The Combinatorics of Resource Sharing

Valmir C. Barbosa
Programa de Engenharia de Sistemas e Computação, COPPE
Universidade Federal do Rio de Janeiro
Caixa Postal 68511, 21945-970 Rio de Janeiro - RJ, Brazil
valmir@cos.ufrj.br

Abstract

We discuss general models of resource-sharing computations, with emphasis on the combinatorial structures and concepts that underlie the various deadlock models that have been proposed, the design of algorithms and deadlock-handling policies, and concurrency issues. These structures are mostly graph-theoretic in nature, or partially ordered sets for the establishment of priorities among processes and acquisition orders on resources. We also discuss graph-coloring concepts as they relate to resource sharing.

Keywords: Deadlock models, deadlock detection, deadlock prevention, concurrency.

1 Introduction

The sharing of resources by processes is one of the most fundamental issues in the design of computer systems, and stands at the crux of most efficiency considerations for those systems. When referred to with such generality, processes can stand for any of the computing entities one finds at the various levels of a computer system, and likewise resources are any of the means necessary for those entities to function. Resources tend to be scarce (or to get scarce shortly after being made available), so the designer of a computer system at any level must get involved with the task of devising allocation policies whereby the granting of resources to processes can take place with at least a minimal set of guarantees.

One such guarantee is of the so-called *safety* type, and in essence forbids the occurrence of *deadlock* situations. A deadlock situation is characterized by the permanent impossibility for a group of processes to progress with their tasks due to the occurrence of a condition that prevents at least one needed resource from being granted to each of the processes in that group. Another guarantee one normally seeks is a *liveness* guarantee, which imposes bounds on the wait that any process must undergo between requesting and being granted access to a resource, and thereby ensures that *lockout* situations never happen.

There are difficulties of various sorts associated with designing and analyzing resource-sharing policies. Some of these difficulties refer to the choice and use of mathematical models that can account properly for the relevant details of the resource-sharing problem at hand. Similarly, there are difficulties that stem from the inherent asynchronism that typically characterizes the behavior of processes in a computer system. This asynchronism, though essential in depicting most computer systems realistically, tends to introduce subtle obstacles to the design of correct algorithms.

In this paper, we are concerned with the several combinatorial models that have proven instrumental in the design and analysis of resource-sharing policies. The models that we consider are essentially of graph-theoretic nature, and relate closely to the aforementioned safety and liveness issues. The essential notation that we use is the following. The set of processes is denoted by $\mathcal{P} = \{P_1, \ldots, P_n\}$, and the set of resources by $\mathcal{R} = \{R_1, \ldots, R_m\}$. For $P_i \in \mathcal{P}$, $\mathcal{R}_i \subseteq \mathcal{R}$ is the set of resources to which P_i may request access. Similarly, for $R_p \in \mathcal{R}$, $\mathcal{P}_p \subseteq \mathcal{P}$ is the set of processes that may request access to R_p. Clearly, for $1 \leq i \leq n$ and $1 \leq p \leq m$, $P_i \in \mathcal{P}_p$ if and only if $R_p \in \mathcal{R}_i$. Also, we let $\mathcal{R}_{ij} = \mathcal{R}_i \cap \mathcal{R}_j$ for $1 \leq i, j \leq n$, and $\mathcal{P}_{pq} = \mathcal{P}_p \cap \mathcal{P}_q$ for $1 \leq p, q \leq m$.

Example 1 *If* $\mathcal{P} = \{P_1, P_2, P_3, P_4, P_5\}$ *and* $\mathcal{R} = \{R_1, R_2, R_3, R_4, R_5, R_6\}$ *with* $\mathcal{R}_1 = \{R_1, R_2\}$, $\mathcal{R}_2 = \{R_2, R_3, R_6\}$, $\mathcal{R}_3 = \{R_3, R_4, R_6\}$, $\mathcal{R}_4 = \{R_4, R_5, R_6\}$, *and* $\mathcal{R}_5 = \{R_1, R_5\}$, *then* $\mathcal{P}_1 = \{P_1, P_5\}$, $\mathcal{P}_2 = \{P_1, P_2\}$, $\mathcal{P}_3 = \{P_2, P_3\}$, $\mathcal{P}_4 = \{P_3, P_4\}$, $\mathcal{P}_5 = \{P_4, P_5\}$, *and* $\mathcal{P}_6 = \{P_2, P_3, P_4\}$. *In addition, we have the following nonempty sets:*

$$
\begin{array}{ll}
\mathcal{R}_{12} = \{R_2\} & \mathcal{P}_{12} = \{P_1\} \\
\mathcal{R}_{15} = \{R_1\} & \mathcal{P}_{15} = \{P_5\} \\
\mathcal{R}_{23} = \{R_3, R_6\} & \mathcal{P}_{23} = \{P_2\} \\
\mathcal{R}_{24} = \{R_6\} & \mathcal{P}_{26} = \{P_2\} \\
\mathcal{R}_{34} = \{R_4, R_6\} & \mathcal{P}_{34} = \{P_3\} \\
\mathcal{R}_{45} = \{R_5\} & \mathcal{P}_{36} = \{P_2, P_3\} \\
 & \mathcal{P}_{45} = \{P_4\} \\
 & \mathcal{P}_{46} = \{P_3, P_4\} \\
 & \mathcal{P}_{56} = \{P_4\}
\end{array}
$$

The following is how the remainder of the paper is organized. In Section 2, we provide an outline of the generic computation that is carried out by the members of $\mathcal{P}$ in order to share the resources in $\mathcal{R}$. Such an outline is given as an asynchronous distributed algorithm, and aims at emphasizing the communication that must take place among processes for resource sharing. This communication comprises at least messages for requesting and granting access to resources. Depending on how such messages are composed and handled by the processes, one gets one of the various *deadlock models* that have appeared in the literature. These models are our subject in Section 3. The two sections that follow (Sections 4 and 5) are devoted to the combinatorics underlying the two broad classes of deadlock-handling policies, namely those of *detection* and *prevention* strategies, respectively. We then move, in Section 6, to a prevention policy that generalizes one of policies discussed in Section 5 and for which an abacus-like graph structure is instrumental. Section 7 contains a brief discussion of the case of high demand for resources by the processes. Section 8 discusses the relationship that exists between concurrency in resource sharing and the various chromatic indicators of a graph. Concluding remarks follow in Section 9.

In this paper, all lemma and theorem proofs are omitted, but references are given to where they can be found.

2 Resource-sharing computations

The model of computation that we assume in this section is the standard *fully asynchronous* (or simply *asynchronous*) model of distributed computing [2]. In this model, every member of $\mathcal{P}$ possesses a local, independent clock, having therefore a time basis that is totally uncorrelated to that of any other process. In addition, all communication among processes take place via point-to-point message passing, requiring a finite (though unpredictable) time for message delivery. Messages are sent over bidirectional communication channels, of which there exists one for every $P_i, P_j \in \mathcal{P}$ such that $\mathcal{R}_{ij} \neq \emptyset$. That is, every two processes with the potential to share at least one resource are directly interconnected by a bidirectional communication channel. If we let $\mathcal{C}$ denote the set of such channels, then the undirected graph $G = (\mathcal{P}, \mathcal{C})$, having one vertex for each process and one edge for each channel, can be used to represent the system over which our resource-sharing computations run. In G, and for $1 \le p \le m$, the vertices in $\mathcal{P}_p$ induce a completely connected subgraph (a *clique* [6]). We assume that G is a connected graph, as processes belonging to different connected components never interfere with each other. In the context of Example 1, G is the graph shown in Figure 1.

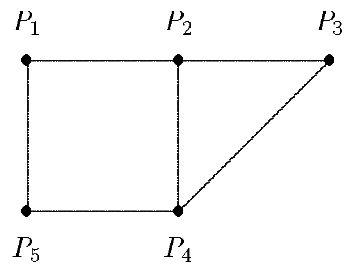


Figure 1: The graph G for Example 1

In the computations that we consider, a process executes the following four procedures.

- REQUEST;
- CHECK_PRIORITY;
- COMPUTE;
- CLEAN_UP.

Each of these procedures is executed atomically in response to a specific event, as follows. When the need arises for the process to compute on shared resources, it executes the REQUEST procedure. Typically, this will entail sending to some of its neighbors in G messages requesting exclusive access to resources shared with them. The reception of one such message causes the receiving process to execute CHECK_PRIORITY, whose outcome will guide the process's decision as to whether grant or not the requested exclusive access. If the process does decide to grant the request, then a message carrying this information is sent back to the requesting process, which upon receipt executes COMPUTE. This procedure is a test to see whether the process already holds exclusive access to enough resources to carry out its computation, which it does in the affirmative case; it keeps waiting, otherwise. If and when the resource-sharing computation is completed, the process engages in a message exchange with its neighbors in G by executing the CLEAN_UP procedure. This message exchange may revise priorities and cause previously withheld requests to be granted.

This outline is admittedly far too generic in several aspects, but already it provides the background for the key questions underlying the establishment of a resource-sharing policy. For example: To which resources does a process request exclusive access in REQUEST when in need for shared resources? At which point when executing COMPUTE does it decide it may proceed with its computation? How do the CHECK_PRIORITY and CLEAN_UP procedures cooperate to handle the priority issue properly? Answers to these questions have been given in the context of several application areas, and along with numerous models and algorithms. Addressing them in detail is beyond our intended scope, but in Section 3 we present the abstraction of deadlock models, which summarizes the issues that are critical to our discussion of the combinatorics of resource sharing.

Note that the initial test performed by COMPUTE may entail waiting on the part of the calling process. Clearly, then, and depending on how the priority issue is handled, here lies the possibility for unbounded wait, which is directly related to the safety and liveness guarantees we may wish to provide. The approaches here vary greatly, and may be grouped into two broad categories. On the "optimistic" side, one may opt for a somewhat loose priority scheme and risk the loss of those guarantees. In such cases, the loss of safety leads to the need for the capability of detecting deadlocks [11, 17, 23]. The opposing, more "conservative" side is the side of those strategies which "by design" guarantee safety and liveness, thereby preventing their loss beforehand.

As we demonstrate in the remainder of the paper, both categories give rise to interesting combinatorial structures and properties, especially as they relate to the deadlock issue. We then end this section by defining what will be meant henceforth by deadlock, although still somewhat informally. As we go through the various combinatorial structures that relate closely to deadlocks, such informality will dissipate. A subset of processes $\mathcal{S} \subseteq \mathcal{P}$ is in deadlock if and only if every process in $\mathcal{S}$ is waiting for a condition that ultimately can only be relieved by another member of $\mathcal{S}$. Obviously, then, deadlocks are stable properties: once they take hold of a group of processes, only the external intervention that eventually follows detection may break it. Prevention strategies, by contrast, seek never to let them happen.

3 Deadlock models

A deadlock model is an abstraction of the rules that govern the wait of processes for one another as they execute the procedures REQUEST, CHECK_PRIORITY, COMPUTE, and CLEAN_UP discussed in Section 2. Deadlock models are defined on top of a dynamic graph, called the *wait-for graph* and henceforth denoted by W.

W is the directed graph $W = (\mathcal{P}, \mathcal{W})$, having the same vertex set as G (one vertex per process) and the directed edges in $\mathcal{W}$. This set is such that an edge exists directed from process P_i to process P_j if and only if P_i has sent P_j a request for exclusive access to some resource that they share and is waiting either for P_j to grant the request or for the need for that resource to cease existing as grant messages are received from other processes. For $P_i \in \mathcal{P}$, we let $\mathcal{O}_i \subseteq \mathcal{P}$ be the set of processes towards which edges are directed away from P_i in $\mathcal{W}$.

It follows from the definition of W that the only processes that may be carrying out some computation on shared resources are those that are *sinks* in W (vertices whose adjacent edges are all incoming). All other processes are waiting for exclusive access to the resources they need. Clearly, then, a necessary condition for a deadlock to exist in W is that W contain a directed cycle.

Fact 1 *If a deadlock exists in W, then W contains a directed cycle.*

Example 2 *In the context of Example 1, suppose a deadlock has happened involving processes P_2, P_3, and P_4. Suppose also that process P_1 is waiting for resource R_2, which is held by P_2, which in turn is waiting for R_3, held by P_3, which is waiting for R_4, held by P_4. If, in addition, R_6 is held by P_2 and awaited by P_4, then the corresponding W is the one shown in Figure 2, with the directed cycle on P_2, P_3, and P_4.*

In Section 4, after we have gone through a variety of deadlock models in the remainder of this section, we will come to the conditions that are sufficient for a deadlock to exist in W.

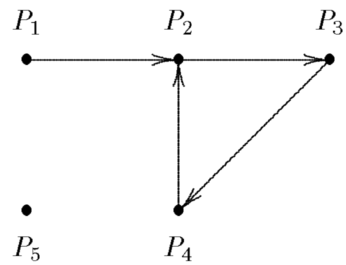


Figure 2: The graph W for Example 2

This is the sense in which graph W is a dynamic structure: although its vertex set is always the same, as the processes interact with one another by executing the aforementioned procedures, its edge set changes. Normally, given the view we have adopted of the resource-sharing computation as an asynchronous distributed computation, one must bear in mind the fact that it only makes sense to refer to W as associated with some *consistent global state* of the computation [2, 9]. For our purposes, however, such an association does not have to be explicit, so long as one understands the dynamic character of W.

What determines the evolution of W by allowing for changes in the set $\mathcal{W}$ of directed edges is the deadlock model that holds for the computation. The deadlock models that have been investigated to date are the ones we discuss next. In essence, what each of these deadlock models does is to specify rules for vertices that are not sinks in W to become sinks.

- **The AND model:** In the AND model, a process P_i can only become a sink when its wait is relieved by all processes in $\mathcal{O}_i$. This model characterizes situations in which a conjunction of resources is needed by P_i [8, 18, 21].

- **The OR model:** In the OR model, it suffices for process P_i to be relieved by one of the processes in $\mathcal{O}_i$ in order for its wait to finish. The OR model characterizes situations in which any one of a group of resources (a disjunction of resources) is needed by P_i [8, 18, 20, 21].

- **The x-out-of-y model:** In this model, there are two integers, x_i and y_i, associated with process P_i. Also, $y_i = |\mathcal{O}_i|$, meaning that process P_i is in principle waiting for communication from every process in $\mathcal{O}_i$. However, in order to be relieved from its wait condition, it suffices that such communication arrive from any x_i of those y_i processes. The x-out-of-y model can then be used to characterize situations in which P_i starts by requiring access permissions in excess of what it really needs, and then withdraws the requests that may still be pending when the first x_i responses are received [7, 8, 18].

- **The AND-OR model:** In the AND-OR model, there are $t_i \geq 1$ subsets of $\mathcal{O}_i$ associated with process P_i. These subsets are denoted by $\mathcal{O}_i^1, \ldots, \mathcal{O}_i^{t_i}$ and must be such that $\mathcal{O}_i = \mathcal{O}_i^1 \cup \cdots \cup \mathcal{O}_i^{t_i}$. In order for process P_i to be relieved from its wait condition, it must receive grant messages from all the processes in at least one of $\mathcal{O}_i^1, \ldots, \mathcal{O}_i^{t_i}$. For this reason, these t_i subsets of $\mathcal{O}_i$ are assumed to be such that no one is contained in another. Situations that the AND-OR model characterizes are those in which P_i perceives several conjunctions of resources as equivalent to one another and issues requests for several of them with provisions to withdraw some of them later [3, 8, 18, 21].

- **The disjunctive x-out-of-y model:** In this model, associated with process P_i are $u_i \geq 1$ pairs of integers, denoted by $(x_i^1, y_i^1), \ldots, (x_i^{u_i}, y_i^{u_i})$. These integers are such that $y_i^1 =$

$|\mathcal{Q}_i^1|, \ldots, y_i^{u_i} = |\mathcal{Q}_i^{u_i}|$, where $\mathcal{Q}_i^1, \ldots, \mathcal{Q}_i^{u_i}$ are subsets of $\mathcal{O}_i$ such that $\mathcal{O}_i = \mathcal{Q}_i^1 \cup \cdots \cup \mathcal{Q}_i^{u_i}$. In order to be relieved from its wait condition, P_i must be granted access to shared resources by either x_i^1 of the y_i^1 processes in $\mathcal{Q}_i^1$, or x_i^2 of the y_i^2 processes in $\mathcal{Q}_i^2$, and so on. Of course, it makes no sense for one of these groups of processes to be a subset of another, which is then assumed not to be the case. This model characterizes situations similar to those characterized by the x-out-of-y model, and generalizes that model by allowing for a disjunction on top of it [8, 18].

As one readily realizes, these five models are not totally uncorrelated and a strict hierarchy exists in which a model generalizes the previous one in the sense that it contains as special cases all the possible wait conditions of the other. For example, the x-out-of-y model generalizes the AND model with $x_i = y_i$ and the OR model with $x_i = 1$ for all $P_i \in \mathcal{P}$. Likewise, and also for all $P_i \in \mathcal{P}$, the AND-OR model also generalizes the AND model with $t_i = 1$ and the OR model with $|\mathcal{O}_i^1| = \cdots = |\mathcal{O}_i^{t_i}| = 1$.

Despite this ability of both the x-out-of-y model and the AND-OR model to generalize both the AND and OR models, they are not equivalent to each other. In fact, the AND-OR model is more general than the x-out-of-y model, while the converse is not true. In order for the AND-OR model to express a general x-out-of-y condition, it suffices that, for all $P_i \in \mathcal{P}$, $t_i = \binom{y_i}{x_i}$ and $|\mathcal{O}_i^1| = \cdots = |\mathcal{O}_i^{t_i}| = x_i$.

Example 3 *Suppose that we have, for some $P_i \in \mathcal{P}$, $\mathcal{O}_i = \{P_j, P_k, P_\ell\}$. In the x-out-of-y model, $y_i = 3$. If $x_i = 2$, then in the AND-OR model we have, equivalently, $t_i = \binom{3}{2} = 3$, $\mathcal{O}_i^1 = \{P_j, P_k\}$, $\mathcal{O}_i^2 = \{P_j, P_\ell\}$, and $\mathcal{O}_i^3 = \{P_k, P_\ell\}$.*

To finalize our discussion on how the five deadlock models are related to one another, note that the AND-OR model and the disjunctive x-out-of-y model are equivalent to each other. In order to see that the AND-OR model generalizes the disjunctive x-out-of-y model, let $t_i = v_i^1 + \cdots + v_i^{u_i}$, where

$$v_i^1 = \binom{y_i^1}{x_i^1}, \ldots, v_i^{u_i} = \binom{y_i^{u_i}}{x_i^{u_i}},$$

for all $P_i \in \mathcal{P}$. In addition, v_i^1 of the sets $\mathcal{O}_i^1, \ldots, \mathcal{O}_i^{t_i}$ must have cardinality x_i^1 and be subsets of $\mathcal{Q}_i^1$, the same holding for the other superscripts $2, \ldots, u_i$.

That the disjunctive x-out-of-y model generalizes the AND-OR model is simpler to see. For such, it suffices that, for all $P_i \in \mathcal{P}$, we let $u_i = t_i$ and $\mathcal{Q}_i^1 = \mathcal{O}_i^1, \ldots, \mathcal{Q}_i^{u_i} = \mathcal{O}_i^{t_i}$, along with $x_i^1 = y_i^1, \ldots, x_i^{u_i} = y_i^{u_i}$.

Example 4 *Let $\mathcal{O}_i = \{P_j, P_k, P_\ell, P_t\}$ for some $P_i \in \mathcal{P}$. In the disjunctive x-out-of-y model, suppose we have $u_i = 2$, $\mathcal{Q}_i^1 = \{P_j, P_k\}$, and $\mathcal{Q}_i^2 = \{P_k, P_\ell, P_t\}$, yielding $y_i^1 = 2$ and $y_i^2 = 3$. If $x_i^1 = x_i^2 = 2$, then in the AND-OR model we have $t_i = \binom{2}{2} + \binom{3}{2} = 4$, $\mathcal{O}_i^1 = \{P_j, P_k\}$, $\mathcal{O}_i^2 = \{P_k, P_\ell\}$, $\mathcal{O}_i^3 = \{P_k, P_t\}$, and $\mathcal{O}_i^4 = \{P_\ell, P_t\}$. Had we started out with this AND-OR setting, then for the disjunctive x-out-of-y model we would have $\mathcal{Q}_i^1 = \{P_j, P_k\}$, $\mathcal{Q}_i^2 = \{P_k, P_\ell\}$, $\mathcal{Q}_i^3 = \{P_k, P_t\}$, and $\mathcal{Q}_i^4 = \{P_\ell, P_t\}$. We would also have $x_i^1 = y_i^1 = x_i^2 = y_i^2 = x_i^3 = y_i^3 = x_i^4 = y_i^4 = 2$. Clearly, this is equivalent to the disjunctive x-out-of-y scenario of the beginning of this example.*

4 Graph structures for deadlock detection

As we remarked in Section 2, computations that make no *a priori* provisions against the occurrence of deadlocks must, if the need arises, resort to techniques for the detection of deadlocks. Detecting the existence of a deadlock in the wait-for graph W can become the detection of a graph-theoretic property on W if we are able to characterize conditions on W that are sufficient for the existence of deadlocks. As we discuss in this section, such a property exists for all the deadlock models of Section 3. However, not always is it the case that detecting this graph-theoretic property directly is the most efficient means of deadlock detection. When this happens not to be the case, alternative approaches must be employed, usually based on some form of simulation of the sending of grant messages.

We start with the AND model, and recognize immediately that the presence of a directed cycle in W is not only a necessary condition for the existence of a deadlock in W but also a sufficient condition. This is so because, in the AND model, every process requires grant messages to be received on all edges directed away from it, which clearly is precluded by the existence of a directed cycle.

Fact 2 *In the AND model, a deadlock exists in W if and only if W contains a directed cycle.*

In Figure 3, we show two wait-for graphs in the AND model. Circular arcs joining edges directed away from vertices are meant to indicate that the AND model is being used. By Fact 2, there is deadlock in the W of Figure 3(a), but not in Figure 3(b).

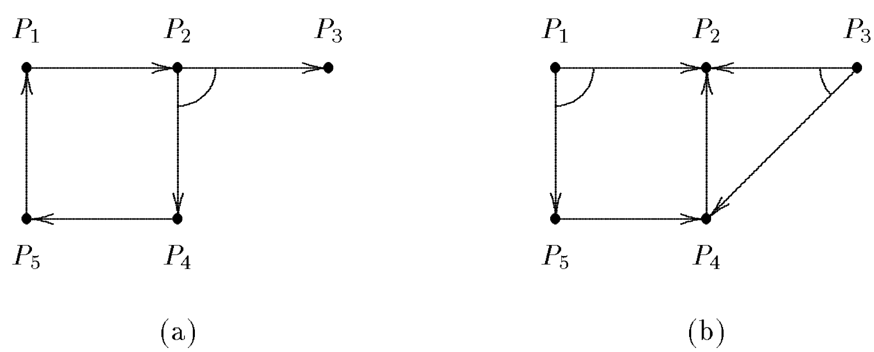


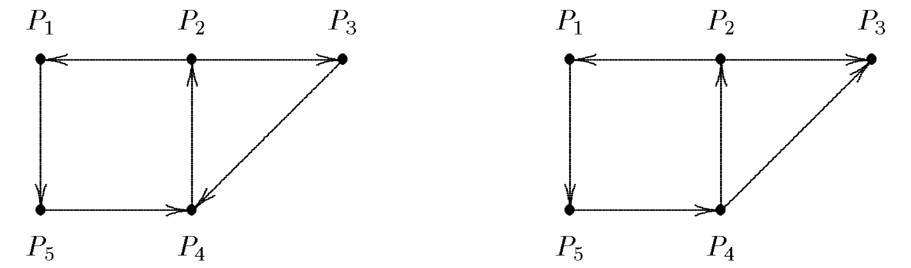
Figure 3: Two wait-for graphs in the AND model

The case of the OR model is more subtle, and it is instructive to start by realizing that the presence of a directed cycle in W is no longer sufficient for the existence of deadlocks. Clearly, so long as a directed path exists in W from every process to at least one sink, then no deadlock exists in W even though a directed cycle may be present. Formalizing this notion requires that we consider the definition of a *knot* in W.

For $P_i \in \mathcal{P}$, let $\mathcal{T}_i \subseteq \mathcal{P}$ be the set of vertices that can be reached from P_i through a directed path in W. This set includes P_i itself, and is known as the *reachability set* of P_i [6]. We say that a subset of vertices $\mathcal{S} \subseteq \mathcal{P}$ is a knot in W if and only if $\mathcal{S}$ has at least two vertices and, for all $P_i \in \mathcal{S}$, $\mathcal{T}_i = \mathcal{S}$. By definition, then, no member of a knot has a sink in its reachability set, which characterizes the presence of a knot in W as the sufficient condition we have sought under the OR model. As it turns out, in fact, this condition is also necessary, being stronger than the necessary condition established by Fact 1.

Theorem 1 [16] *In the OR model, a deadlock exists in W if and only if W contains a knot.*

The wait-for graphs of Figure 4 are for the OR model. A knot is present in Figure 4(a) (involving all vertices), but not in Figure 4(b). Thence, by Theorem 1, there is deadlock in part (a) of the figure but not in part (b), in which P_3 is a sink that can be reached from all other processes.



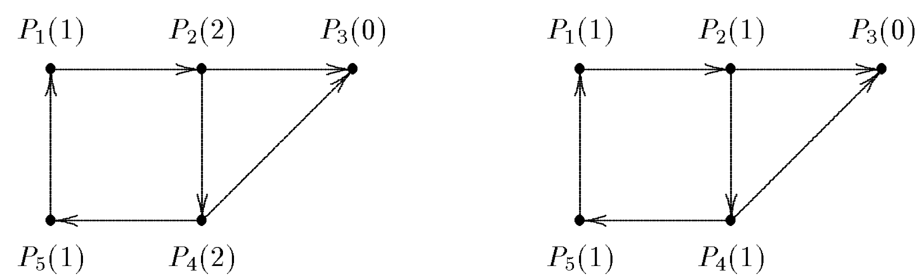
(a) (b)

Figure 4: Two wait-for graphs in the OR model

In order to identify sufficient conditions on W that account for the existence of deadlocks in the remaining deadlock models, we must consider W in a more explicit conjunction with the deadlock model than we have done so far. Let us first consider the x-out-of-y model, and suppose that a subset of vertices $\mathcal{S} \subseteq \mathcal{P}$ can be identified having the property that, for all $P_i \in \mathcal{S}$, $|\mathcal{O}_i \cap \mathcal{S}| > y_i - x_i$. Under these circumstances, it is clear that no member of $\mathcal{S}$ can ever receive the number of grant messages it requires, because at least one of such messages would necessarily have to originate from within $\mathcal{S}$. In this paper, we let a set such as $\mathcal{S}$ be called a $(y-x)$-*knot*, whose existence in W can also be shown to be necessary for deadlocks to exist. As in the case of the OR model, this condition is stronger than the necessary condition of Fact 1.

Theorem 2 [18] *In the x-out-of-y model, a deadlock exists in W if and only if W contains a $(y-x)$-knot.*

An illustration is given in Figure 5, with an integer in parentheses next to the name of each vertex to indicate its x value. A $(y-x)$-knot appears in Figure 5(a), but not in Figure 5(b). The $(y-x)$-knot of Figure 5(a) involves the vertices of the square. By Theorem 2, there is deadlock in the W of part (a) of the figure, but not in that of part (b). Note that P_3 is a sink reachable from all vertices in both graphs, but this is to no avail in the graph of part (a).



(a) (b)

Figure 5: Two wait-for graphs in the x-out-of-y model

We now turn to a discussion of sufficient conditions for deadlocks to exist in W under the AND-OR model. As we discussed in Section 3, the AND-OR model and the disjunctive x-out-of-y model are equivalent to each other, and for this reason the conditions that we come to identify as sufficient under the AND-OR model will also be sufficient under the disjunctive x-out-of-y model if only we perform the transformation described in Section 3.

Our starting point is the following definition. Consider a subgraph W' of W having vertex set $\mathcal{P}$, and for process P_i let $\mathcal{O}_i' \subseteq \mathcal{P}$ be the set of vertices towards which directed edges from P_i exist in W'. In addition, for process P_i let $\mathcal{O}_i'$ be such that $\mathcal{O}_i' \cap \mathcal{O}_i^1, \ldots, \mathcal{O}_i' \cap \mathcal{O}_i^{t_i}$ are all singletons. We call such a subgraph a *b-subgraph* of W, where the "b" is intended to convey the notion that each directed edge in W' relates to a "bundle" of directed edges stemming from the same vertex in W [3]. An illustration is given in Figure 6 of a wait-for graph in part (a) and one of its b-subgraphs in part (b). Circular arcs around vertices in Figure 6(a) indicate the "AND" groupings of neighbors that constitute vertices' waits.

Intuitively, a b-subgraph of W represents one of the various "OR" possibilities that are summarized in W under the AND-OR model, provided that we consider such possibilities "globally," i.e., over all processes. As it turns out, the existence of a knot in at least one of the b-subgraphs of W is necessary and sufficient for a deadlock to exist in W. The knot that in this case exists in that b-subgraph is called a *b-knot* in W [3].

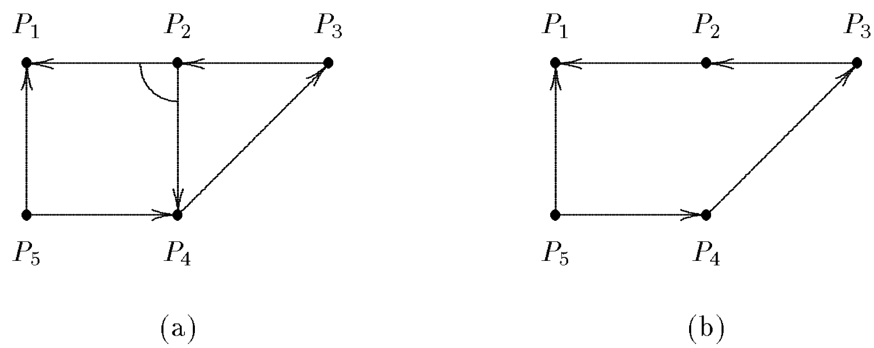


Figure 6: W and one of its b-subgraphs

Theorem 3 [3] *In the AND-OR model, a deadlock exists in W if and only if W contains a b-knot.*

We show in Figure 7 another of the b-subgraphs of the wait-for graph W of Figure 6(a). This b-subgraph has a knot spanning the processes in the triangle, which by Theorem 3 characterizes deadlock. In fact, in W it is easy to see that P_2 requires a relieve signal not only from P_1 (this one must come eventually) but also from P_4 (which will never come).

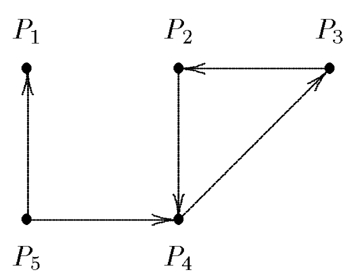


Figure 7: A b-subgraph with a knot

5 Partially ordered sets and deadlock prevention

In the remaining sections (Sections 5 through 8), we address the AND model exclusively. For this model, by Facts 1 and 2 we know that the existence of a directed cycle in the wait-for graph W is necessary and sufficient for a deadlock to exist. The fact that this condition is necessary, in particular, allows us to look for design strategies that prevent the occurrence of deadlocks beforehand by precluding the appearance of directed cycles in W.

Of course, we also know from Fact 1 that directed cycles in W are necessary for a deadlock to exist regardless of the deadlock model. However, for deadlock models other than the AND model, we have seen in Section 4 that there may exist a directed cycle in W without the corresponding existence of a deadlock. As a matter of fact, we have seen that structures in W much more complicated than directed cycles are necessary for deadlocks to exist. Preventing the occurrence of cycles in those other models is then too restrictive, while preventing the occurrence of the more general structures appears to be too complicated. That is why our treatment of deadlock prevention is henceforth restricted to the AND model.

Resource-sharing problems for the AND model are often referred to as the *dining* or *drinking philosophers problem* [10, 13], depending, respectively, on whether every process P_i always requests access to all the resources in $\mathcal{R}_i$ or not. In the remainder of this section, we discuss two prevention strategies for such problems. Both strategies are based on the use of a partially ordered set (a poset), in the first case to establish dynamic priorities among processes, in the second to establish a static global order for resource request.

5.1 Ordering the processes

Consider the graph G that represents the sharing of resources among processes, and let ω be an acyclic orientation of its edges. That is, ω assigns to each edge in $\mathcal{C}$ (the edge set of G) a direction in such a way that no directed cycle is formed. This orientation establishes a partial order on the set $\mathcal{P}$ of G's vertices, so G oriented by ω can be regarded as a poset.

This poset is dynamic, in the sense that the acyclic orientation changes over time, and can be used to establish priorities for processes that are adjacent in G to use shared resources when there is conflict. More specifically, consider a resource-sharing computation that does the following. A process sends requests for all resources that it needs, and must, upon receiving a request, decide whether to grant access to the resource immediately or to do it later. What the process does is to check whether the edge between itself and the requesting process is oriented outwards by ω. In the affirmative case, it grants access to the resource either immediately or upon finishing to use it (if this is the case). In the negative case, it may either grant access (if it does not need the resource presently) or delay the granting until after it has acquired all the resources it needs and used them. Whenever a process finishes using a group of resources, it causes all edges presently oriented towards itself to be oriented outward, thereby changing the acyclic orientation of G locally. These reversals of orientation constitute priority reversals between the processes involved.

We see, then, that an acyclic orientation establishes a priority for resource usage between every two neighbors in G, and that this priority is reversed back and forth between them as they succeed in using the resources they need. The crux of this mechanism is the simple property that the local changes a process causes to the acyclic orientation always maintain its acyclicity, and therefore its poset nature. If ω' is the acyclic orientation that results from the application of such a local change, then we have the following.

Lemma 1 [10] *If ω is acyclic, then ω' is acyclic.*

Note that Lemma 1 holds even if ω' results from local changes applied to ω by more than one process concurrently. We show such a pair of orientations in Figure 8, where the processes that do the reversal are P_1 and P_4. In the dining-philosopher variant of the resource-sharing computation, such a group of processes does necessarily constitute an *independent set* of G (a set whose members are all nonneighbors) [6].

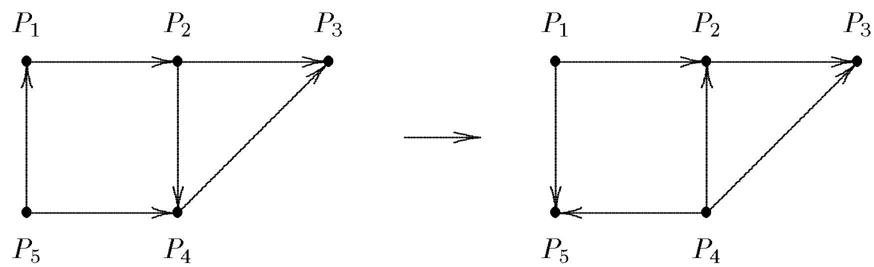


Figure 8: Edge reversal on acyclic orientations

The acyclicity of the changing orientation of G is crucial in guaranteeing that deadlocks never occur. To see this, consider the wait-for graph W that is formed when processes cannot grant access to resources immediately. According to the computation we outlined above, this happens when a process, say P_i, receives a request from a neighbor P_j but cannot grant it immediately because it too needs the resource in question and furthermore holds priority over P_j. Clearly, in both W and the acyclic orientation ω of G that gives priorities, the edge between P_i and P_j is oriented from P_j to P_i. In other words, W is always a subgraph of G oriented by ω, and by Lemma 1 never contains a directed cycle, which by Fact 1 implies the absence of deadlocks. If we refer to computations such as the one we described as *edge-reversal computations* [1], then we have the following.

Theorem 4 [10] *Every edge-reversal computation is deadlock-free.*

Not only do the orientations of G ensure the absence of deadlocks, but they can be easily seen to ensure liveness guarantees as well. Owing to the relationship of the computations we are considering to the dining-philosopher paradigm, such guarantees are referred to as the absence of *starvation*. That no starvation ever occurs comes also from the absence of directed cycles in W: as the orientations of edges are reversed and W evolves, the farthest sinks for which a process is ultimately waiting come ever closer to it, until it too becomes a sink eventually and its wait ceases.

Theorem 5 [10] *Every edge-reversal computation is starvation-free, and the worst-case wait a process must undergo is $O(n)$.*

We note that, in Theorem 5, the wait of a process is measured as the length of "causal chains" in the sending of grant messages, as is customary in the field of asynchronous distributed algorithms [2].

5.2 Ordering the resources

The graph G that underlies all our resource-sharing computations has one vertex per process and undirected edges connecting any two processes with the potential to share at least one resource. The undirected graph we introduce now and use throughout the end of the section is, by contrast, built on resources for vertices, and has undirected edges connecting pairs of resources that are potentially shared by at least one process. This graph is denoted by $H = (\mathcal{R}, \mathcal{E})$, where $\mathcal{E}$ contains an edge between R_p and R_q if and only if $\mathcal{P}_{pq} \neq \emptyset$. H is a connected graph (because G is connected) and contains a clique on $\mathcal{R}_i$ for $1 \leq i \leq n$. The graph H for Example 1 is shown in Figure 9.

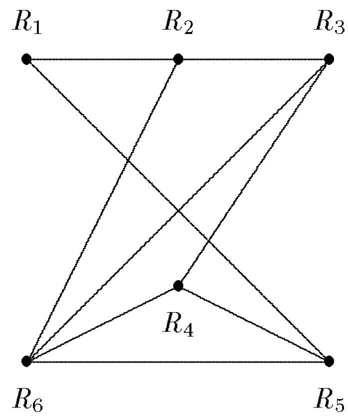


Figure 9: The graph H for Example 1

Our interest in graph H comes from the possibility of constructing a poset on its vertices by orienting its edges acyclically, similarly to what we did previously on G. More specifically, let φ be an acyclic orientation of H, and for $R_p, R_q \in \mathcal{R}$ say that R_p *precedes* R_q if and only if a directed path exists from R_p to R_q in H oriented by φ. One acyclic orientation φ for the graph of Figure 9 is given in Figure 10. Note that the resources in $\mathcal{R}_i$ for any $P_i \in \mathcal{P}$ are necessarily totally ordered by the "precedes" relation.

Now consider the following resource-sharing computation. When a process needs access to a group of shared resources, it sends requests to the neighbors with which it shares those resources according to the partial order implied by φ. The rule to be followed is simple: a process only sends requests for a resource R_p after all grants have been received for the resources that it needs and that precede R_p. The sending of grant responses to requests for a particular resource R_p is regulated by an $O(|\mathcal{P}_p|)$-time distributed procedure on the vertices belonging to the clique in G that corresponds to that resource. This procedure for the acquisition of a single resource must itself be deadlock- and starvation-free [2, 25].

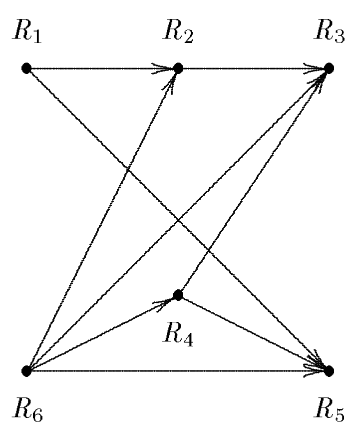


Figure 10: The graph H for Example 1 oriented acyclically

Because φ is acyclic, the evolving wait-for graph W can never contain a directed cycle, so by Fact 1 no deadlocks ever occur. The absence of directed cycles in W comes from the fact that such a cycle would imply a "hold-and-wait" cyclic arrangement of the processes, which is precluded by the acyclicity of φ. We call these resource-sharing computations *acquisition-order computations*, for which the following holds.

Theorem 6 [19] *Every acquisition-order computation is deadlock-free.*

In addition to the safety guarantee given by Theorem 6, and similarly to the case of edge-reversal computations, for acquisition-order computations it is also the case that liveness guarantees can be given. In this case, however, liveness does not come from the shortening distance to sinks in evolving acyclic orientations, but rather from the fact that directed distances as given by an acyclic orientation are always bounded.

In order to be more specific regarding the liveness of acquisition-order computations, let us consider a *coloring* of the vertices of H. Such a coloring is an assignment of colors (natural numbers) to vertices in such a way that neighbors in H get different colors. If H can be colored with c colors for some $c > 0$, then we say that it is *c-colorable* [6].

Lemma 2 [12] *If H is c-colorable, then there exists an acyclic orientation of H according to which the longest directed path in H has no more than $c - 1$ edges.*

If process P_i is the only one to be requesting resources in an acquisition-order computation, then it waits for resources no longer than is implied by the size of $\mathcal{R}_i$, the vertex set of a clique in H. So its wait is given at most by the longest directed path in H according to the acyclic orientation φ fixed beforehand. If H is known to be c-colorable, then by Lemma 2 P_i's wait is bounded from above by c. When other processes are also requesting resources, then let h be the maximum $|\mathcal{P}_p|$ for $1 \leq p \leq m$, that is, the maximum number of processes that may use a resource (this is the size of a clique in G). We have the following.

Theorem 7 [19] *Every acquisition-order computation is starvation-free, and, if H is c-colorable, then the worst-case wait a process must undergo is $O(ch^c)$.*

In Figure 9, the assignment of color 0 to R_3 and R_5, color 1 to R_2 and R_4, and color 2 to R_1 and R_6, makes the graph 3-colorable. One of the acyclic orientations complying with Lemma 2 is the one shown in Figure 10.

Note, in all this discussion, that it must be known beforehand that H is c-colorable so that φ can be built. Also, by Theorem 7, it is to one's advantage to seek as low a value of c as can be efficiently found. Seeking the optimal value of c is equivalent to computing the graph's *chromatic number* (the minimum number of colors with which the graph can be colored) [6], and constitutes an *NP*-hard problem [15].

6 The graph abacus

In this section, we return to the edge-reversal computation discussed in Section 5.1 and consider the following generalization thereof. Associated with each process P_i is an integer $r_i > 0$ to be used to control the dynamic evolution of priorities as given by the succession of acyclic orientations of G. These numbers are to be used in such a way that, as the computation progresses, and for any two neighbors P_i and P_j in G, the ratio of the number of times P_i has priority over P_j to the number of times P_j has priority over P_i "converges" to r_j/r_i in the long run [4, 14]. The special case of Section 5.1 is obtained by setting r_i to the same number for all $P_i \in \mathcal{P}$. In that case, neighbors always have alternating priorities and the ratio is therefore 1.

We refer to computations with this generalized control of priorities as *bead-reversal computations*, in allusion to the following implementation, which views G's edges as the rods along which the beads of a generalized abacus (a graph abacus) are slid back and forth. For $(P_i, P_j) \in \mathcal{C}$, let e_{ij} beads be associated with edge (P_i, P_j). In order for P_i to have priority over P_j, there has to exist at least r_i beads on P_i's side of the edge and strictly less than r_j on P_j's side. When this is the case, the change in priority is performed by moving r_i of those beads towards P_j.

In an bead-reversal computation, the rule for process P_i is the following. Upon terminating its use of the shared resources, send r_i beads to the other end of every edge on which at least r_i beads are on P_i's side. Just as with edge-reversal computations, it is possible to associate an orientation of G's edges to the priority scheme of bead-reversal computations. For such, an edge is oriented towards P_i if and only if there are at least r_i beads on P_i's side of the edge. In order to preserve the syntactic constraints that an edge must be amenable to orientation towards any of the two possible directions, and that it has to be oriented in exactly one direction at any time, we must clearly have

$$\max\{r_i, r_j\} \leq e_{ij} \leq r_i + r_j - 1.$$

But it is possible to obtain a precise value for e_{ij} within this range, and also to come up with a criterion for an initial distribution of the beads along the edges of G in such a way as to provide the desired safety and liveness guarantees. Safety is in this case associated with the acyclicity of the orientations of G as they change, while liveness refers to achieving the desired ratios. As in the case of Section 5.1, we aim at an acyclic wait-for graph W (for deadlock-freedom, by Fact 1). As for liveness, since achieving the desired ratios already implies starvation-freedom, what we aim at are computations for which those ratios are achieved, henceforth called *ratio-compliant*.

We begin with the subgraph G_{ij} of G having for vertices the neighbors P_i and P_j in G, along with the single edge between them. In what follows, g_{ij} is the greatest common divisor of r_i and r_j.

Theorem 8 [4] *If $e_{ij} = r_i + r_j - g_{ij}$, then every bead-reversal computation on G_{ij} is deadlock-free and ratio-compliant.*

Theorem 8 makes no provisions as to the distribution of the e_{ij} beads on the single edge of G_{ij}, and does as such hold for any of the $(r_i + r_j)/g_{ij}$ possible distributions, as we see in Figure 11. In that figure, $r_i = 2$ and $r_j = 3$ (this is indicated in parentheses by the vertices' identifications), and an evolution of bead placements is shown from left to right. For each configuration, the number of beads on each of the edge's end is indicated by small numbers. The corresponding orientation of the edge is also shown. Note that all five possible distributions of beads appear, and that from the last one we return to the first. When we consider the entirety of G, however, the question of how to place the beads on G's edges becomes crucial.

We begin the analysis of the general case by introducing some additional notation and definitions. First, let K denote the set of all the simple cycles in G (those with no repeated vertices). Membership of vertex P_i in $\kappa \in$ K is denoted by $P_i \in \kappa$, and membership of edge (P_i, P_j) in $\kappa \in$ K is denoted by $(P_i, P_j) \in \kappa$. Now, for $\kappa \in$ K, let κ^+ and κ^- denote the two possible traversal directions of κ, chosen arbitrarily. We use a_{ij}^+ to denote the number of beads placed on edge (P_i, P_j) on its far end as it is traversed in the κ^+ direction, and a_{ij}^- likewise for the κ^- direction. Obviously, at all times we have $a_{ij}^+ + a_{ij}^- = e_{ij}$.

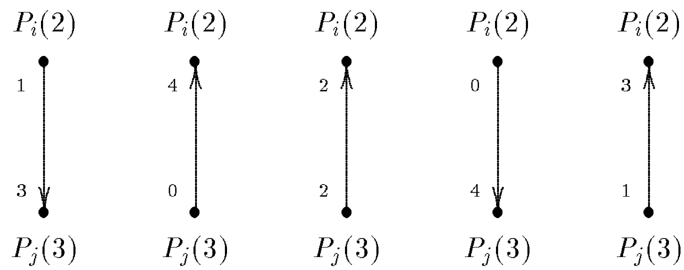


Figure 11: Bead reversal on G_{ij}

For $\kappa \in \mathrm{K}$, let

$$\rho(\kappa) = \sum_{P_i \in \kappa} r_i$$

and

$$\sigma(\kappa) = \max \left\{ \sum_{(P_i,P_j)\in\kappa} a^+_{ij}, \sum_{(P_i,P_j)\in\kappa} a^-_{ij} \right\}.$$

According to these equations, $\rho(\kappa)$ is the sum of r_i over all vertices P_i of κ, while $\sigma(\kappa)$ denotes the total number of beads found on κ as it is traversed in the κ^+ direction or along the κ^- direction, whichever is greatest. In addition, it is easy to see that both $\rho(\kappa)$ and $\sigma(\kappa)$ are time-invariant. We are now ready to state the counterpart of Theorem 8 for G as a whole.

Theorem 9 [4] *If $e_{ij} = r_i + r_j - g_{ij}$ for all $(P_i, P_j) \in \mathcal{C}$ and $\sigma(\kappa) < \rho(\kappa)$ for all $\kappa \in \mathrm{K}$, then every bead-reversal computation is deadlock-free and ratio-compliant.*

One interesting special case that can be used to further our insight into the dynamics of bead-reversal computations is the case of graphs without (undirected) cycles, that is, cases in which G is a tree. In such cases, $\mathrm{K} = \emptyset$ and Theorems 8 and 9 become equivalent to each other.

To finalize this section, we return to the wait-for graph W to analyze its acyclicity. As in the case of edge-reversal computations (cf. Section 5.1), W is a subgraph of an oriented version of G. In the case of bead-reversal computations, the orientation of edge (P_i, P_j) is from P_i towards P_j if and only if there are at least r_j beads placed on the P_j end of the edge (and, necessarily, fewer than r_i on P_i's end). If, as in Section 5.1, we regard W as the graph the results when processes cannot send grant messages, then the directed edges of W coincide with those of G oriented as we just discussed. What this means is that Theorem 9 is an indirect statement on the acyclicity of W: if a directed cycle exists in W, then obviously for the corresponding underlying κ we have $\sigma(\kappa) \geq \rho(\kappa)$, which characterizes an orientation of G that is not acyclic either.

7 The case of heavy loads

A heavy-load situation occurs when, in the resource-sharing computation, processes continually require access to all the shared resources they may have access to, and endlessly go through an acquire-release cycle. Situations such as this bring to the fore interesting issues (some of which will be discussed in Section 8) that are relevant in the context of our discussions in Sections 5.1 and 6, in which we addressed edge-reversal and bead-reversal computations, respectively.

While it is conceivable that, in normal situations, such computations may still be deadlock-free even if the corresponding orientations of G have cycles, the same cannot happen under heavy loads. This is so because what those orientations do is to provide priority. In a light-load regime, a cyclic dependency in the priority scheme may go unnoticed if the pattern of resource demand by the processes happens never to cause a directed cycle in W.

Under heavy loads, on the other hand, the acyclicity of G's orientations is strictly necessary. For bead-reversal computations, in particular, a more strict form of Theorem 9 holds.

Theorem 10 [4] *If $e_{ij} = r_i + r_j - g_{ij}$ for all $(P_i, P_j) \in \mathcal{C}$, then every heavy-load bead-reversal computation is deadlock-free and ratio-compliant if and only if $\sigma(\kappa) < \rho(\kappa)$ for all $\kappa \in \mathrm{K}$.*

The attentive reader will have noticed that violating the inequality of Theorem 10 does not necessarily lead to a directed cycle in G's orientation (or in W). The significance of the theorem, however, is that such a cycle is certain to be created at some time if the inequality is violated. What the theorem does is to provide a criterion for the establishment of initial conditions (bead placement) that is necessary even though at first no cycle might be created otherwise.

Example 5 *Let G be the complete graph on three vertices, and let $r_1 = 1$, $r_2 = 2$, and $r_3 = 3$. Then $e_{12} = 2$, $e_{13} = 3$, and $e_{23} = 4$. Employing the same convention as in Figure 11, and identifying the κ^+ direction of traversal with the clockwise direction for the single simple cycle κ, we show in Figure 12 two possibilities for bead placement. The one in part (a) has $\sigma(\kappa) = 5$, while $\sigma(\kappa) = 6$ for part (b). We have $\rho(\kappa) = 6$ for this example, so $\sigma(\kappa) < \rho(\kappa)$ in part (a), whereas $\sigma(\kappa) = \rho(\kappa)$ in part (b). Although both orientations are acyclic, the reader can check easily that the evolution of the bead placement in Figure 12(b) will soon lead to a directed cycle, while for the other acyclicity will be indefinitely preserved. This is, of course, in accordance with Theorem 10.*

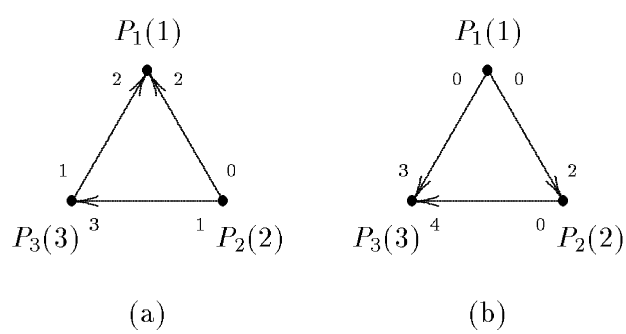


Figure 12: Two possible bead placements

8 Concurrency and graph coloring

From a purely algorithmic perspective, heavy-load situations such as introduced in Section 7 provide a simpler means of implementing edge-reversal and bead-reversal computations than the overall scheme discussed in Section 2. In a heavy-load regime, the need for processes to explicitly request and grant resources becomes moot, because the reversal of priorities (edge orientation or beads) can be taken to signify that permission is granted (or partially granted, in the case of bead reversals) to access shared resources.

Given this simplification, the following is how an edge-reversal computation can be implemented. Start with an acyclic orientation of G. A process computes on shared resources when it is a sink, then reverts all edges adjacent to it and waits to become a sink once again. A bead-reversal computation is similarly simple, as follows. Start with a placement of the beads that not only leads to an acyclic orientation of G but also complies with the inequality prescribed by Theorem 10. A process P_i computes when it is a sink (has enough beads on all adjacent edges), then sends r_i beads to each of its neighbors in G. This is repeated until P_i ceases being a sink, at which time it waits to become a sink again.

Heavy-load situations also raise the question of how much concurrency, or parallelism, there can be in the sharing of resources. While neighbors in G are precluded from sharing resources concurrently, processes that are not neighbors can do it, and how much of it they can do depends on the initial conditions that are imposed on the computation (acyclic orientation of G or bead placement). In the remainder of this section, we discuss this issue of concurrency for edge-reversal computations only, but a similar discussion can be done for bead-reversal computations as well [4].

The simpler means to carry out this concurrency analysis is to abandon the asynchronous model of computation we have been assuming (cf. Section 2) and to assume full synchrony instead. In the *fully synchronous* (or simply *synchronous*) model of distributed computation [2], processes are driven by a common global clock that issues ticks represented by the integer $s \geq 0$. At each tick, processes compute and send messages to their neighbors, which are assumed to get those messages before the next tick comes by.

An edge-reversal computation under the synchronous model is an infinite succession of acyclic orientations of G. If these orientations are $\omega_0, \omega_1, \ldots$, then, for $s \geq 0$, ω_{s+1} is obtained from ω_s by turning every sink in ω_s into a *source* (a vertex with all adjacent edges directed outward). The number of distinct acyclic orientations of G is finite, so the sequence $\omega_0, \omega_1, \ldots$ does eventually become periodic, and from this point on it contains an endless repetition of a number of orientations that we denote by $p(\omega_0)$ (this notation is meant to emphasize that the acyclic orientations that are repeated periodically are fully determined by ω_0). Let these $p(\omega_0)$ orientations be called *periodic orientations* from ω_0.

Lemma 3 [5] *The number of times a process is a sink in the periodic orientations from ω_0 is the same for all processes.*

We let $m(\omega_0)$ denote the number asserted by Lemma 3, and let $m_i(s)$ denote the number of times process P_i is a sink in the subsequence $\omega_0, \ldots, \omega_s$.

Intuitively, it should be obvious that the amount of concurrency achieved from the initial conditions given by ω_0 depends chiefly on the periodic repetition that is eventually reached. In order to make this more formal, let $Conc(\omega_0)$ denote this amount of concurrency, and define it as

$$Conc(\omega_0) = \lim_{s \to \infty} \frac{1}{sn} \sum_{P_i \in \mathcal{P}} m_i(s).$$

That is, we let the concurrency from ω_0 be the average, taken over time and over the number of processes, of the total number of sinks in the sequence $\omega_0, \ldots, \omega_s$ as $s \to \infty$ (the existence of this limit, which is implicitly assumed by the definition of $Conc(\omega_0)$, is only established in what follows, so the definition is a little abusive for the sake of notational simplicity).

Theorem 11 [5] $Conc(\omega_0) = m(\omega_0)/p(\omega_0)$.

Theorem 11 characterizes concurrency in a way that emphasizes the dynamics of edge-reversal computations under heavy loads. But the question that still remains is whether a characterization of concurrency exists that does not depend on the dynamics to be computed, but rather follows from the structure of G as oriented by ω_0.

This question can be answered affirmatively, and for that we consider once again the set K of all simple cycles in G. For $\kappa \in \mathrm{K}$, we let $c^+(\kappa, \omega_0)$ be the number of edges in κ that are oriented by ω_0 in one of the two possible traversal directions of κ. Likewise for $c^-(\kappa, \omega_0)$ in the other direction. The number of vertices in κ is denoted by $|\kappa|$.

Theorem 12 [5] *If G is a tree, then $Conc(\omega_0) = 1/2$; otherwise, then*

$$Conc(\omega_0) = \min_{\kappa \in \mathrm{K}} \frac{\min\{c^+(\kappa, \omega_0), c^-(\kappa, \omega_0)\}}{|\kappa|}.$$

Except for the case of trees, by Theorems 11 and 12 we know that the amount of concurrency of an edge-reversal computation is entirely dependent upon ω_0, the initial acyclic orientation. The problem of determining the ω_0 that maximizes concurrency is, however, *NP*-hard, so an exact efficient procedure to do it is unlikely to exist in general [5].

Example 6 *When G is a ring on five vertices, we have a representation of the original dining philosophers problem [13]. For this case, consider the sequence of acyclic orientations depicted in Figure 13, of which any one can be taken to be ω_0. We have $m(\omega_0) = 2$, $p(\omega_0) = 5$, and $Conc(\omega_0) = 2/5$. This concurrency value follows from either Theorem 11 or Theorem 12.*

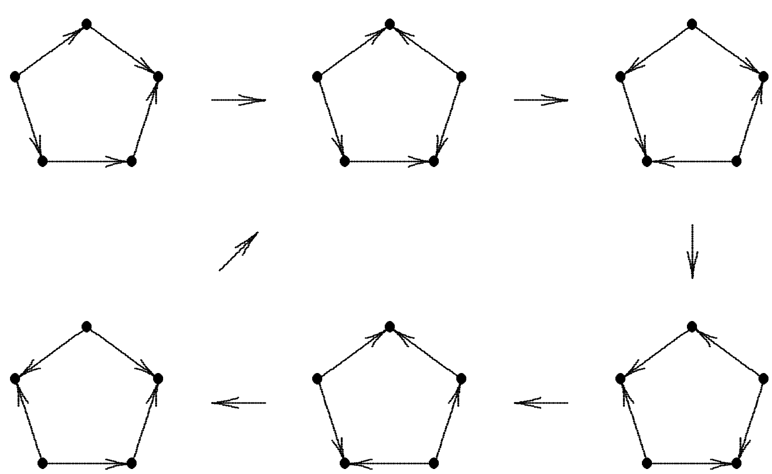


Figure 13: A heavy-load case of edge reversal

Another interesting facet of this concurrency issue is that it relates closely to various forms of coloring the vertices of G. Consider, for example, the *k-tuple coloring* of the vertices of G obtained as follows [24]. Assign k distinct colors to each vertex in such a way that no two neighbors share a color. This type of coloring generalizes the coloring discussed in Section 5.2, for which $k = 1$. The minimum number of colors required to provide G with a k-tuple coloring is its *k-chromatic number*.

In the context of edge-reversal computations, note that the choice of an initial acyclic orientation ω_0 implies, by Lemma 3, that G admits an $m(\omega_0)$-tuple coloring requiring a total of $p(\omega_0)$ colors. If these colors are natural numbers, then neighbors in G get colors that are "interleaved," in the following sense. For two neighbors P_i and P_j, let $c_i^1, \ldots, c_i^z$ and $c_j^1, \ldots, c_j^z$ be their colors, respectively, with $z = m(\omega_0)$. Then either $c_i^1 < c_j^1 < \cdots < c_i^z < c_j^z$ or $c_j^1 < c_i^1 < \cdots < c_j^z < c_i^z$.

So the question of maximizing concurrency is, by Theorem 11, equivalent to the question of minimizing the ratio of the total number of interleaved colors to the number of colors per vertex (this is the ratio $p(\omega_0)/m(\omega_0)$) by choosing ω_0 appropriately. The optimal ratio thus obtained, denoted by $\bar{\chi}(G)$, is called the *interleaved multichromatic* (or *interleaved fractional chromatic*) *number of G* [5]. When the interleaving of colors is not an issue, then what we have is the graph's *multichromatic* (or *fractional chromatic*) *number* [22].

Letting $\chi(G)$ denote the chromatic number of G and $\chi^*(G)$ its multichromatic number, we have

$$\chi^*(G) \leq \bar{\chi}(G) \leq \chi(G).$$

A graph G is shown in Figure 14 for which $\chi^*(G) = 5/2$, $\bar{\chi}(G) = 8/3$, and $\chi(G) = 3$, all distinct therefore. One of the orientations that correspond to $\bar{\chi}(G) = 8/3$ is the one shown in the figure.

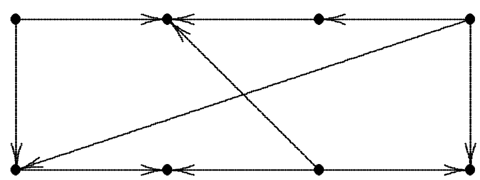


Figure 14: A graph G for which $\chi^*(G) < \bar{\chi}(G) < \chi(G)$

9 Concluding remarks

Distributed computations over shared resources are complex, asynchronous computations. Performing such computations efficiently while offering a minimal set of guarantees has been a challenge for several decades. At present, though problems still persist, we have a clear understanding of several of the issues involved and have met successfully many of those challenges.

Crucial to this understanding has been the use of precise modeling tools, aiming primarily at clarifying the timing issues involved, as well as the combinatorial structures that underlie most of concurrent computations. In this paper, we have concentrated on the latter and outlined some of the most prominent combinatorial concepts on which the design of resource-sharing computations is based. These have included graph structures and posets useful for handling the safety and liveness issues that appear in those computations, and for understanding the questions related to concurrency.

Acknowledgements

The author is thankful to Mario Benevides and Felipe França for many fruitful discussions on the topics of this paper. Support has been provided by the Brazilian agencies CNPq and CAPES, the PRONEX initiative of Brazil's MCT under contract 41.96.0857.00, and by a FAPERJ BBP grant.

References

[1] V. C. Barbosa, "Concurrency in systems with neighborhood constraints," Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles, CA, 1986.

[2] V. C. Barbosa, *An Introduction to Distributed Algorithms*, The MIT Press, Cambridge, MA, 1996.

[3] V. C. Barbosa and M. R. F. Benevides, "A graph-theoretic characterization of AND-OR deadlocks," Technical report ES-472/98, COPPE/UFRJ, Rio de Janeiro, Brazil, July 1998.

[4] V. C. Barbosa, M. R. F. Benevides, and F. M. G. França, "Sharing resources at nonuniform access rates," Technical report ES-412/96, COPPE/UFRJ, Rio de Janeiro, Brazil, November 1996.

[5] V. C. Barbosa and E. Gafni, "Concurrency in heavily loaded neighborhood-constrained systems," *ACM Trans. on Programming Languages and Systems* **11** (1989), 562–584.

[6] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*, North-Holland, New York, NY, 1976.

[7] G. Bracha and S. Toueg, "Distributed deadlock detection," *Distributed Computing* **2** (1987), 127–138.

[8] 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.

[9] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Trans. on Computer Systems* **3** (1985), 63–75.

[10] K. M. Chandy and J. Misra, "The drinking philosophers problem," *ACM Trans. on Programming Languages and Systems* **6** (1984), 632–646.

[11] K. M. Chandy, J. Misra, and L. M. Haas, "Distributed deadlock detection," *ACM Trans. on Computer Systems* **1** (1983), 144–156.

[12] R. W. Deming, "Acyclic orientations of a graph and chromatic and independence numbers," *J. of Combinatorial Theory B* **26** (1979), 101–110.

[13] E. W. Dijkstra, "Cooperating sequential processes," in: F. Genuys (ed.), *Programming Languages*, Academic Press, New York, NY, 1968, 43–112.

[14] F. M. G. França, "Neural networks as neighbourhood-constrained systems," Ph.D. Thesis, Imperial College, London, UK, 1994.

[15] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, New York, NY, 1979.

[16] R. C. Holt, "Some deadlock properties of computer systems," *ACM Computing Surveys* **4** (1972), 179–196.

[17] E. Knapp, "Deadlock detection in distributed databases," *ACM Computing Surveys* **19** (1987), 303–328.

[18] A. D. Kshemkalyani and M. Singhal, "Efficient detection and resolution of generalized distributed deadlocks," *IEEE Trans. on Software Engineering* **20** (1994), 43–54.

[19] N. A. Lynch, "Upper bounds for static resource allocation in a distributed system," *J. of Computer and System Sciences* **23** (1981), 254–278.

[20] J. Misra and K. M. Chandy, "A distributed graph algorithm: knot detection," *ACM Trans. on Programming Languages and Systems* **4** (1982), 678–686.

[21] 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.

[22] E. R. Scheinerman and D. H. Ullman, *Fractional Graph Theory: A Rational Approach to the Theory of Graphs*, Wiley, New York, NY, 1997.

[23] M. Singhal, "Deadlock detection in distributed systems," *IEEE Computer* **22** (1989), 37–48.

[24] S. Stahl, "n-tuple colorings and associated graphs," *J. of Combinatorial Theory B* **20** (1976), 185–203.

[25] J. L. Welch and N. A. Lynch, "A modular drinking philosophers algorithm," *Distributed Computing* **6** (1993), 233–244.