Runs of Distributed Systems and Partial Orders

Claude Jard
Ecole Normale Supérieure de Cachan
Université Européenne de Bretagne, IRISA
Campus de Ker-Lann, F-35170 Bruz cedex, France
Claude.Jard@bretagne.ens-cachan.fr

Abstract

Twenty-five years ago, one recognized that dynamic behaviors of computer systems distributed on a network can be nicely captured and modeled by discrete partial orders. This paper recalls the main interactions between the two scientific domains, distributed systems and partial orders. This is done for three subjects: 1/ on-the-fly coding of the message causality, 2/ abstraction, and 3/ on-line verification of ordering properties.

1. Introduction

A distributed application running on a network of processors involves a set of autonomous sequential processes, cooperating to a common goal. The cooperation is implemented by the asynchronous exchange of messages.

The behavior of such an application can be modeled by a set of events. An event represents an action performed by a process. It is supposed to take place at a given time in the local reference time of the process. Typically, the events are produced by an actual instrumentation of the computer code: some printing statements for example, executed by some sensors added to the software processes. We qualified these events as *observable* (low-level) events.

It is well-known since the eighties [12] that sets of events of distributed runs can be structured as partial orders of causality. This is done by considering that two events are (causally) ordered if they take place on the same process, or if they are separated by a message exchange. Two events that are not causally ordered are said concurrent.

Precisely, let us consider a set of n sequential processes, denoted $\{P_i\}_{1\leq i\leq n}$, which communicate with each other through reliable directed communication channels. We can view the execution of each process P_i isolated as a sequence of events, corresponding to the state changes that take place in the process. This defines a total order $\widehat{O_i} = (O_i, <_{O_i})$, where O_i is the set of events and the relation $<_{O_i}$ is just the (local) time. The proper granularity defining the observable events varies from application to application. In addition to these observable events (O), we must consider the *communication events* (C), which partially synchronize the observable events. Following Lamport's definition of the *happened before* relation, we say that event x *directly happened before* event y, denoted by $x <_d y$, if

- x and y are events in the same process and x precedes immediately y; or
- x is the sending of a message m and y is the receiving of m.

Birth and death of processes can also be modeled by the happened before relation, provided that a child process is started by the reception of the first message from its father, and that its death is the sending of its last message to its father.

The transitive closure of the $<_d$ relation is the happened before relation $<_{O \cup C}$. A suborder $\widetilde{O'} = (O', <_{O'})$ induced by an order $\widetilde{O} = (O, <_O)$ is an order such that $O' \subseteq O$ and $\forall x, y \in O', x <_{O'} y$ iff $x <_O y$. The partial order of causality between the observable events is the suborder $\widetilde{O} = (O, <_O)$ induced by the order $\widetilde{O \cup C}$. For $x \in O$, we note $\downarrow_O x$ the set of all predecessors of x in $\widetilde{O}$, x excluded, and $\downarrow_O^{im} x$ (resp. $\uparrow_O^{im} x$) is the set of all immediate predecessors (resp. successors) of x in $\widetilde{O}$. We note $\prec_O$ the covering relation. $Cov(\widetilde{O})$ is its graph representation (the Hasse diagram).

Let us now consider a particular application, called "trace checking" [8], in which all the observable events are sent (asynchronously using other communication channels) to a particular process, called the "observer". The role of the observer is to check some properties on the set of observed events during the execution of the system (see Figure 1). The goal is to perform the verification on the fly, event by event. We will see a method based on the on-line construction of a state graph by the observer.

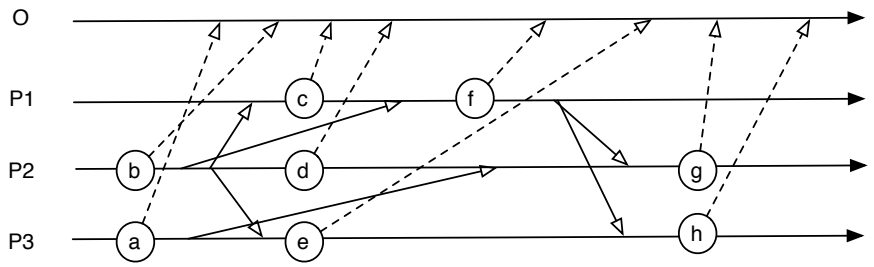

Figure 1: The trace checking application.

This application addresses the three aspects we mentionned, and particularly how the observer can compute the partial order between the observable events from the information communicated by the processes.

2. Dependency tracking (on-the-fly coding)

There exist two major ways to track the dependency information:

- *Transitive dependency tracking* [15, 13, 14, 4]: complete *happened before* information is tracked. That is, the set of predecessors are accumulated upon the different receptions.
- *Direct dependency tracking* [10]: processes do not learn of dependencies resulting from chains of two or more messages.

2.1. Transitive tracking and vector timestamps

This method [4] is based on a management of local counters piggybacked upon interaction. Loosely speaking, the principle of the method is to associate to each observed event x the set of its (observed) predecessors

$\downarrow_O x$.

The set $\downarrow_O x$ can be coded at run time by a list $\mathcal{V}_x$ whose j^{th} component contains the number of predecessors of x on process P_j : $\mathcal{V}_x.j = |\downarrow_O x \cap O_j|$. Property 1 shows how to retrieve the information from the coding ($site(y)$ is the name of the process $P_{site(y)}$ on which y actually occurs):

Property 1 $\forall x, y \in O,\ y \in \downarrow_O x \Leftrightarrow \mathcal{V}_y.site(y) < \mathcal{V}_x.site(y)$

The Fidge's algorithm:
- *each process P_i maintains a list L_i, initialized to $L_i.i = 0$,*
- *when an event x is observed: $\mathcal{V}_x := L_i; L_i.i := L_i.i + 1$,*
- *upon sending a message to P_j : L_i is piggybacked towards P_j,*
- *upon receiving from P_j : $L_i := max(L_i, L_j)$[1] where L_j is the piggybacked list from P_j.*

In our example of Figure 1, $\mathcal{V}_a = [000], \mathcal{V}_b = [000], \mathcal{V}_c = [010], \mathcal{V}_d = [010], \mathcal{V}_e = [011], \mathcal{V}_f = [110], \mathcal{V}_g = [221], \mathcal{V}_h = [212]$. The major problem with this method is the size of the lists L_i and $\mathcal{V}_x$, which can be equal to the number of processes created during the execution. That means that if there is a lot of births and deaths of processes, the size of the lists continuously grows, even if the number of processes simultaneously "alive" remains small.

2.2. Direct dependency tracking and scalar timestamps

The idea of the Direct Dependency coding of Zwaenepoel et al.[10, 5] is to have on each site P_i a list whose j^{th} component is the number of events which occurred on P_j before the last interaction from P_j to P_i. Using our notations, we associate to each observed event x the set $\mathcal{D}_x = \{y \in O : y \lhd x\}$ where $\forall x \in O_i, \forall y \in O_j, y \lhd x$ iff $i = j$ and $y <_i x$ or $\exists x', y'$ (y' is the sending of a message on P_j, x' the corresponding receipt on P_i) such that $y <_{Oj} y'$ and $x' <_{Oi} x$.

As previously, we can code the set D_x by a list of integers, whose i^{th} component contains the number of the direct predecessors of x on process P_i: $\mathcal{D}_x.i = |\{y \in O_i : y \lhd x\}|$. Property 2 shows how to retrieve the information from the coding:

Property 2 $\forall x, y \in O,\ y \in \mathcal{D}_x \Leftrightarrow \mathcal{D}_y.site(y) < \mathcal{D}_x.site(y)$

The Zwaenepoel's algorithm:
- *each process P_i maintains a list L_i, initialized to $L_i.i = 0$,*
- *when an event x is observed : $\mathcal{D}_x := L_i; L_i.i := L_i.i + 1$,*
- *upon sending a message to P_j : $L_i.i$ is piggybacked towards P_j,*
- *upon receiving from P_j : $L_i.j := max(L_i.j, L_j.j)$.*

The algorithm shows that we just have to piggyback a simple scalar, so this method offers a weak and bounded intrusion. Unfortunately, the examination of the timestamps associated with the observable events does not permit to decide their ordering, which can be lost by an "invisible" transitivity. This trouble can be avoided by the satisfaction of the NIVI assumption (NIVI for Non InVisible transitive Interaction), which states that after an interaction from P_j to P_i, we must have an observed event on P_i before any interaction from P_i to any other process P_k. This is the case in our example of Figure 1.

[1] max is performed component by component.

Another way of thinking is to consider that $\mathcal{D}_x$ approximates $\downarrow_O x$. In fact, the causal order is an extension of the order induced by the direct dependency tracking. Pursuing this concept of approximation, we can come back to the Lamport's seminal paper [12] on message causality. This paper also proposed a notion of scalar timestamp, which was in fact a particular coding L: for a given observable event x, $L(x) = height(\downarrow_O x)$. This coding defines a weak order, which is an extension of the causal order. It can be refined, and [3] has proposed a coding by intervals, which becomes exact if and only if the partial order of observable events is an interval order. Many questions remain open in that subject, and the idea is to find interesting codings for particular classes of distributed runs. The main difficulty is to preserve abstraction. In most cases, subclasses that are realistic in terms of communication patterns (remote procedure call paradigm, synchronization barriers, ...) are not stable when considering suborders.

3. Event Filtering

The actual ordering of observed events in front of the observer may be arbitrary. At a given moment, the observer has already received a subset H of O. What subset H of O must be known by the observer to compute the suborder $\widetilde{H}$ induced by $\widetilde{O}$? If any subset is suitable for the computation, we say the coding offers a *strong consistency*.

3.1. Transitive dependency

We saw in the previous section that the vector timestamps compute the set $\downarrow_O x$ for any observable event x. Once such a set is known, it becomes straightforward to compute the *happened before* relation on any subset of observed events, the set of observed events being itself any subset of the set of observable events, as expressed by the Property 3.

Property 3 $\forall H \subseteq O, \forall x \in H,\ \downarrow_H^{im} x \subseteq \downarrow_O x \cap H \subseteq \downarrow_H x$

3.2. Direct dependency

Unfortunately, the direct dependency coding does not ensure strong consistency. To compute the order $\widetilde{H}$, the set H must be downward closed in $\widetilde{O}$ (also called an *IDEAL* of $\widetilde{O}$). Formally, $\forall x \in H, \forall y \in O, y <_O x$ implies $y \in H$. This implies that the observer cannot integrate an event x into the order $\widetilde{H}$ before the reception of all predecessors of x. Now, we can state :

Property 4 $\forall O \subseteq I : NIVI(O), \forall H \subseteq O : IDEAL(H), \forall x \in H,\ \downarrow_H^{im} x \subseteq D_x \subseteq \downarrow_H x$

3.3. On–line construction of the covering digraph of the order

Compared to the vector timestamps coding, the direct dependency coding decreases the intrusion due to piggybacking. The price to be paid is that we are only able to construct 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. Using the chain decomposition induced by processes of $\widetilde{O}$, the algorithm runs in $\mathcal{O}(|O| + n^2)$ for each event.

4. Event Glueing

In many applications, an interesting feature of the observer is to be able to construct "macro-events" from observable events of a lower level. So, correlated events can be presented as a single event. A reasonable goal is that this abstraction remains an order, and that it can be constructed on the fly. Glueing events of a set O can be defined by an equivalence relation $\sim \subseteq O \times O$. The result of the abstraction is the quotient of O by $\sim$, denoted $O/_\sim$, i.e. the set of classes $(\{[x]_\sim\}_{x \in O})$. An example is given in Figure 2a.

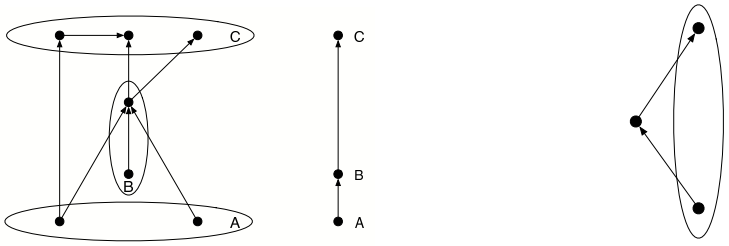

Figure 2: Figure a (on the left) : the macro-events are formed with the surrounding events. Figure b (on the right) : the quotient builds a loop.

Unfortunately, the quotient does not preserve the underlying order (see Figure 2b). We must restrict the $\sim$ relation.

Definition 1 *An equivalence relation $\sim \subseteq O \times O$ is* consistent *with the order $<_O$ if the following condition holds: for all $x_0 \dots x_{2n}$ such that $x_{2i} \sim x_{2i+1}$ and $x_{2i+1} \prec_O x_{2(i+1)}$ for $0 \leq i < n$, if $x_0 = x_{2n}$ then $x_1 \sim \dots \sim x_{2n}$.*

The trivial abstractions, corresponding to $\forall x \in O$, $[x]_\sim = \{x\}$ and $\forall x \in O$, $[x]_\sim = O$, are consistent with any order.

Property 5 *If $\sim$ is consistent with $\widetilde{O}$, $\widetilde{O_\sim}$ is a partial order.*

This notion of consistency implies the notion of convex abstraction developed in [2], which is too precise to preserve the order.

To find non trivial abstractions, we propose to label the events.

Definition 2 *Let us consider a set of types T, and a typing function $\mathcal{T}$ labeling the events of O. The equivalence relation $\sim$ is said* type closed *if $\forall x, y \in O$, $\mathcal{T}(x) = \mathcal{T}(y) \implies x \sim y$. A type closed consistent equivalence is said a* CTC-equivalence.

The set of CTC-equivalences of $O \times O$ is ordered by set inclusion saying that a relation is thinner than an other if every class of the former is included in a class of the latter.

Property 6 *This order forms a lattice.*

The minimal element M is called the *atomic decomposition* of the run observed as O. In the particular case of the NIVI condition, it corresponds to the notion of atom defined in [1]. We keep the same name "atom" for the classes of the $\sim_M$ CTC-equivalence. [7] has shown that we can decompose in atoms a distributed run O in a time $O(|O|^2)$. In [6], we proposed an incremental version of this algorithm to be able to abstract an execution on the fly.

5. Checking of Regular Properties

5.1. State graph and the ideal lattice

For a process P_i, the notion of local state can be easily defined by the occurrence of an observable event, which represents the current state. Due to the sequential nature of the processes, the set of its preceeding events on P_i are in the past and the set of successors are in the future. It is tempting to consider the notion of global state as the union of local states of each process in the system. Unfortunately, all the unions are not consistent with the causality induced by communication (sending a given message must precede the receipt of this message). So, a global state is a set of observable events that have their causal predecessors in the past. Thus, we can define the notion of global state as sets H of observable events that are closed by causal precedence, i.e. $H = \downarrow_O H$, or ideals of the partial order $\widetilde{O}$, denoted $I(O)$.

The set of global states, usually called the state graph in model-checking, is thus the ideal lattice of $\widetilde{O}$, which is known to be distributive. See Figure 3 to have a look of the state graph of our example.

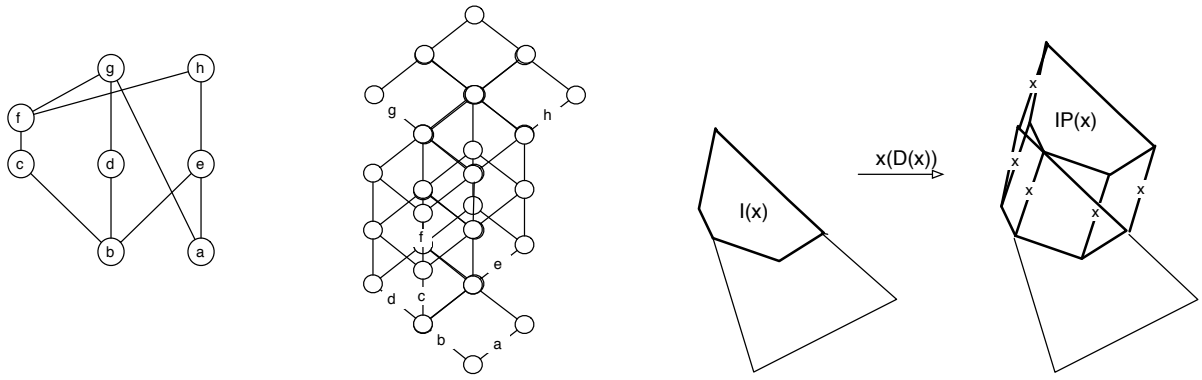

Figure 3: The Hasse diagram of the causal order of Figure 1 and its distributive lattice of ideals, which can grow on-line.

Since our goal is to compute on the fly the state graph of a distributed execution, we studied correlations between ideal lattices yielded by a poset and by one of its subposet when the missing vertex is a maximal one. The ideals grow by adding one element at a time:

Property 7 *Let $\widetilde{Q}$ be a poset, $\forall I, J \in I(Q)$, $I \prec_{I(Q)} J \Longleftrightarrow I \subset J$ and $|J \setminus I| = 1$.*

$J \setminus I$ is called the *label* of the edge $I \prec_{I(Q)} J$.

Let $\widetilde{P}$ be a poset and x one of its maximal element. As one can see in Figure 3, $\widetilde{I(P)}$ is structured in three different parts. The upper part $IP(x)$ is formed with the ideals containing x: $IP(x) = \{I \in I(P) : I \cap \{x\} \neq \emptyset\}$. The medium part $I(x)$ is formed with the ideals containing the immediate predecessors of x and not containing x: $I(x) = \uparrow^{im}_{I(P)}(\downarrow_P x) \setminus IP(x)$. $I(x)$ and $IP(x)$ are two isomorphic sublattices. The lower part is formed with the remaining ideals. This is formalized as follows.

Property 8 *Let $\widetilde{P}$ be a poset, let $x \in Max(P, \widetilde{P})$, let $\widetilde{P'}$ be the subposet $\widetilde{P \setminus \{x\}}$ and $Cov(\widetilde{I(P')}) = (I(P'), E_{I(P')})$. Let $I(x) = \uparrow^{im}_{\widetilde{I(P')}}(\downarrow_{\widetilde{P}} x)$. Let $G(Q) = (Q, E_Q)$ be an acyclic directed graph isomorphic to $Cov(\widetilde{I(x)})$ by a map ϕ.*
Then, the acyclic directed graph $G(Z) = (Z, E_Z)$ where $Z = Q \uplus I(P')$ and E_Z is defined by:

1. $\forall q_1, q_2 \in Q \times Q : q_1 q_2 \in E_Z \Longleftrightarrow \phi(q_1)\phi(q_2) \in E_{I(P')}$
2. $\forall p, q \in I(P') \times Q : pq \in E_Z \Longleftrightarrow q = \phi^{-1}(p)$
3. $\forall p_1, p_2 \in I(P') \times I(P') : p_1 p_2 \in E_Z \Longleftrightarrow p_1 p_2 \in E_{I(P')}$

is isomorphic to $Cov(\widetilde{I(P)})$ by the map φ: $z \longmapsto \begin{cases} \phi(z) \cup \{x\} & \text{if } z \in Q \\ z & \text{otherwise} \end{cases}$

5.2. On the fly computation of the state graph

Assuming that the number of processes involved in a distributed computation is finite, using our previous theorem, we are now able to give an algorithm for an on the fly computation of a distributed execution state graph (assuming the knowledge of the covering causality relation) [9].

Let $\widetilde{P}$ be an order, let $\{P_i\}_{1 \leq i \leq n}$ be a chain decomposition of $\widetilde{P}$ given by the processes and let Δ be the map (which extends the vector clock coding of causality) defined by:

$$\begin{array}{rcl} \Delta & : & I(P) \longrightarrow \mathbb{N}^k \\ & & I \longmapsto (\mid I \cap P_i \mid)_{1 \leq i \leq k} \end{array}$$

This map embeds the lattice into the $(\mathbb{N}^k, \leq_{\mathbb{N}^k})$ lattice.

Property 9 *$\forall I, J \in I(P)$, the following properties are equivalent:*

(i) $I \leq_{I(P)} J$
(ii) $\Delta(I) \leq_{\mathbb{N}^n} \Delta(J)$
(iii) *All maximal chains from I to J in $\widetilde{I(P)}$ have length $len(I, J) = \sum_{i=1}^{n} (\Delta(J)[i] - \Delta(I)[i])$. $\forall i \in \{1, \ldots, k\}$, $\Delta(J)[i] - \Delta(I)[i] \geq 0$. And there is at least one maximal chain.*

The "on-line state construction algorithm"

Input:
(1) $Cov(\widetilde{P})$ with a chain decomposition [2] $\{P_i\}_{1 \leq i \leq n}$.
(2) For any $y \in P$, $\delta(y) = (|\downarrow^{im}_{\widetilde{P}} y \cap P_i|)_{1 \leq i \leq n}$.

[2] The chain decomposition on $\widetilde{P}$ is an extension of the chain decomposition on $\widetilde{P'}$, that is: $\forall i,\ 1 \leq i \leq n$ such that $i \neq site(x)$ we have $\{P_i\} = \{P'_i\}$ and $\{P_{site(x)}\}$ is $\{P'_{site(x)}\}$ with x added as maximal element.

(3) A vertex $x \in Max(P, \widetilde{P})$ and the set $D(x) = \downarrow_P^{im} x$.

(4) When $P' = P \setminus \{x\}$ and $\widetilde{P}/_{P'} = \widetilde{P'}$, $Cov(\widetilde{I(P')})$, such that:

(a) Each $y \in P'$ is directly related to its corresponding $\downarrow_{P'}^{im} y$ in $Cov(\widetilde{I(P')})$.

(b) Each edge yz in $Cov(\widetilde{I(P')})$ is labeled by $I_z \setminus I_y$ where I_y (resp. I_z) is the ideal of $\widetilde{P'}$ corresponding to the vertex y (resp. z) in $\widetilde{I(P')}$.

(c) Outgoing edges of any vertices in $Cov(\widetilde{I(P')})$ are stored ordered by increasing index of the chain their label belong to.

Body:

(1) Find $\downarrow_{P'}^{im} D(x)$ in $Cov(\widetilde{I(P')})$.

(2) Build a directed graph $G(Q)$ isomorphic to $Cov(\widetilde{I(x)})$, by a map ϕ (where $I(x) = \{I \in I(P'), \downarrow_{P'}^{im} D(x) \leq_{I(P')} I\}$).

(3) For any $I \in I(x)$, create the directed edge $(I, \phi^{-1}(I))$ with label x and store this edge according to the storage order.

(4) Create a link between x and $\phi^{-1}(\downarrow_{P'}^{im} D(x))$.

Output:

1. $Cov(\widetilde{I(P')})$, such that:

(a) Each $y \in P$ is directly related to its corresponding $\downarrow_P^{im} y$ in $Cov(\widetilde{I(P)})$.

(b) Each edge yz in $Cov(\widetilde{I(P)})$ is labeled by $I_z \setminus I_y$ where I_y (resp. I_z) is the ideal of $\widetilde{P}$ corresponding to the vertex y (resp. z) in $\widetilde{I(P)}$.

(c) Outgoing edges of any vertices in $Cov(\widetilde{I(P)})$ are stored ordered by increasing index of the chain their label belong to.

Property 10 *[9]* $\widetilde{I(O)}$ *can be computed in time complexity:* $O(n(\mid I(O) \mid + \mid O \mid^2))$.

5.3. Model-checking

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.

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.

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 $I \stackrel{a}{\longrightarrow} J$ iff $I \prec_{I(O)} J$ and a is the label edge, denoted $\lambda(J - I)$.

Let us now define the class of regular properties defined by finite automata on finite words. To 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 the bottom to I.

The satisfaction of a property ϕ in an ideal I is defined by: $I \models \phi$ iff $\lambda(Path(I)) \subseteq \mathcal{L}(\phi)$

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

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: $I(O) \models \phi$ iff $O \models \phi$

From the above definitions, we can easily deduce that: $\forall I \in I(O)$, $I \models \phi \Leftrightarrow \phi(I) \subseteq F$ 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.

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:

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

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.

Since each ideal is attributed with a state of the property automaton, the complexity of the checking algorithms is linear in the product of the size of the lattice and the size of the automaton.

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

6. Conclusion

This paper has presented some practical links between finite partial order theory and the analysis of distributed runs of computer systems. This was illustrated on the problem of trace-checking. Several notions seems interesting at the interface between these two disciplines:

- the on-the-fly coding of causality, with possible use of approximations seen as order extensions;
- the question of abstraction by glueing events, which seems close to notions of hierarchical decomposition of orders,
- the relation between states and ideals, which forces to revisit the question of lattice construction in an on-line context. The user of sub-lattices (like the lattice of maximal antichains [11]) could have some applications in the context of trace-checking.

References

[1] Mohan Ahuja, Ajay D. Kshemkalyani, and Timothy Carlson. A basic unit of computation in distributed systems. In *ICDCS*, pages 12–19, 1990.

[2] Twan Basten, Thomas Kunz, James P. Black, Michael H. Coffin, and David J. Taylor. Vector time and causality among abstract events in distributed computations. *Distrib. Comput.*, 11(1):21–39, 1997.

[3] Claire Diehl and Claude Jard. Interval approximations of message causality in distributed executions. In *STACS*, pages 363–374, 1992.

[4] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, 1991.

[5] J. Fowler and Willy Zwaenepoel. Causal distributed breakpoints. In *ICDCS*, pages 134–141, 1990.

[6] T. Gazagnaire and C. Jard. Abstraire à la volée les événements d'un système réparti. In *NOTERE, 7ème Conférence Internationale sur les Nouvelles Technologies de la Répartition*, pages 241–252, 2007.

[7] L. Helouet and P. Le Maigat. Decomposition of message sequence charts. In S. Graf and C. Jard, editors, *2nd Workshop on SDL and MSC (SAM2000)*, Grenoble, France, June 2000.

[8] Claude Jard, Thierry Jéron, Guy-Vincent Jourdan, and Jean-Xavier Rampon. A general approach to trace-checking in distributed computing systems. In *ICDCS*, pages 396–403, 1994.

[9] Claude Jard, Guy-Vincent Jourdan, and Jean-Xavier Rampon. On-line computations of the ideal lattice of posets. *ITA*, 29(3):227–244, 1995.

[10] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *J. Algorithms*, 11(3):462–491, 1990.

[11] Guy-Vincent Jourdan, Jean-Xavier Rampon, and Claude Jard. Computing on-line the lattice of maximal antichains of posets. *Order*, 11:197–210, 1994.

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

[13] Friedeman Mattern. On the relativistic structure of logical time in distributed systems. *Parallel and Distributed Algorithms*, pages 215–226, 1989.

[14] A. Prasad Sistla and Jennifer L. Welch. Efficient distributed recovery using message logging. In *PODC*, pages 223–238, 1989.

[15] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, 1985.