Efficiently Verifiable Sufficient Conditions for Deadlock-freedom of Large Concurrent Programs (Technical Report)

Paul C. Attie and Hana Chockler
College of Computer Science, Northeastern University,
Cullinane Hall, 360 Huntington Avenue,
Boston, Massachusetts 02115.
Email: {attie,hanac}@ccs.neu.edu

May 17, 2004

Abstract

We present two polynomial-time algorithms for automatic verification of deadlock-freedom of large finite-state concurrent programs. We consider shared-memory concurrent programs in which a process can nondeterministically choose amongst several (enabled) actions at any step. Our algorithms are sound but incomplete: if they return a positive answer, then the program is indeed deadlock-free, while a negative answer conveys no information: the program might be deadlock free, but "fails" the test that our algorithms apply.

Our algorithms apply to programs which are expressed in a particular syntactic form, in which variables are shared between pairs of processes, and the synchronization code for each pair of interacting processes is expressed separately from synchronization code for other (even intersecting) pairs of processes. The first algorithm is an improvement of the deadlock check of [2]. Its complexity is polynomial in all its parameters, as opposed to the exponential complexity of the deadlock check of [2]. It also scales well and can be applied to very large concurrent programs. The second algorithm involves a conceptually new construction of a global wait-for graph for all processes. Its running time is also polynomial in all its parameters, and it is more discriminating than the first algorithm. The algorithms apply to systems with similar and with dissimilar processes.

We illustrate our algorithms by applying them to several examples of concurrent programs that implement mutual exclusion and priority queues. To the best of our knowledge, this is the first work that describes polynomially checkable conditions for assuring deadlock freedom of systems of similar and dissimilar processes.

1 Introduction

One of the important correctness properties of concurrent programs is the absence of *deadlocks*, e.g. as defined in [19]: "a set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause." Most approaches to deadlock assume that the "event" that each process waits for is the release of a resource held by another process. We refer to this setting as the *resource allocation* setting. Four conditions are necessary for a deadlock to arise [5, 13]: (1)

resources can be held by at most one process, (2) processes can hold some resources while waiting to acquire several (more than 1, in general) others, (3) resources cannot be taken away from a process (no preemption), (4) a cyclical pattern of waiting amongst the involved processes. The exact pattern of waiting required to cause a deadlock depends on the specific resource model, and can be depicted in terms of a *wait-for-graph* (WFG), i.e, a graph whose edges depict the "wait-for" relationships between processes. The following models have been formulated [14]: (1) AND model: a process blocks iff one or more of the resources it has requested are unavailable, (2) OR model: a process blocks iff all of the resources it has requested are unavailable, (3) AND-OR model: a process can use any combination of AND and OR operators in specifying a resource request, (4) k-out-of-n: a process requests any k resources out of a pool of n resources. For the AND-model, deadlock arises if the WFG contains a cycle. For the OR-model, deadlock arises if the WFG contains a knot, i.e., a set of processes each of which can reach exactly all the others by traversing wait-for edges. To our knowledge, no graph-theoretic construct characterizing deadlock in the AND-OR or the k-out-of-n models is known [14].

In this paper, we address a version of the deadlock problem that is more general than the resource-based model. We consider the deadlock problem in the case that the event which each process waits for is the truthification of a predicate over shared state. Thus, we deal with a shared variables model of concurrency. However, our approach is applicable in principle to other models such as message passing or shared events. We exploit the representation of concurrent programs in a form where the synchronization between processes can be factored out, so that the synchronization code for each pair of interacting processes is expressed separately from that for other pairs, even for two pairs that have a process in common. This "pairwise" representation was introduced in [2], where it was used to synthesize programs efficiently from CTL specifications.

Traditionally, three approaches to dealing with deadlock have been investigated: (1) deadlock detection and recovery: since a deadlock is stable, by definition, it can be detected and then broken, e.g., by the preemption, rollback, or termination of an involved process. (2) deadlock avoidance: avert the occurrence of a deadlock by taking appropriate action. Deadlock avoidance algorithms have been devised for the resource-based formulation of deadlock [19], (3) deadlock prevention: prevent a deadlock from arising by design. In particular, attempt to negate one of the four conditions mentioned above for the occurrence of deadlock. As Tanenbaum [19] observes, attempting to negate any of the first three conditions is usually impractical, and so we are left with condition (4): a cyclical pattern of waiting.

Related work As shown in [15], deadlock-freedom analysis is NP-complete even for simple systems. Therefore, the research in this area concentrated either on the search for efficiently checkable sufficient conditions for deadlock-freedom, or on improving the complexity of the check in some special cases.

There are two approaches in the literature for preventing the cyclical waiting in concurrent systems. One approach is presented in [2] for systems of similar processes. Essentially, this approach concentrates on checking sufficient conditions for assuring deadlock freedom. The conditions should be efficiently checkable. Attie and Emerson [2] formulate a condition, checkable in subsystems consisting of $n + 2$ processes, where n is the maximal branching degree of a state-node in a state-transition graph that represents the behavior of processes (essentially, n reflects the degree of non-determinism of a single process in a state). This condition implies absence of deadlocks in the whole system. The size of the subsystems that have to be checked is exponential in n, which makes checking the condition infeasible for systems that allow a high degree of non-determinism. Most model checking algorithms can be applied to verifying deadlock freedom. The main impediment is state-explosion. Some approaches to ameliorating state-explosion are by using a partial order instead of an interleaving model [10, 12, 11, 17], or by using symmetry reductions [9], These approaches however have worst case running time exponential in the number of processes in a system.

Our Contribution In this paper we follow the first approach to prevention of deadlock. We present two sufficient conditions for assuring deadlock freedom and describe efficient algorithms for checking these conditions. The first condition is suitable for concurrent programs consisting of similar processes, while the second applies to arbitrary concurrent programs. Essentially, the first condition is a modification of the condition presented in [2] and studies the "wait-for-graph" for presence of supercycles (supercycles represent cyclical waiting patterns). The check for the condition presented in [2] has run-time complexity, which is exponential in the branching degree of state-nodes in the "wait-for-graph". In this paper, we present a condition that can be verified in polynomial time in all the input parameters: the number of processes, the size of a single process, and the branching degree of state-nodes. Moreover, the space complexity of the check is polynomial in the size of a single process. Therefore, this condition can be efficiently checked even on very large concurrent programs. We note that the condition in [2] is formulated for systems of similar (isomorphic) processes. However, it is easy to see that the restriction to similar processes does not play any role in the proof of correctness, and thus can be lifted.

The second condition is more complex, and also more discriminating. This condition is based on studying the global wait-for graph that presents the wait-for relations between all local states. We note that, unlike the previous work on deadlock detection, the graph contains only local states, and thus its size is always polynomial in the number of processes and in the size of a single process. We prove that deadlock results in a cyclic waiting condition in the graph (aka "supercycle"). However, not all supercycles correspond to reachable deadlocked states. A supercycle can result from an inconsistent or unreachable state, and thus presence of supercycles alone is not sufficient to declare a deadlock. We present an efficient algorithm for detection of supercycles in the global wait-for graph and for checking the discovered supercycles for inconsistency and reachability. For each supercycle discovered in the graph, we present an efficient algorithm for detection of inconsistencies and unreachability.

To the best of our knowledge, this is the first work that describes sufficient and polynomially checkable conditions for deadlock avoidance in systems of similar and dissimilar processes. We have implemented our pairwise representation using the XSB logic programming system [18]. This implementation provides a platform for implementing the algorithms in this paper.

2 Technical Preliminaries

2.1 Model of concurrent computation

We consider finite-state concurrent programs of the form $P = P_1 \| \cdots \| P_K$ that consist of a finite number n of fixed sequential processes $P_1, \ldots, P_K$ running in parallel. Each P_i is a *synchronization skeleton* [8], that is, a directed multigraph where each node is a (local) state of P_i (also called an i-*state* and is labeled by a unique name (s_i), and where each arc is labeled with a guarded command [7] $B_i \rightarrow A_i$ consisting of a guard B_i and corresponding action A_i. With each P_i we associate a set $\mathcal{AP}_i$ of *atomic propositions*, and a mapping V_i from local states of P_i to subsets of $\mathcal{AP}_i$: $V_i(s_i)$ is the set of atomic propositions that are true in s_i. As P_i executes transitions and changes its local state, the atomic propositions in $\mathcal{AP}_i$ are updated. Different local states of P_i have different truth assignments: $V_i(s_i) \neq V_i(t_i)$ for $s_i \neq t_i$.

Atomic propositions are not shared: $\mathcal{AP}_i \cap \mathcal{AP}_j = \emptyset$ when $i \neq j$. Other processes can read (via guards) but not update the atomic propositions in $\mathcal{AP}_i$. There is also a set of shared variables $x_1, \ldots, x_m$, which can be read and written by every process. These are updated by the action A_i. A *global state* is a tuple of the form $(s_1, \ldots, s_K, v_1, \ldots, v_m)$ where s_i is the current local state of P_i and $v_1, \ldots, v_m$ is a list giving the current values of $x_1, \ldots, x_m$, respectively. A guard B_i is a predicate on global states, and so can reference any atomic proposition and any shared variable. An action A_i is a parallel assignment statement that updates the shared variables. We write just A_i for *true* $\to A_i$ and just B_i for $B_i \to skip$, where $skip$ is the empty assignment.

We model parallelism as usual by the nondeterministic interleaving of the "atomic" transitions of the individual processes P_i. Let $s = (s_1, \ldots, s_i, \ldots, s_K, v_1, \ldots, v_m)$ be the current global state, and let P_i contain an arc from node s_i to s'_i labeled with $B_i \to A_i$. We write such an arc as the tuple $(s_i, B_i \to A_i, s'_i)$, and call it a P_i-*move* from s_i to s'_i. We use just *move* when P_i is specified by the context. If B_i holds in s, then a permissible next state is $s' = (s_1, \ldots, s'_i, \ldots, s_K, v'_1, \ldots, v'_m)$ where $v'_1, \ldots, v'_m$ are the new values for the shared variables resulting from action A_i. Thus, at each step of the computation, a process with an enabled arc is nondeterministically selected to be executed next. The *transition relation* R is the set of all such (s, s'). The arc from node s_i to s'_i is *enabled* in state s. An arc that is not enabled is *blocked*.

Let S^0 be a given set of initial states in which computations of P can start. A *computation path* is a sequence of states whose first state is in S^0 and where each successive pair of states is related by R. A state is *reachable* iff it lies on some computation path. Let S be the set of all reachable global states of P, and redefine R to restrict it to $S \times S$, i.e, to reachable states. Then, $M = (S^0, S, R)$ is the *global state transition diagram* (GSTD) of P.

2.2 Pairwise normal form

We will restrict our attention to concurrent programs that are written in a certain syntactic form, as follows. Let $\oplus, \otimes$ be binary infix operators. A *general guarded command* [2] is either a guarded command as given in Section 2.1 above, or has the form $G_1 \oplus G_2$ or $G_1 \otimes G_2$, where G_1, G_2 are general guarded commands. Roughly, the operational semantics of $G_1 \oplus G_2$ is that either G_1 or G_2, but not both, can be executed, and the operational semantics of $G_1 \otimes G_2$ is that both G_1 or G_2 must be executed, that is, the guards of both G_1 and G_2 must hold at the same time, and the bodies of G_1 and G_2 must be executed simultaneously, as a single parallel assignment statement. For the semantics of $G_1 \otimes G_2$ to be well-defined, there must be no conflicting assignments to shared variables in G_1 and G_2. This will always be the case for the programs we consider. We refer the reader to [2] for a comprehensive presentation of general guarded commands.

A concurrent program $P = P_1 \| \cdots \| P_K$ is in *pairwise normal form* iff the following four conditions all hold: (1) every move a_i of every process P_i has the form $a_i = (s_i, \otimes_{j \in I(i)} \oplus_{\ell \in \{1, \ldots, n_j\}} B^j_{i,\ell} \to A^j_{i,\ell}, t_i)$, where $B^j_{i,\ell} \to A^j_{i,\ell}$ is a guarded command, I is an irreflexive symmetric relation over $\{1 \ldots K\}$ that defines a "interconnection" (or "neighbors") relation amongst processes, and $I(i) = \{j \mid (i,j) \in I\}$, (2) variables are shared in a pairwise manner, i.e., for each $(i,j) \in I$, there is some set $\mathcal{SH}_{ij}$ of shared variables that are the only variables that can be read and written by both P_i and P_j, (3) $B^j_{i,\ell}$ can reference only variables in $\mathcal{SH}_{ij}$ and atomic propositions in $\mathcal{AP}_j$, and (4) $A^j_{i,\ell}$ can update only variables in $\mathcal{SH}_{ij}$.

For each neighbor P_j of P_i, $\oplus_{\ell \in [1:n]} B^j_{i,\ell} \to A^j_{i,\ell}$ specifies n alternatives $B^j_{i,\ell} \to A^j_{i,\ell}$, $1 \leq \ell \leq n$ for the interaction between P_i and P_j as P_i transitions from s_i to t_i. P_i must execute such an interaction with each of its neighbors in order to transition from s_i to t_i. We emphasize that I is not necessarily the set of all pairs, i.e., there can be processes that do not directly interact by reading each others atomic propositions or reading/writing pairwise shared variables. We do not assume, unless otherwise stated, that processes are isomorphic, or *similar* (we define process similarity later in this section).

We will usually use a superscript I to indicate the relation I, e.g., process P_i^I, and P_i^I-move a_i^I. We define $a_i^I.start = s_i$, $a_i^I.guard_j = \bigvee_{\ell \in \{1,\ldots,n_j\}} B_{i,\ell}^j$, and $a_i^I.guard = \bigwedge_{j \in I(i)} a_i.guard_j$. If $P^I = P_1^I \parallel \ldots \parallel P_K^I$ is a concurrent program with interconnection relation I, then we call P^I an *I-system*.

In pairwise normal form, the synchronization code for P_i^I with one of its neighbors P_j^I (i.e., $\oplus_{\ell \in \{1,\ldots,n_j\}} B_{i,\ell}^j \to A_{i,\ell}^j$). is expressed separately from the synchronization code for P_i^I with another neighbor P_k^I (i.e., $\oplus_{\ell \in \{1,\ldots,n_k\}} B_{i,\ell}^k \to A_{i,\ell}^k$) We can exploit this property to define "subsystems" of an I-system P as follows. Let $J \subseteq I$ and $range(J) = \{i \mid \exists j : (i,j) \in J\}$. If a_i^I is a move of P_i^I then define $a_i^J = (s_i, \otimes_{j \in J(i)} \oplus_{\ell \in \{1\ldots n\}} B_{i,\ell}^j \to A_{i,\ell}^j, t_i)$. Then the *$J$-system* P^J is $P_{j_1}^J \parallel \ldots \parallel P_{j_n}^J$ where $\{j_1, \ldots, j_n\} = range(J)$ and P_j^J consists of the moves $\{a_i^J \mid a_I^I$ is a move of $P_j^I\}$. Intuitively, a J-system consists of the processes in $range(J)$, where each process contains only the synchronization code needed for its J-neighbors, rather than its I-neighbors. If $J = \{\{i,j\}\}$ for some i,j then P_J is a *pair-system*, and if $J = \{\{i,j\},\{j,k\}\}$ for some i,j,k then P_J is a *triple-system*. For $J \subseteq I$, $M_J = (S_J^0, S_J, R_J)$ is the GSTD of P^J as defined in Section 2.1, and a global state of P^J is a *J-state*. If $J = \{\{i,j\}\}$, then we write $M_{ij} = (S_{ij}^0, S_{ij}, R_{ij})$ instead of $M_J = (S_J^0, S_J, R_J)$.

Also, if s_J is a J-state, and $J' \subseteq J$, then $s \upharpoonright J'$ is the J'-state that agrees with s on the local state of all $P_j \in range(J')$ and the value of all variables $x_{ij} \in \mathcal{SH}_{ij}$ such that $i,j \in range(J')$, i.e, the projection of s onto the processes in J'. If $J' = \{\{i,j\}\}$ then we write $s \upharpoonright J$ as $s \upharpoonright ij$. Also, $s \upharpoonright i$ is the local state of P_i in s. Let $reachable(P_i) = \{s_i \mid \exists j \in I(i), s_{ij} \in S_{ij} : s_{ij} \upharpoonright i = s_i\}$, that is, $reachable(P_i)$ is the set of local states of P_i that are reachable in some pair-system involving P_i.

Two processes P_i and P_j are *similar* if they are isomorphic to each other up to the change of indices [2]. A concurrent program $P = P_1 \| \cdots \| P_K$ *consists of similar processes* if for each $1 \leq i,j \leq K$, we have that P_i and P_j are similar.

[2, 1, 3] give, in pairwise normal form, solutions to many well-known problems, such as dining philosophers, drinking philosophers, mutual exclusion, k-out-of-n mutual exclusion, two-phase commit, and replicated data servers. Attie [4] shows that any finite-state concurrent program can be rewritten (up to strong bisimulation) in pairwise normal form. The restriction to pairwise normal form enables us to mechanically verify certain correctness properties very efficiently. For example, safety properties that can be expressed over pairs of processes can be verified in time $O(K^2N^2)$, where K is the number of processes and N is the size of the largest process. Exhaustive state-space enumeration would of course require $O(N^K)$ time. In this paper, we exploit pairwise normal form to devise sufficient conditions for deadlock-freedom that can be checked in time polynomial in both N and K.

2.3 The Wait-For-Graph

The wait-for-graph for an I-state s gives all of the blocking relationships in s.

Definition 2.1 (*Wait-For-Graph $W_I(s)$***)** *Let s be an arbitrary I-state. The* wait-for-graph $W_I(s)$ *of s is a directed bipartite AND-OR graph, where*

1. *The AND nodes of $W_I(s)$ (also called* local-state nodes*) are the i-states $\{s_i \mid s_i = s{\restriction}i, i \in \{1 \dots K\}\}$*[1]*;*
2. *The OR-nodes of $W_I(s)$ (also called* move nodes*) are the moves $\{a_i^I \mid i \in \{1 \dots K\}$ and a_i^I is a move of P_i^I and $s{\restriction}i = a_i^I.start\ \}$*
3. *There is an edge from s_i to every node of the form a_i^I in $W_I(s)$;*
4. *There is an edge from a_i^I to s_j in $W_I(s)$ if and only if $\{i, j\} \in I$ and $a_i^I \in W_I(s)$ and $s{\restriction}ij(a_i^I.guard_j) = false$.*

The AND-nodes are the local states $s_i = s{\restriction}i$ of all processes when the global state is s, and the OR-nodes are the moves a_i^I such that local control in P_i^I is currently at the start state of a_i^I, i.e., all the moves that are candidates for execution. There is an edge from s_i to each move of the form a_i^I. Nodes s_i are AND nodes since P_i^I is blocked if and only if *all* of its possible moves are blocked. There is an edge from a_i^I to s_j iff a_i^I is blocked by P_j^I: a_i^I can be executed in s only if $s{\restriction}ij(a_i^I.guard_j) = true$ for all $j \in I(i)$; if there is some j in $I(i)$ such that $s{\restriction}ij(a_i^I.guard_j) = false$, then a_i^I cannot be executed in state s. The nodes labeled with moves are OR nodes, since a_i^I is blocked iff *some* neighbor P_j^I of P_i^I blocks a_i^I. Note, however, that we cannot say that P_i^I itself is blocked by P_j^I, since there could be another move b_i^I in P_i^I such that $s{\restriction}ij(b_i^I.guard_j) = true$, i.e., b_i^I is not blocked by P_j^I (in state s), so P_i^I can progress in state s by executing b_i^I.

In the sequel, we use $s_i \longrightarrow a_i^I \in W_I(s)$ to denote the existence of an edge from s_i to a_i^I in $W_I(s)$, and $a_i^I \longrightarrow s_j \in W_I(s)$ to denote the existence of an edge from a_i^I to s_j in $W_I(s)$. We also abbreviate $((s_i \longrightarrow a_i^I \in W(s)) \wedge (a_i^I \longrightarrow s_j \in W(s)))$ with $s_i \longrightarrow a_i^I \longrightarrow s_j \in W(s)$, and similarly for longer "wait-chains." For $J \subseteq I$ and J-state s_J we define $W_J(s_J)$ by replacing I by J and $\{1 \dots K\}$ by $range(J)$ in the above definition.

2.4 Establishing Deadlock-freedom: Supercycles

Deadlock is characterized by the presence in the wait-for-graph of a graph-theoretic construct called a *supercycle* [2]:

Definition 2.2 (Supercycle) *SC is a supercycle in $W_I(s)$ if and only if all of the following hold:*

1. *SC is nonempty,*
2. *if $s_i \in SC$ then $\forall a_i^I : a_i^I \in W_I(s)$ implies $s_i \longrightarrow a_i^I \in SC$, and*
3. *if $a_i^I \in SC$ then $\exists s_j : a_i^I \longrightarrow s_j \in W_I(s)$ and $a_i^I \longrightarrow s_j \in SC$.*

Note that SC is a subgraph of $W_I(s)$. If an i-state s_i is in a supercycle SC, then every move of P_i^I that starts in s_i is also in SC and is blocked by some other I-process P_j^I which has a j-state s_j in SC (note that a process has at most one local state in SC, and we say that the process itself is in SC). It follows that no I-process in SC can execute any of its moves, and that this situation persists forever.

In Figure 1 we give an example of a wait-for-graph for a three process system. And-nodes (processes) are shown as $\bullet$, and or-nodes (moves) are shown as $\circ$. Each process P_i, $i \in \{1, 2, 3\}$ has two moves a_i

[1]In [2] state nodes are denoted by processes P_i and not by local states, since they consider wait-for-graphs for each state of the system separately; in this paper, we study wait-for-graphs that encompass all blocking conditions for all local nodes of all processes together; hence we need to distinguish between different local state-nodes of the same process.

and b_i. Since every move has at least one outgoing edge, i.e., is blocked by at least one process, the figure also an example of a supercycle. In fact, several edges can be removed and still leave a supercycle (for example, $a_3 \longrightarrow P_1$, $b_3 \longrightarrow P_2$, $a_2 \longrightarrow P_1$ can all be removed). Thus, the figure contains several subgraphs that are also supercycles.

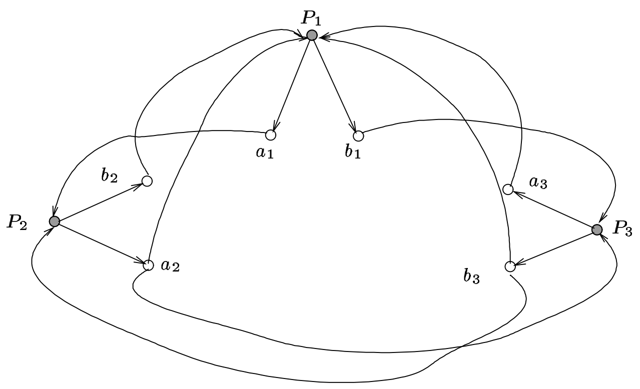


Figure 1: Example of a wait-for-graph.

From [2], we have that the absence of supercycles in the wait-for-graph of a state implies that there is at least one enabled move in that state:

Proposition 2.3 ([2]) *If $W_I(s)$ is supercycle-free, then some move a_i^I has no outgoing edges in $W_I(s)$, and so can be executed in state s.*

We say that s is *supercycle-free* iff $W_I(s)$ does not contain a supercycle. We assume that all initial states of the I-system are supercycle free. That is, we do not allow initial states that contain deadlocks.

3 Improving the deadlock freedom condition from [2]

3.1 The wait-for-graph condition

Consider the following condition:

$$\begin{array}{l} \text{For every reachable } I\text{-state } t \text{ in } M_I \\ \text{such that } s \xrightarrow{k} t \in R_I \text{ for some reachable } I\text{-state } s, \\ \quad \neg\exists a_j^I : (a_j^I \longrightarrow P_k^I \in W_I(t)) \text{ or} \\ \quad \exists a_k^I \in W_I(t) : (\bigwedge_{\ell \in \{1 \ldots K\}} (a_k^I \longrightarrow P_\ell^I \notin W_I(t))). \end{array} \qquad \text{(a)}$$

This condition implies that, after P_k^I executes a transition, either P_k^I blocks no move of another process, or P_k^I itself has an enabled move. Thus P_k^I cannot be in a supercycle. Hence, this transition of P_k^I could not have *created* a supercycle; any supercycle present after the transition must also have been present before the transition. Since initial states are supercycle-free, we conclude, by induction on computation path length, that every reachable I-state is supercycle-free.

Let $t_k.moves$ denote the set of moves $\{a_k^I \mid a_k^I \in P_k^I$ and $a_k^I.start = t_k\}$. It is proved in [2] that it is enough to check condition (a) for all J-systems for $J \in \mathcal{J}$, where $\mathcal{J}$ is the set of all interconnection relations of the form $\{\{j,k\},\{k,\ell_1\},\{k,l_2\},\ldots,\{k,\ell_n\}\}$, where $n = |(t{\upharpoonright}k).moves|$,

$1 \le j, k, \ell_1 \ldots, \ell_n \le K$, $k \notin \{j, k, \ell_1 \ldots, \ell_n\}$. That is, we check:

For every $t_k \in reachable(P_k)$,
for all $J \in \mathcal{J}$ in which there exists a reachable J-state t_J
such that $s_J \xrightarrow{k} t_J$ for some J-reachable state s_J and $t_J{\restriction}k = t_k$,
$\neg \exists a_j^J \,:\, (a_j^J {\longrightarrow} P_k^J \in W_J(t_J))$ or
$\exists a_k^J \in W_J(t_J) \,:\, (\bigwedge_{\ell \in \{l_1,\ldots,l_n\}} (a_k^J {\longrightarrow} P_\ell^J \notin W_J(t_J)))$. (b)

This condition implies an algorithm that checks all possible subsystems J of the form $\{\{j, k\}, \{k, \ell_1\}, \ldots, \{k, \ell_n\}\}$. The algorithm is exponential in n, thus for large n constructing the subsystems is infeasible.

In the condition (b), the variables that appear under the disjunction over the moves of P_k^J in t_k ($\exists a_k^J \in W_J(t_J)$) do not involve a_J^J, and so we can move the disjunction over the moves a_k^J of P_k^J before the conjunction over the moves a_j^J of P_j^J ($\neg\exists a_j^J$). We get the following equivalent condition:

For every $t_k \in reachable(P_k)$,
for all $J \in \mathcal{J}$ in which there exists a reachable J-state t_J
such that $s_J \xrightarrow{k} t_J$ for some J-reachable state s_J and $t_J{\restriction}k = t_k$,
$\exists a_k^J \in W_J(t_J) \,:$
$\neg\exists a_j^J \,:\, (a_j^J {\longrightarrow} P_k^J \in W_J(t_J))$ or $(\bigwedge_{\ell \in \{l_1,\ldots,l_n\}} (a_k^J {\longrightarrow} P_\ell^J \notin W_J(t_J)))$.

Let $J_i = \{\{j, k\}, \{k, \ell_i\}\} \subseteq J$, for $1 \le i \le n$. Then, for each move a_k^J, $\bigwedge_{\ell \in \{\ell_1,\ldots,\ell_n\}} (a_k^J \to P_\ell^J \notin W_J(t_J))$ holds iff

$$\bigwedge_{J_i : 1 \le i \le n} (a_k^{J_i} \to P_{\ell_i}^{J_i} \notin W_{J_i}(t_J{\restriction}J_i)), \tag{1}$$

The last equation follows from wait-for-graph projection [2, Proposition 6.5.4.1].

Equation 1 is checked with respect to all systems of three processes, for all reachable states of the J-system. In order to avoid constructing the J-system, we check the following condition, which requires only construction of the J_i-systems.

For every $t_k \in reachable(P_k)$,
there exists a move $a_k^J \in t_k.moves$
such that for all relations $J_i = \{\{j, k\}, \{k, \ell_i\}\} \subseteq J$
in which there exists a reachable state t_{J_i}
such that $s_{J_i} \xrightarrow{k} t_{J_i}$ for some J_i-reachable state s_{J_i} and $t_{J_i}{\restriction}k = t_k$,
we have $\neg\exists a_j^{J_i} \,:\, (a_j^{J_i} {\longrightarrow} P_k^{J_i} \in W_{J_i}(t_{J_i}))$ or
$(a_k^{J_i} {\longrightarrow} P_{\ell_i}^{J_i} \notin W_{J_i}(t)))$. (c)

In other words, condition (c) holds if either P_k blocks no move of another process or there exists a move of P_k that is not blocked in any of the triple systems J_i.

Theorem 3.1 *If condition (c) holds, then (1) condition (b) holds, and (2) the I-system is deadlock-free.*

Proof: Condition (b) implies deadlock-freedom [2]. Thus, it suffi ces to prove that the condition (c) implies the condition (b). We prove the contrapositive. Assume that M_J does not satisfy condition (b) in a state t_k^J. Then, there exists a move a_j^J of P_j such that $a_i^J \to P_k^J \in W_J(t_J)$, and for each move $a_k^J \in (t_k^J{\restriction}k).moves$, there exists a process P_l^J such that $a_k^J \to P_l^J \in W_J$. Let $\{a_{k,1}^J, \ldots, a_{k,n}^J\} =$

$(t_k^J \lceil k).moves$. For a move $a_{k,i}^J$, let P_{l_i} be the process that blocks it in t_k^J, for $1 \leq i \leq n$. Now we show that condition (c) is violated. Indeed, for each interprocess relation $J_i = \{\{j,k\},\{k,l_i\}\}$, for $1 \leq i \leq n$, we have that $P_k^{J_i}$ blocks the move $a_j^{J_i}$ of the process $P_j^{J_i}$. In addition, for each move $a_{k,i}$ of P_k there exists a relation $J_i = \{\{j,k\},\{k,l_i\}\}$ in which $a_{k,i}$ is blocked. □

Intuitively, checking condition (c) involves constructing all triples of processes with P_k being the middle process. Since the size of a triple system is polynomial in the size of a single process, and the number of triples is polynomial in the number of processes in the system, the check is polynomial in all parameters. Now we describe the algorithm formally and analyze its time and space complexity. For a process P_k, we determine the set S_k of reachable local states of P_k. For every $t_k \in S_k$, we determine the set $t_k.moves$ of outgoing moves of P_k from t_k. The J_i-systems in the wait-for-graph assumption are of the form $J_i = \{\{j,k\},\{k,l_i\}\}$, where $k \neq j$ and $k \neq l_i$. We define the set of all J-systems under consideration as

$$\mathcal{J}(t_k) = \{J_i : J_i = \{\{j,k\},\{k,l_i\}\}, k \neq j, l_i \text{ and } l_i \in \{l_1,\ldots,l_n\}\},$$

where $n = |t_k.moves|$.

For every $J_i \in \mathcal{J}$ we generate the state-transition diagram M_{J_i} and the wait-for-graph $W_{J_i}(t_k^{J_i})$. For each $a_k \in t_k.moves$, we evaluate

$$\bigwedge a_j^{J_i} \,:\, (a_j^{J_I} \longrightarrow P_k^{J_i} \notin W_{J_i}(t_k^{J_i})) \vee (a_k^{J_i} \rightarrow P_{l_i}^{J_i} \notin W_{J_i}(t_k^{J_i})) \qquad (2)$$

for all $J_i \in \mathcal{J}(t_k)$, where $a_k^{J_i} \lceil k = a_k$.

If for all t_k there exists $a_k \in t_k.moves$ for which Equation 2 holds for all $J_i \in \mathcal{J}$, then we conclude that the system is deadlock-free. We formalize the procedure given above in Figure 2, and compute its time complexity. The procedure clearly terminates, since the range of all loop variables is finite. Furthermore, upon termination, condition (c) holds if and only if $WGflag$ is set to “true.” By definition of M_{J_i}, its size is bounded by $O(N^3)$, where N is the size of a single process.

We consider each step of the procedure and compute its worst-case time complexity. The complexity of step 2.3.2.2 is $O(|M_{J_i}|)$, which is $O(N^3)$. The step 2.3.2.2 is performed $|\mathcal{J}| = O(K^3)$ times. The step 2.3.2 is performed $|t_k.moves| = n$ times. Thus, the total runtime complexity is $O(K^3N^3n)$, which is polynomial in the number of processes, the size of a single process, and the branching degree of a process. The space complexity is bounded by the size of a triple system, which is $O(N^3)$.

Example 3.2 [Deadlock detection in the general resource allocation problem] In this example we apply our method to the general resource allocation problem presented in Lynch [16, Chapter 11]. The problem is defined by the *explicit resource specification* for n processes. An explicit resource specification $\mathcal{R}$ for n processes consists of a universal finite set R of (unsharable) resources and sets $R_i \subseteq R$ for all $1 \leq i \leq n$, where R_i is the set of resources that process P_i requires to execute. We describe here an instance of the resource allocation problem in which there is a potential deadlock and show how this deadlock can be detected by studying triples of processes. We assume that each process needs at least one resource in order to execute. We first consider a naive algorithm in which each process chooses the order of requests for resources non-deterministically. That is, if a process P_i needs resources $\{1,\ldots,k\}$, it non-deterministically acquires resource $1 \leq r_1 \leq k$, then a resource $r_2 \in \{1,\ldots,k\} \setminus r_1$, etc. After the last resource has been acquired, P_i executes. Clearly, if a resource r is already allocated to another

```
0. $WGflag := true$;
1. $S_k := reachable(P_k)$;
2. for all $t_k \in S_k$
    2.1 $n := |t_k.moves|$;
    2.2 $\mathcal{J}(t_k) := \{J_i \mid J = \{\{j,k\},\{k,l_i\}\}$;
    2.3 for all $a_k \in t_k.moves$
        2.3.1 reset the value of Equation 2 to true;
        2.3.2 for all $J_i$ in $\mathcal{J}(t_k)$
            2.3.2.1 generate $M_{J_i}$;
            2.3.2.2 evaluate Equation 2; if false, break;
    2.4 if for all $a_k$ the value of Equation 2 is false, set WGflag to false; return;
```

Figure 2: Procedure that checks the wait-for-graph assumption.

process, P_i cannot acquire it. If at some state in the resources allocation all remaining resources are allocated to other processes, P_i cannot proceed.

We show that in the case where $R_i = R$ for all $1 \leq i \leq n$ (every process needs all resources) and $|r| \leq n$, the system has deadlocked states. The skeleton of a single process P_i is given in Figure 3. The states in which P_i requires resources are labeled with sets of resources already acquired by P_i. For sets $S_1, S_2 \subseteq R$ of resources, there is a move from the state labeled with S_1 to the state labeled with S_2 iff $S_2 = S_1 \cup \{r\}$ for some resource r, and the move is guarded by $\otimes_{j \neq i} x^i_{ijr} = i$. There is a move $S \rightarrow C_i$ iff $S = R$ (that is, a process can move to the state where it executes only after it acquired all its resources).

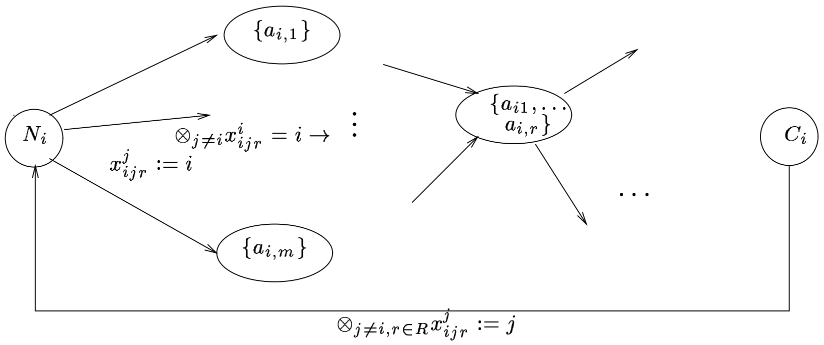

$\{a_{i,1}\}$
N_i
$\otimes_{j \neq i} x^i_{ijr} = i \rightarrow$
$x^j_{ijr} := i$
$\{a_{i1}, \ldots a_{i,r}\}$
C_i
$\{a_{i,m}\}$
$\otimes_{j \neq i, r \in R} x^j_{ijr} := j$

Figure 3: The skeleton of a process in the resource allocation problem

For $1 \leq i, j \leq n$ and $1 \leq r \leq |R|$, the variables x^i_{ijr} are shared variables that are written by P_j and read by P_i only and determine the allocation of the resource r. A process P_i can acquire the resource r only if $x^i_{ijr} = i$ for all $1 \leq j \leq n$. Upon acquiring the resource r, the process P_i sets x^j_{ijr} to i, for all $1 \leq j \leq n$. The state of P_i is determined by the subset of resources P_i currently holds. That is,

P_i is in state $\langle a_{i,1},\ldots,a_{i,l}\rangle$ iff P_i holds the resources $\{r_1,\ldots,r_l\}$. The system has a single initial state $[N_1\,N_2\ldots N_n]$. The process P_i executes in the state C_i for $1\leq i\leq n$. Let $|R|=m$. For simplicity, we assume $n=m$ (if $n>m$ the deadlock follows from the deadlock for $n=m$). The system reaches the deadlocked state $[\langle a_{1,1}\rangle\,\langle a_{2,2}\rangle\ldots\langle a_{m,m}\rangle]$ (i.e., the state in which the process P_i holds the resource r_i for $1\leq i\leq m$ via the following path.

$$\begin{array}{l}
[N_1\,N_2\ldots N_m]\xrightarrow{1}\\
[x^i_{1i1}=1 \text{ for } i\in\{2,\ldots,m\},\ \langle a_{1,1}\rangle\,N_2\ldots N_m]\xrightarrow{2}\\
[x^i_{1i1}=1 \text{ for } i\in\{2,\ldots,m\},\ x^i_{2i2}=2 \text{ for } i\in\{1,\ldots,m\}\setminus\{2\},\ \langle a_{1,1}\rangle\,\langle a_{2,2}\rangle\,N_3\ldots N_m]\xrightarrow{3}\ldots\\
\ldots[x^i_{1i1}=1 \text{ for } i\in\{2,\ldots,m\},\ldots x^r_{mrm}=r \text{ for } r\in\{1,\ldots,m-1\}\ \langle a_{1,1}\rangle\,\langle a_{2,2}\rangle\ldots\langle a_{m,m}\rangle]
\end{array}$$

It is easy to see that the last state is a deadlock state, i.e., has no enabled outgoing moves. Indeed, every process holds exactly one resource, and thus no process can continue the allocation and execute. This scenario arises when the process P_1 requests the fi rst resource, the process P_2 requests the second resource, etc. until the process P_m requests the m-th resource.

The deadlock can be detected by checking condition (c) in the J_{ik}-systems $J_{ik}=\{\{i,m\},\{m,k\}\}$. In the state $[\langle a_{1,1}\rangle\,\langle a_{2,2}\rangle\ldots\langle a_{m,m}\rangle]$ of the J_{ik}-system the process P_m blocks the move $\langle a_{i,i}\rangle\rightarrow\langle a_{i,i},a_{i,m}\rangle$ of P_i (that is, the move that tries to allocate the resource m to P_i), and the process P_k blocks the move $\langle a_{m,m}\rangle\rightarrow\langle a_{m,m},a_{m,k}\rangle$ of P_m by setting the variable x^m_{kmk} to k (that is, holding the resource k). Thus, for each move $\langle a_{m,m}\rangle\rightarrow\langle a_{m,m},a_{m,k}\rangle$ there exists a triple-system J_{ik} in which this move is blocked, and therefore condition (c) fails. We note that the number of processes and the number of resources can be arbitrarily large. Indeed, checking condition (c) requires only constructing triples of processes, and the number of triples is polynomial in the total number of processes in the system.

Now consider the hierarchical resource allocation presented in Lynch [16, Chapter 11]. In this case, there is a global hierarchy between processes, and the resource is acquired to the process with the highest priority that requests it. The system is deadlock-free. However, condition (c) fails, giving a false deadlock indication. The reason for its failure is existence of waiting chains of length three in the system, despite the fact that cyclical waiting pattern never ocurs. In the next section we present a more complex (and more discriminating) algorithm that shows deadlock freedom of hierarchical resource allocation.

We study one more example of deadlock detection, and then we modify the example so that, while condition (c) fails, the system is in fact deadlock-free. In Section 4 we show a more discriminating test, which show deadlock-freedom of the example.

Example 3.3 [Application of our algorithm to detect possible deadlock] Consider the system of K similar processes with a single initial state $[N_1\ldots N_K]$. The skeleton of a single process P_i is given in Figure 4. The processes attempt to enter one of the two critical sections C and D.

For $i,j\in\{1\ldots K\}, i\neq j$, the variables x^i_{ij},y^i_{ij} are shared variables that are written by P_j and read by P_i only. The shared variable x^i_{ij} controls the move of P_i from T_i to C_i, which can be made only when $x^i_{ij}=i$. Likewise, y^i_{ij} controls the move of P_i from T_i to D_i, which can be made only when $y^i_{ij}=i$. When process P_j moves from the state N_j to the state T_j, it non-deterministically chooses how to update the variables x^i_{ij}, y^i_{ij}, for $i\in\{1\ldots K\}\setminus\{j\}$. In the state T_j, if both moves of P_j are enabled, it chooses the next move non-deterministically. For $K=4$, the system can reach a deadlocked state $[T_1,T_2,T_3,T_4]$ via the following path.

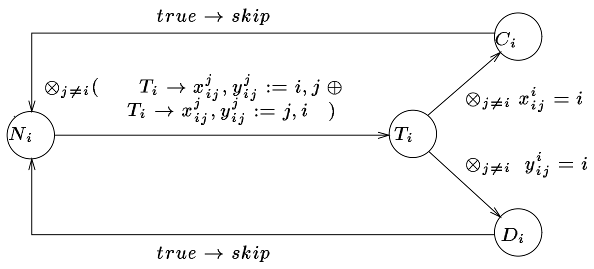

$true \to skip$
C_i
$\otimes_{j\neq i}(\quad T_i \to x^j_{ij}, y^j_{ij} := i, j \oplus$
$T_i \to x^j_{ij}, y^j_{ij} := j, i \quad)$
$\otimes_{j\neq i}\ x^i_{ij} = i$
N_i
T_i
$\otimes_{j\neq i}\ y^i_{ij} = i$
D_i
$true \to skip$

Figure 4: The skeleton of a process in the deadlock example for four processes.

$$
\begin{aligned}
&[N_1\, N_2\, N_3\, N_4] \xrightarrow{1} \\
&[x^i_{1i} = 1 \text{ for } i \in \{2,3,4\},\ T_1\, N_2\, N_3\, N_4] \xrightarrow{2} \\
&[x^i_{1i} = 1 \text{ for } i \in \{2,3,4\},\ y^j_{2j} = 2 \text{ for } j \in \{1,3,4\},\ T_1\, T_2\, T_3\, T_4] \xrightarrow{3} \\
&[x^i_{1i} = 1 \text{ for } i \in \{2,3,4\},\ y^j_{2j} = 2 \text{ for } j \in \{1,3,4\},\ x^k_{3k} = 3 \text{ for } k \in \{1,2,4\},\ T_1\, T_2\, T_3\, N_4] \xrightarrow{4} \\
&[x^i_{1i} = 1 \text{ for } i \in \{2,3,4\},\ y^j_{2j} = 2 \text{ for } j \in \{1,3,4\},\ x^k_{3k} = 3 \text{ for } k \in \{1,2,4\} \\
&\ y^l_{4l} = 4 \text{ for } l \in \{1,2,3\},\ T_1\, T_2\, T_3\, T_4\,]
\end{aligned}
$$

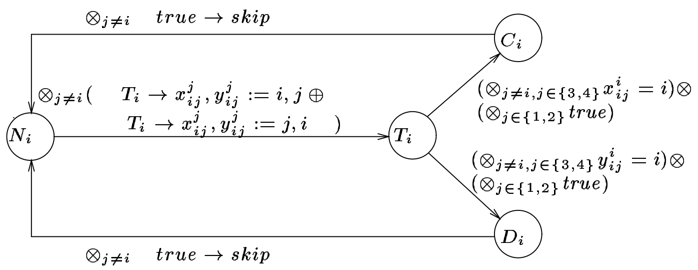
It is easy to see that the last state is a deadlock state, i.e., has no enabled outgoing moves. Indeed, the outgoing move $T_1 \to C_1$ is blocked by the process P_3, and the outgoing move $T_1 \to D_1$ is blocked by the processes P_2 and P_4, and similarly for other processes. This can be detected by checking condition (c) in the J-system $J = \{\{1,4\},\{4,2\}\}$. In the state $[T_1\, T_4\, T_2]$ of this J-system the process P_4 blocks the move $T_1 \to D_1$ of P_1, and the process P_2 can block the move $T_4 \to C_4$ of P_4 by setting the variable x^4_{24} to 2, or the move $T_4 \to D_4$ by setting the variable y^4_{24} to 2. Thus, condition (c) fails for this J-system.

We note that P_2 can block only one move of P_4 at a time. However, since all processes are similar, in the K-process system P_2 can block one move and another process can block another move of P_4, so that this deadlock scenario applies for any number $K \geq 4$ of processes.

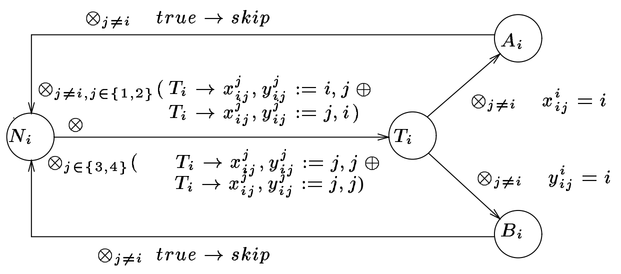
The requirement of processes similarity is therefore crucial. In Section 3.4 we describe an example of a system with non-similar processes and show that in this case we get a false deadlock indication from checking condition (c).

Example 3.4 [False positive deadlock indication in a system with dissimilar processes] In this example we describe a situation in which the deadlock-freedom check described in Section 3 may be too conservative. Consider a modifi cation of Example 3.3 for a system of four processes, in which the processes P_1 and P_2 cannot block the processes P_3 and P_4, that is, P_i always assigns j to x^j_{ij} when $i \in \{1,2\}$ and $j \in \{3,4\}$, rather than nondeterministically choosing between assigning i (thereby blocking P_j) and assigning j. Condition (c) still fails in the state $[T_1\, T_2\, T_3\, T_4]$ with P_4 being the process that made the last move. The system is not deadlocked, however, since the only process that can block P_4 is P_3, and it can block only one move of P_4 at the same time. Thus, P_4 always has at least one enabled move. Figure 5 shows the skeletons of processes in this example. The system has a single initial state $[N_1\, N_2\, N_3\, N_4]$. The processes P_1 and P_2 access the critical sections A and B, and the processes P_3 and P_4 access the critical sections C and D. The processes P_1 and P_2 can block each other, and the processes P_3 and P_4

can block each other and the processes P_1 and P_2. Condition (c) fails in the state $[T_1\, T_2\, T_3\, T_4]$ with P_4 being the process that made the last move. Indeed, in the system $J = \{\{1,4\},\{4,3\}\}$, the process P_4 blocks the process P_1 and both moves of P_4, when viewed separately, can be blocked by P_3. However, the system is not deadlocked in the state $[T_1\, T_2\, T_3\, T_4]$ for all reachable combinations of values of the shared variables. The process P_3 can block only one move of P_4 at the same time, and the processes P_1 and P_2 cannot block P_4. Thus, P_4 always has at least one enabled move.


$\otimes_{j\neq i}$ $true \to skip$
C_i
$\otimes_{j\neq i}($ $T_i \to x^j_{ij}, y^j_{ij} := i, j \oplus$
$T_i \to x^j_{ij}, y^j_{ij} := j, i$ $)$
N_i
T_i
$(\otimes_{j\neq i, j\in\{3,4\}} x^i_{ij} = i)\otimes$
$(\otimes_{j\in\{1,2\}} true)$
$(\otimes_{j\neq i, j\in\{3,4\}} y^i_{ij} = i)\otimes$
$(\otimes_{j\in\{1,2\}} true)$
D_i
$\otimes_{j\neq i}$ $true \to skip$

P_i for $i \in \{3,4\}$.
The processes P_3, and P_4 can block each other and the processes P_1 and P_2


$\otimes_{j\neq i}$ $true \to skip$
A_i
$\otimes_{j\neq i, j\in\{1,2\}}(T_i \to x^j_{ij}, y^j_{ij} := i, j \oplus$
$T_i \to x^j_{ij}, y^j_{ij} := j, i)$
$\otimes$
$\otimes_{j\in\{3,4\}}($ $T_i \to x^j_{jj}, y^j_{ij} := j, j \oplus$
$T_i \to x^j_{ij}, y^j_{ij} := j, j)$
N_i
T_i
$\otimes_{j\neq i}$ $x^i_{ij} = i$
$\otimes_{j\neq i}$ $y^i_{ij} = i$
B_i
$\otimes_{j\neq i}$ $true \to skip$

P_i for $i \in \{1,2\}$.
The processes P_1 and P_2 can only block each other

Figure 5: Modifi ed example for four dissimilar processes.

The test given in the next section, which is more complex but also more discriminating, verifi es that this example is in fact deadlock-free.

4 More Complex and Discriminating Condition

We defi ne a *global wait-for graph* $\mathcal{W}$ which contains the union of all $W_I(s)$, for all reachable I-states s.

Definition 4.1 $(\mathcal{W})$ *The graph $\mathcal{W}$ is as follows. The nodes of $\mathcal{W}$ are*

1. *the states s_i such that $i \in \{1 \dots K\}$ and $s_i \in reachable(P_i)$;*
2. *the moves a_i such that $i \in \{1 \dots K\}$, a_i is a move of P_i, and $a_i.start = s_i$ for some node s_i;*

and the edges are:

1. *an edge from s_i to every a_i such that $a_i.start = s_i$;*
2. *for $(i,j) \in I$ and a_i a move a_i of P_i, there is an edge from a_i to s_j iff $\exists s_{ij} \in S_{ij} : s_{ij}\restriction j = s_j \wedge s_{ij}(a^I_i.guard_j) = false\}$.*

We can view $\mathcal{W}$ as either a directed graph or as an AND-OR graph. When viewed as an AND-OR graph, the AND-nodes are the local states s_i of all processes (which we call local-state nodes) and the OR-nodes are the moves a_i (which we call move nodes). We shall use MSCC to abbreviate "maximal strongly connected component" in the sequel. The following proposition is an immediate consequence of the definitions.

Proposition 4.2 *For every reachable I-state s, $W_I(s)$ is a subgraph of $\mathcal{W}$.*

Proposition 4.3 *Let s be a reachable I-state, and assume that $W_I(s)$ contains a supercycle SC. Then, there exists a subgraph SC' of SC such that:*

1. *SC' is nontrivial, i.e., contains more than one node,*
2. *SC' is itself a supercycle, and*
3. *there exists a maximal strongly connected component $\mathcal{W}'$ of $\mathcal{W}$ such that SC' is a subgraph of $\mathcal{W}'$.*

Proof: By Definition 2.2, SC is not acyclic, hence SC contains at least one (nontrivial) MSCC. Consider the acyclic component graph obtained by shrinking each MSCC down to a single node. This graph, being acyclic, contains at least one node with no outgoing edges. Let SC' be an MSCC of SC corresponding to such a node. Thus, no node in SC' has an outgoing edge to a node outside of SC'. Suppose SC' contains a single local-state node s_i. Then, by Definition 2.2 and our restriction (in Section 2.1) that a synchronization skeleton contains no "dead ends," SC' must contain at least one node of the form a_i. On the other hand, suppose that SC' contains a single move node a_i. Since a_i is not enabled in global state s, it must be blocked by at least one local-state node s_j $(= s\lceil j)$. Hence, s_j is a node in SC'. Thus, in either case, SC' contains at least two nodes. Thus clause (1) is established.

We showed above that no node in SC' has an outgoing edge to a node outside of SC'. Thus, every move-node in SC' is blocked only by local-state nodes in SC'. Hence, by clause (1) and Definition 2.2, SC' itself is a supercycle. Hence, clause (2) is established.

Suppose SC' does not occur within a single MSCC of $\mathcal{W}$. Then there exist two MSCC's $\mathcal{W}_1$, $\mathcal{W}_2$ of $\mathcal{W}$, and two nodes v_1, v_2 such that v_1 occurs in both SC' and $\mathcal{W}_1$ and v_2 occurs in both SC' and $\mathcal{W}_2$. Since SC' is a strongly connected subgraph of $\mathcal{W}$, v_1 is reachable from v_2 in $\mathcal{W}$ and vice-versa. Hence $\mathcal{W}_1$ and $\mathcal{W}_2$ are not *maximal* strongly connected components, contrary to assumption. Hence clause (3) is established. $\square$

Proposition 4.4 *If $\mathcal{W}$ is acyclic, then for all reachable I-states s, $W_I(s)$ is supercycle-free.*

Proof: We establish the contrapositive. Suppose $W_I(s)$ contains a supercycle SC for some $s \in S_I^r$. Then by Proposition 4.2, SC is a subgraph of $\mathcal{W}$. But SC contains a cycle. Hence $\mathcal{W}$ is not acyclic. $\square$

We now present a test for supercycle-freedom. In the following we will view $\mathcal{W}$ as a regular directed graph, rather than an AND-OR graph. The test is given by the procedure CHECK-SUPERCYCLE($\mathcal{W}$) below, which works as follows. We first find the maximal strongly connected components (MSCC's) of $\mathcal{W}$. If no nontrivial MSCC's exist, then $\mathcal{W}$ is acyclic and so the I-system is supercycle-free by Proposition 4.4. Otherwise, we execute the following check for each local-state node t_k in $\mathcal{W}$. If the check marks t_k as “safe”, this means that no transition by P_k that ends in state t_k can create a supercycle where one did not exist previously. If all local-state nodes in $\mathcal{W}$ are marked as “safe”, then we conclude that no transition by any process in the I-system can create a supercycle. Given that all initial I-states are supercycle-free, this then implies that every reachable I-state is supercycle free, and so the I-system is deadlock-free. The check for t_k is as follows. If t_k does not occur in a nontrivial MSCC of $\mathcal{W}$, then, by Proposition 4.3, t_k cannot occur in any supercycle, so mark t_k as safe and terminate. Otherwise, invoke CHECK-STATE(t_k, C), where C is the nontrivial MSCC of $\mathcal{W}$ in which t_k occurs. Our test is sound but not complete. If some t_k is not marked “safe”, then we have no information about the possibility of the occurrence of supercycles.

```
CHECK-SUPERCYCLE($\mathcal{W}$)
1.  Find the maximal strongly connected components of $\mathcal{W}$
2.  **for** each MSCC $C$ of $\mathcal{W}$ that consists of a single node
        **if** the node is a local-state node **then** mark it “safe”
3.  **for** each MSCC $C$ of $\mathcal{W}$ that contains more than one node
        **for** each local-state node $s_i$ of $C$, invoke CHECK-STATE($s_i, C$)
4.  **if** all local-state nodes in $\mathcal{W}$ are marked “safe”, **then return** (“No supercycle possible”)
    **else return** (“Inconclusive”)
```

```
CHECK-STATE($t_k, C$)
1.  Construct a subgraph $SC$ of $C$ as follows.
        Let $SC$ initially be $C$
        Remove from $SC$ every $s_k$ such that $s_k \in reachable(P_k) - \{t_k\}$
        **repeat** until no more nodes to remove from $SC$
            **if** $a_j$ is a node in $SC$ with no outgoing edges in $SC$ **then**
                let $s_j$ be the unique node such that $s_j \longrightarrow a_j \in SC$
                remove $s_j$ and $a_j$ and their incident edges from $SC$
2.  Compute the maximal strongly connected components of $SC$
3.  **if** $t_k$ is not in some MSCC of $SC$ **then** mark $t_k$ as “safe” and terminate.
    **else** Let $MC$ be the MSCC of $SC$ containing $t_k$
4.  **for** all $(s_j, a_j, t_k, a_k, s_\ell)$ such that $s_j \longrightarrow a_j \longrightarrow t_k \longrightarrow a_k \longrightarrow s_\ell \in MC$
        Let $J = \{\{j,k\},\{k,\ell\}\}$
        **if** there exists a state $s_J$ of $M_J$ such that:
            $s_J$ is reachable along a path in $M_J$ that ends in a transition by $P_k$, and
            $s_j \longrightarrow a_j \longrightarrow t_k \longrightarrow a_k \longrightarrow s_\ell \in W_J(s_J)$
        **then** mark all the nodes and edges in $s_j \longrightarrow a_j \longrightarrow t_k \longrightarrow a_k \longrightarrow s_\ell$
5.  Remove from $MC$ all nodes and edges within two hops from $t_k$ (in either direction)
     that are unmarked. Call the resulting graph $MC'$
6.  Calculate the maximal strongly connected components of $MC'$
7.  **if** $t_k$ does not lie in an MSCC of $MC'$ **then** mark $t_k$ as safe
```

The procedure CHECK-STATE(t_k, C) tests whether the wait-for edges involving t_k and two other local states s_j, s_ℓ can all be generated *simultaneously* in the triple system consisting of processes j, k, ℓ. This generation must also be by a *reachable* transition of process k. If this generation cannot occur in all such triple systems, then the wait-for-edges in the vicinity of s_k cannot be present simultaneously in some global state in a combination that results in a supercycle containing t_k.

Theorem 4.5 *If all local-state nodes in $\mathcal{W}$ are marked as "safe," then the I-system P^I is supercycle-free.*

Proof: The proof idea is, roughly, that if all the wait-chains involving a local-state node t_k are unreachable in the relevant J-machines, then t_k cannot be part of a supercycle. The detailed proof is as follows.

We establish the contrapositive of the statement of the theorem. Let M_I be the global-state transition diagram of P^I. Assume that there exists a reachable global state v of M_I such that $W_I(v)$ contains a supercycle SC. Let SC' be a supercycle as given by Proposition 4.3 (there may be more than one such supercycle—we choose one arbitrarily).

Since v is a reachable global state, there exists at least one finite computation path $\pi = s_0 \dots v$ ending in v. By assumption, all initial global states are supercycle-free. Also, a supercycle is stable by definition: once created, it persists forever, since none of the involved processes can make a transition and thereby change their state. Hence, there exist states s, t along π such that (1) SC' is not contained in $W_I(s')$ for any state s' along the prefix $s_0 \dots s$ of π, and (2) SC' is contained in $W_I(t')$ for every state t' along the suffix $t \dots v$ of π.

Let P_k be the process that executes the transition from s to t along π. Since SC' is strongly connected, we have:

$$t_j \longrightarrow a_j \longrightarrow t_k \longrightarrow a_k \longrightarrow t_\ell \in SC' \tag{a}$$

where $t_j = t{\upharpoonright}j$, $t_k = t{\upharpoonright}k$, $t_\ell = t{\upharpoonright}\ell$, a_j is an arbitrarily chosen P_j-move in SC', a_k is an arbitrarily chosen P_k-move in SC', $(j,k) \in I$, $(k,\ell) \in I$, and j may or may not be equal to ℓ.

Now SC' is a subgraph of $W_I(t)$, by construction, hence by (a):

$$t_j \longrightarrow a_j \longrightarrow t_k \longrightarrow a_k \longrightarrow t_\ell \in W_I(t). \tag{b}$$

By Proposition 4.2, $W_I(t)$ is a subgraph of $\mathcal{W}$. So, from (b):

$$t_j \longrightarrow a_j \longrightarrow t_k \longrightarrow a_k \longrightarrow t_\ell \in \mathcal{W}. \tag{c}$$

Let $J = \{(j,k),(k,\ell)\}$, and let $t_J = t{\upharpoonright}J$. By [2, Proposition 6.5.4.1][2] and (b), we have

$$t_j \longrightarrow a_j \longrightarrow t_k \longrightarrow a_k \longrightarrow t_\ell \in W_J(t_J). \tag{d}$$

Now t is reachable in M_I. Hence by [2, Corollary 6.4.5], t_J is reachable in M_J. From this and (d), it follows that all the nodes in the wait-chain $t_j \longrightarrow a_j \longrightarrow t_k \longrightarrow a_k \longrightarrow t_\ell$ are marked in line 5 of CHECK-STATE(t_k, C). Thus, none of these nodes are removed in line 6. Since a_j and a_k are arbitrarily chosen, it follows that all nodes of SC' that are within two hops of t_k in either direction are marked in line 4 of CHECK-STATE(t_k, C). Thus, no nodes of SC' are deleted in line 5.

[2]Technically, the results of [2] were established only for concurrent programs with similar processes. However, the specific results that we use here were established without resorting to the similarity assumption, and so they also hold for concurrent programs with dissimilar processes.

Hence, SC' is a subgraph of the graph MC' that results after all deletions in line 5 are performed. Since SC' is strongly-connected, SC' lies in an MSCC of MC'. Since t_k is a node of SC', it follows from line 7 that t_k is not marked as safe. Hence the contrapositive is established. $\square$

Proposition 4.6 *Let N be the size of the largest I-process (number of local states plus number of I-moves). Then the size of $\mathcal{W}$ (number of nodes and edges) is $O(K^2N^2)$.*

Proof: Each process contributes $O(N)$ nodes to $\mathcal{W}$, and there are K processes. Thus, $\mathcal{W}$ contains $O(KN)$ nodes. Hence, $\mathcal{W}$ contains $O(K^2N^2)$ edges. $\square$

Theorem 4.7 *The running time of CHECK-SUPERCYCLE($\mathcal{W}$) is $O(K^4N^4)$.*

Proof: Line 1 of CHECK-SUPERCYCLE($\mathcal{W}$) takes time $O(K^2N^2)$, using the algorithm in [6, Chapter 22].

Lines 2 and 4 of CHECK-SUPERCYCLE($\mathcal{W}$) clearly take time $O(K^2N^2)$,

Line 3 of CHECK-SUPERCYCLE($\mathcal{W}$) invokes CHECK-STATE(t_k, C). Line 1 of CHECK-STATE(t_k, C) takes time $O(|C|)$ + the time to compute $reachable(P_k)$. $O(|C|)$ is $O(K^2N^2)$ since C is a subgraph of $\mathcal{W}$. $reachable(P_k)$ can be computed in time $O(K^2N^2)$ since there are $O(K^2)$ pair-systems, each of size $O(N^2)$. Line 2 of CHECK-STATE(t_k, C) takes time $O(K^2N^2)$ since C is a subgraph of $\mathcal{W}$. Line 3 of CHECK-STATE(t_k, C) takes time $O(K^2N^2)$ using a straightforward graph search. Line 4 takes time $O(K^3N^3)$ since there are $O(K^3)$ three-process systems, each of size $O(N^3)$. Searching for wait-chains with MC is clearly in $O(K^3N^3)$. Line 5 takes time $O(N^2K^2)$, since $|MC|$ is $O(N^2K^2)$. Line 6 takes time $O(N^2K^2)$, since $|MC'|$ is $O(N^2K^2)$. Line 7 takes time $O(N^2K^2)$, using a straightforward graph search. Thus, a single invocation of CHECK-STATE(t_k, C) runs in time $O(K^3N^3)$. CHECK-STATE(t_k, C) is invoked at most $O(KN)$ times, since $\mathcal{W}$ contains $O(KN)$ nodes, and CHECK-STATE(t_k, C) is invoked at most once for each local-state node t_k in $\mathcal{W}$. Hence the total running time of line 3 of CHECK-SUPERCYCLE($\mathcal{W}$) is $O(K^4N^4)$.

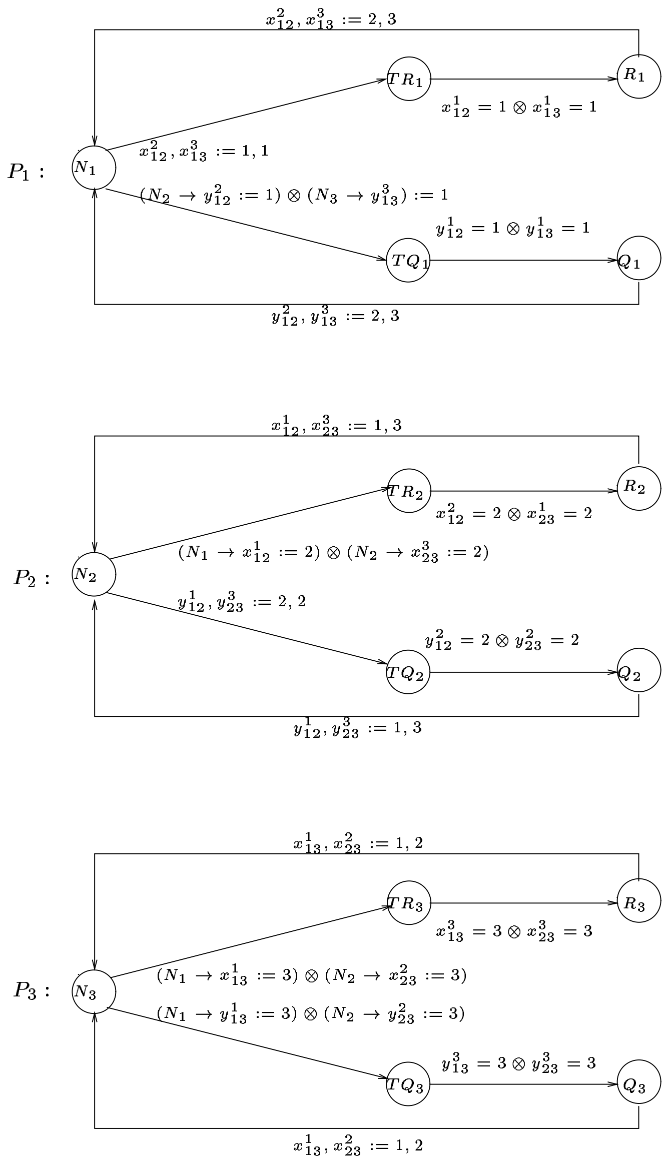
Thus, CHECK-SUPERCYCLE($\mathcal{W}$) runs in time $O(K^4N^4)$. $\square$

It may be possible to improve the runtime complexity of the algorithm using more sophisticated graph search strategies. For example, for each three-process system, we could collect all the wait-chains together and search for them all at once within the global state-transition graph (GSTD) of the three-process system. Wait-chains that are found could then be marked appropriately for subsequent processing.

It is not too hard to verify that the global wait-for graph for the hierarchical resource allocation strategy that we discussed in Section 3 is acyclic. Indeed, a supercycle in a wait-for graph represents a cyclical waiting pattern between processes. However, a hierarchy establishes a total order between processes, and the transitions in the graph represent blocking conditions, which can occur only when moves of a process with a lower priority are blocked by a process with higher priority. Thus, waiting conditions form chains, and not cycles in the wait-for graph. In a more general situation, the requirement of total hierarchical order can be relaxed for a subset of resources. Clearly, in this case deadlock can occur, depending on the sets of resources that each process attempts to acquire and the order of requests. Our algorithm can effi ciently detect deadlocks in these cases.

Now we turn to some more examples, for which we explicitly construct the wait-for graph and show the presence or absence of supercycles.

Example 4.8 [Verification of deadlock freedom] We study a special case of resource allocation problem [16] that we presented in Section 3. In this system, there are two resources (we refer to them as priority queues) and the additional parameter is the set of priorities of processes for the queues. Consider an I-system where the processes are partitioned into 3 classes, and are accessing two priority queues R and Q. The first class of processes has the highest priority for R, and the second class of processes has the highest priority for Q. The skeletons of processes P_1, P_2, and P_3 that represent the three classes of processes are described in Figure 6.



P_1 :
N_1
TR_1
R_1
TQ_1
Q_1
$x^2_{12}, x^3_{13} := 2, 3$
$x^1_{12} = 1 \otimes x^1_{13} = 1$
$x^2_{12}, x^3_{13} := 1, 1$
$(N_2 \rightarrow y^2_{12} := 1) \otimes (N_3 \rightarrow y^3_{13}) := 1$
$y^1_{12} = 1 \otimes y^1_{13} = 1$
$y^2_{12}, y^3_{13} := 2, 3$
P_2 :
N_2
TR_2
R_2
TQ_2
Q_2
$x^1_{12}, x^3_{23} := 1, 3$
$x^2_{12} = 2 \otimes x^1_{23} = 2$
$(N_1 \rightarrow x^1_{12} := 2) \otimes (N_2 \rightarrow x^3_{23} := 2)$
$y^1_{12}, y^3_{23} := 2, 2$
$y^2_{12} = 2 \otimes y^2_{23} = 2$
$y^1_{12}, y^3_{23} := 1, 3$
P_3 :
N_3
TR_3
R_3
TQ_3
Q_3
$x^1_{13}, x^2_{23} := 1, 2$
$x^3_{13} = 3 \otimes x^3_{23} = 3$
$(N_1 \rightarrow x^1_{13} := 3) \otimes (N_2 \rightarrow x^2_{23} := 3)$
$(N_1 \rightarrow y^1_{13} := 3) \otimes (N_2 \rightarrow y^2_{23} := 3)$
$y^3_{13} = 3 \otimes y^3_{23} = 3$
$x^1_{13}, x^2_{23} := 1, 2$

Figure 6: The example of three processes with two priority queues.

For processes in the same class and processes in different classes that have the same priority, the access to a queue is FIFO. There can be only one process at a time at the head of each queue (i.e., in state R_i for queue R and Q_i for queue Q). The system has a single initial state $[N_1\, N_2\, N_3, \ldots, N_K]$. For clarity, we omit the shared variables that control the FIFO access to queues between the processes in the same class. Intuitively, a deadlock can occur if there are several processes with the same priority in a trying state (TR_i or TQ_i). However, the guards on transitions to trying states guarantee that a process enters a trying state iff either there is no other process is in the trying state, or the other process in the trying state has a lower priority.

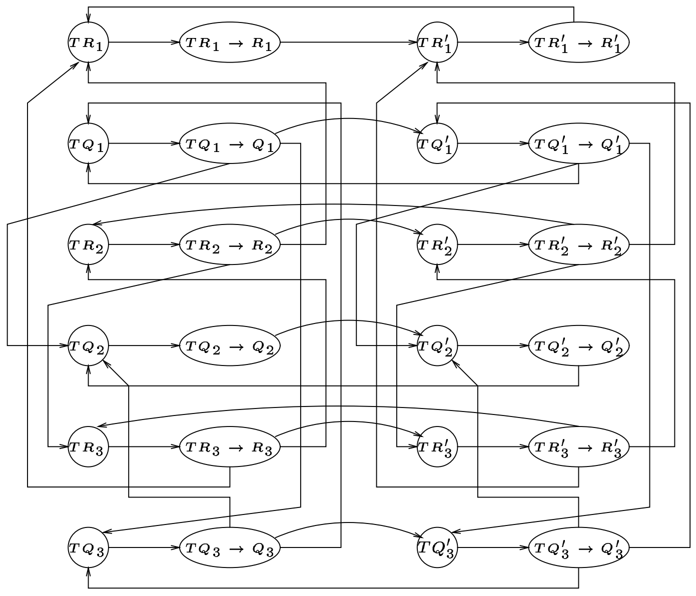


Figure 7: A part of the global graph.

Figure 7 gives a fragment of $\mathcal{W}$ for this system. The move nodes are labeled with the name of the source and target states and a directed edge between them (for example, the node that corresponds to the move from TR_1 to R_1 is labeled with $TR_1 \to R_1$). For clarity, the graph shows only two processes P_i and P'_i from each class, for $1 \leq i \leq 3$, and only the trying states and their outgoing moves. It also omits the wait-for edges between the primed and unprimed processes from different classes. Still, it is easy to see that the graph contains many supercycles. We can prove that these supercycles are unreachable using the algorithm CHECK-STATE. Indeed, for all triples of processes that include an edge in a supercycle, all processes in the triple are in their trying states. These states are unreachable in the the triple systems. Thus, the I-system passes our check, and hence is deadlock-free.

Example 4.9 We now return to example 3.4. Recall, that the system in the example consists of 4 processes P_1, P_2, P_3, and P_4 accessing two critical sections, where the processes P_3 and P_4 can block all other processes, and the processes P_1 and P_2 can only block each other. The test in Section 3 failed for this system in example despite it being deadlock-free. We sketch the application of the test in this section as follows. We apply CHECK-STATE to each state that participates in a supercycle. The only supercycles in the global wait-for graph are created by two processes - either P_1 and P_2, or P_3 and P_4 in their trying states. By checking triples of processes, we can show that a trying state T_i and the assignment to shared variables that blocks two moves of another process are unreachable in all triples, for $1 \leq i \leq 4$. Thus, the system passes the test and is deadlock-free.

Example 4.10 [A system with a deadlocked state] In this example we describe a system with a reachable deadlocked state and demonstrate the evidence for the deadlock in the the global wait-for graph. The system consists of two dissimilar processes P_1 and P_2 accessing two priority queues R and Q. The skeletons of the processes are shown in Figure 8. The system has a single initial state $[N_1\,N_2]$. The process P_1 has priority over P_2 in the access to R, and the process P_2 has priority over P_1 in the access to Q. Both processes access both queues during one execution round. The skeletons of the processes are presented in Figure 8.

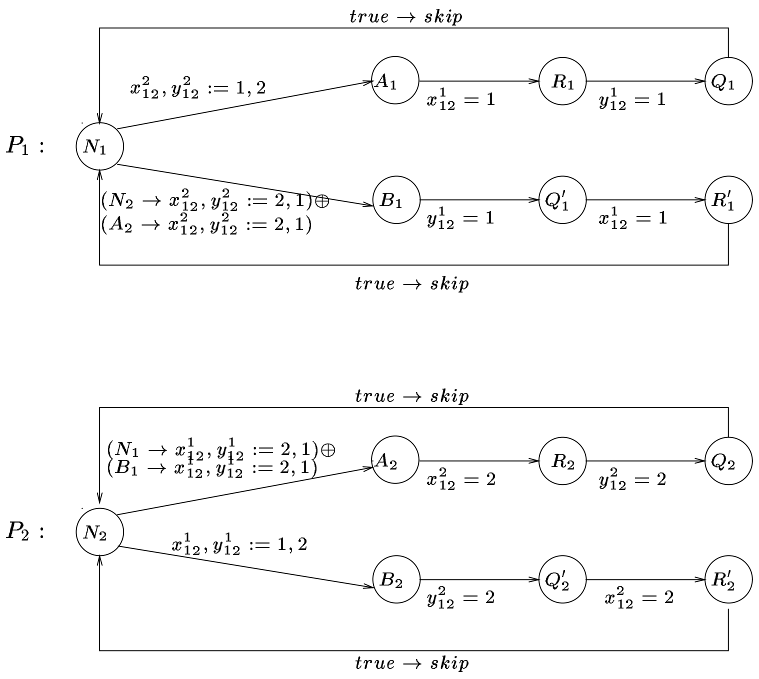
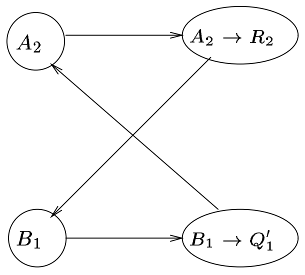


Figure 8: The example of two processes with two priority queues.

The global state $[B_1\,A_2]$ is reachable from the initial state $[N_1\,N_2]$ by the following path.

$$[N_1\,N_2] \xrightarrow{1} [y^2_{12} := 1, B_1\,N_2] \xrightarrow{2} [x^1_{12} := 2, B_1\,A_2].$$

This state is deadlocked, since the process P_1 is waiting for the process P_2 to release Q, and the process P_2 is waiting for the process P_1 to release R. This cyclic waiting can be discovered by examining the global wait-for graph for supercycles. The drawing presents a fragment of the graph that contains the supercycle for the deadlocked state $[B_1\,A_2]$.



5 Conclusions

We studied the problem of deadlock detection and avoidance in concurrent systems. We presented two sufficient conditions for assuring deadlock-freedom in a concurrent system. The first condition is suitable for systems of similar processes, while the second condition is suitable for systems of dissimilar processes. We presented efficient algorithms for computing both conditions. Our algorithms are polynomial in the number of processes and in the size of a single process.

References

[1] P. C. Attie. Synthesis of large concurrent programs via pairwise composition. In *CONCUR'99: 10th International Conference on Concurrency Theory*, number 1664 in LNCS, Aalborg, Denmark, Aug. 1999. Springer-Verlag.

[2] P. C. Attie and E. A. Emerson. Synthesis of concurrent systems with many similar processes. *ACM Trans. Program. Lang. Syst.*, 20(1):51–115, Jan. 1998.

[3] P.C. Attie. Synthesis of large dynamic concurrent programs from dynamic specifications. Technical report, Northeastern University, Boston, MA, 2003. Available at `http://www.ccs.neu.edu/home/attie/pubs.html`.

[4] P.C. Attie. Finite-state concurrent programs can be expressed pairwise. Submitted, 2004.

[5] E.G. Coffman, M.J. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3:67–78, June 1971.

[6] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms; Second Edition*. MIT Press and McGraw-Hill, 2001.

[7] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1976.

[8] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2:241 – 266, 1982.

[9] E. A. Emerson and A. P. Sistla. Symmetry and model checking. In *Proceedings of the 5th International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 463–477, Berlin, 1993. Springer-Verlag.

[10] P. Godefroid. *Partial Order Methods for the Verification of Concurrent Systems*. PhD thesis, University of Liege, 1994.

[11] P. Godefroid, D. Peled, and M. Staskauskas. Using partial-order methods in the formal validation of industrial concurrent programs. *Transactions on Software Engineering*, 22(7):496–507, Jan. 1996.

[12] P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110(2):305–326, 1991.

[13] R. C. Holt. Some deadlock properties of computer systems. *ACM Computing Surveys*, 4(3):179–196, 1972.

[14] E. Knapp. Deadlock detection in distributed databases. *ACM Comput. Surv.*, 19(4):303–328, Dec. 1987.

[15] P. Ladkin and B. Simons. Compile-time analysis of communicating processes. In *Proc. International Conference on Supercomputing*, pages 248–259. ACM Press, 1992.

[16] N. A. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, San Francisco, California, USA, 1996.

[17] D. Peled. Partial order reduction: Model-checking using representatives. In *MFCS*, 1996.

[18] B. Rex. Inference of k-process behavior from two-process programs. Master's thesis, School of Computer Science, Florida International University, Miami, FL, April 1999.

[19] A. S. Tanenbaum. *Modern Operating Systems, second edition*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 2001.