Detecting Diamond Necklaces in Labeled Dags (A Problem from Distributed Debugging)

Michel Hurfin, Michel Raynal

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Detecting Diamond Necklaces in Labeled Dags (A Problem from Distributed Debugging)

Michel Hurfin, Michel Raynal

N° 2838

Mars 1996

THÈME 1

apport de recherche

ISSN 0249-6399

Detecting Diamond Necklaces in Labeled Dags
(A Problem from Distributed Debugging)

Michel Hurfin, Michel Raynal

Thème 1 — Réseaux et systèmes
Projet Adp

Rapport de recherche n ° 2838 — Mars 1996 — 13 pages

Abstract: The problem tackled in this paper originates from the debugging of distributed applications. Execution of such an application can be modeled as a partially ordered set of process states. The debugging of control flows (sequences of process states) of these executions is based on the satisfaction of predicates by process states. A process state that satisfies a predicate inherits its label. It follows that, in this context, a distributed execution is a labeled directed acyclic graph (dag for short). Debug or determine if control flows of a distributed execution satisfies some property amounts to test if the labeled dag includes some pattern defined on predicate labels.

This paper first introduces a general pattern (called *diamond necklace*) which includes classical patterns encountered in distributed debugging. Then an efficient polynomial time algorithm detecting such patterns in a labeled dag is presented. To be easily adapted to an on-the-fly detection of the pattern in distributed executions, the algorithm visits the nodes of the graph according to a topological sort strategy.

Key-words: Distributed systems, On the fly global predicate detection

(Résumé : tsvp)

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

Détecter des colliers de diamants dans des graphes étiquetés, orientés et sans circuits

Résumé : Le problème abordé dans cet article a été rencontré dans le cadre d'une étude sur la mise au point des application réparties. L'exécution de telles applications peut être modélisée par un ensemble partiellement ordonné d'états locaux des processus. L'analyse des flots de contrôle (séquences d'états locaux) de ces exécutions est fondée sur le fait que les états locaux satisfont des prédicats. Un état local qui satifait un prédicat hérite de son étiquette. En conséquence, dans ce contexte, une exécution répartie est représentée par un graphe étiqueté, orienté et sans circuits. Mettre au point ou déterminer si un flot de contrôle satisfait une propriété revient à tester si le graphe étiqueté inclut un motif défini à l'aide des étiquettes des prédicats.

Cet article présente tout d'abord un motif général (appelé *collier de diamants*) qui inclut les motifs classiques rencontrés dans le domaine de la mise au point des programmes répartis. Ensuite, un algorithme performant qui permet de détecter de tels motifs dans un graphe étiqueté et dont la complexité en temps est polynomial, est exposé. Afin d'être facilement adaptable pour détecter au vol des motifs lors d'une exécution répartie, l'algorithme visite les sommets du graphe selon la stratégie du tri topologique.

Mots-clé : Systèmes répartis, Détection au vol de prédicats globaux.

1 Introduction

This paper presents an algorithm to detect a sophisticated pattern (called *diamond necklace*) in a labeled directed acyclic graph. The problem solved by this algorithm originated from the detection of properties of distributed computations in our current effort to design and implement a facility for debugging distributed programs [9]. These programs are composed of a finite set of processes cooperating by the only means of message passing. From an initial state a process produces a sequence of process states according to its program text. In the context of the debugging of distributed programs, a distributed execution is usually modeled as a partially ordered set of process states [7]. Informally, process state s_1 precedes s_2 if both have been produced by the same process with s_1 first, or if s_1 has been produced by some process before it sent a message to another process and the receiver process produced s_2 after receiving this message; this causal precedence relation is nothing else than Lamport's "happened before" relation expressed on process states [12]. A directed path of process states starting from an initial process state is usually called a control flow.

We designed and implemented several distributed algorithms that on-the-fly detect properties on control flows of distributed computations [10, 5, 4]. Basically a property is defined as a language on an alphabet of predicates (a predicate being a boolean expression in which appear variables of a single process); a pattern is a word of this language. If a local state satisfies a given predicate, it inherits its label: so words can be associated with each control flow. Finally a control flow satisfies a property if one of its words belongs to the language defining the property, *i.e.*, if it matches some pattern. Such an approach has been formalized in [1]. These properties are fundamentally sequential in the sense they consider each control flow separately.

Sequential properties are not powerful enough to express patterns which are on several control flows. An example of such a property is the following one: "there is a process state s_1 satisfying a predicate P_1 causally preceding a process state s_2 satisfying a predicate P_2 and all paths of process states starting at s_1 and ending at s_2 satisfy some sequential property". A logic able to express such non-sequential properties has been introduced in [6].

Here we abstract from distributed executions and consider labeled directed acyclic graphs. We first define (Section 2) a general type of patterns (diamond necklace) for labeled dags which includes as particular cases sequential and non-sequential patterns useful in distributed debugging, and then (Section 3) we present an algorithm to detect these patterns. In order to be adaptable to on-the-fly distributed detection in the context of distributed debugging, it is required that the algorithm visits the nodes of the dag according to a topological sort strategy.

So this paper solves a new problem (to our knowledge), namely deciding if a labeled dag includes some specific pattern, that we met in designing and implementing a distributed debugging facility.

2 Diamond Necklaces

2.1 Labeled Dags

Let $G = (V, E)$ be a finite dag with n vertices. Notations v, v', v_i, v^i are used to represent elements of V. Let v_i and v_j be two vertices of V; $\mathcal{P}(v_i, v_j)$ is the set of all the paths in G from v_i to v_j.

$$\mathcal{P}(v_i, v_j) = \{(v^1, v^2, \cdots, v^u) \mid (v^1 = v_i) \wedge (v^u = v_j) \wedge (\forall i, 1 \le i < u, (v^i, v^{i+1}) \in E)\}$$

In order to facilitate the explanation of the algorithm we suppose that G has a source vertex and a sink vertex denoted v_1 and v_n, respectively. By definition:

$$\forall v_i \in V, \quad \left\{ \begin{array}{l} \quad \mathcal{P}(v_i, v_1) = \emptyset \\ \wedge \\ \quad \mathcal{P}(v_n, v_i) = \emptyset \\ \wedge \\ \quad v_i \neq v_1 \iff \mathcal{P}(v_1, v_i) \neq \emptyset \\ \wedge \\ \quad v_i \neq v_n \iff \mathcal{P}(v_i, v_n) \neq \emptyset \end{array} \right.$$

Let Σ be a finite set of l labels: $\Sigma = \{a_1, a_2, \cdots, a_l\}$. The set of all strings over the alphabet Σ is denoted by Σ^*. λ is a labeling function that maps edges of G to sets of labels. If $(v_i, v_j) \in E$, $\lambda(v_i, v_j)$ denotes the set of labels associated with the edge (v_i, v_j). We assume the "empty" label ϵ is implicitly associated with every edge for which the labeling function defines no label. G^λ denotes the dag G with labeling λ.[1]

For each pair of vertices (v_i, v_j) of the graph, $\mathcal{L}(v_i, v_j)$ represents the set of words defined by considering all possible labeling of all paths starting at v_i and ending at v_j. More formally:

$$\mathcal{L}(v_i, v_j) = \{a_1 a_2 \cdots a_u \in \Sigma^* \mid \exists\, (v^1, v^2, \cdots, v^u, v^{u+1}) \in \mathcal{P}(v_i, v_j),\ \forall i,\ 1 \le i \le u,\ a_i \in \lambda(v^i, v^{i+1})\}$$

Let R^k be the name of a property defined as a set of words (language $\mathcal{L}(R^k)$) on the alphabet Σ.

2.2 The Primitive Pattern SOME

Let v_i and v_j be two vertices of G and R^k be a property. The pair (v_i, v_j) satisfies the pattern SOME(R^k) if there is a path from v_i to v_j such that at least one of the labelings of the path is a word of $\mathcal{L}(R^k)$. More formally:

$$((v_i, v_j) \models SOME(R^k)) \equiv \mathcal{L}(v_i, v_j) \cap \mathcal{L}(R^k) \neq \emptyset$$

[1] We assign labels to each arc of the graph rather than to each vertex. When the goal is to detect properties of distributed computations, each vertex represents a local state and, in that case, the labels of all the predicates satisfied by a local state v are assigned to all incoming arcs of vertex v.

2.3 The Primitive Pattern ALL

The pair (v_i, v_j) satisfies the pattern ALL(R^k) if all labelings of all paths from v_i to v_j belong to $\mathcal{L}(R^k)$. More formally:[2]

$$((v_i, v_j) \models ALL(R^k)) \equiv (\mathcal{P}(v_i, v_j) \neq \emptyset) \wedge (\mathcal{L}(v_i, v_j) \subseteq \mathcal{L}(R^k))$$

2.4 The General Pattern

This pattern is an alternating sequence of primitive patterns SOME and ALL. An ALL pattern reminds of a diamond and two consecutive diamonds are connected by a link, *i.e.*, a pattern SOME, the whole pattern forming a necklace of diamonds. The alternating sequence is denoted $R^1\ R^2\ R^3\ \cdots\ R^m$.

A sequence of $m+1$ vertices $(v^1, v^2, v^3, v^4, \cdots, v^m, v^{m+1})$ is a solution of the general pattern (*i.e.*, $(v^1, v^2, v^3, v^4, \cdots, v^m, v^{m+1}) \models R^1\ R^2\ R^3\ \cdots\ R^m$), if these vertices satisfy the following constraints:

- $(v^1 = v_1) \wedge (v^{m+1} = v_n)$
- $\forall\ k,\ 1 \leq 2k+1 \leq m,\ (v^{2k+1}, v^{2k+2}) \models SOME(R^{2k+1})$
- $\forall\ k,\ 2 \leq 2k \leq m,\ (v^{2k}, v^{2k+1}) \models ALL(R^{2k})$

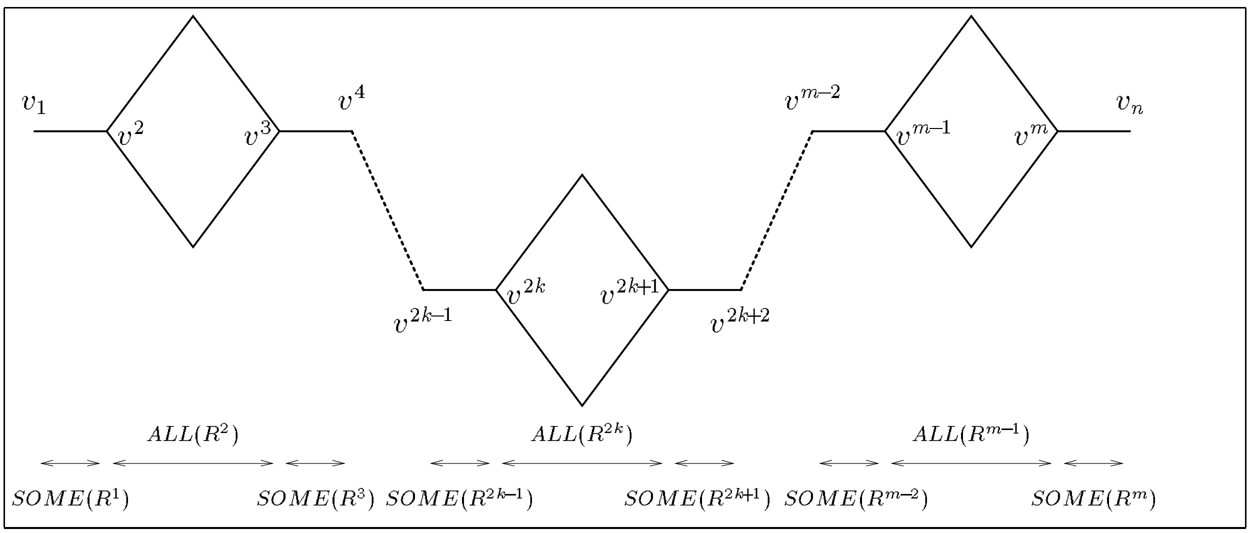

Figure 1: A diamond necklace pattern

[2] This condition can also be expressed as follow: $(\mathcal{P}(v_i, v_j) \neq \emptyset) \wedge (\mathcal{L}(v_i, v_j) \cap (\Sigma^* - \mathcal{L}(R^k)) = \emptyset)$

Figure 1 gives a pictorial representation of a diamond necklace. A line from v^{2k-1} to v^{2k} represents a path satisfying a pattern SOME, and a diamond-shaped plane figure represents a diamond starting at v^{2k} and terminating at v^{2k+1}.

The following prefix notation will be used in what follows. Let us consider the subgraph of G whose v_1 and v^k are the source and the sink vertices (*i.e.*, all maximal paths of this subgraph start at v_1 and terminate at v^k). If $(v^1, v^2, \cdots, v^k) \models R^1\ R^2\ \cdots\ R^{k-1}$ then we say the sequence $(v^1, v^2, \cdots, v^k)$ is a solution of the prefix $R^1\ R^2\ \cdots\ R^{k-1}$ of the pattern.

3 A Detection Algorithm

3.1 Regular Properties

For the sake of simplicity, in what follows, we consider only properties R^k whose corresponding languages $\mathcal{L}(R^k)$ are regular [8]. Moreover, these properties are sufficient to solve practical problems encountered in distributed debugging.

Let $\mathcal{A}^k$ $(1 \leq k \leq m)$ be the finite automaton recognizing $\mathcal{L}(R^k)$. Formally, an automaton is a tuple $\mathcal{A}^k = (Q^k, \Sigma, \delta^k, q_0^k, F^k)$, where Q^k is a finite set of states (q_x^k is one of these states), Σ is a finite alphabet (equal to the set of l labels associated with vertices of the graph), q_0^k is the initial state, F^k is the set of final states and δ^k its transition function mapping $Q^k \times \Sigma$ to 2^{Q^k}.

All automata $\mathcal{A}^{2k}$ $(2 \leq 2k \leq m)$ are supposed to be deterministic and complete; automata $\mathcal{A}^{2k+1}$ $(1 \leq 2k+1 \leq m)$ can be non-deterministic.

3.2 Visiting the Graph

The algorithm proposed in this paper visits the vertices of G, starting from v_1. When it visits a new vertex v, it computes information necessary to detect the property (*i.e.*, the diamond necklace). The traversal is done in the following manner: a vertex v is visited after all its predecessors (all vertices v_i such that $\mathcal{P}(v_i, v) \neq \emptyset$); *i.e.* the visit is done according to a topological sort strategy. [3]

Without such a visit requirement, we could envisage to detect occurrences of the general pattern with the whole labeled dag to our disposal. In such a case, a naive solution would consists in examining all the possible sequence of $m-1$ vertices $\{v^2, \cdots, v^m\}$, candidates to be a solution. In the worth case, there are $\binom{n-2}{m-1}$ candidates sets. For each automaton R^k, the intersection of the language $\mathcal{L}(v^k, v^{k+1})$ with the language $\mathcal{L}(R^k)$ (respt. language $\Sigma^* - \mathcal{L}(R^k)$) must be non-empty (respt. empty). Classical techniques (product of automata [11]) can be applied to realize these tests. The time complexity of this approach $\mathrm{O}(n^m)$ will be

[3] This visit strategy is particularly interesting in the context of on-the-fly detection of properties of distributed executions. Actually, in that case, the partially ordered set of local states is generated on-the-fly by the execution itself: due to this visit strategy of the vertices (local states) of the graph, the detection algorithm can be easily superimposed [2] on such an execution.

compared with the time complexity of the algorithm presented in this paper which is also polynomial.[4]

3.3 Detecting $(v_1, v) \models SOME(R^k)$

To facilitate the understanding of the general algorithm (Section 3.5), we first present simpler algorithms which constitute building blocks of the general one.

A variable $state(v, k)$ is associated with each vertex v; its definition is the following one:

$$states(v,k) = \{ q_x^k \mid \exists w \in \mathcal{L}(v_1, v) \text{ such that } q_x^k \in \delta^k(q_0^k, w) \}$$

By visiting the vertices of G, starting from v_1 and using the traversal strategy explained above, the value of $states(v, k)$ is computed as indicated by Figure 2 (Initially: $states(v_1, k) = \{q_0^k\}$).

It follows that answering to the question "$(v_1, v) \models SOME(R^k)$" is equivalent to test the following predicate:

$$\exists q_x^k \in states(v,k) : q_x^k \in F^k$$

> **begin**
> $states(v,k) := \emptyset$;
>   **foreach** v_p such that $((v_p, v) \in E)$:
>   **foreach** $q_x^k \in states(v_p, k)$:
>   **foreach** $a \in \lambda(v_p, v)$:
> $states(v,k) := states(v,k) \cup \{\delta^k(q_x^k, a)\}$;
>   **endfor**
>   **endfor**
>   **endfor**
> **end**

Figure 2: Visit of a vertex $v \neq v_1$

3.4 Detecting $(v_1, v) \models ALL(R^k)$

The previous discussion about the computation of $states(v, k)$ is still valid. Only the predicate to decide "$(v_1, v) \models ALL(R^k)$" has to be defined. As indicated we constraint all the automata recognizing a language whose property appears in an ALL pattern to be deterministic. With such a constraint the decision test becomes:

$$\forall q_x^k \in states(v,k) : q_x^k \in F^k$$

[4] Therefore, this algorithm can also be used to detect efficiently diamond necklaces in dags such as the lattice of global states of distributed computations which may be constructed at a designated process [3].

3.5 Detecting Diamond Necklace Patterns

Determine the set of solutions $(v^1, v^2, \cdots, v^m, v^{m+1})$ requires to analyze all the words associated with all possible labelings of all the paths. This demands to keep information related to word analyses and to launch the next automaton each time a prefix of the pattern has been recognized. As indicated previously, it is supposed that vertices of G are visited according to the strategy explained in Section 3.2, v_1 being the first vertex visited.

Launching automata

When v is visited, if $(v^1, v^2, \cdots, v^{2k}, v)$ is solution of the prefix $R^1\ R^2\ \cdots\ R^{2k}$ then a copy of the automaton $\mathcal{A}^{2k+1}$ has to be launched in order to start the search for a vertex v' such that $(v, v') \models SOME(R^{2k+1})$.

Similarly, if $(v^1, v^2, \cdots, v^{2k+1}, v)$ is solution of the prefix $R^1\ R^2\ \cdots\ R^{2k+1}$ then a copy of $\mathcal{A}^{2k+2}$ has to be launched to search for a vertex v' such that $(v, v') \models ALL(R^{2k+2})$.

Data structure to record past word analyses

As an automaton $\mathcal{A}^k$ can be launched from any vertex, the data structure $states(v, k)$ has to be enriched to record the vertices in which copies of $\mathcal{A}^k$ have been started. An array of m variables $start\_states$ is associated with each vertex v; $start\_states(v, k)$ is a set of pairs (v_i, q_x^k) whose first component is a vertex of G and second component is a state of Q^k. Its semantics is the following one:

$$(v_i, q_x^k) \in start\_states(v, k) \iff \begin{cases} \text{a copy of } \mathcal{A}^k \text{ has been started in } v_i \\ \wedge \\ \exists\ w \in \mathcal{L}(v_i, v) \text{ such that } q_x^k \in \delta^k(q_0^k, w) \end{cases}$$

These data structures keep a record of all the word analyses done in the past of the vertex v that is currently visited.

The algorithm

The procedure described in Figure 3 specify the set of actions executed when a vertex v is visited. Two tasks have to be done.

1. All the copies of all the automata previously launched, in the past of v, have to progress in word recognition (lines 1–5).

   If $(v_p, v) \in E$ and $start\_states(v_p, k) \neq \emptyset$ then copies of $\mathcal{A}^k$ have been previously launched. From $(v_i, q_x^k) \in start\_states(v_p, k)$, we conclude first that a copy of $\mathcal{A}^k$ has been launched in v_i and second that there is at least one word $w \in \mathcal{L}(v_i, v_p)$ such that $q_x^k \in \delta^k(q_0^k, w)$. So the algorithm makes this copy of $\mathcal{A}^k$ progress according to labelings of the edge (v_p, v).

```
Procedure Visit (v : vertex);
begin
/* Recognition */
   if (v = v_1) then
      start_states(v_1, 1)  :=  { (v_1, q_0^1) };
      for  k := 2  to  m :
           start_states(v_1, k)  :=  ∅;
      endfor
   else
      /* v is not the least vertex of G */
      /* All predecessor of v have been already visited */
      for  k := 1  to  m :
(1)        start_states(v, k)  :=  ∅;
(2)        foreach  v_p  such that  ((v_p, v) ∈ E) :
(3)             foreach  (v_i, q_x^k) ∈ start_states(v_p, k) :
(4)                  foreach  a ∈ λ(v_p, v) :
(5)                       start_states(v, k)  :=  start_states(v, k) ∪ {(v_i, δ^k(q_x^k, a))};
                     endfor
                endfor
           endfor
      endfor
   endif
/* Launching a copy of the automaton A^{k+1} */
   for  k := 1  to  m − 1 :
        if (k mod 2 = 0) then
(6)        if (∃ v_i such that: (∀ (v_i, q_x^k) ∈ start_states(v, k) : q_x^k ∈ F^k)) then
              /* A pattern ALL has been recognized: (v_i, v) ⊨ ALL(R^k) */
(7)           start_states(v, k + 1)  :=  start_states(v, k + 1) ∪ {(v, q_0^{k+1})};
           endif
        else
(8)        if (∃ (v_i, q_x^k) ∈ start_states(v, k) such that: q_x^k ∈ F^k) then
              /* A pattern SOME has been recognized: (v_i, v) ⊨ SOME(R^k) */
(9)           start_states(v, k + 1)  :=  start_states(v, k + 1) ∪ {(v, q_0^{k+1})};
           endif
        endif
   endfor
   if (v = v_n) then
      output_solutions(m + 1, v_n);
   endif
(10) /* If interested by only one solution, call the procedure reduction (See Section 3.6) */
end
```

Figure 3: General algorithm

It is important to note that all the copies of the automata previously launched continue their analysis till the vertex v_n is visited. This is necessary as we do not known in advance if a partial solution will give rise to a solution.

2. When a prefix of the general pattern has been recognized, a new copy of the next automaton has to be launched (lines 6–9).

   If there is an automaton $\mathcal{A}^k$ such that a copy of $\mathcal{A}^k$ has been launched in some v_i and $(v_i, v) \models ALL(R^k)$ (when k is an even number), or $(v_i, v) \models SOME(R^k)$ (when k is an odd number), then a copy of the automaton $\mathcal{A}^{k+1}$ has to be launched from v.

The set of all the solutions is obtained with the procedure described in Figure 4 by calling $output\_solutions(m+1, v_n)$. If the whole pattern has not been recognized at the end of the computation, it is also possible to find out the longest prefix of the diamond necklace for which partial solution exist.

It is important to note that actions executed when visiting a vertex v only depend on values of variables $start\_states$ of v's immediate predecessors (This allows to adapt the algorithm to an on the fly detection when used in debugging distributed applications [5]).

It is also important to note that if we are only interested in the simpler problem which consists in deciding if an a priori given set of $m+1$ vertices is a solution, then the data structures and the algorithm can be greatly simplified.

3.6 Deciding if there exists a solution

The previous algorithm finds all the solutions, *i.e.*, all sets of vertices $(v^1, v^2, \cdots, v^m, v^{m+1})$ satisfying the pattern in G^λ. If we are only interested in knowing if there is a solution, the contents of variables $start\_states(v, k)$ can be reduced in the following way. The procedure *reduction* (line 10) decreases the size of variables $start\_states$. This procedure does the following actions. At the end of the visit of vertex v, a pair (v_i, q_x^k) belonging to the set $start\_states(v, k)$ is suppressed if one of the two following predicates is true.

1. **Predicate P1**:
   k is an odd number and there exists another pair (v_j, q_x^k) in $start\_states(v, k)$.

   From $(v_i, q_x^k) \in start\_states(v, k)$, we deduce that there exists at least one word $w1$ such that $w1 \in \mathcal{L}(v_i, v)$ and $q_x^k \in \delta^k(q_0^k, w1)$. Similarly, we conclude that there exists also a word $w2$ such that $w2 \in \mathcal{L}(v_j, v)$ and $q_x^k \in \delta^k(q_0^k, w2)$. Therefore, if there exists a word $w3$ such that $F^k \cap \delta^k(q_x^k, w3) \neq \emptyset$ then we can conclude that both words $w1.w3$ and $w2.w3$ belong to $\mathcal{L}(R^k)$. So, if we are not interested in computing

[5] [5] presents such an algorithm which detects on the fly the simple primitive pattern: $(v_1, v_n) \models SOME(R_1)$. In that case, there is only one pair of vertices that can be a solution; moreover this pair is defined a priori. For this very simple pattern, variables needed for the detection reduce to a boolean array whose size is equal to the number of states of the corresponding automaton. Each process of the distributed application which is debugged, manages a copy of this array and each application message piggybacks the value of the sender process array. In the general case, every process has to manage an array $start\_states[1..m]$ and messages have to carry the value of this array.

```
Solution: array[1..m + 1] of vertex;

Procedure output_solutions (k : integer;  v : vertex);
begin
    Solution[k] := v;
    if (k = 1) then
       print(Solution);
    else
       if (k mod 2 = 0) then
          /* Continue with all v_i such that (v_i, v) |= ALL(R^k) */
          foreach v_i such that: (∀ (v_i, q_x^k) ∈ start_states(v, k) : q_x^k ∈ F^k))
              output_solutions(k − 1, v_i);
          endfor
       else
          /* Continue with all v_i such that (v_i, v) |= SOME(R^k) */
          foreach v_i such that: (∃ (v_i, q_x^k) ∈ start_states(v, k) : q_x^k ∈ F^k))
              output_solutions(k − 1, v_i);
          endfor
       endif
    endif
end
```

Figure 4: Enumerating the set of solutions

all solutions, it is sufficient to indicate that q_x^k is a state in which a copy of automaton $\mathcal{A}^k$ arrived after the vertex v has been visited.

2. **Predicate P2**:
   k is an even number, $\mathcal{L}(R^k)$ is a suffix language and there exists another pair (v_j, q_y^k) in $start\_states(v, k)$ such that there is path from v_i to v_j (*i.e.* $\mathcal{P}(v_i, v_j) \neq \emptyset$).

   For each word $w3 \in \mathcal{L}(v_j, v)$, there exists at least one word $w1 \in \mathcal{L}(v_i, v)$ such that $w1 = w2.w3$.
   If $\mathcal{L}(R^k)$ is a suffix language, $\forall w \in \Sigma^*, w1.w \in \mathcal{L}(R^k) \Rightarrow w3.w \in \mathcal{L}(R^k)$. Therefore, if v' is a vertex such that $\mathcal{P}(v, v') \neq \emptyset$ then $(v_i, v') \models ALL(R^k) \Rightarrow (v_j, v') \models ALL(R^k)$. It follows that only (v_j, q_y^k) has to be memorized if we are not interested in computing all solutions [6].

[6] In [10], a particular simple kind of diamond necklaces called atomic sequences is defined. The language associated to each diamond contains all the words built with all the symbols of an alphabet except those containing a particular forbidden symbol. Such a language is a suffix language. Consequently, the second reduction rule explained above can by applied in this particular case.

3.7 Complexity

During an on-the-fly detection and when one try to find all the solutions, the storage complexity of this algorithm is $O(m.n^2.r)$ where m is the number of automata (*i.e.*, the length of the diamond necklace), n is the number of vertices in the graph and r is the maximal number of states of an automaton (*i.e.* $r = max\{r^k \mid 1 \leq k \leq m\}$ with $r^k = \mid Q^k \mid$). Note that automaton $\mathcal{A}^1$ is launched only once when vertex v_1 is visited. Therefore, the size of the structure $start\_states(v,1)$ is bounded by r^1 whereas the size of $start\_states(v,k)$ is bounded by $(p_v.r^k + 1)$ if $2 \leq k \leq m$ (where p_v is the number of immediate predecessors of vertex v).

Let $t_x^k(a) = \mid \delta^k(q_x^k, a) \mid$ and let $t^k = max\{max\{t_x^k(a) \mid a \in \Sigma\} \mid q_x^k \in Q^k\}$. Note that $t^k = 1$ if automaton $\mathcal{A}^k$ is deterministic. Let $t = max\{t^k | 1 \leq k \leq m\}$. Assume that elements (v_i, q_x^k) of $start\_states(v,k)$ are sorted according to the first component. The time complexity of the general algorithm is

$$O(m.n^3.r.t.l)$$

where l is the number of labels in Σ. If $k \geq 2$, computation of the set $start\_states(v,k)$ requires less than $p_v^2.r^k.t^k.l$ insertions of elements.

The time complexity of this algorithm is cubic whereas the complexity of the naive approach described in Section 3.2 is $O(n^m)$. Note that, when $m = 1$, the naive approach consists in determining the product of two automata.

When the two reduction rules are applied, the size of the structure $start\_states(v,k)$ is bounded by $(s.r^k)$ where s is the width of the partial order (*i.e.*, the size of the largest antichain). In the dag corresponding to the execution of a distributed application, the value of s is bounded by the number of processes observed during the debugging activity. In this case, the storage and time complexities of the algorithm also decrease .

4 Conclusion

The problem tackled in this paper originated from the debugging of distributed applications. Execution of such an application can be modeled as a partially ordered set of process states. The debugging of control flows (sequences of process states) of these executions is based on the satisfaction of predicates by process states. A process state that satisfies a predicate inherits its label. It follows that, in this context, a distributed execution is a labeled directed acyclic graph. Debug or determine if control flows of a distributed execution satisfies some property amounts to test if the labeled acyclic graph includes some pattern defined on predicate labels.

This paper first introduced a general pattern (called *diamond necklace*) which includes classical patterns encountered in distributed debugging. Then an algorithm detecting such patterns in a labeled acyclic graph has been presented. To be easily adapted to an on-the-fly detection of the pattern in distributed executions, the algorithm has been based on a visit of the nodes of the graph according to a topological sort. Its time complexity is polynomial.

Acknowledgments

The authors would like to thank Didier Caucal and Jean-Xavier Rampon whose comments greatly improved both the content and the presentation of the paper.

References

[1] Ö. Babaoğlu, E. Fromentin, and M. Raynal: "A Unified Framework for the Specification and Run-time Detection of Dynamic Properties in Distributed Computations", *The Journal of Systems and Software*, special issue on Soft. Eng. for Distributed Computing, To appear 1996.

[2] L. Bougé, and N. Francez: "A compositional approach to superimposition", In *Proc. of the* 15^{th} *ACM SIGACT-SIGPLAN Symposium on Principle of Programming Languages*, pp. 240–249, San Diego, California, January 1988.

[3] R. Cooper, and K. Marzullo: "Consistent Detection of Global Predicates", In *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 163–173, Santa Cruz, California, May 1991.

[4] E. Fromentin, C. Jard, G. Jourdan, and M. Raynal: "On-the-fly Analysis of Distributed Computations", *Information Processing Letters 54*, pp. 267–274, 1995.

[5] E. Fromentin, M. Raynal, V.K. Garg, and A.I. Tomlinson: "On the fly testing of regular patterns in distributed computations", In *Proc. of the the* 23^{rd} *Int. Conf. on Parallel Processing*, pp. 73–76, St. Charles, IL, August 1994.

[6] V.K. Garg, A.I. Tomlinson, E. Fromentin, and M. Raynal: "Expressing and Detecting General Control Flow Properties of Distributed Computations", In *Proc. of the 7th IEEE Symposium on Parallel and Distributed Processing*, pp. 432–438, San-Antonio (USA), October 1995.

[7] V.K. Garg and B. Waldecker: "Detection of Unstable Predicates in Distributed Programs", In *Proc. of the 12*th *Int. Conf. on Foundations of Software Technology and Theoretical Computer Science*, Springer Verlag, LNCS 652, pp. 253–264, New Delhi, India, December 1992.

[8] M.A. Harrison: "Introduction to Formal Language Theory", Addison-Wesley series in computer science, 1978.

[9] M. Hurfin, N. Plouzeau, and M. Raynal: "A Debugging Tool for Estelle Distributed Programs", *Journal of Computer Communications*, 28(5), pp. 328–333, May 1993.

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

[11] J.E. Hopcroft and J.D. Ullman: "Introduction to Automata Theory, Languages, and Computation", Addison-Weslay Publishing Company, 418 pages, 1979.

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

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, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
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)
ISSN 0249-6399