Efficiently Verifiable Conditions for Deadlock-freedom of Large Concurrent Programs (Extended Abstract)

Paul C. Attie[1] [2] and Hana Chockler[2] [3]

[1] College of Computer Science, Northeastern University,
360 Huntington Avenue, Boston, Massachusetts 02115.
attie@ccs.neu.edu, hanac@ccs.neu.edu
[2] MIT CSAIL, 32 Vassar street,
Cambridge, MA, 02139, USA.
attie@theory.csail.mit.edu, hanac@theory.lcs.mit.edu
[3] Department of Computer Science, WPI,
100 Institute Road, Worcester, MA 01609, USA.

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. As shown in [23], deadlock-freedom analysis is NP-hard even for concurrent programs of restricted form (no nondeterministic choice). Therefore, research in this area concentrates either on the search for efficiently checkable sufficient conditions for deadlock-freedom, or on improving the complexity of the check in some special cases. In this paper, we present two efficiently checkable sufficient conditions for deadlock freedom.

Our algorithms apply to programs which are expressed in a particular syntactic form, in which variables are shared between pairs of processes. The first algorithm improves the complexity of the deadlock check of Attie and Emerson [4] to polynomial in all parameters, as opposed to the exponential complexity of [4]. 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. We illustrate our algorithms by applying them to several examples of concurrent programs that implement resource allocation and priority queues. To the best of our knowledge, this is the first work that describes polynomially checkable conditions for assuring deadlock freedom of large concurrent programs.

1 Introduction

One of the important correctness properties of concurrent programs is the absence of *deadlocks*, e.g. as defined in [28]: "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 [10, 20]: (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); and (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): a graph whose edges depict the "wait-for" relationships between processes. The following models have been formulated [22]: (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; and (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 [22].

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 [4], 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 [28], (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 [28] 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 [23], deciding the deadlock-freedom of a finite-state concurrent program is NP-hard even for constrained programs in which each process consists of a finite prefix followed by an infinite loop.

Most model checking algorithms can be applied to verifying deadlock freedom. The main impediment is state-explosion. Some approaches to ameliorating state-explosion are to use a partial order instead of an interleaving model [16–18, 26], using symbolic model checking [21, 7, 8, 25] or by using symmetry reductions [2, 9, 13, 14]. These approaches, however, have worst case running time exponential in the number of processes in a system, and often rely on the processes being similar. (Roughly, two processes are similar if the code for one can be obtained from the code for the other by replacing process indices). Our first algorithm has better accuracy (i.e., returns a positive answer for deadlock-free programs) when processes are similar, but our second algorithm does not depend on similarity in any way.

In [1, 19] sufficient conditions for verifying deadlock-freedom are given, but it is not shown that these can be evaluated in polynomial time. Also, no example applications are given.

Attie and Emerson [4] formulate a condition that is sufficient but not necessary for deadlock-freedom. Checking this condition requires the construction of the automata-theoretic product of $n+2$ processes, where n is the maximum branching degree of a state-node in a state-transition graph that represents the behavior of processes (essentially, n reflects the degree of "local" nondeterminism of a single process in a state). The $n+2$ processes are arranged in a "star" configuration with a central process P_k and $n+1$ "satellite" processes. The condition is that after every transition of P_k, either P_k does not block another process, or P_k has another enabled transition. Hence P_k cannot be part of a cyclical waiting pattern in either case. Since this product has size exponential in n, checking the condition is infeasible for concurrent programs that have a high degree of local nondeterminism. While the condition in [4] is formulated for systems of similar (isomorphic) processes, the restriction to similar processes does not play any role in the proof of correctness given in [4], and thus can be removed.

Our Contribution In this paper we follow the approach of [4] to deadlock prevention. We present two sufficient conditions for assuring deadlock freedom and describe efficient (polynomial time) algorithms for checking these conditions.

The first condition is a modification of the condition presented in [4], but can be checked by constructing the product of only three processes (triple-systems). Roughly, the idea is to check the condition "after a transition, P_k either does not block another process, or is itself enabled" in systems of only three processes. We show that this implies the original condition of [4], and so implies deadlock-freedom by the results of that paper. Since only triple-systems are model-checked, the condition can be checked in time polynomial in all the input parameters: the number of processes, the size of a single process, and the branching degree of state-nodes of a process. Moreover, the space complexity of the check is polynomial in the size of a single process, and the checks for all triples can be performed sequentially, thus memory can be reused. Therefore, this condition can be efficiently checked even on very large concurrent programs.

The second condition is more complex and also more discriminating. This condition is based on constructing the "global wait-for graph," a bipartite graph whose nodes are the local states of all processes, and also the possible transitions that each process can execute. The edges of this graph represent "pairwise wait-for" conditions: if in the system consisting of P_i and P_j executing their synchronization code in isolation (pair-system), there is a transition a_i of P_i that is blocked in some state where the local state of P_j is s_j, then there is an edge from a_i to s_j. Since only pair-systems need be checked, the global wait-for-graph can be constructed in polynomial time. Existence of a deadlock implies the existence of a subgraph of the global wait-for-graph in which every process is blocked by some other processes in the subgraph. We call such a subgraph a *supercycle* and define it formally in the sequel. One could check the global wait-for-graph for the occurrence of supercycles, but the results of [23] imply that this cannot be done in polynomial time. Instead we check the global wait-for-graph for the occurrence of subgraphs of a supercycle. If these subgraphs are not present, then the supercycle cannot be present either, and so our check succeeds in verifying deadlock-freedom. If these subgraphs are present, then the supercycle may or may not be present, and so our check is inconclusive.

To the best of our knowledge, this is the first work that describes sufficient and polynomially checkable conditions for deadlock-freedom of large concurrent programs. We have implemented our pairwise representation using the XSB logic programming system [27]. This implementation provides a platform for implementing the algorithms in this paper. Due to the lack of space, all proofs and many technical details are omitted from this version. The full version can be found at authors' home pages.

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* [15], 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 [12] $B_i \to A_i$ 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 $s = (s_1, \dots, s_i, \dots, s_K, v_1, \dots, v_m)$ 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 $s' = (s_1, \dots, s'_i, \dots, s_K, v'_1, \dots, v'_m)$ where $v'_1, \dots, v'_m$ 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. We write $states(M)$ for S.

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* [4] 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 [4] 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 $a_i = (s_i, \otimes_{j \in I(i)} \oplus_{\ell \in \{1, \dots, n_j\}} B^j_{i,\ell} \to A^j_{i,\ell}, t_i)$, where $B^j_{i,\ell} \to A^j_{i,\ell}$ is a guarded command, I is an irreflexive symmetric relation over $\{1 \dots K\}$ that defines a "interconnection" (or "neighbors") relation amongst processes[4], 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 $B^j_{i,\ell} \to A^j_{i,\ell}$, $1 \le \ell \le n$ for the interaction between P_i and P_j as P_i transitions from s_i

[4] In other words, I is the topology of the connection network.

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 ($\otimes_{j \in I(i)}$ specifies this). 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. For $a_i^I = (s_i, \otimes_{j \in I(i)} \oplus_{\ell \in \{1,\dots,n_j\}} B_{i,\ell}^j \to A_{i,\ell}^j, t_i)$, we define $a_i^I.start = s_i$, $a_i^I.guard_j = \bigvee_{\ell \in \{1,\dots,n_j\}} B_{i,\ell}^j$, and $a_i^I.guard = \bigwedge_{j \in I(i)} a_i.guard_j$. We write $a_i^I \in P_i^I$ when a_i^I is a move of P_i^I. If $P^I = P_1^I \| \dots \| P_K^I$ is a concurrent program with interconnection relation I, then we call P^I an *I-system*. Global states of P^I are called *I-states*.

In pairwise normal form, the synchronization code for P_i^I with one of its neighbors P_j^I (i.e., $\oplus_{\ell \in \{1,\dots,n_j\}} B_{i,\ell}^j \to A_{i,\ell}^j$) is expressed separately from the synchronization code for P_i^I with another neighbor P_k^I (i.e., $\oplus_{\ell \in \{1,\dots,n_k\}} B_{i,\ell}^k \to A_{i,\ell}^k$). 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 $a_i^J = (s_i, \otimes_{j \in J(i)} \oplus_{\ell \in \{1 \dots n\}} B_{i,\ell}^j \to A_{i,\ell}^j, t_i)$. We also use $a_i^I \upharpoonright J$ for a_i^J, to emphasize the projection onto the subrelation J. Then the *J-system* P^J is $P_{j_1}^J \| \dots \| P_{j_n}^J$ where $\{j_1, \dots, j_n\} = range(J)$ and P_j^J consists of the moves $\{a_i^J \mid a_I^I \text{ is a move of } P_j^I\}$. 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. Two processes P_i and P_j are *similar* if they are isomorphic to each other up to a change of indices [4, p. 78]. 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.

[4, 3, 5] 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 [6] shows that any finite-state concurrent program can be rewritten (up to strong bisimulation) in pairwise normal form. Thus, the algorithms we present here are applicable to any concurrent program, up to strong bisimulation.

2.3 The Wait-For-Graph

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

Definition 1 ($Wait\text{-}For\text{-}Graph\ W_I(s)$)**.** *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 $\{s{\upharpoonright}i \mid i \in \{1 \dots K\}\}$[5];*
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 $a_i^I.start = s{\upharpoonright}i\ \}$*
3. *There is an edge from $s{\upharpoonright}i$ to every node of the form a_i^I in $W_I(s)$;*
4. *There is an edge from a_i^I to $s{\upharpoonright}j$ in $W_I(s)$ if and only if $\{i, j\} \in I$ and $a_i^I \in W_I(s)$ and $s{\upharpoonright}ij(a_i^I.guard_j) = false$.*

The AND-nodes are the local states s_i $(= s{\upharpoonright}i)$ 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 iff *all* of its possible moves are blocked. There is an edge from a_i^I to s_j $(= s{\upharpoonright}j)$ iff a_i^I is blocked by P_j^I: a_i^I can be executed in s only if $s{\upharpoonright}ij(a_i^I.guard_j) = true$ for all $j \in I(i)$; if there is some j in $I(i)$ such that $s{\upharpoonright}ij(a_i^I.guard_j) = false$, 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. We cannot, however, 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 $s{\upharpoonright}ij(b_i^I.guard_j) = true$, 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

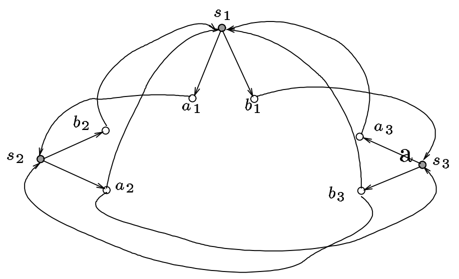
Deadlock is characterized by the presence in the wait-for-graph of a graph-theoretic construct called a *supercycle* [4]:

Definition 2 (Supercycle). *Let s be an I-state and $s_i = s{\upharpoonright}i$ for all $i \in \{1 \dots K\}$. 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 $a_i^I \longrightarrow s_j \in SC$.*

[5] In [4] 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.

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 the figure on the right we give an example of a wait-for-graph for a three process system. And-nodes (local states of processes) are shown as •, and or-nodes (moves) are shown as ∘. Each process P_i, $i \in \{1, 2, 3\}$ has two moves a_i and b_i in the local state s_i. Since every move has at least one outgoing edge, i.e., is blocked by at least one process, the figure is also an example of 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.

From [4], 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 1 ([4]). *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 Attie-Emerson Deadlock Freedom Condition

In this section we improve the Attie and Emerson [4] deadlock-freedom check (the wait-for-graph assumption of [4]). Consider the following condition.

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

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 $t_k.moves = \{a_k^I \mid a_k^I \in P_k^I \wedge a_k^I.start = t_k\}$. It is proved in [4] that it is enough to check condition (a) for all local states t_k of P_k^I and for all J-systems for $J \in \mathcal{J}$, where $\mathcal{J}$ is the set of all interconnection relations of the form $\{\{j,k\},\{k,\ell_1\},\{k,\ell_2\},\ldots,\{k,\ell_n\}\}$, and $n = |t_k.moves|$, $1 \leq j,k,\ell_1 \ldots,\ell_n \leq K$, $k \notin \{j,\ell_1 \ldots,\ell_n\}$. 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 must construct M_J, and so is exponential in n. It is thus impractical for large n.

Let $J_i = \{\{j,k\},\{k,\ell_i\}\} \subseteq J$, for $1 \leq i \leq n$.[6] Then, for each move a_k^J and state $t_J \in states(M_J)$, $\forall \ell \in \{\ell_1,\ldots,\ell_n\} : a_k^J \to t_J{\restriction}\ell \notin W_J(t_J)$ holds iff

$$\forall i : 1 \leq i \leq n : a_k^{J_i} \to t_J{\restriction}\ell_i \notin W_{J_i}(t_J{\restriction}J_i). \tag{1}$$

The last equation follows from wait-for-graph projection [4, Proposition 6.5.4.1].

Equation 1 is checked with respect to all systems of three processes, for all reachable states of these triple-systems. To avoid constructing the J-system, we check the following condition (b), which requires constructing only J_i-systems. Define $triple{-}reachable(k) = \{t_k : (\forall J = \{\{j,k\},\{k,\ell\}\} \subseteq I : (\exists t_J \in states(M_J) : t_J{\restriction}k = t_k))\}$. That is, $triple - reachable(k)$ is the set of local states t_k of P_k such that in every triple system J_i involving P_k there is a reachable state t_{J_i} that projects onto t_k. Then, the appropriate condition is:

```
$\forall t_k \in triple - reachable(k)$
    $\exists a_k \in t_k.moves$
        $\forall J_i = \{\{j,k\},\{k,\ell_i\}\} \subseteq I$
            $\forall t_{J_i}$ such that $t_{J_i} \in states(M_{J_i})$ and $t_{J_i}{\restriction}k = t_k$
            and $s_{J_i} \xrightarrow{k} t_{J_i}$ for some $s_{J_i} \in states(M_{J_i})$:
                $(\neg\exists a_j^{J_i} : a_j^{J_i} \longrightarrow t_k \in W_{J_i}(t_{J_i}))$ or
                $(a_k^{J_i} \longrightarrow t_{J_i}{\restriction}\ell_i \notin W_{J_i}(t_{J_i})))$ for $a_k^{J_i} = a_k{\restriction}J_i$.    (b)
```

Condition (b) 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. In either case, in every system $J = \{\{j,k\},\{k,l_1\},\ldots,\{k,l_n\}\}$, either P_k has an enabled move, or P_k does not block any move of P_j. Hence, in the I-system, P_k cannot be involved in a deadlock. Note that if the state t_J that projects onto t_{J_i} for all J_i is reachable in the J-system, then condition (b) implies the deadlock-freedom condition of [4] for the J-system. The converse always holds.

Theorem 1. *If condition (b) holds, then the I-system P^I is deadlock-free.*

Intuitively, checking condition (b) 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.

We check condition (b) as follows. For every process P_k, we compute the set $S_k = triple - reachable(k)$, and the set $\mathcal{J}_k$ of all triple-systems J_i which have P_k as the "middle' process:

$$\mathcal{J}_k = \{J_i : J_i = \{\{j,k\},\{k,\ell_i\}\} \wedge J_i \subseteq I \wedge k \neq j, \ell_i\}.$$

[6] Since $J \subseteq I$ and I is irreflexive, we have $k \neq i, \ell_i$.

For every $t_k \in S_k$, we compute the set $t_k.moves$ of outgoing moves of P_k from t_k. Then, for each $a_k \in t_k.moves$ and each $J_i \in \mathcal{J}_k$, we find every state $t_{J_i} \in states(M_{J_i})$ such that $t_{J_i}\restriction k = t_k \wedge (\exists s_{J_i} \in states(M_{J_i}) : s_{J_i} \xrightarrow{k} t_{J_i})$. This can be done by a graph search of M_{J_i}. We then evaluate

$$(\forall a_j^{J_i} : a_j^{J_i} \longrightarrow t_k \notin W_{J_i}(t_{J_i})) \vee (a_k^{J_i} \to t_{J_i}\restriction \ell_i \notin W_{J_i}(t_{J_i})) \qquad (2)$$

where $a_k^{J_i} = a_k\restriction J_i$ and $a_j^{J_i}$ ranges over all moves of $P_j^{J_i}$ such that $a_j^{J_i}.start = t_{J_i}\restriction j$, i.e., the moves of process j in the J_i-system which start in the local state that process j has in state t_{J_i}.

If for all $k \in \{1 \ldots K\}$ and 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 as the procedure CHECK-TRIPLES(P^I).

```
CHECK-TRIPLES($P^I$)
0.  for all $k \in \{1 \ldots K\}$
1.      $S_k := triple - reachable(P_k)$
2.      $\mathcal{J}_k := \{J_i \mid J_i = \{\{j,k\},\{k,\ell_i\}\} \wedge J_i \subseteq I\}$
3.      for all $t_k \in S_k$
            for all $a_k \in t_k.moves$
                for all $J_i$ in $\mathcal{J}_k$
                    generate $M_{J_i}$
                    for all $t_{J_i}$ such that $t_{J_i}\restriction k = t_k \wedge (\exists s_{J_i} \in states(M_{J_i}) : s_{J_i} \xrightarrow{k} t_{J_i})$
                        evaluate Equation 2
                if Equation 2 was found true for all $J_i$ and all $t_{J_i}$ then mark $t_k$
4.  if $\forall k \in \{1 \ldots K\}$: all $t_k \in S_k$ are marked, then return ("No supercycle possible")
    else return ("Inconclusive")
```

Upon termination of CHECK-TRIPLES(P^I), condition (b) holds iff "No supercycle possible" is returned. Termination is assured since all loops are finite.

Let b be the branching factor of a process, i.e., the maximum value of $|t_k.moves|$ over all $k \in \{1 \ldots K\}$ and all $t_k \in triple - reachable(P_k)$.

Theorem 2. *The time complexity of procedure* CHECK-TRIPLES(P^I) *is* $O(K^3N^4b)$, *and the space complexity is* $O(N^3)$.

We apply our check to the general resource allocation problem [24, Chapter 11]. For a system of n processes, an explicit resource specification $\mathcal{R}$ consists of a universal finite set R of (unsharable) resources and sets $R_i \subseteq R$ for all $i \in 1, \ldots, n$, where R_i is the set of resources that process P_i requires to execute.

Example 1 (Deadlock detection in the general resource allocation problem). In this example, we describe an solution to the 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 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. It can be shown that condition (b) fails, and indeed there is a deadlocked state in the system (in which each process is trying to acquire a resource already acquired by another process). In the full version we present the formal and detailed description of this example.

Now consider the hierarchical resource allocation presented in Lynch [24, 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 (b) 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 Section 4 we present a more complex (and more discriminating) test that shows deadlock freedom of hierarchical resource allocation.

In the following example we demonstrate false deadlock indication. It describes a system in which there are two types of processes, and only processes from one type can block other processes. The deadlock-freedom condition from [4] (the "wait-for-graph assumption") is satisfied, since it considers systems of $m + 2$ processes, m being the branching degree of a single process. Since condition (b) checks blocking for each outgoing move separately, it does not detect unreachability of the blocking state.

Example 2. We give here only the brief informal description of the example. For the formal description including the skeletons of participating processes the reader is refered to the full version of the paper. 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. Consider a triple in which P_3 is the middle process. In its trying state it has two outgoing moves for accessing two critical sections. Both moves can be blocked by process P_4 separately, depending on the state of the process P_4. That is, the process P_4 blocks the move of the process P_3 that attempts to access the same critical section as P_4. The condition (b) fails. At the same time, the condition in [4] passes, since it checks blocking conditions for both moves of P_3 at the same time. Then, it is easy to see that there are no two processes that can block both moves of P_3 simultaneously. In Section 4 we show that the absence of reachable supercycles can be detected by examining the global wait-for graph for this system.

Example 2 illustrates that while condition (b) implies the deadlock-freedom condition of [4], the opposite is not true. That is, there exist cases in which condition (b) fails, while the more discriminating condition of [4] is satisfied, and hence the system is deadlock-free. This happens when the blocking state is reachable for each triple separately, but not for the J-system with $m + 2$ processes.

4 A More Complex and Discriminating Deadlock-freedom Check

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

Definition 3. ($\mathcal{W}$) *The graph $\mathcal{W}$ is as follows. The nodes of $\mathcal{W}$ are*

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

and the edges are:

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

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 use MSCC to abbreviate "maximal strongly connected component" in the sequel.

Proposition 2. *For every reachable I-state s, $W_I(s)$ is a subgraph of $\mathcal{W}$.*

Proposition 3. *Let s be a reachable I-state, and assume that $W_I(s)$ contains a supercycle SC. Then, there exists a nontrivial subgraph SC' of SC which is itself a supercycle, and which is contained within a maximal strongly connected component of $\mathcal{W}$.*

Note that a supercycle is strongly connected, but is not necessarily a maximal strongly connected component.

Proposition 4. *If $\mathcal{W}$ is acyclic, then for all reachable I-states s, $W_I(s)$ is supercycle-free.*

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. 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 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 can be removed 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 $(s_j, a_j^I, t_k, a_k^I, s_\ell)$ such that $s_j \longrightarrow a_j^I \longrightarrow t_k \longrightarrow a_k^I \longrightarrow s_\ell \in MC$
        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
            $s_j \longrightarrow a_j^J \longrightarrow t_k \longrightarrow a_k^J \longrightarrow s_\ell \in W_J(s_J)$
        then mark all the nodes and edges in $s_j \longrightarrow a_j^I \longrightarrow t_k \longrightarrow a_k^I \longrightarrow s_\ell$
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 chain from some local state s_j to some j-move a_j to state t_k to some ℓ-move a_ℓ to some state s_ℓ can arise from a reachable transition of process P_k in the triple system consisting of processes P_j, P_k, P_ℓ. If so, then all these states and moves are marked and are retained, since they might form part of a supercycle involving t_k. After all such "length 5" chains have been examined, all nodes within 2 hops of t_k are removed, since these nodes cannot possibly be part of a supercycle involving t_k. If this removal process causes t_k to no longer be contained in an MSCC, then t_k cannot possibly be an essential part of a supercycle, since every supercycle is "essentially" contained inside a single MSCC, since removing all parts of the supercycle outside the MSCC still leaves a supercycle (see Proposition 3).

In summary, we check for the existence of subgraphs of a potential supercycle that are wait-chains of length 5. If enough of these are absent, then no supercycle can be present. Our check could be made more accurate by using longer length chains, but at the cost of greater time complexity.

Theorem 3. *If all local-state nodes in $\mathcal{W}$ are marked as "safe," then the I-system P^I is supercycle-free.*

Proposition 5. *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 $O(K^2N^2)$.*

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

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 efficiently detect deadlocks in these cases.

The following proposition relates the deadlock-freedom check of Section 3 and the check introduced in this section.

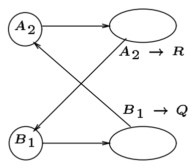
Proposition 6. *If procedure* CHECK-TRIPLES(P^I) *returns "No supercycle possible," then so does procedure* CHECK-SUPERCYCLE($\mathcal{W}$).

5 Examples

In this section, we study several examples of deadlock-free and deadlock-prone instances of the resource allocation problem [24, Chapter 11] and summarize the results obtained by using our algorithms. Due to the lack of space, many details are omitted here. They can be found in the full version.

Example 3 (Deadlock-free instance with two resources). We study a special case of resource allocation problem [24] 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. 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. Intuitively, a deadlock can occur if there are several processes with the same priority in a trying state. 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. We note that the unreachability of supercycles in the wait-for graph is evident already by considering triple-systems, and thus condition (b) is also satisfied.

Example 4 (Deadlock-prone instance with two resources). In this example we describe a system with a reachable deadlocked state and demonstrate the evidence for the deadlock in the global wait-for graph. The system consists of two dissimilar processes P_1 and P_2 accessing two priority queues R and Q.



A deadlocked state $[B_1 A_2]$ can be reached in which process P_1 is in local state B_1, waiting for process P_2 to release Q, and process P_2 is in local state A_2, waiting for process P_1 to release R. This cyclic waiting can be discovered by examining the global wait-for graph for supercycles.

The drawing above presents a fragment of the graph that contains the supercycle for the deadlocked state $[B_1\ A_2]$. The node labeled $B_1 \to Q$ is the move of P_1 that acquires Q, and the move labeled $A_2 \to R$ is the move of P_2 that acquires R. Condition (b) fails for the triple system $J_1 = \{\{P_1, P_2\}, \{P_2, P_1\}\}$, and thus the cyclic waiting is discovered by applying CHECK-TRIPLES(P^I).

Example 5 (Overlapping sets of resources). For a process P_i, let R_i be the set of resources P_i needs to acquire in order to execute. For each process P_k in the system, there exist two different processes P_i and P_j such that $R_i \cap R_k \neq \emptyset$ and $R_j \cap R_k \neq \emptyset$. Also, the order of acquiring the resources is non-deterministic for each process. In this case, condition (b) fails, thus indicating a possible deadlock. It is easy to see that the system is indeed deadlock-prone.

Example 6 (Processes with rollback). Now we construct an example for which the condition (b) described in Section 3 fails, although there is no deadlock. In this example, we have two types of processes. One type is the processes that acquire and lock resources one-by-one without the ability to rollback, as in the previous examples. The second type is the processes that rollback in case they encounter that one of the required resources is not available. In this case, condition (b) fails, although there is no deadlock.

6 Summary and Conclusions

examples	existence of deadlock	algorithm from Section 3	algorithm from Section 4
Example 1	deadlock	deadlock	deadlock
Example 2	no deadlock	deadlock	no deadlock
Example 3	no deadlock	no deadlock	no deadlock
Example 4	deadlock	deadlock	deadlock
Example 5	deadlock	deadlock	deadlock
Example 6	no deadlock	deadlock	no deadlock

The inset table summarizes the deadlock detection results for the instances of resource allocation problem (both in the previous section and in Section 3). We note that although we did not demonstrate this explicitly, it is easy to verify that the deadlock detection algorithm of [4] recognizes deadlock correctly in all the examples studied in this paper. Our fist algorithm is very simple and has a polynomial complexity in all its parameters. The negative answer from this algorithm, that is, if the system satisfies the condition (b), eliminates the need to invoke more complex and time-consuming algorithms. In cases where the system fails the condition (b), it might be necessary to invoke the more discriminating algorithm from Section 4. While this algorithm is more complicated, its complexity is still polynomial in all the parameters of the system.

By closely examining the instances of the resource allocation problem we studied, we can see that the algorithm from Section 3 gives false positive deadlock indications in systems with dissimilar processes, where there are some processes with "more blocking power" than the others and the number of potentially blocking processes is smaller than the branching degree of a single process. The algorithm from Section 4 is more subtle, and is suitable for systems of any number of dissimilar processes.

In conclusion, the success of our approach in verifying the deadlock-freedom of many variants and instances of the resource allocation problem is evidence of its wide applicability.

References

1. A. Aldini and M. Bernardo. A general approach to deadlock freedom verification for software architectures. In *FM 2003*, pp. 658–677, LNCS 2805.
2. Tamarah Arons, Amir Pnueli, Sitvanit Ruah, Jessie Xu, and Lenore D. Zuck. Parameterized verification with automatically computed inductive assertions. In *CAV*, pp. 221–234, 2001.
3. P. C. Attie. Synthesis of large concurrent programs via pairwise composition. In *CONCUR*, LNCS 1664, 1999.
4. P. C. Attie and E. A. Emerson. Synthesis of concurrent systems with many similar processes. *ACM Trans. Program. Lang. Syst.*, 20(1):51–115, 1998.
5. P.C. Attie. Synthesis of large dynamic concurrent programs from dynamic specifications. Technical report, NEU, Boston, MA, 2003.

6. P.C. Attie. Finite-state concurrent programs can be expressed pairwise. Technical report, NEU, Boston, MA, 2004.
7. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *TACAS*, pp. 193–207, 1999.
8. E. M. Clarke, O.Grumberg, and D. Peled. *Model Checking.* MIT Press, Cambridge, MA, 2000.
9. E.M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *FMSD*, 9(2), 1996.
10. E.G. Coffman, M.J. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3:67–78, 1971.
11. T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms; Second Edition.* MIT Press and McGraw-Hill, 2001.
12. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall Inc., 1976.
13. E. A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *CAD*, pp. 236–254, 2000.
14. E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *FMSD*, 9(1/2):105–131, 1996.
15. E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2:241 – 266, 1982.
16. P. Godefroid. *Partial Order Methods for the Verification of Concurrent Systems.* PhD thesis, University of Liege, 1994.
17. P. Godefroid, D. Peled, and M. Staskauskas. Using partial-order methods in the formal validation of industrial concurrent programs. *Trans. on Soft. Eng.*, 22(7):496–507, 1996.
18. P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110(2):305–326, 1991.
19. Gregor Goessler and Joseph Sifakis. Component-based construction of deadlock-free systems. In *FSTTCS*, pp. 420–433, LNCS 2914, 2003.
20. R. C. Holt. Some deadlock properties of computer systems. *ACM Comput. Surv.*, 4(3):179–196, 1972.
21. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *LICS* , pp. 1–33, 1990.
22. E. Knapp. Deadlock detection in distributed databases. *ACM Comput. Surv.*, 19(4):303–328, 1987.
23. P. Ladkin and B. Simons. Compile-time analysis of communicating processes. In *Proc. Int. Conf. on Supercomputing*, pp. 248–259, 1992.
24. N. A. Lynch. *Distributed Algorithms.* Morgan-Kaufmann, 1996.
25. Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.
26. D. Peled. Partial order reduction: Model-checking using representatives. In *MFCS*, 1996.
27. 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.
28. A. S. Tanenbaum. *Modern Operating Systems, second edition.* Prentice-Hall, 2001.