PUBLICATION
INTERNE
N^{o} 817

ON THE FLY TESTING OF REGULAR PATTERNS IN DISTRIBUTED COMPUTATIONS

EDDY FROMENTIN, MICHEL RAYNAL, VIJAY K GARG
AND ALEX TOMLINSON

ISSN 1166-8687

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 99 84 71 00 – Fax : (33) 99 84 71 71

ON THE FLY TESTING OF REGULAR PATTERNS IN DISTRIBUTED COMPUTATIONS

Eddy Fromentin*, Michel Raynal*, Vijay K Garg** and Alex Tomlinson**

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet Adp

Publication interne n° 817 — Avril 1994 — 13 pages

Abstract: A class of properties of distributed computations is described and an algorithm which detects them is presented. This class of properties called regular patterns allows the user to specify an expected (or unwanted) behavior of a computation as sequences of relevant events (or as sequences of local predicates that must be successively verified). The sequences are defined by a finite state automaton (hence the name regular patterns). A computation verifies the property if and only if one of its causal paths matches a sequence.

Key-words: behavioral property, distributed computation, distributed debugging, on the fly detection, regular pattern.

(Résumé : tsvp)

This paper will appear in the proceedings of the 23th Int. Conf. on Parallel Processing, Pennsylvanie State Univ., August 1994.

*IRISA – Campus de Beaulieu, 35042 RENNES cedex – FRANCE, e-mail:{fromenti, raynal}@irisa.fr
**Dept. of ECE – University of Texas at AUstin, Austin TX, e-mail: {vijay,alext}@pine.ece.utexas.edu

Centre National de la Recherche Scientifique
(URA 227) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Détection au vol de motifs réguliers dans une exécution répartie

Résumé : Nous définissons ici une classe de propriétés pour les exécutions réparties d'un programme distribué ainsi qu'un algorithme qui les detecte au vol: les motifs réguliers. Cette classe permet à un utilisateur de spécifier un comportement souhaité ou non au moyen de séquences d'événements pertinents ou de séquences de prédicats qui doivent être successivement vérifiés. Ces séquences sont exprimées à l'aide d'automates d'états finis (d'où l'emploi du terme *régulier*). Une exécution répartie vérifie la propriété ainsi exprimée si et seulement si un de ses chemins causaux vérifie une des séquences admises par l'automate.

Mots-clé : propriété comportementale, exécution répartie, déverminage de programmes répartis, detection au vol, motifs réguliers.

1 INTRODUCTION

The study of behavioral properties of parallel and distributed computations is important for analysis, testing and debugging. By "behavioral property" we mean a property which expresses some order notion [3, 8, 15]. This is a typical situation in the analysis of synchronization in distributed computations where we want to detect if a sequence of causally related relevant events (or a sequence of local states that satisfy some predicates) has been produced by a computation.

One systematic way to detect such properties consists first in building the lattice associated with a distributed execution [4, 5] and then in traversing this lattice to detect the considered behavioral pattern [2, 4]. A node of this lattice represents a possible global state of the distributed computation and an edge from one node to another represents an event that would make the computation progress from the first to the second global state. The kinds of patterns that can be studied in this way have very few limitations but the associated cost is very high since this method requires building the lattice (even if it is possible to build and to exploit it on the fly, during the execution of the analyzed computation as in [1, 2, 4]).

To prevent this cost, some authors have defined particular behavioral patterns whose detection does not require building the lattice or using a centralized manager. These detection algorithms are superimposed on the distributed computation and use a simple piggybacking technique to ensure consistency of the detection. One of the first of these behavioral patterns has been introduced by Miller and Choi [12]; called *linked predicates* such a pattern describes a causal sequencing of local states verifying some predicates ([6, 8, 12] present algorithms to detect such patterns). In [8], Hurfin et al. generalize this pattern to a more general *atomic sequence of predicates*; here some events can be forbidden between each pair of consecutive relevant events.

This paper presents a class of behavioral patterns that includes linked predicates and atomic sequences as particular cases. This class of properties called regular patterns allows the user to specify a behavioral property of a computation as sequences of relevant events (or of local predicates that must be verified). These sequences are defined by a finite state automaton, hence the name regular pattern. The property is detected as soon as a causal path of the computation matches one of these sequences. The detection algorithm works on the fly without building a complex and expensive data structure such as a lattice.

The paper is structured as follows: Section 2 presents the model of distributed computations. Section 3 defines regular patterns and the meaning of the claim "this computation satisfies this pattern". Section 4 is devoted to the presentation of a detection algorithm for regular patterns.

2 MODEL OF DISTRIBUTED COMPUTATIONS

2.1 The underlying system

The underlying system that executes distributed programs is composed of n nodes that can exchange messages. Neither shared memory nor a global clock is available. Messages are exchanged through reliable but not necessarily FIFO channels. Transmission delays are finite but unpredictable.

2.2 Distributed computations

A distributed program is made of n processes P_i which communicate and synchronize only by means of message passing. A distributed computation describes the execution of a distributed program.

2.2.1 Partial order on events

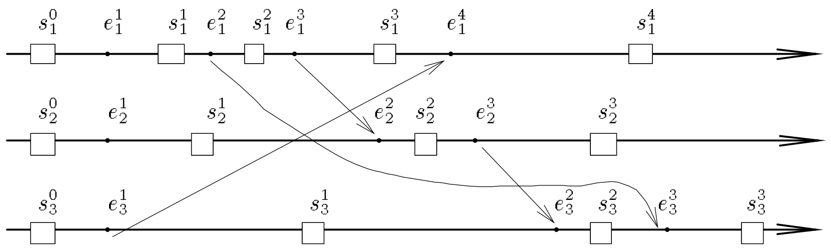
The activity of each process P_i is modeled by a sequence of events h_i called the history of P_i: $h_i = e_i^1\, e_i^2\, e_i^3 \cdots$ (where e_i^x is the x^{th} event executed by P_i). An event is a send event, a receive event or an internal event. Let $\overset{e}{\rightarrow}$ be the classical binary relation defined by Lamport on events [10]: $e_i^x \overset{e}{\rightarrow} e_j^y$ iff $i = j$ and $x + 1 = y$ (in that case we note $e_i^x <_{im}^{e} e_j^y$), or e_i^x is the sending of m and e_i^y the reception of m, or $\exists e_k^z$ such that $e_i^x \overset{e}{\rightarrow} e_k^z$ *and* $e_k^z \overset{e}{\rightarrow} e_j^y$.

Let H be the set of all the events. A distributed computation is a partially ordered set (poset) $\widehat{H} = (H, \overset{e}{\rightarrow})$.

2.2.2 Partial order on local states

Each process P_i is initially in the local state s_i^0. Event e_i^x transforms P_i's local state s_i^{x-1} into s_i^x. Let S be the set of local states of all the processes. Moreover, let $\overset{s}{\rightarrow}$ be the following binary relation on local states, $s_i^x \overset{s}{\rightarrow} s_j^y$ iff one of the following hold: $i = j$ and $x + 1 = y$ (in that case we note $(s_i^x <_{im}^{s} s_j^y)$, or e_i^{x+1} is the sending of m and e_i^y the reception of m, or $\exists s_k^z$ such that $s_i^x \overset{s}{\rightarrow} s_k^z$ *and* $s_k^z \overset{s}{\rightarrow} s_j^y$.

This relation on local states is Lamport's relation applied to local states. A poset $\widehat{S} = (S, \overset{s}{\rightarrow})$ is associated with each distributed computation $\widehat{H} = (H, \overset{e}{\rightarrow})$. Figure 1



s_1^0 e_1^1 s_1^1 e_1^2 s_1^2 e_1^3 s_1^3 e_1^4 s_1^4
s_2^0 e_2^1 s_2^1 e_2^2 s_2^2 e_2^3 s_2^3
s_3^0 e_3^1 s_3^1 e_3^2 s_3^2 e_3^3 s_3^3

Figure 1: A distributed execution

displays a distributed execution in the classical space-time diagram (black circles are events, white squares are local states). To obtain the representation of an associated poset $\widehat{H}$ or $\widehat{S}$ we consider only relevant elements (either events or local states) and add all edges due to transitivity.

2.2.3 Reduction of a distributed execution

Each poset $\widehat{H} = (H, \xrightarrow{e})$ can be associated with a relation called H-reduction, $\widehat{H_r} = (H, \xrightarrow{e}_r)$, where relation $\xrightarrow{e}_r$ is relation $\xrightarrow{e}$ from which have been suppressed: all transitivity edges due to the transitive closure of $<^e_{im}$ and all transitivity edges connecting events from two distinct processes[1].

Figure 2 displays the H-reduction associated with the computation shown in Figure 1 (edge (e_1^2, e_3^3) has been suppressed as it is a transitivity edge). The S-reduction $\widehat{S_r} = (S, \xrightarrow{s}_r)$ is defined similarly from $\widehat{S}$.

A causal path of $\widehat{H}$ (respt. $\widehat{S}$) is a directed path in the associated H-reduction $\widehat{H_r}$ (respt. in the associated S-reduction $\widehat{S_r}$). $\widehat{H_r}$ (respt. $\widehat{S_r}$) will be used to reason about sequences of relevant events (respt. sequences of local states satisfying some predicates).

[1]If messages deliveries respect causal order [13], there are no such transitivity edges.

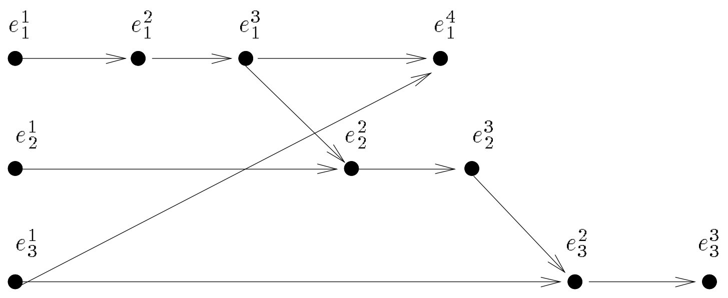


Figure 2: The H-reduction of the poset $\widehat{H}$

3 REGULAR PATTERNS

3.1 Regular pattern

A regular pattern P is a language (set of words) defined on a vocabulary V, by a finite state automaton. The regular pattern and the corresponding finite state automaton are given the same name P.

3.2 Labelling

In order to recognize a pattern on a distributed execution, it is necessary to label events (or local states) with symbols of V. In order to ignore irrelevant events (or local states) the empty symbol ϵ belongs to V.

The labelling function λ associates with each event e (respt. to each local state s) a set of labels $\lambda(e)$ (respt. $\lambda(s)$). Considering the causal path $C = e^1 e^2 e^3 \cdots$, $\lambda(C)$ represents the set $\{x_1 x_2 x_3 \cdots \mid x_i \in \lambda(e^i)\}$. In others words $\lambda(C)$ is the set of all the words that can be build from C with all possible labellings.

3.3 Detection of a pattern

A pattern P is verified in a distributed computation $\widehat{H}$ if there is a causal path C such that an element of $\lambda(C)$ is recognized by P. If P is on events, C is a path of $\widehat{H_r}$; if P is on local states C is a path of $\widehat{S_r}$.

3.4 Examples

Example **1**

Consider the following sequence of local predicates $\Phi_1 = \varphi_1\varphi_2\varphi_3$ [6, 8, 12]. This sequence describes a pattern that is verified if there is a causal path in $\widehat{S_r}$ in which there is a local state of $P_{x(1)}$ satisfying φ_1 followed by a local state of $P_{x(2)}$ satisfying φ_2 followed by a local state of $P_{x(3)}$ satisfying φ_3.

All local states of $P_{x(i)}$ that satisfy φ_i are labeled φ_i. Other states are labeled ϵ. Φ_1 is true if there exist a causal path that matches the regular expression $\varphi_1\varphi_2\varphi_3$.

Example **2**

Consider the more sophisticated predicate $\Phi_2 = [\theta_1]\varphi_2[\theta_3]\varphi_4[\theta_5]$ where θ_i and φ_j are local predicates [8]. Φ_2 is true if there is a causal path in which as before there is a local state s_2 in $P_{x(2)}$ satisfying φ_2 and a local state s_4 in $P_{x(4)}$ satisfying φ_4 and additionally on this causal path all local states preceding s_2 do not verify θ_1, all local states in between s_2 and s_4 do not verify θ_3 and all local states following s_4 do not verify θ_5 (see [8] for more details about atomic sequences of predicates).

A local state that satisfies a local predicate θ or φ is labeled by this predicate. Local states of a process on which no local predicates φ or θ are defined are labeled ϵ. Φ_2 is true if there is a causal path recognized by the following regular expression: $(\varphi_2 + \theta_3 + \varphi_4 + \theta_5)^*\varphi_2\ (\theta_1 + \varphi_2 + \varphi_4 + \theta_5)^*\varphi_4\ (\theta_1 + \varphi_2 + \theta_3 + \varphi_4)^*$.

4 AN ON THE FLY TESTING ALGORITHM

4.1 The finite state automaton

Let P be the finite state automaton (deterministic or not) describing the regular pattern we are interested in. $P = (Q, V, q_0, Q_F, \delta)$ with: Q the set of states, V the set input symbols (including ϵ), q_0 the initial state, Q_F the set of final states and δ the transition function[2].

Figure 3 displays the finite state automaton associated with the pattern of example 2 in Section 3.4.

4.2 Working on the H(or S)-reduction

In order to work on the H(or S)-reduction, processes are equipped with a vector clock system [11]. Each process P_i is augmented with a vector $vc_i[1\cdots n]$ whose elements

[2] δ is such that $(q, x) \overset{\delta}{\mapsto} Q'$ with $Q' \subseteq Q$ and $(q, \epsilon) \overset{\delta}{\mapsto} \{q\}$ for all states $q \in Q$.

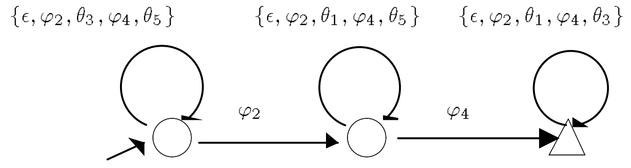

$\{\epsilon, \varphi_2, \theta_3, \varphi_4, \theta_5\}$
$\{\epsilon, \varphi_2, \theta_1, \varphi_4, \theta_5\}$
$\{\epsilon, \varphi_2, \theta_1, \varphi_4, \theta_3\}$
φ_2
φ_4

Figure 3: A finite state automaton for an atomic sequence

are initialized to 0. Each time it produces an event, $vc_i[i]$ is incremented by a positive value. Each message m piggybacks the value $vc(m)$ of the vector clock of its sender. When it receives a message m, process P_i updates vc_i to $max(vc_i, vc(m))$[3]. When it arrives at P_i, message m adds a transitivity edge to the H(or S)-reduction if and only if $vc_i > vc(m)$[4] [14].In that case this receive is ignored from the point of view of the detection. In the following, we do not describe vector clock management.

4.3 The detection algorithm

A controller CTL_i is superimposed on each process P_i of the underlying computation. In addition to vector clocks, CTL_i manages an array $A_i[Q]$ of boolean values with the following meaning: $A_i[q]$ is true iff there is a causal path C ending at the current local state of P_i (or at the last event it produced such that a word of $\lambda(C)$ puts P in state q.

$A_i[q_0]$ is initially the only entry with value $True$. CTL_i executes the following statements: S_1, S_2 and S_3. Statement S_1 is executed each time an event e_i is produced by a process P_i (or each time a new local state s_i is entered). Statements S_2 and S_3 are associated with each communication message.

[3]max is done element per element.

[4]$vc_i > vc(m) \Leftrightarrow \forall j \in 1 \cdots n : vc_i[j] \geq vc(m)[j]$ and $\exists j \in 1 \cdots n : vc_i[j] > vc(m)[j]$.

```
$S_1$: Each time an event $e_i$ is produced by $P_i$:
    **let** $X = \lambda(e_i)$; % or $X = \lambda(s_i)$ %
    **foreach** $x \in X$ **do**
        $A^x[Q] := (False, \cdots, False)$;
        **foreach** $q \in Q$ **such that** $A_i[q]$ **do**
            **foreach** $r \in \delta(q, x)$ **do** $A^x[r] := True$ **od**
        **od**
    **od**
    **foreach** $q \in Q$ **do** $A_i[q] := \bigvee_{x \in X} A^x[q]$; **od**
```

```
$S_2$: When $P_i$ sends a message $m$:
    **if** the regular pattern is on events **then**
        execute statement $S_1$ where $e_i$ is this send event;
    Piggyback $A_i$ on $m$ before it is sent;
```

```
$S_3$: When $P_i$ receives a msg $m$ piggybacked with $A(m)$:
    **if** $m$ does not create a transitivity edge **then**
        **foreach** $q \in Q$ **do** $A_i[q] := A_i[q] \vee A(m)[q]$; **od**
    **if** the regular pattern is on events **then**
        execute $S_1$ where $e_i$ is this receive event.
```

4.4 Correctness proof

The proof is done for regular patterns of events (a similar reasoning can be done for regular patterns of local predicates). Consider the H-reduction associated with a distributed computation. Let e be a node of this partial order and A_e the value of the array A associated with the event e by the previous algorithm. Let $CP(e)$ be the set of causal paths ending at e.

The proof consists in showing that for any event e the following proposition PR is true.

PR $A_e[q]$ is true iff there exists a causal path $C \in CP(e)$ such that: $y \in \lambda(C)$ and the word y puts the automaton in state q.

Proof: Augment each process P_i by a fictitious initial event e_i^0 labeled ϵ. The proof is done by induction on the rank of events in the augmented H-reduction (the rank of an event is the length of the longest causal path ending at e).

Due to initialization of arrays A_i, the proposition is trivially true for all initial events which have rank 1. Now assuming proposition *PR* is true for all events with rank less than r, consider an event e of rank r. Events, other than e, belonging to a path of $CP(e)$ have a rank less than r. Two cases have to be considered:

1. e is an internal or a send event.
   It follows that e has only one predecessor e', and e' has rank $r-1$. Statement $S-1$ checks all possible transitions from all automaton states q' such that $A_{e'}[q']$ with all labels of $\lambda(e)$. It follows that the array A_e, associated with e whose rank is r, satisfies proposition *PR*.

2. e is a receive event.
   In this case, e has two predecessors e' and e'' and at least one of them has rank $r-1$. Statement S_3 merges $A_{e'}$ and $A_{e''}$ into an intermediate array which is used as the input array when S_3 calls S_1. The same reasoning as in case 1 applies and proposition *PR* is verified for e.

It follows proposition *PR* is true for all events of rank r.

□

5 RELATED WORK

As mentioned in section 3, linked predicates [6, 8, 12] and atomic sequences of predicates [8] can be expressed as regular patterns. In the case of linked predicates, the associated automaton is linear and messages have only to piggyback the last recognized state of this automaton.

In [9], Jard et al. propose to check for regular properties of distributed computations. These properties are described by finite state automata. To answer the question "is this property verified by the distributed computation ?" Jard et al. consider the lattice of global states representing all possible observations of the distributed computation (an observation is a path in this lattice from the initial to the final global state [4]). "Some" satisfaction is claimed when at least one path of the lattice is recognized by the automaton whereas "every" satisfaction is claimed when each path of the lattice is recognized by the automaton (two distinct paths of the lattice can be associated with two distinct sentences both accepted by the automaton). In our algorithm the verification is done, on the fly, on the paths of the partial order generated by the execution.

6 CONCLUSION

This paper has introduced a class of behavioral patterns for distributed computations. Such patterns are causal sequences of relevant events (or causal sequences of local states that satisfy some predicates) defined by a finite state automaton (hence the name regular patterns). Theoretical results (about complementary, union, etc) on finite state automata can simplify the task of designing and verifying such behavioral patterns. An algorithm that detects these regular patterns on the fly has been proposed; it needs neither complex data structures (such as a lattice), nor a central monitor, nor additional messages; it only requires that each message piggybacks a boolean array (whose size is the number of states of the finite state automaton) and a vector clock if causal order is not ensured by the underlying system.

An implementation of this detection algorithm is currently in progress in our distributed debugging facility [7].

Acknowledgements

We are grateful to Ö. Babaoğlu, M. Hurfin and Cl. Jard for interesting discussions about distributed detection of predicates. This work has been supported in part by the Commission of European Communities under ESPRIT Programme BRA 6360 (BROADCAST), by the French CNRS under the grant Parallel Traces and by a French-Israeli grant on distributed computing.

References

[1] Ö. Babaoğlu and K. Marzullo. *Consistent global states of distributed systems: fundamental concepts and mechanisms, in Distributed Systems*, chapter 4. ACM Press, Frontier Series, S.J. Mullender Ed., (1993).

[2] Ö. Babaoğlu and M. Raynal. "Specification and detection of behavioral patterns in distributed computations", In *Proc. of 4th IFIP WG 10.4 Int. Conference on Dependable Computing for Critical Applications*, Springer Verlag Series in Dependable Computing, San Diego, (January, 1994).

[3] P.C. Bates and J.C. Wileden. “High-level debugging of distributed systems: the behavioral abstraction approach”, *Journal of Systems and Software*, (December, 1983) 4(3):255–264.

[4] R. Cooper and K. Marzullo. “Consistent detection of global predicates”, In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 167–174, Santa Cruz, California, (May, 1991).

[5] 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).

[6] 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).

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

[8] M. Hurfin, N. Plouzeau, and M. Raynal. “Detecting atomic sequences of predicates in distributed computations”, In *Proc. ACM workshop on Parallel and Distributed Debugging*, pages 32–42, San Diego, (May, 1993). (Reprinted in SIGPLAN Notices, Dec. 1993).

[9] 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).

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

[11] 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).

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

[13] M. Raynal, A. Schiper, and S. Toueg. "The causal ordering abstraction and a simple way to implement it", *Information Processing Letters*, (1991), 30:343–350.

[14] A. Schiper, J. Eggli, and A. Sandoz. "A new algorithm to implement causal ordering", In J.C. Bermond and M. Raynal, editors, *Proc. 3rd Workshop on Distributed Algorithms*, pages 219–232, Springer Verlag, LNCS 392, Nice, France, (September, 1989).

[15] R. Schwarz and F. Mattern. "Detecting causal relationships in distributed computations : in search of the holy grail", *Distributed Computing*, (1994), 7(3), pages 149–174.