Appl. Math. and Comp. Sci., 1997, vol.7, No.1, 205–216

ON-THE-FLY DETECTION OF A CLASS OF WORD-BASED PATTERNS IN LABELLED DAGs†

MICHEL HURFIN*, MICHEL RAYNAL*

The problem tackled in this paper originates from the debugging of distributed applications. Execution of such an application can be modelled as a partially-ordered set of process states. The debugging of control flows (sequences of process states) of these executions is based on satisffying the predicates by process states. A process state that satisfies a predicate inherits its label. In this context, it follows that a distributed execution is a labelled directed acyclic graph (DAG for brevity). To debug or to determine if control flows of a distributed execution satisfy some property amounts to testing if the labelled DAG includes some pattern defined on predicate labels.

This paper first introduces a general pattern (called *the diamond necklace*) which includes classical patterns encountered in distributed debugging. Then an efficient polynomial-time algorithm detecting such patterns in a labelled 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.

1. Introduction

This paper presents an algorithm to detect a sophisticated pattern (called *the diamond necklace*) in a labelled directed acyclic graph. The problem solved by this algorithm originated from the detection of properties of distributed computations in our current efforts to design and implement a facility for debugging distributed programs (Hurfin *et al.*, 1993a). These programs are composed of a finite set of sequential processes cooperating only by means of message passing. The concurrent execution of all the processes on a network of processors is called a distributed computation. The computation is asynchronous: each process evolves at its own speed and messages are exchanged through communication channels whose transmission delays are finite but arbitrary. During a computation, each process executes a sequence of actions. At a given time-moment, the local state of a process is defined by the values of the local variables managed by this process. From an initial state, a process produces a sequence of process states according to its program text. In the context

† A first version of this paper, entitled "Detecting diamond necklaces in labelled DAGs (A problem from distributed debugging)", appeared in Proc. 22nd Int. Workshop *Graph-Theoretic Concepts in Computer Science* published in the LNCS series of Springer-Verlag (June 12–14, 1996, Como, Italy).

* IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France, e-mail: {hurfin,raynal}@irisa.fr.

of debugging distributed programs, a distributed execution is usually modelled by a partially-ordered set of process states (Garg and Waldecker, 1992). Due to the asynchronous nature of a distributed computation, programmers refer to a classical causal precedence relation between states rather than to a non-available global clock. Informally, a process state s_1 precedes s_2 if they have both 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 in terms of process states (Lamport, 1978). 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 detect on-the-fly some properties of control flows in distributed computations (Fromentin *et al.*, 1994; 1995; Hurfin *et al.*, 1993b). Basically, a property is defined as a language on an alphabet of predicates (a predicate being a Boolean expression in which variables of a single process appear); 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 (Babaoğlu *et al.*, 1996). These properties are fundamentally sequential in the sense that 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 as follows: "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 (Garg *et al.*, 1995).

Here we abstract from distributed executions and consider labelled directed acyclic graphs (DAGs), in which vertices represent local states and edges represent dependent relation over states. We first define (Section 2) a general type of patterns (diamond necklace) for labelled 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 visit the nodes of the DAG according to a topological sort strategy.

In this way, the paper solves a new problem (to our knowledge), i.e. deciding if a labelled 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. We will use the symbols v, v', v_i, and v^i to denote elements of V. Let v_i and v_j be two vertices of V.

Then $\mathcal{P}(v_i, v_j)$ is the set of all the paths in G from v_i to v_j:

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

To facilitate the explanation of the algorithm, we suppose that G has a source vertex and a sink vertex denoted by v_1 and v_n, respectively. By definition,

$$\forall v_i \in V, \quad \begin{cases} \mathcal{P}(v_i, v_1) = \emptyset \ \wedge \\ \mathcal{P}(v_n, v_i) = \emptyset \ \wedge \\ v_i \neq v_1 \Longleftrightarrow \mathcal{P}(v_1, v_i) \neq \emptyset \ \wedge \\ v_i \neq v_n \Longleftrightarrow \mathcal{P}(v_i, v_n) \neq \emptyset \end{cases}$$

Let Σ be a finite set of l labels: $\Sigma = \{a_1, a_2, \ldots, a_l\}$. The set of all strings over the alphabet Σ is denoted by Σ^*. Moreover, λ is a labelling function that maps edges of G to sets of labels. If $(v_i, v_j) \in E$, then $\lambda(v_i, v_j)$ denotes the set of labels associated with the edge (v_i, v_j). We assume that the "empty" label ϵ is implicitly associated with every edge for which the labelling function defines no label. G^λ denotes the DAG G with labelling 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 labellings of all paths starting at v_i and ending at v_j. More formally,

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

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. 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 labellings of the path is a word of $\mathcal{L}(R^k)$. More formally,

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

[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. Primitive Pattern ALL

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

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

2.4. General Pattern

This pattern is an alternating sequence of primitive patterns SOME and ALL. An ALL pattern resembles 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 by $R^1\ R^2\ R^3\ \cdots\ R^m$.

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

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

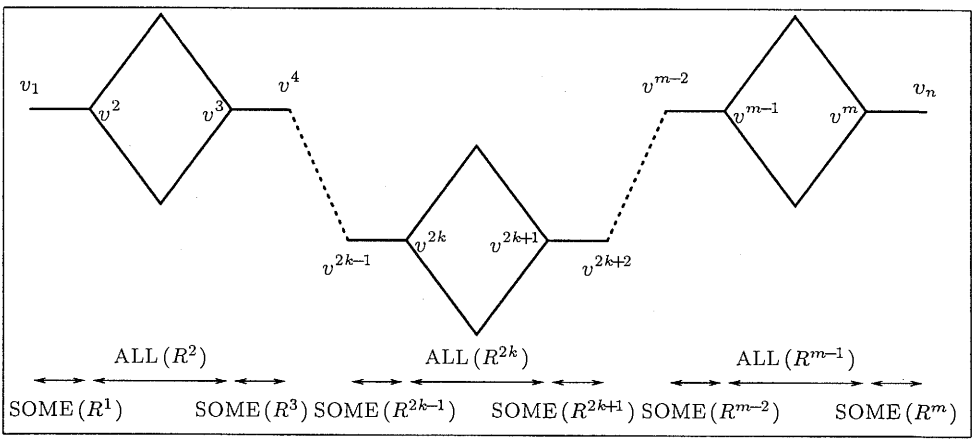

Fig. 1. A diamond necklace pattern.

Figure 1 gives a pictorial representation of the 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 respectively the source and sink vertices (i.e. all

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

maximal paths of this subgraph start at v_1 and terminate at v^k). If $(v^1, v^2, \ldots, v^k) \models R^1\ R^2\ \cdots\ R^{k-1}$, then we say that the sequence $(v^1, v^2, \ldots, v^k)$ is a solution of the prefix $R^1\ R^2\ \cdots\ R^{k-1}$ of the pattern.

3. 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 (Harrison, 1978). 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 quintuple $\mathcal{A}^k = (Q^k, \Sigma, \delta^k, q_0^k, F^k)$, 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 edges of the graph), q_0^k stands for the initial state, F^k is the set of final states, and δ^k denotes its transition function mapping $Q^k \times \Sigma$ to 2^{Q^k}.

We extend δ to map $Q^k \times \Sigma^*$ to 2^{Q^k} by reflecting sequences of inputs as follows: Let w be a sequence of inputs, a_u be a basic predicate, and q be a state. Then

(1) $\delta(q, \epsilon) = \{q\}$ and

(2) $\delta(q, w \cdot a_u) = \left\{ q_i \mid q_i \in \delta(q_j, a_u) \text{ where } q_j \in \delta(q, w) \right\}$

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 labelled DAG at our disposal. In such a case, a naive solution would consists in examining all possible sequences of $m-1$ vertices $\{v^2, \ldots, v^m\}$, which are candidates to be a solution. In the case considered, there are $\binom{n-2}{m-1}$ candidate sets. For each automaton R^k, the intersection of the language $\mathcal{L}(v^k, v^{k+1})$ with the language $\mathcal{L}(R^k)$ (resp. the language $\Sigma^* - \mathcal{L}(R^k)$) must be non-empty (resp. empty). Classical techniques (a product of automata (Hopcroft and Ullman, 1979)) can be applied to realize these tests. The time complexity of

[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 (Bougé and Francez, 1988) on such an execution.

this approach $O(n^m)$ will be compared with the time complexity of the algorithm presented in this paper which is also polynomial.[4]

3.3. Detecting $(v_1, v) \models \mathbf{SOME}(R^k)$

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 $states(v,k)$ is associated with each vertex v; its definition is as follows:

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

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 in Fig. 2 (initially, $states(v_1,k) = \{q_0^k\}$).

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

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

```
begin
    states(v,k) := ∅;
    foreach v_p such that ((v_p, v) ∈ E) :
        foreach q_x^k ∈ states(v_p, k) :
            foreach a ∈ λ(v_p, v) :
                states(v,k) := states(v,k) ∪ {δ^k(q_x^k, a)};
            endfor
        endfor
    endfor
end
```

Fig. 2. Visit of a vertex $v \neq v_1$.

3.4. Detecting $(v_1, v) \models \mathbf{ALL}(R^k)$

The previous discussion of the computation of $states(v,k)$ is still valid. Only the predicate to decide "$(v_1, v) \models \mathrm{ALL}(R^k)$" 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 (Cooper and Marzullo, 1991).

3.5. Detecting Diamond Necklace Patterns

To determine the set of solutions $(v^1, v^2, \ldots, v^m, v^{m+1})$ requires an analysis of all the words associated with all possible labellings of all the paths. This demands keeping information related to word analyses and launching 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 $(v^1, v^2, \ldots, v^{2k}, v)$ is a 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 $(v, v') \models \text{SOME}(R^{2k+1})$.

Similarly, if $(v^1, v^2, \ldots, v^{2k+1}, v)$ is a 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 $(v, v') \models \text{ALL}(R^{2k+2})$.

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 at 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 the second component is a state of Q^k. Its semantics is as follows:

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

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

Algorithm

The procedure described in Fig. 3 specifies 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 labellings of the edge (v_p, v).

   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 know in advance if a partial solution will give rise to a solution.

```
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 predecessors of v have already been 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 only by one solution, call the procedure reduction
        (See Section 3.6) */
end
```

Fig. 3. General algorithm.

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 $(v_i, v) \models \mathrm{ALL}(R^k)$ (when k is even), or $(v_i, v) \models \mathrm{SOME}(R^k)$ (when k is odd), 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 Fig. 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 a partial solution exists.

It is important to note that actions executed when visiting a vertex v depend only on values of variables *start_states* of v's immediate predecessors (this allows us to adapt the algorithm to 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 $m+1$ vertices $(v^1, v^2, \ldots, v^m, v^{m+1})$ 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 performs 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 also exists a word $w2$ such that $w2 \in \mathcal{L}(v_j, v)$ and $q_x^k \in \delta^k(q_0^k, w2)$. Thus, if a word $w3$ is such that $F^k \cap \delta^k(q_x^k, w3) \neq \emptyset$, 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 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.

[5] The paper (Fromentin *et al.*, 1994) presents such an algorithm. It detects on-the-fly the simple primitive pattern: $(v_1, v_n) \models \mathrm{SOME}(R_1)$. 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
```

Fig. 4. Enumerating the set of solutions.

2. **Predicate P2**: k is even, $\mathcal{L}(R^k)$ is a suffix language and there exists another pair (v_j, q_y^k) such that there is a 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 $(v_i, v') \models \text{ALL}(R^k) \Rightarrow (v_j, v') \models \text{ALL}(R^k)$. It follows that only (v_j, q_y^k) has to be memorized if we are not interested in computing all solutions.[6]

3.7. Complexity

During an on-the-fly detection and when one tries to find all solutions, the storage complexity of this algorithm is $O(m \cdot n^2 \cdot r)$, 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 denotes the maximal number of states of an automaton (i.e. $r = \max\{r^k \mid 1 \leq k \leq m\}$ with $r^k = |Q^k|$). Note that the automaton $\mathcal{A}^1$ is launched only once, when vertex

[6] In (Hurfin *et al.*, 1993b), a particularly simple kind of diamond necklaces called atomic sequences is defined. The language associated with each diamond contains all the words built of 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.

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 $(p_v \cdot r^k + 1)$ if $2 \leq k \leq m$ (where p_v is the number of immediate predecessors of vertex v).

Let $t_x^k(a) = |\delta^k(q_x^k, a)|$ 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 the 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 with respect to the first component. The time complexity of the general algorithm is $O(m \cdot n^3 \cdot r \cdot t \cdot l)$, where l is the number of labels in Σ. If $k \geq 2$, computation of the set $start\_states(v,k)$ requires less than $p_v^2 \cdot r^k \cdot t^k \cdot l$ 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 (described in Section 3.6) are applied, the size of the structure $start\_states(v,k)$ is bounded by $s \cdot r^k$, 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 modelled as a partially-ordered set of process states. The debugging of control flows (sequences of process states) of these executions is based on satisfying the 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 labelled directed acyclic graph. To debug or to determine if control flows of a distributed execution satisfy some property amounts to testing if the labelled acyclic graph includes some pattern defined on predicate labels.

This paper first introduced a general pattern (called *the diamond necklace*) which includes classical patterns encountered in distributed debugging. Then an algorithm detecting such patterns in a labelled acyclic graph has been presented. To be easily adapted to an on-the-fly detection of the pattern in distributed executions, the algorithm is 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 presentation of the paper.

References

Babaoğlu Ö., Fromentin E. and Raynal M. (1996): *A unified framework for the specification and run-time detection of dynamic properties in distributed computations.* — Systems and Software, Vol.33, No.3, pp.287–298.

Bougé L. and Francez N. (1988): *A compositional approach to superimposition.* — Proc. 15th ACM SIGACT-SIGPLAN Symp. *Principle of Programming Languages*, San Diego, California, pp.240–249.

Cooper R. and Marzullo K. (1991): *Consistent detection of global predicates.* — Proc. ACM/ONR Workshop *Parallel and Distributed Debugging*, Santa Cruz, California, pp.163–173.

Fromentin E., Jard C., Jourdan G. and Raynal M. (1995): *On-the-fly analysis of distributed computations.* — Information Processing Letters, Vol.54, No.2, pp.267–274.

Fromentin E., Raynal M., Garg V.K. and Tomlinson A.I. (1994): *On-the-fly testing of regular patterns in distributed computations.* — Proc. 23rd Int. Conf. *Parallel Processing*, St. Charles, IL, pp.73–76.

Garg V.K., Tomlinson A.I., Fromentin E. and Raynal M. (1995): *Expressing and detecting general control flow properties of distributed computations.* — Proc. 7th IEEE Symp. *Parallel and Distributed Processing*, San-Antonio, USA, pp.432–438.

Garg V.K. and Waldecker B. (1992): *Detection of unstable predicates in distributed programs.* — Proc. 12th Int. Conf. *Foundations of Software Technology and Theoretical Computer Science*, New Delhi, India, Springer Verlag, LNCS 652, pp.253–264.

Harrison M.A. (1978): *Introduction to Formal Language Theory.* — New York: Addison-Wesley.

Hurfin M., Plouzeau N. and Raynal M. (1993a): *A debugging tool for estelle distributed programs.* — Comp. Communications, Vol.28, No.5, pp.328–333.

Hurfin M., Plouzeau N. and Raynal M. (1993b): *Detecting atomic sequences of predicates in distributed computations.* — Proc. ACM Workshop *Parallel and Distributed Debugging*, San Diego, pp.32–42.

Hopcroft J.E. and Ullman J.D. (1979): *Introduction to Automata Theory, Languages, and Computation.* — New York: Addison-Wesley.

Lamport L. (1978): *Time, clocks and the ordering of events in a distributed system.* — Communications of the ACM, Vol.21, No.7, pp.558–565.

Received: March 12, 1996