$ denote the event such that:
$(last\_modifier(t) <_{\pi(t)} t)$
and
$(last\_modifier(t) \in O$ *or* $last\_modifier(t)$ *is a receive event*$)$
and
$\nexists y$ *such that* $(y \in O$ *or* y *is a receive event*$)$ *and* $(last\_modifier(t) <_{\pi(t)} y$ *and* $y <_{\pi(t)} t)$
or such that
$last\_modifier(t) = \perp_{\pi(t)}$ if there is neither receive nor an observable event on $P_{\pi(t)}$ before t.

Remark that if t is an internal event or a send event, the algorithm does not change the array $\Phi pred_{\pi(t)}[1..n]$. So, in order to proof the proposition, it is sufficient to prove that (see Figure 5):

$$\forall x \in O, \forall j \in 1..n : \Phi pred[j](last\_modifier(x)) = \Phi(im\_pred_j\ x)$$

.

The proof is on the rank of x. More formally, let $\mathcal{M}^n$ and $\mathcal{O}^n$ be subsets of O such that:

$$\mathcal{M}^0 = \{x \in O : \nexists y \in O : y <_O x\} \qquad \mathcal{O}^0 = O$$
$$\mathcal{M}^n = \{x \in \mathcal{O}^{n-1} : \nexists y \in \mathcal{O}^{n-1} : y <_O x\} \qquad \mathcal{O}^n = \mathcal{O}^{n-1} \setminus \mathcal{M}^{n-1}$$

With these notations we have to prove the following:

$$\forall r, \forall x \in \mathcal{M}^r, \forall j \in 1..n : \Phi pred[j](last\_modifier(x)) = \Phi(im\_pred_j\ x)$$

Moreover we suppose there is no infinite chains CC of O, such that $\exists x \in \mathcal{M}^n, y \in \mathcal{M}^{n+1} : x = min(CC), y = max(CC)$. This states that, for the distributed computations we consider, there is no an infinity of events between two events (this hypothesis will be called *finiteness hypothesis* in the following).

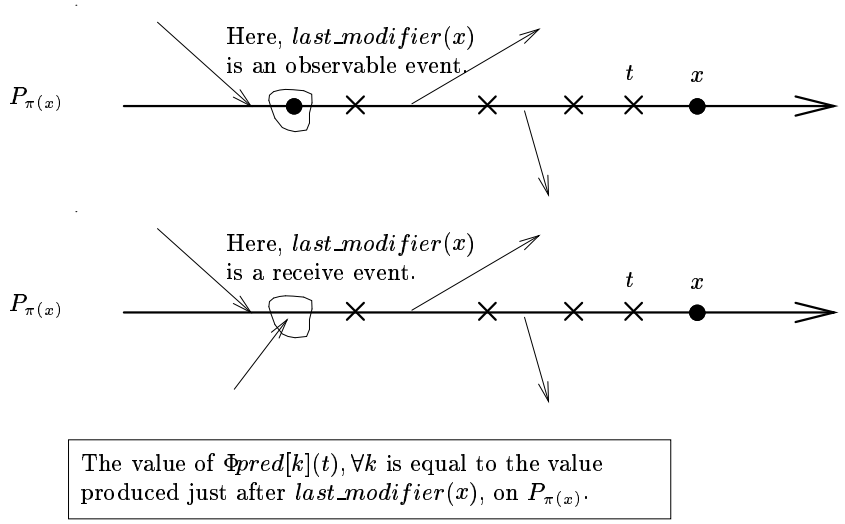


Figure 5: Meaning of $last\_modifier(x)$.

Base case

Let $x \in \mathcal{M}^0$. x is a minimum of O, so $\forall j \in 1..n : \Phi(im\_pred_j\ x) = \{q_0\}$. As there is no observable event before x, $last\_modifier(x)$ is $\perp_{\pi(x)}$ or a receive event. If $last\_modifier(x) = \perp_{\pi(x)}$ then the property follows from S1. If $last\_modifier(x)$ is a receive event, consider $s \in E$ its associated send event. If $\exists k : \Phi pred[k](last\_modifier(x)) \neq \{q_0\}$ then, due to statement S4, we have $\Phi pred[k](s) \neq \{q_0\}$ and so $\Phi pred[k](last\_modifier(s)) \neq \{q_0\}$. Applying inductively this reasonning, there is a $\perp_m$ such that $\Phi pred[k](\perp_m) \neq \{q_0\}$ which is impossible. So the proposition is true for $x \in \mathcal{M}^0$.

Induction case

Let r an integer, assume that the property is true for all rank $r' < r$, we have to show:

$$\forall x \in \mathcal{M}^r, \forall j \in 1..n : \Phi pred[j](last\_modifier(x)) = \Phi(im\_pred_j\ x)$$

Lemma $\forall r' < r$, $\forall y \in \mathcal{M}^{r'}$, *we have* $\Phi pred[\pi(y)](y) = \Phi(y)$ *and*
$\forall j \neq \pi(y) : \Phi pred[j](y) = \{\}$.
The Proof of this lemma follows from induction hypothesis and from statement S2.

Now, there are two cases to consider: $last\_modifier(x)$ is an observable event or a receive event.

1. $last\_modifier(x)$ is an observable event.
   It follows from definition of $last\_modifier(x)$ that $\forall j \neq \pi(x) : im\_pred_j\ x = \{\}$ and $im\_pred_{\pi(x)}\ x = \{last\_modifier(x)\}$. Then the proposition follows directly from the previous lemma.

2. $last\_modifier(x)$ is a receive event.
   Let $y = max(O_{\pi(y)} \cap pred\ x)$, in order to proof the proposition, we have to show the following property:

$$\begin{array}{lcl} \Phi pred[\pi(y)](last\_modifier(x)) = \{\} & \Leftrightarrow & im\_pred_{\pi(y)}\ x = \{\} \\ \Phi pred[\pi(y)](last\_modifier(x)) = \Phi(y) & \Leftrightarrow & im\_pred_{\pi(y)}\ x = \{y\} \end{array}$$

   which is equivalent to (by definition of $last\_modifier(x)$):

$$\begin{array}{lcl} \Phi pred[\pi(y)](last\_modifier(x)) = \{\} & \Leftrightarrow & \exists z \in O : y <_E z \wedge z <_E last\_modifier(x) \\ \Phi pred[\pi(y)](last\_modifier(x)) = \Phi(y) & \Leftrightarrow & \nexists z \in O : y <_E z \wedge z <_E last\_modifier(x) \end{array}$$

   Let $s \in E$ be the send event associated with $last\_modifier(x)$. Let t be the event $last\_modifier(last\_modifier(x))$ (see Figure 6).

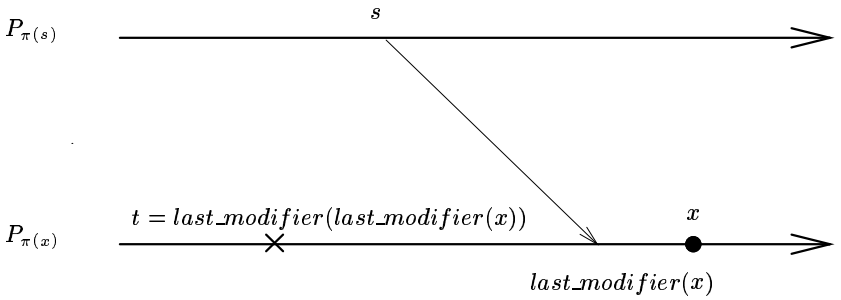


Figure 6: Two cases to consider.

   The proposition is proved by considering the two cases:

   (a) $\nexists z \in O : y <_E z \wedge z <_E last\_modifier(x) \Rightarrow$
   $\Phi pred[\pi(y)](last\_modifier(x)) = \Phi(y)$;

(b) $\exists z \in O : y <_E z \wedge z <_E last\_modifier(x) \Rightarrow$
$\Phi pred[\pi(y)](last\_modifier(x)) = \{\}$.

(case 2a)

$\nexists z \in O : y <_E z \wedge z <_E last\_modifier(x) \Rightarrow \Phi pred[\pi(y)](last\_modifier(x)) = \Phi(y)$
By contradiction, assume $\nexists z \in 0 : y <_E z \wedge z <_E last\_modifier(x)$ (H1)
and $\Phi pred[\pi(y)](last\_modifier(x)) \neq \Phi(y)$ (H2).

Considering statement S4 of the algorithm, there are three cases to consider:

i. $V_{\pi(x)}[\pi(y)] < V_{\pi(s)}[\pi(y)]$
then $\Phi pred[\pi(y)](s) \neq \Phi(y)$,
so we have $\Phi pred[\pi(y)](last\_modifier(s)) \neq \Phi(y)$.
If $last\_modifier(s) \in O$ then by H1 it follows that $y = last\_modifier(s)$
and then $\Phi pred[\pi(y)](y) \neq \Phi(y)$ which contradicts the lemma.
If $last\_modifier(s)$ is a receive event then we have
$\nexists z \in O : y <_E z \wedge z <_E last\_modifier(s)$
and $\Phi pred[\pi(y)](last\_modifier(s)) \neq \Phi(y)$.
To proof there is a contradiction, it remains to show that
$\nexists z \in O : y <_E z \wedge z <_E last\_modifier(s) \Rightarrow$
$\Phi pred[\pi(y)](last\_modifier(s)) \neq \Phi(y)$.

ii. $V_{\pi(x)}[\pi(y)] > V_{\pi(s)}[\pi(y)]$
then $\Phi pred[\pi(y)](t) \neq \Phi(y)$,
by applying the same reasonning as in case 2()i, we show that $t \notin O$,
so we have $\nexists z \in O : y <_E z \wedge z <_E t$ and $\Phi pred[\pi(y)](t) \neq \Phi(y)$.
It remains to show that
$\nexists z \in O : y <_E z \wedge z <_E t \Rightarrow \Phi pred[\pi(y)](t) \neq \Phi(y)$.

iii. $V_{\pi(x)}[\pi(y)] = V_{\pi(s)}[\pi(y)]$
then we have to consider two cases:

A. $\Phi pred[\pi(y)](t) = \{\}$,
as $t \notin O$ (by applying the same reasonning as in cases 2()i and 2()ii),
It remain to show that $\nexists z \in O : y <_E z \wedge z <_E t \Rightarrow$
$\Phi pred[\pi(y)](t) \neq \Phi(y)$.

B. $\Phi pred[\pi(y)](t) \neq \{\}$,
then we have $\Phi pred[\pi(y)](s) \neq \Phi(y)$ and the same reasonning as in case 2()i applies.

So considering the finiteness hypothesis, applying recursively the reasonning will fall in the contradiction case : $\Phi pred[\pi(y)](y) \neq \Phi(y)$ which proves 2a.

(case 2b)

$\exists z \in O : y <_E z \wedge z <_E last\_modifier(x) \Rightarrow \Phi pred[\pi(y)](last\_modifier(x)) = \{\}$
By contradiction, assume $\exists z \in 0 : y <_E z \wedge z <_E last\_modifier(x)$ (H3) and
$\Phi pred[\pi(y)](last\_modifier(x)) \neq \{\}$ (H4).

Considering statement S4 of the algorithm, there are three cases to consider:

i. $V_{\pi(x)}[\pi(y)] < V_{\pi(s)}[\pi(y)]$
then $\Phi pred[\pi(y)](s) \neq \{\}$,
so we have $\Phi pred[\pi(y)](last\_modifier(s)) \neq \{\}$.
If $last\_modifier(s) \in O$ there are two cases: $last\_modifier(s) = y$ or $last\_modifier(s) \neq y$.
If $last\_modifier(s) \neq y$ then by statement S2 and by H4 it follow that $\pi(s) = \pi(y)$,
and by H3, $y <_{\pi(y)} last\_modifier(s)$ wich is impossible as
$y = max(O_{\pi(y)} \cap pred\, x)$.
If $last\_modifier(s) = y$ then it contradicts H3.
If $last\_modifier(s)$ is a receive event, we have
$\exists z \in O : y <_E z \wedge z <_E last\_modifier(s)$,
as $\nexists z \in O : y <_E z \wedge z <_E s$ implies (by H3)
$\exists z \in O : y <_E z \wedge z <_E last\_modifier(x)$
which implies $V_{\pi(x)}[\pi(y)] > V_{\pi(s)}[\pi(y)]$ contradicting the hypothesis on timestamps of events.
To prove there is a contradiction, it remains to show that
$\exists z \in O : y <_E z \wedge z <_E last\_modifier(s) \Rightarrow$
$\Phi pred[\pi(y)](last\_modifier(s)) = \{\}$.

ii. $V_{\pi(x)}[\pi(y)] > V_{\pi(s)}[\pi(y)]$
then $\Phi pred[\pi(y)](t) \neq \{\}$.
If $t \in O$ there are two cases: $t = y$ or $t \neq y$.
If $t \neq y$ then by statement S2 and by H4 it follow that $\pi(y) = \pi(t)$,
then $y <_{\pi(y)} t$ wich is impossible as $y = max(O_{\pi(y)} \cap pred\, x)$.
If $t = y$ then it contradicts H3.
If t is a receive event then we have $\exists z \in O : y <_E z \wedge z <_E t$.
It remains to show that $\exists z \in O : y <_E z \wedge z <_E t \Rightarrow$
$\Phi pred[\pi(y)](t) = \{\}$.

iii. $V_{\pi(x)}[\pi(y)] = V_{\pi(s)}[\pi(y)]$
then $\Phi pred[\pi(y)](t) \neq \{\}$ and $\Phi pred[\pi(y)](s) \neq \{\}$
the same reasonning as in case 2()i applies.

So considering the finiteness hypothesis, applying recursively the reasonning will fall in the contradiction cases : $y \neq max(O_{\pi(y)} \cap pred\, x)$ or $\nexists z \in O : y <_E z \wedge z <_E last\_modifier(x)$ which implies 2b. □

5 Particular Cases

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

5.1 Non invisible process participation

In this case we assume there is always 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 participation, is described in Figure 7.

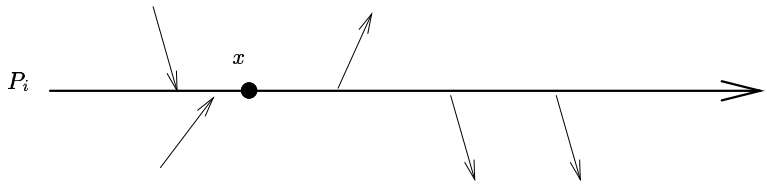


Figure 7: Non invisible process participation.

This assumption has the following immediate consequence: when a process P_i sends a message we have always:

$$\Phi pred_i[k] := \{\}, \forall k \in 1..n, k \neq i$$

It follows that only the vector $V_i[1..n]$ and the set $\Phi pred_i[i]$ have to be piggybacked by the message sent. Statement S4 can be simplified accordingly and becomes:

S4: When P_i receives from P_k a message piggybacking V_k and $\Phi pred_k[k]$:

$\forall j \neq k$: **if** $V_i[j] \leq V_k[j]$ **then** $\Phi pred_i[j] := \{\}$ **fi**;
if $V_i[k] < V_k[k]$ **then** $\Phi pred_i[k] := \Phi pred_k[k]$ **fi**;
$\forall j \in 1..n : V_i[j] := max(V_i[j], V_k[j])$;

5.2 Non invisible communication

Here we assume all communication events are observed in the following way: there is always an observable event just before every send event or just after every receive event (by "just before" or "just after" we mean there is neither a send nor a receive event between). In that case each observable event has exactly one or two immediate predecessors: always one on the same process and in the case of two, another on the process wich sent the last received message. In Figure 8, x has two immediate predecessors z and y; x' as only x as immediate predecessor ((t, x') is not an edge on a "longest control flow").

It follows from this simplifying assumption that the array $\Phi pred_i[1..n]$ is no more necessary, a simple $\Phi pred_i$ being sufficient to compute $\Phi(x)$ for each observable event x. The algorithm becomes for each process P_i:

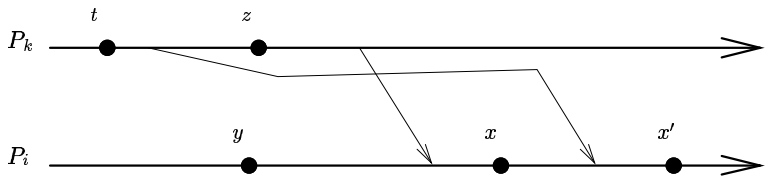


Figure 8: Non invisible communication.

S1: initialization:

$\forall j \in 1..n : V_i[j] := 0;$
$\Phi pred_i := \{q_0\}\,;$

S2: When P_i produces an observable event x:

$V_i[i] := V_i[i] + 1;$
$\Phi pred_i := \{\delta(q, \lambda(x)), \forall q \in \Phi pred_i\}\,;$
% $\Phi(x) = \Phi pred_i$ %

S3: When P_i sends a message to P_k:

$V_i[1..n]$ and $\Phi pred_i$ are added to the message;

S4: When P_i receives from P_k a message piggybacking V_k and $\Phi pred_k$:

case
$V_i[k] \geq V_k[k]$ **then** skip
$V_i[i] = V_k[i]$ **then** $\Phi pred_i := \Phi pred_k$
else $\Phi pred_i := \Phi pred_i \cup \Phi pred_k$
end-case
$\forall j \in 1..n : V_i[j] := max(V_i[j], V_k[j]);$

5.3 Considering all paths

If we are interested not in the "longest control flows" as defined by $C(x)$ for each observable event x, but in control flows ending at x (cf. discussion in Section 3.1) vector clocks become useless (remember they are used to consider only immediate predecessors when a messsage is received). The analysis algorithm simplifies accordingly.

S1: initialization:

$\Phi pred_i := \{q_0\}$;

S2: When P_i produces an observable event x:

$\Phi pred_i := \{\delta(q, \lambda(x)), \forall q \in \Phi pred_i\}$;
% $\Phi(x) = \Phi pred_i$ %

S3: When P_i sends a message to P_k:

add $\Phi pred_i$ to the message;

S4: When P_i receives from P_k a message piggybacking $\Phi pred_k$:

$\Phi pred_i := \Phi pred_i \cup \Phi pred_k$

6 Related works

The algorithm introduced in [4] to detect linked predicates and the algorithm which detects regular patterns in distributed computation introduced in [3] are two particular uses of the simplified algorithm described in Section 5.3 which considers all control flows ending at observable events.

The algorithm described in [1] that computes the immediate predecessors of an observable event x is a special case of the general analysis algorithm where all observable events have the same label l and the automaton has only one state q_0 whith $\delta(q_0, l) = q_0$. With such an automaton, when an observable event x is produced by P_i we have:

$$(\exists y : im\_pred_j \ x = \{y\}) \Leftrightarrow \Phi pred_i[j] \neq \{\}$$

7 Conclusion

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 terminates at this event. A labelling function allows the user to consider these sequences as words on some alphabet and the algorithm checks wether these words belongs to some language (defined by a finite state automaton). It has been shown that according to the contraints on the position of observable events with respect to communication events, the analysis algorithm can be simplified.

References

[1] 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.

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

[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* 23^{rd} *International Conference on Parallel Processing*, St. Charles, IL, August 1994.

[4] 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.

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

[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*, pages 32–42, San Diego, CA, May 1993. (Reprinted in SIGPLAN Notices, Dec. 1993).

[7] C. Jard, T. Jeron, G.V. Jourdan, and Rampon J.X. A general approach to trace-checking in distributed computing systems. In *Proc. IEEE Int. Conf. on DCS*, Poznan, Poland, June 1994.

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

[9] 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.