A Graph-Theoretic Characterization of AND-OR Deadlocks

Valmir C. Barbosa[†]
Mario R. F. Benevides[†‡]

[†]Programa de Engenharia de Sistemas e Computação, COPPE
[‡]Instituto de Matemática
Universidade Federal do Rio de Janeiro
Caixa Postal 68511
21945-970 Rio de Janeiro - RJ, Brazil

{valmir, mario}@cos.ufrj.br

Abstract

We introduce the b-knot as a structure whose existence in a wait-for graph is necessary and sufficient for the existence of a deadlock under the AND-OR model. Unlike the case of other, more restricted deadlock models, for the AND-OR model no such graph structure has heretofore been explicitly identified and characterized. We also show that a well-known asynchronous algorithm for distributed knot detection can be adapted to yield an asynchronous distributed algorithm on the wait-for graph for a node to detect whether it is in a b-knot.

Keywords: AND-OR deadlocks; distributed algorithms; operating systems.

1. Introduction

Let N denote a set of processes in a distributed computation. Informally, a deadlock is said to exist in this computation if a subset $S \subseteq N$ can be identified whose members are all blocked for the occurrence of some condition that can only be relieved by members of the same subset S. A useful abstraction to analyze deadlock situations is the *wait-for graph* $G = (N, E)$, where E is a set of directed edges. For $n_i, n_j \in N$, an edge exists in E directed away from n_i towards n_j if n_i is blocked for some condition that n_j may relieve. G changes dynamically as the computation progresses, so whenever we refer to G we mean the wait-for graph that corresponds to a "snapshot" of the distributed computation in the usual sense of a consistent global state [1, 4].

In general, a necessary condition for the existence of a deadlock in the distributed computation is the existence of a directed cycle in G. In order to discuss more specific necessary conditions and sufficient conditions, we must first introduce additional concepts and notation. For $n_i \in N$, let D_i denote the set of *descendants* of n_i in G (nodes that are reachable from n_i, including itself) and A_i denote the set of *ancestors* of n_i in G (nodes from which n_i is reachable, including itself). Let $O_i \subseteq D_i$ be the set of *immediate descendants* of n_i in G (descendants that are one edge away from n_i) and $I_i \subseteq A_i$ its set of *immediate ancestors* in G (ancestors that are one edge away from n_i). Nodes in $D_i \setminus A_i$ are called *subordinates* of n_i in G.[1]

A *deadlock model* for the distributed computation that underlies G is a collection $W_i^1, \ldots, W_i^{p_i}$ of subsets of O_i for all $n_i \in N$, such that:

[1] $\setminus$ denotes set difference.

- $W_i^1 \cup \cdots \cup W_i^{p_i} = O_i$;
- No two nonempty sets in $W_i^1, \ldots, W_i^{p_i}$ are such that one is a subset of the other;
- In order to exit its blocked state and proceed with its local computation, a node n_i for which $O_i \neq \emptyset$ must receive a signal from all nodes in at least one of the nonempty sets in $W_i^1, \ldots, W_i^{p_i}$.

At this level of generality, the deadlock model is known as the *AND-OR model*, reflecting the need for n_i to be signaled by all members of W_i^1 (if nonempty), or all members of W_i^2 (if nonempty), and so on. If at most one of $W_i^1, \ldots, W_i^{p_i}$ is nonempty for all $n_i \in N$, then the deadlock model is the *AND model*. Similarly, if all nonempty sets in $W_i^1, \ldots, W_i^{p_i}$ are singletons for all $n_i \in N$, then the deadlock model is known as the *OR model*.[2]

A sufficient condition for the existence of a deadlock in the AND model is the same as the general necessary condition mentioned earlier, that is, that a directed cycle exist in G. For the OR model, a necessary and sufficient condition is the existence of a *knot* in G. A knot is a subset $K \subseteq N$ with $|K| > 1$ having the property that, for all $n_i \in K$, $D_i = K$. For details on these conditions and related material, the reader is referred to [6, 10] and the references therein.

In spite of the existence of several approaches to the detection of deadlocks in distributed computations under the AND-OR model (the approaches in [3, 7, 9] are representative recent examples), no graph structure appears to have been identified that accounts for those deadlocks as a necessary and sufficient condition. In this paper, we describe such a structure in Section 2, and in Section 3 give an asynchronous distributed algorithm for a node to detect whether it is in such a structure in G. This algorithm employs the algorithm of [8] for knot detection as a first phase. Concluding remarks follow in Section 4.

2. B-knots

In this section we introduce the notion of a *b-knot* as a structure in G that can be used to characterize deadlocks under the AND-OR model. Unlike the case of directed cycles and knots, the definition of a b-knot requires G to be considered in explicit conjunction with the deadlock model. As will become apparent shortly, the "b" in b-knot is an allusion to the fact that, for the definition of this structure, edges directed away from a node n_i must be considered in "bundles" that relate closely to the sets $W_i^1, \ldots, W_i^{p_i}$.

The definition of a b-knot is based on the definition of a *b-subgraph* of G given the deadlock model. If $G' = (N', E')$ is a subgraph of G, then we say that it is a b-subgraph of G if every $n_i \in N'$ has at most as many immediate descendants in G' as there are nonempty sets in $W_i^1, \ldots, W_i^{p_i}$, and furthermore each of $W_i^1, \ldots, W_i^{p_i}$, if nonempty, includes at least one of those immediate descendants. Given the deadlock model, a subset $K \subseteq N$ is said to be a b-knot if K is a knot in some b-subgraph of G whose node set contains K. Henceforth, we let the deadlock model be implicitly assumed whenever a b-subgraph or a b-knot of G is mentioned.

If G' is a b-subgraph of G and n_i is a node of G', then, for $1 \leq k \leq p_i$, let $W_i'^k$ be the set of nodes contained in W_i^k that by definition appear in G' as immediate descendants of n_i (if W_i^k is nonempty), or $W_i'^k = \emptyset$ (otherwise). It follows that $W_i'^1, \ldots, W_i'^{p_i}$ is an OR model for G', so long as no two nonempty sets in $W_i'^1, \ldots, W_i'^{p_i}$ are identical.

Illustrations of this notion of a b-knot are given in Figures 1 and 2. In both figures, part (a) depicts G with the sets $W_i^1, \ldots, W_i^{p_i}$, whenever nonempty, shown as circular arcs around n_i joining

[2] Another deadlock model of interest is the so-called *k-out-of-n* model, which is more general than both the AND model and the OR model, while being generalized by the AND-OR model. A generalization of the k-out-of-n model is the *disjunctive k-out-of-n model*, this one equivalent to the AND-OR model (see [3], for example, and the references therein, for details).

groups of edges that lead to n_i's immediate descendants. In the case of Figure 1(a), for example, we have $W_6^1 = \{n_1, n_2\}$ and $W_6^2 = \{n_2, n_5\}$. The graph of Figure 1(a) has no b-knots, which by definition means that none of its b-subgraphs has knots. Two of these b-subgraphs are shown in Figures 1(b) and 1(c). In the graph of Figure 2(a), on the other hand, the set $\{n_1, n_2, n_6\}$ is a b-knot. This same set appears as a knot in the b-subgraph of Figure 2(b), while the b-subgraph of Figure 2(c) has no knots.

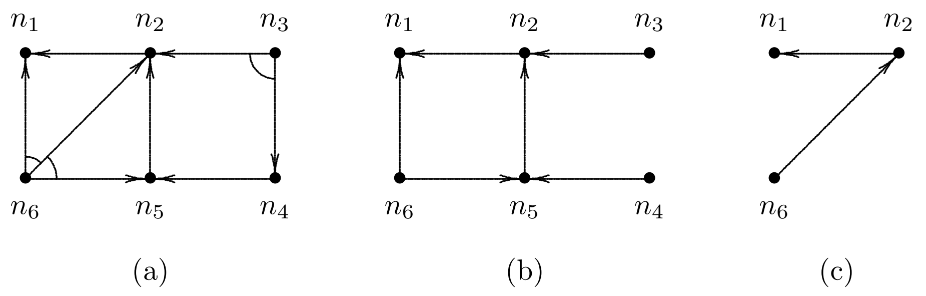

Figure 1. G with no b-knots and two b-subgraphs

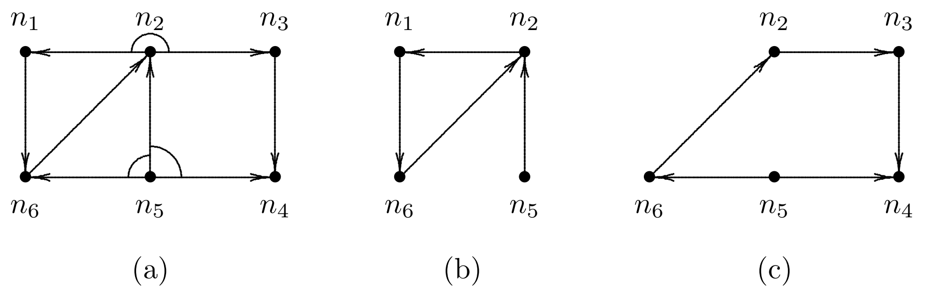

Figure 2. G with a b-knot and two b-subgraphs

Now suppose that node n_i is a sink in G (i.e., n_i has no descendants in G). It follows that n_i is not blocked in G, and it makes sense to consider the subgraph H of G that results from the signaling by n_i that all its immediate ancestors in G need no longer be blocked as far as it is concerned. In H, n_i is isolated (has no ancestors or descendants), and it may happen that one or more of its immediate ancestors in G, say node n_j, has now become a sink (this happens if, for some k such that $1 \le k \le p_j$, $W_j^k = \{n_i\}$). If in H no wait is "superfluous," in a sense that will become clear shortly, then we say that H is *signal-reduced from G by n_i*.

In order to define this signal-reduction from G by n_i precisely, let n_j be an immediate ancestor of n_i in G, and let $\bar{W}_j^1, \ldots, \bar{W}_j^{p_j}$ be the sets that represents n_j's wait condition in H (that is, these sets are part of the deadlock model for H). For $1 \le k \le p_j$, the following is how the set $\bar{W}_j^k$ is obtained in the process of signal-reduction from G by n_i. If there exists k' such that $1 \le k' \le p_j$ and $k' \ne k$, and furthermore $W_j^{k'} \setminus \{n_i\} \subseteq W_j^k \setminus \{n_i\}$, then $\bar{W}_j^k = \emptyset$. Otherwise, $\bar{W}_j^k = W_j^k \setminus \{n_i\}$. In addition to being in consonance with the definition of a deadlock model, this reflects the fact that, in H, it only makes sense for n_j to keep on waiting for signals from nodes in $W_j^k \setminus \{n_i\}$ if no other $W_j^{k'} \setminus \{n_i\}$ exists that already indicates such a wait.

Note that the absence of a b-knot in G implies, by the definition of a b-knot, that G has at least one sink, and therefore there exists a graph that is signal-reduced from G by each of G's sinks. Note also that, in the OR model, the process of signal-reduction preserves all knots existing in the original graph while creating no new knots. This is what happens, then, when that original graph is a b-subgraph of G and its deadlock model is derived from that of G as we discussed earlier.

An illustration is provided in Figure 3 of this process of signal-reduction from a sink. The graph shown in Figure 3(a) is signal-reduced from the one of Figure 1(a) by n_1. Note that not only does n_1 become isolated, but also the edge directed from n_6 to n_5 need no longer exist (once an unblocking signal is received by n_6 from n_1, the only further signal that it needs is from n_2, as a signal from n_5 is irrelevant to its wait condition). The remaining graphs in Figure 3, those in parts (b) and (c), are both b-subgraphs of the graph of Figure 3(a). The one in Figure 3(b) is signal-reduced from the graph in Figure 1(c) by n_1 (this one a b-subgraph of the graph in Figure 1(a)), while the one in Figure 3(c) is not. However, all it takes for the graph of Figure 3(c) to be signal-reduced from that same graph by n_1 is the addition of the isolated n_1 to it.

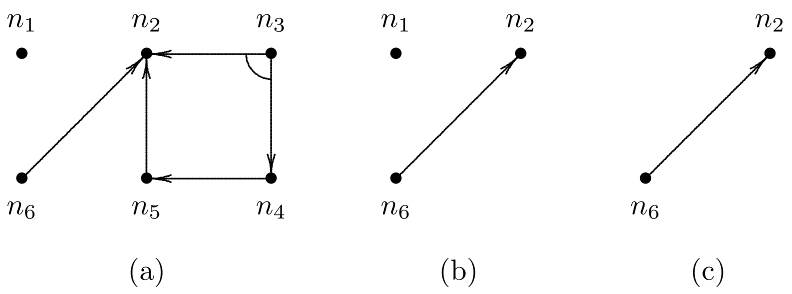

Figure 3. Signal-reduction by n_1

We now demonstrate, after stating and proving a supporting lemma, that the existence of a b-knot in G is closely related to the existence of a deadlock in the AND-OR model.

Lemma 1. *If G has no b-knots and H is signal-reduced from G by one of G's sinks, then H has no b-knots.*

Proof: Let n_i be the sink in G such that H is signal-reduced from G by n_i, and let H' be a b-subgraph of H. If H' is also a b-subgraph of G, then by hypothesis H' has no knots. If H' is not a b-subgraph of G, then H' includes an immediate ancestor n_j of n_i in G for which $W_j^k \neq \emptyset$ and $\bar{W}_j^k = \emptyset$, where $1 \leq k \leq p_j$. In other words, the nonempty W_j^k in G became an empty $\bar{W}_j^k$ in H through the signal-reduction by n_i. In order for this to have happened, there has to exist k' such that $1 \leq k' \leq p_j$, $k' \neq k$, and $W_j^{k'} \supseteq \{n_i\}$ such that $W_j^{k'} \setminus \{n_i\} \subseteq W_j^k \setminus \{n_i\}$. Now consider a b-subgraph G' of G that includes n_i and n_j, and in addition includes an edge directed from n_j to n_i and another directed from n_j to any member of $W_j^k \setminus W_j^{k'}$ (this set is nonempty, by definition of a deadlock model). If n_i is a node of H', then there exists such a G' from which H' is obtained via a signal-reduction by n_i. If n_i is not a node of H', then the result of this signal-reduction is H' enlarged by the isolated n_i. In either case, any knots in H' must also be knots in G'. These, however, are ruled out by hypothesis, so in this case too H' has no knots. It follows that H has no b-knots. ■

Theorem 2. *There exists a deadlock in the AND-OR model if and only if G has a b-knot.*

Proof: If G has a b-knot, then let G' be a b-subgraph of G in which a knot exists. A node in this knot is blocked for the receipt of a signal from at least one of its immediate descendants in G', but the existence of the knot means that such a signal will never be sent. As a consequence, that node is permanently deprived of any progress in the distributed computation, that is, a deadlock exists.

Conversely, suppose that G does not have a b-knot. In order to prove that in this case no deadlock exists, we must show that, if G can only evolve by the removal of edges as signals are sent to unblock waiting nodes, then eventually all waits are eliminated and G stabilizes as a graph with no edges. But this is guaranteed directly by Lemma 1, thence the theorem. ■

If at most one of $W_i^1, \ldots, W_i^{p_i}$ is nonempty for all $n_i \in N$ (this is the AND model), then in every b-subgraph of G every node has at most one immediate descendant. A knot in such a subgraph is a directed cycle, so the condition that Theorem 2 asserts for the AND-OR model becomes the known condition for the existence of a deadlock in the AND model. Similarly, if all nonempty sets in $W_i^1, \ldots, W_i^{p_i}$ are singletons for all $n_i \in N$ (the OR model), then in every b-subgraph node n_i has as many immediate descendants as it has in G. A knot in such a subgraph is then a knot in G as well, and the condition for the AND-OR model given by Theorem 2 is reduced to the condition for the existence of a deadlock in the OR model.

3. Checking membership distributedly

In this section we describe an asynchronous distributed algorithm for a node, say $n_1 \in N$, to detect whether it is in a b-knot in G. The algorithm we give employs a simplified version of the algorithm of [8] as an initial phase. That algorithm has been given for the detection by n_1 of whether it belongs to a knot in G. We discuss that algorithm first.

3.1. Knot detection

The algorithm of [8] (and hence the one we give in this section) is described for the following model of computation. A node in G is identified with a process that can only compute reactively to the reception of messages from other nodes. Upon receiving a message, a node may compute and send messages to any other nodes that are directly connected to it in G, regardless of the directions of the edges. Only node n_1 can compute (and possibly send messages) without being triggered by the arrival of a message, but it must do so only once and behave reactively like the others thereafter. Such a distributed computation is referred to as a *diffusing computation* initiated by n_1. As in the standard model for asynchronous distributed computations [1], every node has an independent time basis, and messages are guaranteed to be delivered with finite (though unpredictable) delays. Finally, we note that the algorithms discussed in this section adopt the same view of [2] for the distributed detection of stable properties, that is, the view that G stands for the (unchanging) wait-for graph at some consistent global state of the underlying distributed computation.

The departing point of the algorithm in [8] is that n_1 is in a knot in G if and only if $D_1 \setminus A_1 = \emptyset$. What the algorithm does is to compute the cardinality of $D_1 \setminus A_1$. In order to achieve this, messages of three types, called *desc*, *anc*, and *ack*, are employed, along with the following suite of variables for node n_i.

descendant$_i$: Boolean variable indicating whether $n_i \in D_1$ (initially set to **true** if $i = 1$, **false** otherwise);

ancestor$_i$: Boolean variable indicating whether $n_i \in A_1$ (initially set to **true** if $i = 1$, **false** otherwise);

subordinate$_i$: Integer variable having value 1 if $n_i \in D_1 \setminus A_1$, 0 otherwise (initially set to 0);

cs$_i$: Integer variable containing the sum of *subordinate*$_k$ over some nodes $n_k \in N$ (initially set to 0).

The algorithm seeks to establish

$$cs_1 = \sum_{n_i \in N} subordinate_i \tag{1}$$

$$= |D_1 \setminus A_1|$$

at global termination. Towards this goal, it proceeds as follows. Node n_1 starts by sending *desc* to its immediate descendants and *anc* to its immediate ancestors, and replies at once with an *ack* upon receiving any messages of these types. Another node n_i forwards the first *desc* it receives to its immediate descendants and the first *anc* it receives to its immediate ancestors. The *ack* that corresponds to a *desc* or *anc* received when n_i does not expect to receive any *ack*'s itself is withheld and only sent when n_i once again comes to expect no further *ack*'s. All other *desc*'s or *anc*'s received by n_i are replied to with an *ack* immediately. The first *desc* and *anc* that n_i receives cause $descendant_i$ and $ancestor_i$, respectively, to be set to **true**.

As one readily recognizes, this algorithm is an instance of the algorithm of [5] (see also [1] for another description) for n_1 to detect the global termination of diffusing computations initiated by itself. This occurs when n_1 has received *ack*'s from all of its immediate descendants and ancestors. In order to guarantee that (1) holds at this moment, the algorithm of [8] employs the following additional rules.

(a) Upon receiving a *desc* or an *anc*, node $n_i \neq n_1$ does

$$cs_i := cs_i - subordinate_i + \begin{cases} 1, & \text{if } descendant_i \textbf{ and not } ancestor_i; \\ 0, & \text{otherwise;} \end{cases}$$

$$subordinate_i := \begin{cases} 1, & \text{if } descendant_i \textbf{ and not } ancestor_i; \\ 0, & \text{otherwise;} \end{cases}$$

(b) Every *ack* that node n_1 sends is sent as $ack(0)$;

(c) Every *ack* that node $n_i \neq n_1$ sends is sent as $ack(cs_i)$, and then cs_i is reset to 0;

(d) Upon receiving an $ack(c)$, node n_i does $cs_i := cs_i + c$.

Now consider a consistent global state of the diffusing computation initiated by n_1, and let C be the sum, over all $ack(c)$'s that are in transit in that global state, of the parameters c. The following is an easy consequence of (a) through (d). At all consistent global states of the diffusing computation initiated by n_1,

$$\sum_{n_i \in N} cs_i + C = \sum_{n_i \in N} subordinate_i. \tag{2}$$

In particular, at all consistent global states at which global termination holds, we have $C = 0$ and, by (c), $cs_i = 0$ for all nodes $n_i \neq n_1$. By (2), it follows that the equality of (1) is achieved by n_1 upon detecting global termination.

3.2. B-knot detection

Let us now turn to the detection by node n_1 of whether it is in a b-knot in G. By definition, what n_1 must detect is whether it participates in a knot in some b-subgraph of G. For $n_i \in N$, let $S_i^1, \ldots, S_i^{q_i}$ be the subsets of O_i having at least one node from each nonempty set in $W_i^1, \ldots, W_i^{p_i}$ and at most as many nodes as there are nonempty sets in $W_i^1, \ldots, W_i^{p_i}$. For one of the q_1 subsets $S_1^1, \ldots, S_1^{q_1}$, an equivalent condition for n_1 to detect is whether there exists a b-subgraph of G that includes n_1 and that subset, but no subordinate of n_1. Node n_1 must be in a b-knot if and only if this condition holds for at least one of $S_1^1, \ldots, S_1^{q_1}$.

Our algorithm comprises two phases. The first phase is a simplification of the algorithm of [8] that removes most of the actions described under (a) through (d). What the simplified version achieves, upon global termination, is the correct computation of $\mathit{subordinate}_i$ for all $n_i \in N$. This first phase employs the same messages and variables we have considered so far, except for the parameters carried by the *ack*'s and the *cs* variables.

The second phase is also initiated by node n_1 and comprises two sub-phases, one "nested" into the other. It employs messages *desc*, *ack_d*, *anc*, and *ack_a*, in addition to the following variable (among others, to be introduced later) for node n_i.

- s_i: Array $[1 \mathinner{..} q_i]$ of Booleans, each indicating whether all b-subgraphs of G that include n_i and the corresponding set in $S_i^1, \ldots, S_i^{q_i}$ also include a subordinate of n_1 (initially set to **false**).

The goal of the second phase of our algorithm is to compute $s_1[k]$ for all k such that $1 \le k \le q_1$. Clearly, if this is achieved at global termination, then n_1 is in a b-knot if and only if

$$\bigwedge_{1 \le k \le q_1} s_1[k] = \mathbf{false}. \tag{3}$$

We start by providing an informal description of the second phase.

Node n_1 initiates the second phase by sending *desc* to all its immediate descendants. It then replies immediately with an *ack_d* to any *desc* it receives. When n_1 has received an *ack_d* for every *desc* it sent, global termination of the second phase has occurred. A node $n_i \neq n_1$, upon receiving the first *desc*, withholds the corresponding *ack_d* and then checks whether it is a subordinate of n_1. If it is not, then it forwards *desc* to all its immediate descendants and waits for no *ack_d*'s to be any longer expected in order to send the *ack_d* it withheld. If it is a subordinate of n_1 and has never received an *anc* message, then it initiates another diffusing computation, whose termination will signal that it may send the *ack_d* it withheld. Upon initiating this diffusing computation, n_i sets $s_i[k]$ to **true** for all k such that $1 \le k \le q_i$. Every further *desc* received by n_i is replied to with an *ack_d* immediately.

The diffusing computation that a subordinate n_i of n_1 initiates proceeds as follows. First n_i sends *anc* to all of its immediate ancestors that are also descendants of n_1 (this can be recorded locally at n_i during the propagation of *desc* messages in the first phase). Node n_i replies immediately with an *ack_a* to any *anc* that it receives, and detects global termination of the diffusing computation it initiated when it receives as many *ack_a*'s as it sent *anc*'s. Upon receiving *anc* from an immediate descendant n_ℓ, a node $n_j \neq n_i$ sets $s_j[k]$ to **true** for all k such that $n_\ell \in S_j^k$, and sends *anc* to all of its immediate ancestors that are also descendants of n_1 if $s_j[1] = \cdots = s_j[q_j] = \mathbf{true}$. The *ack_a* that corresponds to the *anc* it received is withheld in the affirmative case, otherwise it is sent at once. If withheld, it is sent when n_j no longer expects to receive any *ack_a*'s.

A more detailed description of the second phase is given next as Algorithm Compute(s), in which the following additional variables are used at node n_i.

- $\mathit{descendant}_i^k$: Boolean variable indicating whether $n_k \in D_1$ for $n_k \in I_i$ (this variable is assumed to have been set during the first phase as *desc* messages are received);
- $\mathit{subordinate}_i$: Integer variable having value 1 if n_i is a subordinate of n_1, 0 otherwise (this variable is assumed to have been set during the first phase);
- $\mathit{reached\_d}_i$: Boolean variable indicating whether n_i has received at least one *desc* (initially set to **true** if $i = 1$, **false** otherwise);
- $\mathit{expected\_d}_i$: Integer variable containing the number of *ack_d*'s n_i expects to receive (initially set to 0);
- $\mathit{parent\_d}_i$: Variable used to point to a special immediate ancestor of n_i (initially set to **nil**);
- $\mathit{initiator}_i$: Boolean variable indicating whether n_i is one of the nodes that initiate the sending of *anc*'s (initially set to **false**);

expected_a$_i$: Integer variable containing the number of *ack_a*'s n_i expects to receive (initially set to 0);

parent_a$_i$: Variable used to point to a special immediate descendant of n_i (initially set to **nil**).

Algorithm COMPUTE(s) comprises actions (4) through (8), respectively for initiation by node n_1 and for $n_i \in N$ to respond to the reception of a *desc*, an *ack_d*, an *anc*, and an *ack_a*.

Algorithm COMPUTE(s)**:**

```
▷ Initial action by $n_1$:                                                   (4)

    Send $desc$ to all $n_k \in O_i$;
    Set $expected\_d_i$ accordingly.

▷ Action upon receipt by $n_i$ of $desc$ from $n_j \in I_i$:                 (5)

    if $reached\_d_i$ then
        Send $ack\_d$ to $n_j$                                              (5.1)
    else
        begin
            $reached\_d_i$ := true;
            $parent\_d_i := n_j$;
            if $subordinate_i = 0$ then
                begin                                                       (5.2)
                    Send $desc$ to all $n_k \in O_i$;
                    Set $expected\_d_i$ accordingly;
                    if $expected\_d_i = 0$ then
                        Send $ack\_d$ to $parent\_d_i$
                end
            else
                if $s_i[1] = \cdots = s_i[q_i]$ = false then
                    begin                                                   (5.3)
                        $initiator_i$ := true;
                        $s_i[k]$ := true for all $k$ such that $1 \le k \le q_i$;
                        Send $anc$ to all $n_k \in I_i$ such that $descendant_i^k$;
                        Set $expected\_a_i$ accordingly
                    end
        end.

▷ Action upon receipt by $n_i$ of $ack\_d$ from $n_j \in O_i$:               (6)

    $expected\_d_i := expected\_d_i - 1$;
    if $expected\_d_i = 0$ then
        if $i = 1$ then
            Global termination has occurred                                 (6.1)
        else
            Send $ack\_d$ to $parent\_d_i$.                                 (6.2)
```

```
▷ Action upon receipt by $n_i$ of $anc$ from $n_j \in O_i$:                         (7)

    if $initiator_i$ then
        Send $ack\_a$ to $n_j$                                                       (7.1)
    else
        begin
            $s_i[k] :=$ true for all $k$ such that $n_j \in S_i^k$;                  (7.2)
            if $s_i[1] \wedge \cdots \wedge s_i[q_i]$ then
                begin                                                                (7.3)
                    $parent\_a_i := n_j$;
                    Send $anc$ to all $n_k \in I_i$ such that $descendant_i^k$;
                    Set $expected\_a_i$ accordingly;
                    if $expected\_a_i = 0$ then
                        Send $ack\_a$ to $parent\_a_i$
                end
            else
                Send $ack\_a$ to $n_j$                                               (7.4)
        end.

▷ Action upon receipt by $n_i$ of $ack\_a$ from $n_j \in I_i$:                       (8)

    $expected\_a_i := expected\_a_i - 1$;
    if $expected\_a_i = 0$ then
        if $initiator_i$ then
            Send $ack\_d$ to $parent\_d_i$                                           (8.1)
        else
            Send $ack\_a$ to $parent\_a_i$.                                          (8.2)
```

3.3. Correctness and complexity

In this section we first concentrate on establishing the correctness of Algorithm Compute(s) and after that analyze the complexity of the overall algorithm, comprising Algorithm Compute(s) and the first phase that precedes it. Throughout this section, we let node n_i be called an *initiator* if, during the execution of Algorithm Compute(s), n_i ever executes (5.3), thereby setting $initiator_i$ to **true**.

Note that, if n_i is an initiator, then there exists in G a directed path starting at n_1 on which n_i is the first subordinate of n_1 to appear. This is so because the absence of such a path would require n_i to be reached by a *desc* message through another subordinate of n_1 first, which, by (5.3), never happens.

Algorithm Compute(s) is a combination of several diffusing computations, each including the termination-detection mechanism of [5]. The first diffusing computation occurs as a single instance and is initiated by n_1. It uses *desc* and *ack_d* messages, and proceeds according to (4) and (5.2) for the sending of *desc*'s, and according to (5.1), (5.2), (6.2), and (8.1) to send *ack_d*'s.

Each of the other diffusing computations is initiated by an initiator, and employs *anc* messages (sent according to (5.3) and (7.3)) and *ack_a* messages (sent according to (7.1), (7.3), (7.4), and (8.2)).

The following is how the various diffusing computations are combined. At an initiator, by (5.3) the computation initiated by n_1 is suspended and a new computation, based on *anc* and *ack_a* messages, is initiated. Upon termination of this computation, the computation initiated by n_1 is resumed by (8.1). It is an immediate consequence of the results in [5] that all these computations do indeed terminate correctly, that is, the detection of global termination by n_1

in (6.1) is correct. We then concentrate on arguing that, upon global termination of Algorithm COMPUTE(s), s_1 has been computed correctly for use in checking whether (3) holds.

Theorem 3. *Let k be such that $1 \leq k \leq q_1$. Upon global termination of Algorithm* COMPUTE(s), $s_1[k] =$ **true** *if and only if all b-subgraphs of G that include n_1 and S_1^k also include a subordinate of n_1.*

Proof: If $s_1[k] =$ **true** when Algorithm COMPUTE(s) terminates globally, then by (7.2) n_1 must have received an *anc* from an immediate descendant in S_1^k. Let n_1 and the immediate descendants from which n_1 received an *anc* be *marked*, and proceed with the marking of other nodes as follows. If node n_i is marked and is not an initiator, then mark the nodes from which n_i received an *anc*. By (5.3) and (7.3), at least one node in each of the sets $S_i^1, \ldots, S_i^{p_i}$ gets marked, and the marking process halts at initiators or at nodes already marked. Now consider any subgraph of G that includes n_1, S_1^k, and for every n_i that is included, one of the sets $S_i^1, \ldots, S_i^{p_i}$ (unless n_i is not marked or is an initiator, such a set includes a marked node). It follows that this subgraph necessarily includes an initiator, and is therefore a b-subgraph of G that includes n_1, S_1^k, and a subordinate of n_1.

If $s_1[k] =$ **false** upon global termination of Algorithm COMPUTE(s), then by (7.2) no *anc* ever arrived at n_1 from an immediate descendant in S_1^k. We do the marking process again, starting at n_1 and all nodes in S_1^k. For a marked n_i, we mark every node from which n_i received no *anc*. Once again by (5.3) and (7.3), all nodes in at least one of the sets $S_i^1, \ldots, S_i^{p_i}$ get marked, and the marking stops at nodes already marked without ever reaching an initiator. Now consider any subgraph of G that includes n_1, S_1^k, and for every n_i that is included, one of the sets $S_i^1, \ldots, S_i^{p_i}$ whose nodes are all marked. Clearly, this subgraph is a b-subgraph of G that includes n_1 and S_1^k, but no subordinate of n_1. ■

We now turn to an analysis of the complexity of our algorithm to detect membership of n_1 in a b-knot in G. To this end, we employ the standard measures of time and communication complexity for asynchronous distributed algorithms [1]. Also, we let E_1^a be the set of edges that lie on paths directed towards n_1 in G, and E_1^d the set of edges lying on paths directed away from n_1 in G.

Note, initially, by the results in [8], that the time required by the first phase of our algorithm is no larger than $2\big(|A_1 \cup D_1|\big)$. In addition, the first phase requires exactly $2\big(|E_1^a| + |E_1^d|\big)$ messages to be sent. Of these, exactly one *desc* and one *ack* are sent on each member of E_1^d, and exactly one *anc* and one *ack* on each member of E_1^a.

To analyze the complexity of the second phase (Algorithm COMPUTE(s)), note, by (4), (5.2), (5.3), and (7.3), that messages are only sent on edges that connect two descendants of n_1. So the time taken by Algorithm COMPUTE(s) is at most $4|D_1|$. In order to assess the number of messages involved in the algorithm, note that a descendant of n_1 only receives a *desc* message, and therefore starts participating in the computation, if a path exists directed from n_1 to it with no intermediate subordinates of n_1. As a consequence, the number of messages sent by Algorithm COMPUTE(s) is at most $4|E_1^d|$. Of these, at most one *desc/ack_d* pair flows on each member of E_1^d, and likewise at most one *anc/ack_a* pair.

4. Concluding remarks

In this paper we have identified b-knots as graph structures that account for deadlocks in the AND-OR model as a necessary and sufficient condition. Unlike cycles and knots, b-knots are defined in explicit conjunction with the deadlock model.

We have also given an asynchronous distributed algorithm for node n_1 to check whether it is in a b-knot in G. This algorithm extends the algorithm of [8], which is essentially a procedure to count the subordinates of n_1. Our algorithm employs a simplification of this procedure as a first phase (it simply identifies the subordinates of n_1), and then a second phase in which each node that is a descendant of n_1, and from which a subordinate of n_1 is reachable, can detect whether a subordinate of n_1 is reachable from it in all b-subgraphs of G in which it participates. Detection of whether n_1 is in a b-knot follows easily.

Ongoing work includes investigating how existing algorithms to detect AND-OR deadlocks (those in [3, 7, 9], for example) relate to the presence of a b-knot in the wait-for graph, and how they can be improved, if at all, by explicitly considering the b-knot whose presence we have demonstrated to be necessary and sufficient for an AND-OR deadlock to exist. We also point out that the algorithm of Section 3 has been given as an extension of a well-known algorithm for knot detection only. The question of how to exploit the definition of a b-knot more closely in order to improve that algorithm (and then carry out an all-encompassing comparative study of the algorithms that detect AND-OR deadlocks) is subject of ongoing research as well.

Acknowledgments

We acknowledge partial support from CNPq, CAPES, the PRONEX initiative of Brazil's MCT under contract 41.96.0857.00, the LOCUS Project of ProTeM/CNPq, and a FAPERJ BBP grant.

References

1. V. C. Barbosa, *An Introduction to Distributed Algorithms*, The MIT Press, Cambridge, MA, 1996.
2. G. Bracha and S. Toueg, "Distributed deadlock detection," *Distributed Computing* **2** (1987), 127–138.
3. J. Brzezinski, J.-M. Hélary, M. Raynal, and M. Singhal, "Deadlock models and a general algorithm for distributed deadlock detection," *J. of Parallel and Distributed Computing* **31** (1995), 112–125.
4. K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Trans. on Computer Systems* **3** (1985), 63–75.
5. E. W. Dijkstra and C. S. Scholten, "Termination detection for diffusing computations," *Information Processing Letters* **11** (1980), 1–4.
6. E. Knapp, "Deadlock detection in distributed databases," *ACM Computing Surveys* **19** (1987), 303–328.
7. A. D. Kshemkalyani and M. Singhal, "Efficient detection and resolution of generalized distributed deadlocks," *IEEE Trans. on Software Engineering* **20** (1994), 43–54.
8. J. Misra and K. M. Chandy, "A distributed graph algorithm: knot detection," *ACM Trans. on Programming Languages and Systems* **4** (1982), 678–686.
9. D.-S. Ryang and K. H. Park, "A two-level distributed detection algorithm of AND/OR deadlocks," *J. of Parallel and Distributed Computing* **28** (1995), 149–161.
10. M. Singhal, "Deadlock detection in distributed systems," *IEEE Computer* **22** (1989), 37–48.