INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A General Approach to Trace-Checking in Distributed Computing Systems

Claude Jard, Thierry Jéron, Guy-Vincent Jourdan et Jean-Xavier Rampon

N°????

Mars 94

THÈME ??

Rapport de recherche

ISSN 0249-6399

A General Approach to Trace-Checking in Distributed Computing Systems

Claude Jard, Thierry Jéron, Guy-Vincent Jourdan et Jean-Xavier Rampon

Thème ?? — (hors thème INRIA)
Projet Pampa

Rapport de recherche n° ???? — Mars 94 — 18 pages

Abstract: The problem of checking the correctness of distributed computations arises when debugging distributed algorithms, and more generally when testing protocols or distributed applications. For that purpose, one describes the expected behavior (or suspected errors) by a global property: for example, a predicate on process variables, or the set of admissible orderings on observable events. The problem is to check whether this property is satisfied or not during the execution. A relevant model for this study is the partial order of message causality and the associated state graph, called "lattice of consistent cuts". In this paper, we propose a general approach to trace checking, based on partial order theory.

Key-words: Distributed computation, debugging parallel programs, predicate checking, timestamps, causality order, lattice of consistent cuts

(Résumé : tsvp)

A paraıitre dans "14$^{\text{th}}$ International Conference on Distributed Computing Systems", Poznan, Pologne, Juin 1994

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Une approche générale pour la vérification de traces dans les systèmes répartis

Résumé : Le problème de la vérification de la correction de calculs répartis survient lorsque l'on souhaite déboguer des algorithmes distribués, et plus généralement lorsque l'on teste des protocoles ou des applications réparties. Dans ce but on décrit le comportement attendu (ou les erreurs supposées) par une propriété globale: par exemple un prédicat portant sur les variables de processus, ou l'ensemble des ordres admissibles entre événements observables. Le problème est de vérifier si cette propriété est satisfaite ou non au cours de l'exécution. Un modèle adéquat pour cette étude est celui de l'ordre partiel de causalité et le graphe des états associés appelé "treillis des coupes cohérentes". Dans cet article, nous proposons une approche générale à la vérification de trace fondée sur la théorie des ordres partiels.

Mots-clé : Calcul réparti, débogage de programmes répartis, vérification de prédicats, estampillage, ordre de causalité, treillis des coupes cohérentes

1 Introduction

1.1 Problem statement

This paper presents techniques for reliably detecting logical errors in a distributed program, where "logical error" is defined on the intended partial ordering of events. We are only concerned with relative events orderings; no consideration is made of the issues associated with real-time or "performance" debugging.

Faced with an implementation (i.e. a black-box parallel program) and a global property specifying the intended behavior of the program, a natural approach to testing is to insert an observer process which receives reports from the program under test and checks that the property holds. In practice, as it is impossible to trace all the elementary events of the computation (variable assignments, message exchanges, ...), we must select the type of events we want to observe.

When observing a system, the activity of gathering and using run-time information (that we call trace-checking) can be split up into three reasonably independent pipe-lined tasks: the generation of run-time information, achieved by software probes inserted in the source code (possibly assisted by instrumented libraries or hardware components of the machines), the sending of the information to the observer, which makes the occurrence of events observable (note that download may have a certain latency depending on the internal buffers capacity) and finally, the analysis of the information, which consists in interpreting the data in order to provide a diagnostic.

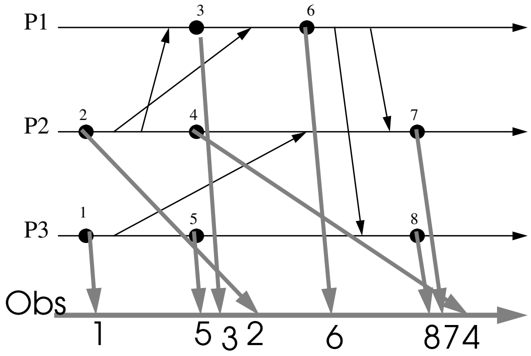

Figure 1: Observer problems

Unfortunately, if no assumption can be made on communication delays or politics of channels (fifo or not), the observer may receive events in an arbitrary order (see figure 1). And without added information, it is impossible for the observer to detect concurrency or causal dependence between observable events. The stand taken here (after [Fid88]) is that the ordering of events observation should not affect the results produced by a parallel testing system. Any pronouncement on the correctness of a particular test must be valid for any possible interleaving of events compatible with the causal ordering induced by the observed computation.

1.2 Work position

In the last ten years, there were several propositions to help in distributed debugging, even in the special case we called trace-checking or predicate detection. The pioneering work in that area was presented in [CL85], and gives an algorithm to detect stable predicates, which uses global snapshots. Since then, the scientific community tries to extend this work to larger classes of properties, said "unstable" [GW92, CM91, BR94]. Different algorithms were proposed to deal with local predicates, conjunction and disjunction of local predicates, sequencing of local predicates... All these algorithms suffer from a lack of homogeneous theoretical structure. Consequently, the different proofs of the algorithms do not even help since they appear ad-hoc. Faced to this situation, we were tempted to study in depth the structure of the lattice of global states (or consistent cuts, or ideals). We think this lattice is a good candidate to found the reflection on predicate detection. The paper defends such a position. We show that understanding the dynamic structure of the lattice directly leads to general algorithms which detect larger classes of predicates than in the previous literature.

The remainder of the paper is organized as follows. We start by exposing the problem of trace observation. We lay stress on the notion of observable event, and on the vector coding of subsets of events. We then introduce a classification of properties based on the ability to decide them on-line. Finally, we show how to check properties on traces. These properties are expressed by finite automata. Though the techniques are well-known in the area of model-checking, we present an algorithm which strictly extends the class of properties that were presented in previous works in distributed debugging.

2 Trace observation

This section is devoted to the problem of trace observation in a distributed environment. We start by defining some terminology which stems from the theory of partial orderings and lattices. Then we recall the notions of "causal relation" and "consistent cuts" for observable events. The remainder of the section is devoted to the on–line reconstruction of the causal relation by an observer.

2.1 Terminology

A partially ordered set (*poset*) $\widetilde{P} = (P, \leq_{\widetilde{P}})$ is a set P of elements, together with an antisymmetric, reflexive and transitive binary relation $\leq_{\widetilde{P}}$. In addition, if $x \leq_{\widetilde{P}} y$ and $x \neq y$, then we write $x <_{\widetilde{P}} y$. If $x \leq_{\widetilde{P}} y$ or $y \leq_{\widetilde{P}} x$ we say that x and y are *comparable* in $\widetilde{P}$. Otherwise, x and y are said *incomparable* in $\widetilde{P}$. If $x \leq_{\widetilde{P}} y$, then x is a *predecessor* of y in P and y is a *successor* of x in P. We say that y is an *immediate successor* of x (or x is an *immediate predecessor* of y) or y *covers* x and note $x \prec_{\widetilde{P}} y$ if $x < y$ and $\forall z \in P, x <_{\widetilde{P}} z \leq_{\widetilde{P}} y \Rightarrow z = y$. The directed graph associated to this covering relation is denoted $Cov(P) = (P, E_P)$.

Let A be a subset of P , the *suborder* of $\widetilde{P}$ on A is $\widetilde{P/_A} = (A, \leq_{\widetilde{P}})$. An *antichain* (resp. a *chain*) is a subset of P in which every pair of elements is incomparable (resp. comparable). A *chain decomposition* in k chains of $\widetilde{P}$ is a partition $\{P_i\}_{i \in [1 \ldots k]}$ of P in which every P_i is a chain of $\widetilde{P}$. $Max(A, \widetilde{P}) = \{a \in A, \forall x \in A, \neg(a <_{\widetilde{P}} x)\}$ is the set of maximal elements of A in $\widetilde{P}$. Similarly, $Min(A, \widetilde{P}) = \{a \in A, \forall x \in A, \neg(x <_{\widetilde{P}} a)\}$ is the set of minimal elements. A has an *infimum* (resp. a *supremum*) in $\widetilde{P}$ if $|Max(\{x \in P,\ \forall y \in A, x \leq_{\widetilde{P}} y\}, \widetilde{P})| = 1$ (resp. $|Min(\{x \in P,\ \forall y \in A, y \leq_{\widetilde{P}} x\}, \widetilde{P})| = 1$).$\widetilde{P}$ is a *lattice* iff every pair of elements has a supremum and an infimum in $\widetilde{P}$.

Let $x \in P$, $\downarrow_P x = \{y \in P, y \leq_{\widetilde{P}} x\}$ (resp. $\uparrow_P x = \{y \in P, x \leq_{\widetilde{P}} y\}$) is the *predecessor set* (resp. the *successor set*) of x in $\widetilde{P}$. $\downarrow_P^{im} x = \{y \in P, y \prec_{\widetilde{P}} x\}$ (resp. $\uparrow_P^{im} x = \{y \in P, x \prec_{\widetilde{P}} y\}$) is the *immediate predecessor set* (resp. the *immediate successor set*) of x in $\widetilde{P}$.

A *linear extension* of a poset $\widetilde{P}$ is a maximal chain C onto P which preserves $\leq_{\widetilde{P}}$, i.e.: $x \leq_{\widetilde{P}} y \Rightarrow x \leq_{\widetilde{C}} y$.

An *ideal* A of a poset $\widetilde{P}$ is a subset closed by precedence: $\forall x \in A, \forall y \in P,\ y \leq_{\widetilde{P}} x \Rightarrow y \in A$. When $Max(A, \widetilde{P})$ is reduced to a singleton $\{x\}$, we say that A is a *sup-irreducible* associated to x. Observe that the set of maximal elements in an ideal forms an antichain and that this provides a one-to-one corresponding between the ideals and the antichains of a poset. The set of ideals $I(P)$ of a poset $\widetilde{P}$, ordered by set inclusion, forms a distributive lattice $\widetilde{I(P)}$ (see [Bir67]).

2.2 Message causality

Formally reasoning on execution traces requires a simple mathematical model. Two different levels can be distinguished:

- The user's level, where the programmer defines what he wants to trace (variable changes, calls to sending procedures, ...). These are actions occurring during the computation. The action occurrences are called *observable events*. The action name defines the event type (or *label*).
- The run-time level which induces some constraints on the relative ordering of observable events. Precisely, that is the causality due to the physical exchange of messages between processes, ensuring that the sending of a message must always precede its reception.

An example of a trace is given in figure 2(a). The observable events are figured as dots on the "time-line" of processes. Arrows represent the message exchanges.

Let us consider a finite set of processes, denoted by P_1 to P_n (one considers for simplification this set is known in advance and stable). We denote $E = X \uplus \bar{X} \uplus O$ the finite set of events occurring during the execution ($\uplus$ denotes the disjoint union). X contains all the sending events (unobservable in principle), $\bar{X}$ contains the corresponding receptions (for the sake of simplicity, communication channels are considered point-to-point) and O is the set of observable events. One can consider that E is split up into the disjoint subsets E_i of events that are local to P_i: $E = \biguplus_{1 \leq i \leq n} E_i$.

Let $\prec_i$ be the sequential ordering of events appearing on process P_i. The causal relationship (whose definition is given in [Lam78]) in E^2 is the smallest relation $\prec$ satisfying :

1. $\forall i \in \{1..n\}, \forall x, y \in E_i,\ \ x \prec_i y \Longrightarrow x \prec y$
2. $\forall x \in X,\ \ x \prec \bar{x}$
3. $\prec$ is transitive

$\prec$ is a partial order. In the sequel, we will only consider the suborder $\widetilde{O}$ induced by the set O of observable events partially ordered by the relation $\prec$. This abstraction capability is interesting as it allows us to not consider communication events. This leads to our trace model: *a distributed trace is modeled by a finite poset $\widetilde{O}$, split up into n disjoint chains and equipped with a labeling function λ from O to an action alphabet Σ.* We say that a trace satisfies the NIVI condition (for "non invisible interaction") if on each process, between a receive event and a following send event there exists at least one observable event i.e. $\forall i \in [1..n], \forall \bar{y_i} \in \bar{X}_i, \forall x_i \in X_i, \bar{y_i} \prec_i x_i \Longrightarrow \exists o_i \in O_i$ s.t. $\bar{y_i} \prec_i o_i \prec_i x_i$.

In practice, the observer has to reconstruct the poset $\widetilde{O}$ from the observed events. More precisely, it has to reconstruct the covering digraph of $\widetilde{O}$, which Hasse diagram is shown in figure 2(b). So we need to associate some kind of coding to events in order to make it possible. The observer receives the event e together with the process number in which e occurs and the coding. Moreover, the construction has to be done on–line. But when an event e is received by the observer, there may be some events which belong to the past of e in the computation and which have not been observed yet. For example, in figure 2 assume that all events but events 5 and 8 have already been observed, and suppose that the observer now receives 8. The event 5, which is an immediate predecessor of 8, is missing. There are two possible techniques in that case. The first one is to compute the suborder of the real causal order on the already observed events. In our example, that means that 1 becomes an immediate predecessor of 8 in the suborder. When 5 arrives, the edge between 1 and 8 has to be removed as it becomes a transitive edge. The second possibility is to wait for a linear extension, that is when an event e is observed and there is some missing events in the past of e, the observer must wait all missing events before taking e into account. The observer still works on–line as we can expect that the lowest events in the order will not be systematically received after the highest ones. Sections 2.3 and 2.4 present two ways of coding the causal relation and algorithms to extract the covering digraph of the poset from its coding.

Once we have constructed a certain part of the poset $\widetilde{O}$, say the subposet $\widetilde{O'}$, we may construct its ideal lattice. If $\widetilde{O'} = \widetilde{O}$, then this lattice is the "lattice of consistent cuts" (see

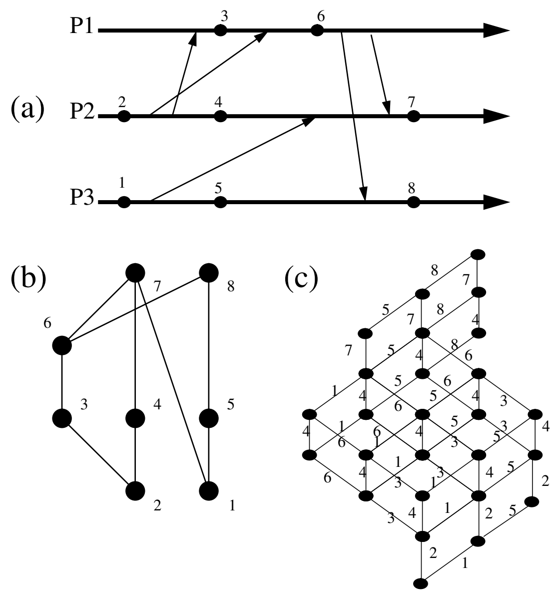

Figure 2: A trace (a), the Hasse diagram of its poset (b) and its ideal lattice (c)

figure 2(c)). We do not describe algorithms to compute it[1]. Efficient and on–line ones may be found in [DJR93]. We just recall the meaning of the ideal lattice. It is a poset whose elements are ideals and the ordering relation is set inclusion. When an ideal I is covered by an ideal J in this lattice, we have $J - I = \{x\}$ for an event $x \in O$. Thus the edge between I and J in the covering digraph of the lattice may be labeled with x. There is a one-to-one mapping between paths from the bottom to the top of the digraph and the linear extensions of the poset $\widetilde{O}$ [BP82]. So this lattice codes all possible interleavings of events and provides a simple way to deal with properties which use concurrency. Since the size of the lattice is much bigger than the size of the order, such a construction has to be well motivated. This point is discussed in section 3.1.

2.3 Vector timestamps

The first idea for efficiently coding the causal relation is to consider the chain decomposition induced by processes. Let us consider the following coding into vectors of integers:

$$\begin{array}{rcccl} \Delta & : & O & \longrightarrow & \mathbb{N}^n \\ & & x & \longmapsto & |{\downarrow_O} x \cap O_i|_{i\in[1\ldots n]} \end{array}$$

[1] When observing events according to a linear extension of the poset, the principle of building the lattice is, after observing an event e, to find the ideal $({\downarrow_O} e) - \{e\}$ in the lattice, and then to duplicate the entire sublattice above this ideal.

i.e. to each observable event x is associated a vector $\Delta(x)$ whose i^{th} component is the number of observable events of P_i which causally precede x. One recognizes the vector clocks of Fidge and Mattern [Fid88, Mat89] (see figure 3).

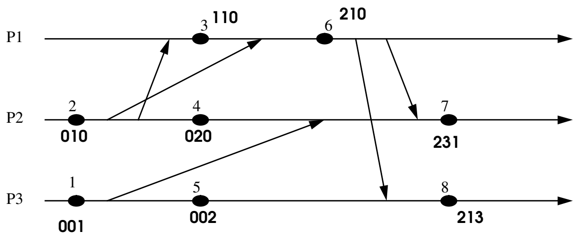

Figure 3: Vector timestamps

This coding is obtained on–line as follows:

- each process P_i maintains a vector Δ_i of size n (initially set to the null vector).
- when an observable event e occurs on P_i, $\Delta_i[i] := \Delta_i[i] + 1$; $\Delta(e) := \Delta_i$.
- when P_i sends a message, the current value V of the vector Δ_i is "piggybacked".
- upon reception on P_i of a message from P_j carrying a vector V, $\forall k \in [1 \ldots n],\ \Delta_i[k] := max(\Delta_i[k], V[k])$.

On–line construction of the covering digraph of the order

At the instant of observation of an event e, we have already constructed the covering digraph of $\widetilde{O_{bef(e)}}$ (the suborder of $\widetilde{O}$ on the events received before e), and we want to deduce the covering digraph of $\widetilde{O_{aft(e)}}$ (the suborder of $\widetilde{O}$ which takes e into account). During the algorithm, the chain decomposition of the suborder induced by processes is maintained. The computation of the covering digraph of $\widetilde{O_{aft(e)}}$ works in two phases:

- Compute $\downarrow^{im}_{\widetilde{O_{aft(e)}}} e$ (resp. $\uparrow^{im}_{\widetilde{O_{aft(e)}}} e$) as follows: on each chain, compute the biggest (resp. lowest) element smaller (resp. bigger) than e and put it into a set L (resp. U). Remove from this set L (resp. U) the elements which are smaller (resp. bigger) than another element in the set. This set is now $\downarrow^{im}_{\widetilde{O_{aft(e)}}} e$ (resp. $\uparrow^{im}_{\widetilde{O_{aft(e)}}} e$).
- Remove the edges between $\uparrow^{im}_{\widetilde{O_{aft(e)}}} e$ and $\downarrow^{im}_{\widetilde{O_{aft(e)}}} e$ (they necessarily exist in $\widetilde{O_{bef(e)}}$) as they become transitive edges in $\widetilde{O_{aft(e)}}$.

This algorithm runs in $\mathcal{O}(|O| + n^2)$ for each event using the fact that the comparison between two events necessitates only the comparison on one coordinate
$\forall x \in O_i, \forall y \in O, x \leq_{\widetilde{O}} y \Leftrightarrow \Delta(x)[i] \leq \Delta(y)[i]$ (comparison of one coordinate).

2.4 Observing the direct dependences

The major drawback of vector clocks is the intrusion induced by the piggybacking of information of size n. However, when the trace satisfies the NIVI condition[2], we can use another timestamping technique called "direct dependences coding" [FZ90, BDM$^+$94] which piggybacks a scalar value. The idea of the direct dependence coding is to maintain on each site P_i a vector of size n which j^{th} coordinate represents the number of events which occurred on site P_j before the last direct communication from P_j. In this communication it suffices to piggyback this number. The coding is defined as follows:

$$
\begin{array}{lcl}
 & & D : O \longrightarrow I\!\!N^n \\
 & & \quad\; x \longmapsto D(x) \quad \text{where for } x \in O_i \\
D(x)[i] & = & |\{ \; y \in O_i, y \preceq x\} | \\
D(x)[j \neq i] & = & |\{ \; y \in O_j \text{ s.t. there is a message sent by } P_j \text{ after y} \\
 & & \quad \text{and received by } P_i \text{ before } x\} |
\end{array}
$$

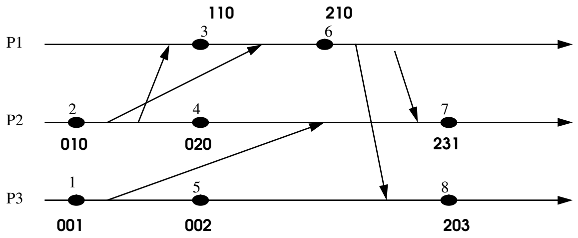

Figure 4: Direct dependences coding

The coding is computed on–line as follows:

- each process P_i maintains a vector $D_i \in I\!\!N^n$ (initially set to the null vector).
- when an observable event e occurs on P_i, $D_i[i] := D_i[i] + 1$; $D(e) := D_i$.
- when P_i sends a message, the scalar value $D_i[i]$ is "piggybacked".
- upon reception on P_i of a message from P_j carrying a value v, $D_i[j] := max(D_i[j], v)$.

This is illustrated in figure 4.

On–line construction of the covering digraph of the order

Compared to the vector timestamps coding, the direct dependence coding decreases the intrusion due to piggybacking. The price to be paid is that we are only able to construct

[2] This is easily obtained at run time by generating fictitious observable events which can be filtered later by the observer.

suborders on ideals of $\widetilde{O}$. So when the observer receives an event e, a new suborder can be computed only if all predecessors of e have already been observed. Then we can awake the construction of suborders using events which have only e as missing event in their past. We use lists *waitingfor[v, i]*, which contain observed events which at least wait for the v^{th} event z on the process P_i to get all their past in the set of observed events. Those events may wait for z either because z is missing or because z is also waiting for other events. To each event y we associate a counter *missing*, which counts the number of index i such that $\exists v, y \in waitingfor[v, i]$. Thus *missing* is null if all events in the past of y have already been observed. For each event x, the algorithm constructs an array L_x of observed events such that for $i \neq site(x)$, $L_x[i]$ is the $D_x[i]^{th}$ event in P_i and $L_x[site(x)]$ is the $D_x[site(x)] - 1^{th}$ event in $P_{site(x)}$. The sentence "compute the list of immediate predecessors of x" in procedure Wake-Up can be performed in two steps. Firstly, we associate to each event x a list $\mathcal{L}_x$ of events such that $\mathcal{L}_x[i] = max\{y \in O_i, y \prec x\}$ (this list corresponds to the set L in the previous section). This is done in $\mathcal{O}(n^2)$ since $\mathcal{L}_x[i] = max\{y \in \bigcup_{y \in L_x} \mathcal{L}_y[i] \cup L_x[i]\}$ Note that this also gives us the transitive closure of the poset since this construction can be used to compute the vector timestamps $\Delta(x)$. Secondly, we deduce $\downarrow_O^{im} x$ from $\mathcal{L}_x$ in $\mathcal{O}(n^2)$ since $\downarrow_O^{im} x = \{y \in \mathcal{L}_x[i], \forall z \in \mathcal{L}_x, \neg(\Delta(y)[i] \leq \Delta(z)[i])\}$. After this, we can proceed with the construction of the lattice or the verification of properties (see section 4). Using the chain decomposition induced by processes of $\widetilde{O}$, the algorithm runs in $\mathcal{O}(|O| + n^2)$ for each event.

```
type
  stamp: array [1..n] of integer;
  event: record
    label:action; D: stamp; site: 1..n; missing: integer;
    L: array [1..n] of event;
    end ;

var
  waitingfor: array [|O|,1..n] of set of event;
procedure NewEvent(a:action; D: stamp; s: integer);
var x: event;
begin
  x.label←a; x.D←D; x.D[s]←x.D[s]-1; x.site← s; x.missing:=0;
  for all i ∈ [1..n] do x.L[i]:= ∅;
  for all i ∈ [1..n] do begin
    if for all y ∈ O_i, y.D[i] ≠ x.D[i] then
      NotOk(x,x.D[i],i);
    else let y ∈ O_i s.t. y.D[i] = x.D[i]
      x.L[i]:=y;
      if y.missing ≠ 0 then NotOk(x,x.D[i],i);
      end ;
   if x.missing=0 then Wake-up(x);
end ;

procedure NotOk(x:event; v,i: integer);
begin
  waitingfor[v,i]:=waitingfor[v,i] ∪{x};
  x.missing:=x.missing+1;
end ;

procedure Wake-up(x: event);
begin
  compute the list of immediate predecessors of x;
  for all z ∈ waitingfor[x.D[x.site],x.site] do
    z.L[x.site]:=x;
    z.missing:=z.missing - 1;
    if z.missing=0 then Wake-up(z);
end ;
end
```

2.5 Garbage collection

The size of the ideal lattice is a practical problem. In order to save some memory during its construction and verification, we informally give the clue to remove the garbage part of the lattice. For the sake of simplicity, we assume that the subposet $\widetilde{O'}$ is constructed according to a linear extension of the complete poset. Let $O_1, \ldots O_n$ be a chain decomposition of $\widetilde{O'}$. And let V be a vector of size n such that $V(i) = y_i$ where $y_i \in O_i$ and $\forall z \in O_i, z \leq_{\widetilde{O'}} y_i$. We assume that each component of V is defined (i.e. we have already observed one event on each site). Let $G = \{z \in O', \forall i \in [1 \ldots n],\ \neg(z \geq_{\widetilde{O'}} y_i)\}$. G is an ideal of $\widetilde{O'}$. Then, on the ideal lattice of $\widetilde{O'}$, an ideal I can be removed iff $I \subseteq G$. This can be easily done using techniques detailed in [DJR93].

3 The complexity of property checking

3.1 What kind of checking?

Independently to the problem of which kind of property we want to check, we have to consider what kind of checking we want. Two main kinds of checking can be considered. In one hand, we may want to enlight one (generally the first) consistent cut verifying a property. For example, the property is an error detection defined on conjunction of local predicates. This kind of checking can be efficiently performed on the poset since it can be done without backtracking. On the other hand, we may be interested in enumerating all consistent cuts verifying the property. There may be several reasons to do that: counting, computing some average value, or listing every "minimal" snapshot verifying the property (i.e. an antichain in the ideal lattice). In this case, checking on the poset will induce backtracking and may lead to a complexity equivalent (or worse) to the construction and the verification on the ideal lattice. This leads to the definition of a general method to check such properties directly on the lattice, instead of ad-hoc methods. Section 4 is a first step in this direction.

3.2 A sketch of classification of properties

Usually, one distinguishes between local properties, which can be decided on a single process, and global ones, which depend on several processes. There is also a distinction between stable properties and unstable ones. We would like to introduce another kind of classification. As we already said, the observer may receive observable events in any order. We say that a consistent cut is *appearing* to the observer when all its maximal elements (for $\prec$) have been observed. This means that in the subposet defined by already observed events, we already have the antichain which defines the ideal of the complete poset corresponding to the consistent cut. We are interested here in determining when the observer can decide a property on appeared consistent cuts.

We will say that a property is *wait-free* (WF for short) if for any ordering in which the observer receives events, the property may be decided on a consistent cut as soon as it appears. For example, the property $a||b$, which means the concurrency of two events labeled with a and b, is WF. Since there is no assumption on the ordering of received events, we can state that a property is WF iff it can be decided by only considering the maximal events of a consistent cut. For example, the property $a \prec b$, which is true for an ideal if it contains a maximal element labeled b and in the past of this event, an event labeled with a, is not WF, since at the instant of observation of b the decision cannot be taken.

However, among non WF properties, some have nice behaviors which sometimes allow decision before reception by the observer of the complete consistent cut, that is they do not necessarily need the knowledge of the whole past. There are properties which, when reaching a certain value (true or false) on an uncomplete ideal, definitely conserves this value on this ideal. For example $a \prec b$ is definitively true in a consistent cut containing a maximal event labeled with b as soon as an event a belonging to the past of b is observed. There is also properties which can be decided when a certain part of the cut is known. For example, $a \prec\!\!\cdot\, b$ ("a is covered by b") can be decided as soon as all the immediate predecessors of b in $\widetilde{O}$ are known.

Note that the membership of properties to the aforementioned classes can be modified when assumptions can be made on the possible orderings in which events are received by the observer.

There is a practical consequence to the above sketch of classification. If we choose a direct dependence coding, in the construction of the lattice we must wait for linear extensions, thus our classification is unaccurate. On the other hand, with the vector timestamp coding, we can always decide a WF property in an ideal as soon as it appears to the observer. This allows an early decision at the cost of a more expensive intrusion.

4 Checking of regular properties

In this section we are interested in the problem of verifying properties of traces during the construction of the ideal lattice. Model-checking is a well known verification technique which consists in deciding whether a logical temporal formula is satisfied by a program. Model-checking is based on the standard model of transition systems which are essentially labeled digraphs used to represent the set of possible behaviors of the program. Numerous works have been developed these ten last years to justify the interest of temporal logics (linear or branching time) in this context, and to design general model-checking algorithms based on graph traversals. We will show that the ideal lattice can be viewed as a labeled transition system. Therefore, no doubt that the aforementioned techniques can be applied to the particular graph of ideals. Though this question has not yet been studied to our knowledge, we expect a great simplification due to the algebraic structure of the graph. One interest here is to take into account the requirement of on–line computation [JJ89].

Surprisingly, the approach in distributed debugging was different. Only very restrictive classes of properties with respect to the general framework provided by temporal logics (see [Pnu86] for example) have been studied. Our feeling is that the ideas about detection algorithms have preceded the properties formalization (one specifies only what is known to be detectable). There is also the wish to limit the exploration of the set of possible behaviors to keep a reasonable algorithmic complexity.

Our ambition is to show that existing techniques can be adapted to this context. As a starting point, we want to describe some general algorithms, able to compute a large class of properties. These properties are specified by finite automata. Consequently one often uses the term automaton verification instead of model-checking. However, as every formula of linear time temporal logic (LTL) interpreted over finite sequences can be automatically translated into a finite automaton which accepts the same models, we already reach the power of LTL.

4.1 Ideal lattice and transition system

A *(deterministic) labeled transition system* is a triple $T = (Q, A, \longrightarrow)$ where Q is the set of states, A the alphabet of actions, and $\longrightarrow \subseteq Q \times A \times Q$ is the deterministic transition relation. The covering digraph of the ideal lattice $\widetilde{I(O)}$ of the poset O can be viewed as a labeled transition system:

$$Tr(I(O)) = < I(O), \Sigma, \longrightarrow >$$

where the set of states is the set $I(O)$, Σ is the alphabet of actions (which occur as observable events), and $\longrightarrow$ is defined by $I \xrightarrow{a} J$ iff $I \prec_{\widetilde{I(O)}} J$ and $\lambda(J - I) = a$.

4.2 Regular properties

Let us now define the class of regular properties defined by finite automata on finite words. In each maximal path of the lattice, the successive encountered labels form a word on Σ. The set of such words is $\lambda(lin(O))$[3] where $lin(O)$ is the set of linear extensions of O.

A finite (deterministic) automaton is a 5-uple $\phi = < \Sigma, Q, q_0, F, \delta >$, where

- Σ is the action alphabet,
- Q is the set of states,
- q_0 is the initial state,
- $F \subseteq Q$ is the set of accepting states,
- $\delta \subseteq Q \times \Sigma \times Q$ is the transition function.

These automata allow to specify properties about the potential or necessary sequencing of actions.

Let $\mathcal{L}(\phi) = \{u \in \Sigma^* \ : \delta^*(u, q_0) \in F\}$ be the language defined by ϕ. Confronting such an automaton with the covering digraph of the lattice requires to consider, for each ideal I, the set $Path(I)$ of all paths from bottom to I. This leads to two possibilities when defining the satisfaction of a property ϕ in an ideal I:

$$\begin{array}{lll} I \models_\forall \phi & \text{iff} & \lambda(Path(I)) \subseteq \mathcal{L}(\phi) \\ I \models_\exists \phi & \text{iff} & \lambda(Path(I)) \cap \mathcal{L}(\phi) \neq \emptyset \end{array}$$

[3]For simplicity, we denote by $\lambda(e_1 \dots e_n)$ the word $\lambda(e_1).\dots.\lambda(e_1)$.

The satisfaction relation $\models_\forall$ (resp. $\models_\exists$) states that an ideal I satisfies ϕ iff *every* (resp. *some*) path in $Path(I)$ is labeled with a sequence accepted by the automaton ϕ.

Let $\neg\phi$ denote the complementary automaton of ϕ. We have $\mathcal{L}(\neg\phi) = \Sigma^* - \mathcal{L}(\phi)$. Since for every $A, B \subseteq \Sigma^*$, we have $A \subseteq B$ iff $A \cap (\Sigma^* - B) = \emptyset$, and using the definitions of the satisfaction relations, we get the usual duality relation: $I \models_\forall \phi \Leftrightarrow I \not\models_\exists \neg\phi$.

The upper element of $\widetilde{I(O)}$ is the ideal O which contains all events of the trace. The satisfaction of ϕ by the lattice $I(O)$ is then defined by:

$$\begin{array}{lcl} I(O) \models_\forall \phi & \text{iff} & O \models_\forall \phi \\ I(O) \models_\exists \phi & \text{iff} & O \models_\exists \phi \end{array}$$

From the above definitions, we can easily deduce that:

$$\begin{array}{lcl} \forall I \in I(O)\ , I \models_\forall \phi & \Leftrightarrow & \phi(I) \subseteq F \\ \forall I \in I(O)\ , I \models_\exists \phi & \Leftrightarrow & \phi(I) \cap F \neq \emptyset \end{array}$$

where $\phi(I) = \delta^*(\lambda(lin(O_{|I})), q_0)$ is the set of states of the automaton ϕ reachable by paths ending in I. The checking algorithms will be automatically derived from these expressions of the satisfaction relations.

4.3 The checking algorithms

As was evoked in the preceding paragraph, the checking algorithms are essentially based on the computation of $\phi(I)$. It is easy to see that $\phi(I)$ can be computed on–line during the lattice construction since it is sufficient to attribute the ideals by states of the automaton ϕ. In fact, the attribute $\phi(I)$ of an ideal I can be computed from the attributes of its immediate predecessors in the lattice[4]:

$$\phi(I) = \bigcup_{J \lessdot^{im}_{I(O)} I} \left(\bigcup_{q \in \phi(J)} \delta(\lambda(J - I), q) \right)$$

The algorithm can be implemented by a breadth-first traversal of the lattice during which each ideal is attributed with a boolean array of size $|Q|$, the i^{th} element indicating whether some sequence leading to I can put the automaton in state i. If we want to extend the algorithm in order to count the number of sequences accepted by the automaton, it suffices to replace the boolean array by an array of integers.

It is straightforward to see that the complexity of the checking algorithms is linear in the size of the lattice and the size of the automaton.

Figure 5 illustrates the result of algorithms for checking $\models_\forall$ and $\models_\exists$. The subset of the lattice denoted E represents the set of ideals I such that $I \models_\exists \phi$ and the subset A consists of ideals satisfying $I \models_\forall \phi$.

[4] This supposes that events are treated according to a linear extension of the poset.

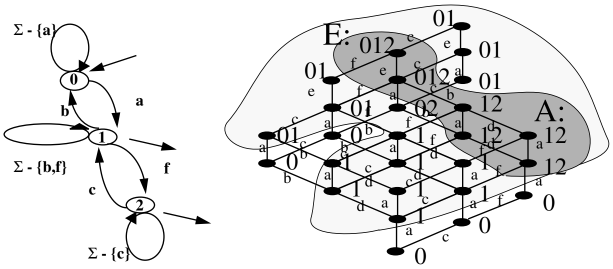

Figure 5: Automaton verification $\models_\forall$ and $\models_\exists$

5 Conclusion and future works

In this paper we have used the poset theory background in order to provide a general framework for on-line checking properties on execution traces of distributed computations. For two different codings of the causal poset we have shown how to on-line construct its covering digraph. We introduce a classification of properties which shows that the drawback of an early construction of the lattice of consistent cuts is the weakening of the class of properties that can be verified on-line. An other consequence is the enlightment of the duality between the price of piggybacking and the generality of on-line checkable properties. We have also shown that properties expressed by finite automata can be checked linearly in the size of the ideal lattice. This was a simple example of the power of dealing with this lattice. All the algorithms presented in this paper have been implemented. Our goal is to provide a tool for the on-line testing of distributed computations. A natural continuation of our work is to search for generic algorithms which could be used to verify some larger class of properties (for example defined on lattice abstractions) and to generalize ad-hoc algorithms on already known temporal logics (like branching time for example).

References

[BDM+94] P. Baldy, H. Dicky, R. Medina, M. Morvan, and JF. Vilarem. Efficient reconstruction of the causal relationship in distributed computations. In *Canada-France Conference on Parallel Computing, Montréal, Canada*, May 1994.

[Bir67] G. Birkhoff. Lattice theory. In *American Mathematica Society Colloquium Publications, Vol. XXV*, 1967.

[BP82] R. Bonnet and M. Pouzet. Linear extension of ordered sets. In I. Rival, editor, *Ordered Sets*, pages 125–170. D. Reidel Publishing Company, 1982.

[BR94] O. Babaoglu and M. Raynal. Specification and verification of behavioral patterns in distributed computations. In *Proc. of the* 4th *Int. Conference on Dependable Computing for Critical Applications*, San Diego, January 1994. Springer Verlag Series.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM TOCS*, 3(1):63–75, 1985.

[CM91] 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.

[DJR93] C. Diehl, C. Jard, and J.X. Rampon. Reachability analysis on distributed executions. In M.C. Gaudel and J.P. Jouannaud, editors, *TAPSOFT*, pages 629–643, Orsay, April 1993. Springer–Verlag, LNCS 668.

[Fid88] J. Fidge. Timestamps in message passing systems that preserve the partial ordering. In *Proc.* 11th *Australian Computer Science Conference*, pages 55–66, February 1988.

[FZ90] J. Fowler and W. Zwaenepoel. Causal distributed breakpoints. In 10th *IEEE International Conference on Distributed Computing Systems*, pages 134–141, 1990.

[GW92] V.K. Garg and B. Waldecker. Detection of unstable predicates in distributed programs. In *Proc. of the 12*th *conf. Foundations of Software Technology and Theoretical Computer Science*, pages 253–264, New Delhi, India, December 1992. Lecture Notes in Computer Science 652, Springer-Verlag.

[JJ89] C. Jard and T. Jéron. On-line model-checking for finite linear temporal logic specifications. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 275–285, Grenoble, France, June 1989. Springer–Verlag, LNCS 407.

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

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard, P. Quinton, M. Raynal, and Y. Robert, editors, *Proc. Int. Workshop on Parallel and Distributed Algorithms Bonas, France, Oct. 1988*. North Holland, 1989.

[Pnu86] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. *LNCS 224, Current Trends in Concurrency*, pages 510–584, 1986.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399