A THEORY FOR DEADLOCKS

Y. C. Tay
W. Tim Loke

CS-TR-344-91

Ausust 1991

A Theory for Deadlocks *

Y.C. Tay and W. Tim Loke
Department of Mathematics
National University of Singapore

Abstract

Deadlock detection is an elementary problem in computer science, yet algorithms for detecting deadlocks over resources in a distributed system are curiously prone to errors. This anomaly calls for a theory that will help us understand existing algorithms and design new algorithms.

Surprisingly, most papers in the literature have either no definition of what a deadlock is, or a bad definition; this fact itself accounts for many of the errors. The first task in developing a theory is therefore to choose an appropriate definition for a deadlock. Since this theory is to be used for the analysis and synthesis of detection algorithms, it should focus on what an algorithm can observe (in this sense, it is an *operational* theory); accordingly, the definition chosen here is in terms of locally observable facts, and does not use real time.

The theory begins with a rigorous formulation of the interaction between processes and resources, and its development is via logical deduction, rather than operational arguments. The results examine the effect of aborting processes on deadlock detection, clarify the difference between a process and a resource, and reveal the structure of detection algorithms.

To illustrate its application, the theory is used to analyze several errors and algorithms in the literature.

* Part of this work was done when Y.C. Tay was on sabbatical at Princeton University. W. Tim Loke was supported by the Science Research Program. This report is a revised version of Research Report 459 (Apr. 1991), Department of Mathematics, National University of Singapore.

1 Introduction

In the previous decade, several algorithms were proposed for detecting deadlocks over resources in a distributed system. Despite the elementary nature of the problem, many of these algorithms contain errors. There are two reasons for this. First, there is a carelessness in the definition of a deadlock: most papers have an incomplete, ambiguous or bad definition, or no definition at all. Second, there is no theory for analyzing detection algorithms. The absence of a theory leads to a variety of techniques that do not draw upon a common body of results, and the ad hoc nature of this endeavor therefore makes it prone to errors.

We study both issues in this paper. Our purpose is to provide a rigorous definition of a deadlock and construct a theory based on this definition. The development of such a theory is overdue: In their surveys of deadlock detection, Elmagarmid [E] attributed the many errors to the "lack of unified means by which researchers may specify their algorithms", and Knapp [K] concluded that "only rigorous proofs, using as little operational argumentation as possible, suffice to show the correctness of these algorithms".

Specifically, this paper presents a rigorous formulation of the assumptions concerning processes and resources, examines the issues in defining a deadlock, develops the theory from our assumptions and definitions, and applies it to an analysis of existing algorithms.

We begin by formalizing the request-to-release cycle in the acquisition of resources by processes. All operational assumptions are distilled as axioms, so that a theory can be constructed by logical deduction instead of through operational arguments. The axioms are in terms of *logical edges* and *logical time*. Logical edges underlie the data structures in detection algorithms, while logical time underlie their proofs of correctness. We thus obtain a unified framework for analyzing detection algorithms.

To motivate our definition of a deadlock, we first review other definitions in the literature and explain why they are unsatisfactory. For example, many papers use process-to-process edges for their waits-for graphs, when in fact the appropriate ones to use are resource-to-resource edges.

All current definitions use real time; in contrast, we define a deadlock in terms of a vector of local times. Our main reason for doing so is as follows: From the point of view of a detector, everything it can deduce must be based on what it *can* observe of what is true at various sites. By adopting a definition that is in terms of local times, we reflect this reality and facilitate the development of a theory that is based on locally observable facts. Such a theory is then immediately applicable to the analysis of existing algorithms and the synthesis of new algorithms. In this sense, the definition provides for an *operational* theory.

We also introduce the concept of a quasi-deadlock. The importance of this concept lies in the conjunction of two facts: (1) what most existing algorithms detect is a quasi-deadlock, and (2) a quasi-deadlock may not be a deadlock — the timing of aborting processes distinguishes one from the other.

However, a deadlock is intuitively equivalent to the existence of a cycle at some (scalar) instant in some waits-for graph. We prove this equivalence, which is in fact partial because (again)

of aborting processes. Besides formalizing our intuition, the result shows that our definition of a deadlock in terms of a vector of local times is a generalization of a definition that is in terms of real time.

We next formalize several concepts that arise in deadlock detection. These concepts are of two types; one type concerns the coexistence of certain logical edges, and the other concerns the ordering in logical time of certain events. We then develop a theory to relate these concepts to a deadlock.

There are two reasons for developing this theory: first, it should help us grasp the ideas behind the various algorithms, and thus clarify the confusion caused by the errors and also bring some coherence to the diversity of techniques; second, new architectures and new applications will call for new tailor-made detection algorithms, and this theory should form a foundation for constructing such algorithms (which would be, one hopes, error-free).

The theory makes three contributions: (1) It examines the role of aborting processes in deadlock detection — hence the concept of a quasi-deadlock. (2) It analyzes the difference between a process and a resource. For instance, it shows that the order in time of formation is important for edges local to a process, but for edges local to a resource, it is the order in the time of deletion that is important. (3) It reveals the structure of a deadlock detector to be a collaboration of the two concepts just mentioned; viz. a typical detector is based on the deductions, via the ordering of some events, that follow from some coexistence observation.

Although this work is motivated by distributed deadlocks, the theory applies to centralized systems as well. And although it is focused on deadlock detection, it also applies to deadlock avoidance — we illustrate this by proving a folk theorem on resource ordering.

The theory is applied to an analysis of several well-known errors and algorithms in the literature. Among the latter are Obermarck's distributed waits-for graphs, Chandy and Misra's probes, Mitchell and Merritt's labels, as well as Chandy and Lamport's snapshots. Our ability to apply, with ease, the theory to these algorithms supports our claim that logical edges and logical time provide a unified framework for analyzing detection algorithms, and that a theory based on observable local facts would be easy to apply.

In the following, Section 2 (Axioms) states formally the assumptions we make about the system, Section 3 (Definition) presents our definition of a deadlock, Section 4 (Theory) develops the theory from Sections 2 and 3, Section 5 (Application) examines the physical aspects of the system and critically surveys the literature, and Section 6 (Conclusion) summarizes what we have learnt and indicates some directions for future work.

2 Axioms

The theory we are constructing will be based on the axioms and lemmas in this section. These axioms serve to make rigorous and explicit all necessary operational assumptions. Our formulation of these axioms here may not be the most concise or the most elegant, but that does not matter; our focus is on the results that follow from the axioms, and we believe that any other rigorous treatment of the deadlock detection problem will yield the same results.

In the same vein, the notation we use may not be the best. Our aim at providing a uniform framework requires that we keep track of the timing of all events (requests, receipt of messages, aborts, etc.), because different algorithms base their actions on different sets of events. Our notation captures all events concisely, if cryptically.

We first describe informally the system we are considering; this description should help the reader tackle the axioms and notation that follow.

The system consists of processes and resources. Processes and resource managers communicate by sending and receiving messages. A process requests for exclusive access to one resource at a time, and does not release resources it has acquired while waiting for an outstanding request. A process may abort at any time, thus releasing all resources it has acquired and canceling all outstanding requests. A resource manager can grant received requests in any order and after arbitrary delays, as long as there is no more than one granted request for the resource at any time.

2.1 Logical Edges

Processes are elements of a nonempty set $\mathcal{P}$, and *resources* are elements of a nonempty set $\mathcal{R}$. Let $\mathcal{Q} = \mathcal{P} \cup \mathcal{R}$. For each $Q \in \mathcal{Q}$, there is a nonempty set of *Q-time* (or *local time*) $\mathcal{T}_Q$ that is totally ordered by some $<_Q$. The elements of $\mathcal{T} = \bigcup_{Q \in \mathcal{Q}} \mathcal{T}_Q$ are called *times*. If $(\mathcal{T}_Q, <_Q) = (\mathcal{T}_{Q'}, <_{Q'})$ for all $Q, Q' \in \mathcal{Q}$, so $\mathcal{T}$ is totally ordered, we say there is a *common clock*. Lamport's logical clock [L] and real time are examples of a common clock. Our model includes the possibility that real time exists, but $\mathcal{T}$ is not real time itself.

All times are logical; we do not assume the existence of physical clocks. The physical implications of our axioms are discussed in Section 5.1.

The predicates $2e$ and $3e$ are defined over $\mathcal{T}^2$ and $\mathcal{T}^3$ respectively, where $e = \vec{P}R$, $P\vec{R}$, $\vec{R}P$ or $R\vec{P}$, and $P \in \mathcal{P}$, $R \in \mathcal{R}$. The atom $2e(T,t)$ is called an *edge*, and t is the *time of formation* of the edge. (Thus the time of formation, as well as the timestamp T, are integral to the definition of the edge.) The atom $3e(T,t,t')$ implies $2e(T,t)$ — see Lemma 1(iii) — and t' is called the *time of deletion* of the edge $2e(T,t)$.

Henceforth, we write $e(T,t)$ for $2e(T,t)$ and $e(T,t,t')$ for $3e(T,t,t')$. An edge $e(T,t)$ is *local* to $Q \in \mathcal{Q}$ if and only if $e = \vec{Q}Q'$ or $e = Q'\vec{Q}$ for some Q', and time s is *local* to $e(T,t)$ if and only if $s \in \mathcal{T}_Q$ where $e(T,t)$ is local to Q. For example, $R\vec{P}(T,t)$ is local to P, and any $s \in \mathcal{T}_R$ is local to $\vec{R}P(T,t)$.

2.2 Logical Time

We assume there are no lost, fraudulent or duplicated messages. This assumption is formalized by the first three axioms, which are for two message predicates *send* and *receive* defined over $\mathcal{Q} \times \mathcal{Q} \times \mathcal{T}$ and $\mathcal{Q} \times \mathcal{Q} \times \mathcal{T} \times \mathcal{T}$ respectively.

Axiom A1 $send(Q, Q', t) \rightarrow (t \in \mathcal{T}_Q) \wedge \exists t'\ receive(Q, Q', t, t')$
[$send(Q, Q', t)$ means Q sends a message to Q' at Q-time t; no message is lost.]

Axiom A2 $receive(Q, Q', t, t') \rightarrow (t' \in \mathcal{T}_{Q'}) \wedge send(Q, Q', t)$
[Q' cannot receive a message that was never sent.]

Axiom A3 $receive(Q, Q', t, t') \wedge receive(Q, Q', t, t'') \rightarrow t' = t''$
[Messages are not duplicated.]

The following macro-predicates are useful in the statement of some of the axioms.

[request message] $req\_msg(P, R, T, T, t') \equiv \vec{P}R(T, T) \wedge receive(P, R, T, t')$
[grant message] $grant\_msg(R, P, T, t, t') \equiv \vec{R}P(T, t) \wedge receive(R, P, t, t')$
[release message] $rel\_msg(P, R, T, t, t') \equiv \exists u\ R\vec{P}(T, u, t) \wedge receive(P, R, t, t')$

Messages and $<_Q$ induce a partial ordering $\prec$ on $\mathcal{T}$, as follows [L]:
For all $t_1, t_2 \in \mathcal{T}$, $t_1 \prec t_2$ if and only if
(i) $t_1 <_Q t_2$ for some $Q \in \mathcal{Q}$, where $t_1, t_2 \in \mathcal{T}_Q$,
(ii) $receive(Q_1, Q_2, t_1, t_2)$ for some $Q_1, Q_2 \in \mathcal{Q}$, where $t_1 \in \mathcal{T}_{Q_1}$ and $t_2 \in \mathcal{T}_{Q_2}$,
or (iii) $t_1 \prec t$ and $t \prec t_2$ for some $t \in \mathcal{T}$.

We assume that the physics of message delivery guarantees the existence of such an ordering. In particular, for $t_1, t_2 \in \mathcal{T}_Q$, $t_1 \prec t_2$ if and only if $t_1 <_Q t_2$. This is a constraint that a common clock must satisfy. We write $t_1 < t_2$ if $t_1 <_Q t_2$ and Q is clear from the context.

2.3 Aborting Processes

A process may abort at any time, but it cannot request for more resources after that. Formally, we have a predicate *abort*, defined over $\mathcal{P} \times \mathcal{T}$ and satisfying two axioms:

Axiom A4 $abort(P, t) \rightarrow (P \in \mathcal{P}) \wedge (t \in \mathcal{T}_P)$
[$abort(P, t)$ means process P aborts at P-time t.]

Axiom A5 $abort(P, p) \wedge \vec{P}R(T, T) \rightarrow\ T < p$
[P issues no more requests after it aborts.]

As before, we define an abort message as follows:
$abort\_msg(P, R, t, t') \equiv abort(P, t) \wedge receive(P, R, t, t')$

2.4 Edges Local to a Process

We first consider edges local to a process. For expository purposes, we assign a *resource manager* M_R to each $R \in \mathcal{R}$. A process P can request a resource R at any time T, a fact represented by the edge $\vec{P}R(T,T)$. It then waits to receive the grant message from M_R. The edge is deleted at time v, i.e. $\vec{P}R(T,T,v)$, when P stops waiting, either because it receives the grant message, or because it aborts. If it receives the grant message, we say it *acquires* R, and this is represented by $R\vec{P}(T,v)$. Sometime later at v', say, P releases R and deletes $R\vec{P}(T,v)$, i.e. $R\vec{P}(T,v,v')$. This description, with further details, is formalized by Axioms A6 to A13.

Axiom A6 $\vec{P}R(T,t) \rightarrow (T \in \mathcal{T}_P) \wedge (t = T) \wedge send(P,R,T)$
[$\vec{P}R(T,T)$ means P sends a request for R at local time T.]

Axiom A7 $\vec{P}R(T,T,v) \rightarrow (v \in \mathcal{T}_P) \wedge \vec{P}R(T,T) \wedge \forall z < v \neg \vec{P}R(T,T,z)$
[P stops waiting for R at time v.]

Axiom A8 $\vec{P}R(T,T) \wedge \forall z < p \neg \vec{P}R(T,T,z) \wedge abort(P,p) \rightarrow \vec{P}R(T,T,p) \wedge send(P,R,p)$
[When P aborts, it cancels outstanding requests.]

Axiom A9 $\vec{P}R(T,T) \wedge \forall z < p \neg \vec{P}R(T,T,z) \wedge grant\_msg(R,P,T,r,p) \rightarrow \vec{P}R(T,T,p)$
[P stops waiting for R when it receives the grant message.]

Axiom A10 $\vec{P}R(T,T,v) \wedge \neg abort(P,v) \leftrightarrow R\vec{P}(T,v)$
[$R\vec{P}(T,v)$ means P requested R at time T and acquires it at time v.]

Axiom A11 $R\vec{P}(T,u) \rightarrow \exists r \; grant\_msg(R,P,T,r,u)$
[P acquires R only if it was granted.]

Axiom A12 $R\vec{P}(T,u,v) \rightarrow (v \in \mathcal{T}_P) \wedge R\vec{P}(T,u) \wedge (u < v) \wedge \forall z < v \neg R\vec{P}(T,u,z) \wedge send(P,R,v)$
[P releases R — only once — through a message to M_R.]

Axiom A13 $R\vec{P}(T,u) \wedge \forall z < p \neg R\vec{P}(T,u,z) \wedge abort(P,p) \rightarrow R\vec{P}(T,u,p)$
[When P aborts, it must release any resource it is holding.]

2.5 Edges Local to a Resource

We next consider edges local to a resource. Suppose process P sends at time T a request for R, which the resource manager M_R receives at time x; we represent this with $P\vec{R}(T,x)$. If M_R grants the request at time y, the edge is also deleted at y, i.e. $P\vec{R}(T,x,y)$. The grant itself is represented by another edge $\vec{R}P(T,y)$, which is deleted, say $\vec{R}P(T,y,y')$, only when M_R receives

a message at time y' from P for releasing R. The details are specified by Axioms A14 to A19.

Axiom A14 $req\_msg(P,R,T,T,x) \wedge \neg(\exists p \exists r\ abort\_msg(P,R,p,r) \wedge (r < x)) \rightarrow P\vec{R}(T,x)$
[$P\vec{R}(T,x)$ means M_R receives at time x a request from P — sent at time T — and the request was not nullified by P aborting.]

Axiom A15 $P\vec{R}(T,x) \rightarrow req\_msg(P,R,T,T,x)$
[M_R notes an outstanding request only if it receives a request message.]

Axiom A16 $P\vec{R}(T,x,y) \rightarrow (y \in \mathcal{T}_R) \wedge P\vec{R}(T,x) \wedge (x < y) \wedge \forall z < y \neg P\vec{R}(T,x,z)$
[M_R cancels an outstanding request only once.]

Axiom A17 $P\vec{R}(T,x,y) \wedge \neg\exists p\ abort\_msg(P,R,p,y) \rightarrow send(R,P,y)$
[If M_R cancels P's request, either P aborted or M_R is sending a grant message.]

Axiom A18 $\exists x\ P\vec{R}(T,x,y) \wedge send(R,P,y) \leftrightarrow \vec{R}P(T,y)$
[$\vec{R}P(T,y)$ means M_R grants R to P via a message sent at R-time y.]

Axiom A19 $\vec{R}P(T,x,y) \rightarrow (y \in \mathcal{T}_R) \wedge \vec{R}P(T,x) \wedge \exists p\ rel\_msg(P,R,T,p,y)$
[M_R retrieves R from P only if P releases it.]

2.6 Constraints on Processes and Resources

The rest of the axioms, A20 to A24, formalize behavioral assumptions about processes and resources. Essentially, they say that resources are not shared and a process does not request a resource it already holds, nor make requests or release resources while waiting for a request to be granted. We need a definition of *existence* to formalize the latter constraints.

We say an edge $e(T,t)$ formed *before* time s if and only if $t \preceq s$. Further, $e(T,t)$ *exists* at time s if and only if $s : e(T,t)$, which is defined by

$$s : e(T,t) \equiv e(T,t) \wedge t \preceq s \wedge \forall t' \preceq s\ \neg e(T,t,t').$$

If $\mathcal{T}$ is real time or if s is local to $e(T,t)$, then $s : e(T,t)$ means the edge has formed and has not been deleted by time s. For example, we can formalize the concept of a blocked process thus: P is *blocked* at local time $t \in \mathcal{T}_P$ if and only if $t : \vec{P}R(T,T)$ for some resource R and some time T. If $\mathcal{T}$ is not real time and s is not local to the edge, say $s \in \mathcal{T}_Q$, then $s : e(T,t)$ intuitively means Q *could have* heard about the formation of the edge, but not its deletion.

Note carefully the meaning of $\neg s : e(T,t)$ — if $e = R\vec{P}$, say, it is possible that $R\vec{P}(T,t)$ and the edge does not exist at some $s \in \mathcal{T}_R$, $t \prec s$, although M_R did not receive the release message from P by time s.

Axiom A20 $\vec{R}P(T,x) \wedge \vec{R}P'(T',x') \wedge (x < x') \rightarrow \exists y \leq x'\ \vec{R}P(T,x,y)$
[If R is granted to P and later to P', then it was earlier retrieved from P.]

Axiom A21 $\vec{R}P(T,x) \wedge \vec{R}P'(T',x) \rightarrow (P = P') \wedge (T = T')$
[R cannot be granted to two different processes simultaneously.]

Axiom A22 $\vec{P}R(T',T') \wedge t : R\vec{P}(T,u) \rightarrow T' \neq t$
[A process does not request a resource it is already holding.]

Axiom A23 $t : \vec{P}R(T,T) \wedge \vec{P}R'(t,t) \rightarrow (T = t) \wedge (R = R')$
[A process does not request any resource while waiting for a resource.]

Axiom A24 $R\vec{P}(T,u,v) \wedge v' : \vec{P}R'(T',T') \rightarrow v \neq v'$
[A process does not release any resource while waiting for a resource.]

Axioms A23 and A24 say that a process has *single locus.*

2.7 Other Constraints

There are three other constraints that would be needed in a real system, but which we omit from our list of axioms because they do not affect the theory:

$$P\vec{R}(T,x) \wedge \forall z < r \neg P\vec{R}(T,x,z) \wedge abort\_msg(P,R,p,r) \rightarrow P\vec{R}(T,x,r) \quad \text{(i)}$$

$$P\vec{R}(T,x,y) \wedge \exists p\ abort\_msg(P,R,p,y) \rightarrow \neg send(R,P,y) \quad \text{(ii)}$$

$$\vec{R}P(T,t) \wedge \forall z < r \neg \vec{R}P(T,t,z) \wedge rel\_msg(P,R,T,p,r) \rightarrow \vec{R}P(T,t,r) \quad \text{(iii)}$$

In (i), M_R is required to cancel outstanding requests from aborted transactions; in (ii), M_R must not grant R to an aborted transaction (see A18); in (iii), M_R must retrieve R when informed that it has been released. Violation of these constraints may make a resource forever unavailable, but it wouldn't affect the soundness of the theory. Note from the omission of these constraints that we do not attempt to be complete in our axiomatization.

2.8 Lemmas

The following lemmas are immediate consequences of the axioms. Several of the claims are "obvious", and serve as operational arguments in the literature. The results are rather unreadable, so we have added annotations. (Incidentally, a comparison between a formal statement and its annotation shows how difficult it would be for an operational argument to be precise.) The proofs follow directly from the axioms, but are even more unreadable, so we put them in an appendix. The reader who wants to be convinced that we use no operational argument in our proofs, or who wants to see how we work with the axioms, should peruse the appendix.

The first thing to note from the axioms is that times of formation and deletion are always

local to the edge. For example, $P\vec{R}(T,x,y)$ implies $x,y \in \mathcal{T}_R$. In trying to grasp these results intuitively, one should take into account the possibility that the time of existence of an edge may not be local to that edge.

The first lemma lists the elementary properties of edges.

Lemma 1

(i) $e(T,t)$ implies $\vec{P}R(T,T)$ and $T \preceq t$ ($T \prec t$ if $e \neq \vec{P}R$).
[All edges originate from a request for a resource.]

(ii) $e(T,t)$ and $e(T,t')$ imply $t = t'$.
[Time of formation of an edge is unique.]

(iii) $e(T,t,t')$ implies $e(T,t)$ and $t < t'$
[An edge can be deleted only if it formed earlier.]

(iv) $e(T,t_1,t_2)$ and $e(T,t_1',t_2')$ imply $t_1 = t_1'$ and $t_2 = t_2'$.
[Time of deletion of an edge is unique.]

(v) $e(T,t,t')$ implies $s : e(T,t)$ for all $t \preceq s \prec t'$.
[An edge exists at all times after its formation and before its deletion.]

(vi) $s : e(T,t)$ implies $s' : e(T,t)$ for all $t \preceq s' \preceq s$.
[If an edge exists at time s, it exists at all earlier times after its formation.]

□

The second lemma makes deductions about the existence and deletion of edges.

Lemma 2

(i) $R\vec{P}(T,u)$ implies $t : \vec{P}R(T,T)$ for all $T \preceq t \prec u$.
[After requesting a resource and before acquiring it, the process must wait.]

(ii) $\vec{R}P(T,x)$ implies for all $T \preceq t \preceq x$, either $t : \vec{P}R(T,T)$ or $abort(P,p)$ for some $p \preceq t$.
[After a request is issued and before it is granted, the process is waiting for the resource.]

(iii) $P\vec{R}(T,x)$ implies for all $T \preceq t \preceq x$, either $t : \vec{P}R(T,T)$ or $abort(P,p)$ for some $p \preceq t$.
[After a request is issued and before it is received, the process is waiting for the resource.]

(iv) $\vec{P}R(T,T,v)$ implies $P\vec{R}(T,x,y)$ for some x and y, $T \prec x < y \prec v$, or $abort(P,v)$.
[P stops waiting for R when P aborts or P is granted R.]

(v) $s : P\vec{R}(T,x)$ implies $s : \vec{P}R(T,T)$ or $abort(P,p)$ for some $p \preceq s$.
[If M_R has not yet granted P's request, then P — if unaborted — must be waiting for R.]

(vi) $s : R\vec{P}(T,u)$ implies $s : \vec{R}P(T,x)$ for some $x \prec u$.
[If a process is holding a resource R, then M_R has not yet retrieved R.]

(vii) Suppose $R\vec{P}(T,u)$ and $\vec{P}R(T',T')$. If $u \leq T'$, then $R\vec{P}(T,u,v)$ for some $v \leq T'$.
[If P is waiting for R, which it has acquired before, then P must have released R earlier.]

(viii) Suppose $\vec{R}P(T,x)$ and $t : \vec{P}R(T,T)$. If $x \preceq t$, then $s : \vec{R}P(T,x)$ for all $x \preceq s \preceq t$.
[M_R could not have retrieved a resource it has granted to P if P is still waiting for it.] □

The third lemma considers the consequences if two edges exist at the same time.

Lemma 3

(i) $s : R\vec{P}(T,u)$ and $s : R\vec{P}(T',u')$ imply $T = T'$ and $u = u'$.
[A process cannot have two requests for the same resource satisfied simultaneously.]

(ii) $s : \vec{P}R(T,T)$ and $s : \vec{P}R'(T',T')$ imply $R = R'$ and $T = T'$.
[A process cannot simultaneously wait for two different resources.]

(iii) $s : R\vec{P}(T,u)$ and $s : \vec{P}R'(T',T')$ imply $u \leq T'$.
[If P is holding R and waiting for R', then R was acquired before the latter request.]

(iv) $s : \vec{R}P(T,x)$ and $s : \vec{R}P'(T',x')$ imply $P = P'$, $T = T'$ and $x = x'$.
[R cannot be simultaneously granted to two different processes.]

(v) Suppose $s : R\vec{P}(T,u)$ and $s : \vec{P}R'(T',T')$. If $R\vec{P}(T,u,v)$, then $\vec{P}R'(T',T',v')$ for some $v' \leq v$.
[If P is holding R and waiting for R', then it must acquire R' before releasing R.]

(vi) Suppose $s : P'\vec{R}(T',x')$ and $s : \vec{R}P(T,x)$. If $P'\vec{R}(T',x',y')$, then either $abort\_msg(P',R,p',y')$ for some p' or $\vec{R}P(T,x,y)$ for some $y \leq y'$.
[If P' is waiting for R, which has been granted to P, then R must be retrieved from P before being granted to P'.]
□

3 Definition

Before defining what a deadlock is, we first review (Section 3.1) the current definitions of a deadlock. The definition we give (Section 3.2) is not based on simultaneity, but we show (Section 3.3) that this definition is equivalent, almost, to the simultaneous existence of a cycle of edges. Even so, our definition of simultaneous existence does not assume there is real time.

3.1 Review: What is a deadlock?

All definitions of deadlocks are in terms of a waits-for graph of some kind, and all current definitions of such a graph assume there is real time. (Deadlocks are not defined in some papers [CKST, CL, HR, K, KS, SN].)

When a deadlock is defined in terms of a waits-for graph, how the edges in this graph are defined becomes crucial. In many definitions, the waits-for graph consists of process-to-process edges, where a process is sometimes called a *transaction*. **Obermarck** [O], for example, define these edges to "represent the wait-for relationship between transactions", but does not define the relationship itself. There are similar omissions in other papers [RBC, KKNR].

For **Mitchell and Merritt** [MiMe], "an edge indicates that one process is waiting on a resource held exclusively by another". That could mean $t : \vec{P}'R(T', T')$ and $t : R\vec{P}(T, u)$ for some R, or $t : \vec{P}'R(T', T')$ and $t : \vec{R}P(T, x)$, or some other possibility. There is a further ambiguity in the location of t. One cannot claim that t is in real time — Mitchell and Merritt's edges are *physical* (i.e. some data structure), so the location of t must be specified. Several other papers adopt a similar definition [GS, KMIT, SKYO, TB].

Wuu and Bernstein use a waits-for graph wherein an edge exists from P' to P if P' "executes a request operation for a resource owned by" P, and this edge exists from the time the request arrives at the resource to the time P releases it [WB]. As in the previous case, the edge definition is ambiguous. Furthermore, the definition of existence leaves out the possibility that when P's request arrives at the resource R, P' may be waiting for R while some P'' is holding R; if now P'' releases R and P acquires it, the definition would give an edge from P' to P that exists from the time P''s request arrives at R, possibly before P even started. Again, several papers share this problem of not considering the consequences when resources are released, as when a process is aborted to resolve a deadlock [CM, KMIT, MeMu, O].

Badal uses both processes and resources in his graph [B], where an edge "from a resource node to a transaction node indicates that the transaction has a lock on the resource", while an edge "from a transaction node to a resource indicates that the transaction has placed an intention lock on that resource", meaning the transaction is waiting to lock it; and "a cycle in the (graph) indicates the existence of a deadlock". Since Badal's lock tables are kept at the resources, the edges are of the form $\vec{R}P(T, x)$ and $P'\vec{R}(T', x')$ respectively. Now consider the case where $\mathcal{T}$ is real time, and let $r < s < t$. Suppose

$$\vec{R}_2 P_1(T_1, x_1),\ P_1\vec{R}_1(T_1', x_1'),\ \vec{R}_1 P_0(T_0, x_0),\ P_0\vec{R}_0(T_0', x_0') \text{ and } \vec{R}_0 P_2(T_2, x_2)$$

all exist at time r. At time s, P_2 releases R_0 and requests R_2, i.e. $R_0\vec{P}_2(T_2, u_2, s)$ and $\vec{P}_2 R_2(s, s)$

for some u_2. If P_2's request reaches R_2 at time t, before P_2's release reaches R_0, then a cycle exists at time t, but it will be broken once R_0's manager receives P_2's release message.

Sanders and Heuberger also use both processes and resources in their waits-for graph [SH], in which an edge from Q' to Q "represents the situation where (Q') is waiting for (Q) to do something", and a deadlock corresponds to a cycle in this graph. By their description, the edges are of the form $\vec{P}R(T,T)$ or $\vec{R}P(T,x)$. Again consider an example where $\mathcal{T}$ is real time, and

$$\vec{R}_2P_1(T_1,x_1),\ \vec{P}_1R_1(T_1',T_1'),\ \vec{R}_1P_0(T_0,x_0),\ \vec{P}_0R_0(T_0',T_0') \text{ and } \vec{R}_0P_2(T_2,x_2)$$

all exist at time r. Suppose $r < s < t$. Now P_2 releases R_0 at time s and then sends at time t a request for R_2, i.e. $R_0\vec{P}_2(T_2,u_2,s)$ and $\vec{P}_2R_2(t,t)$ for some u_2. If P_2's release message does not reach R_0 by time t, then the cycle

$$\vec{R}_2P_1(T_1,x_1),\ \vec{P}_1R_1(T_1',T_1'),\ \vec{R}_1P_0(T_0,x_0),\ \vec{P}_0R_0(T_0',T_0'),\ \vec{R}_0P_2(T_2,x_2) \text{ and } \vec{P}_2R_2(t,t)$$

exists at time t. But there is no deadlock, because once R_0's manager receives the message, $\vec{R}_0P_2(T_2,x_2)$ is deleted and the cycle is broken.

Although the authors' operational description of an edge from Q to Q' seems to be from Q's point of view, their *implementation* of this definition is at Q'. For example, an edge from P to R is inserted into their physical waits-for graph only when the request arrives at R, and this edge is removed when the request is granted. Similarly, insertion and deletion of edges from R to P coincide (respectively) with P's receipt of the grant message and P's release of the resource. Thus a cycle among P_0, P_1, P_2 and R_0, R_1, R_2 in this implementation will instead be of the form

$$R_2\vec{P}_1(T_1,u_1),\ P_1\vec{R}_1(T_1',x_1'),\ R_1\vec{P}_0(T_0,u_0),\ P_0\vec{R}_0(T_0',x_0'),\ R_0\vec{P}_2(T_2,u_2) \text{ and } P_2\vec{R}_2(T_2',x_2').$$

By Corollary 6, the simultaneous existence of such a cycle of edges does in fact imply a deadlock.

We see from this review that most papers in the literature for distributed deadlock detection have either no definition of a deadlock, or a definition that is ambiguous, incomplete, or erroneous.

3.2 Definition

How should a deadlock be defined? If we follow the common perception of a deadlock as a cycle of processes waiting for each other simultaneously, then the definition would have the form "there is a deadlock at time t if and only if $\mathcal{C}(t)$", where $\mathcal{C}$ is a statement about the existence of a cycle. Such a definition would be unsatisfactory because the fundamental characteristic of a deadlock is not that the processes are in a cycle, but that they must wait forever for the resources they requested. Moreover, what would t be?

The nature of t is related to the existence of real time, over which there is a controversy. To paraphrase Pratt's argument [P], for three processes, one on Earth, one orbiting Jupiter, and one swinging pass the sun, what definition of real time would be relativistically acceptable?

If we do not assume the existence of real time, then there is no concept of a global state and t would be some local time. In that case, the meaning of "there is a deadlock at time t" becomes

unintuitive, because t is not physically related to the local times elsewhere in the system. It is then necessary to specify the local times $t_1, \ldots, t_n$ for the processes that are involved, so the definition would instead be of the form "there is a deadlock at $t_1, \ldots, t_n$ if and only if $\mathcal{D}(t_1, \ldots, t_n)$", where $\mathcal{D}$ is some predicate. However, we should insist that such a definition agree with our intuition in the presence of real time.

There is another, more important, reason why we favor a definition that is in terms of a vector of times, and which we will give later (Section 3.4). In any case, we are now led to the following scheme for defining a deadlock: The definition will be in terms of $\mathcal{D}(t_1, \ldots, t_n)$, where $\mathcal{D}$ is a condition on processes that causes them to wait forever. Furthermore, if real time exists, then $\mathcal{D}(t_1, \ldots, t_n)$ must be equivalent to $\mathcal{C}(t)$, where $\mathcal{C}$ specifies the existence of a cycle.

Let n be a positive integer. A *resource* n-cycle is a set of edges $P_i \vec{R}_i(T_i', x_i')$ and $\vec{R}_i P_{i-1}(T_{i-1}, x_{i-1})$ where $i \in \mathbf{Z}_n$ (i.e. $i = 0, 1, \ldots, n-1$ and addition and subtraction are modulo n). The following illustrates a resource 3-cycle:

$$P_2 \xrightarrow[(T_2', x_2')]{} R_2 \xrightarrow[(T_1, x_1)]{} P_1 \xrightarrow[(T_1', x_1')]{} R_1 \xrightarrow[(T_0, x_0)]{} P_0 \xrightarrow[(T_0', x_0')]{} R_0 \xrightarrow[(T_2, x_2)]{} P_2$$

Thus, Badal defines a deadlock as the (simultaneous) existence of a resource n-cycle of edges.

Similarly, a *process n-cycle* is a set of edges $R_{i+1} \vec{P}_i(T_i, u_i)$ and $\vec{P}_i R_i(T_i', T_i')$ where $i \in \mathbf{Z}_n$. The following illustrates a process 2-cycle:

$$R_0 \xrightarrow{(T_1, u_1)} P_1 \xrightarrow{(T_1', T_1')} R_1 \xrightarrow{(T_0, u_0)} P_0 \xrightarrow{(T_0', T_0')} R_0$$

Intuitively, since processes have single locus (A23, A24) and resources are not shared (A20, A21), we expect the processes, as well as the resources, in a deadlock to be distinct. Formally, we say $P_0, \ldots, P_{n-1}$ and $R_0, \ldots, R_{n-1}$ in a process n-cycle are *distinct up to cyclic repetitions* if and only if $n = qc$ for some positive integers q and c, $c \geq 2$, $P_0, \ldots, P_{c-1}$ are distinct, $R_0, \ldots, R_{c-1}$ are distinct, and for all $i \in \mathbf{Z}_n$, $\alpha_i = \alpha_{i+c}$ for $\alpha = P, R, T, T', u$. Thus, the process n-cycle consists of q repetitions of a cycle with c distinct processes.

We say processes $P_0, \ldots, P_{n-1}$ are *quasi-deadlocked* at $p_0, \ldots, p_{n-1}$ over resources $R_0, \ldots, R_{n-1}$ if and only if for each $i \in \mathbf{Z}_n$,

(a) $p_i \in \mathcal{T}_{P_i}$, $p_i : R_{i+1} \vec{P}_i(T_i, u_i)$ and $p_i : \vec{P}_i R_i(T_i', T_i')$ for some T_i, T_i' and u_i, and

(b) there is x_{i-1} such that $\vec{R}_i P_{i-1}(T_{i-1}, x_{i-1})$ but $\neg P_i \vec{R}_i(T_i', x_i', y_i')$ for all $x_i', y_i' \leq x_{i-1}$.

Further, we say $P_0, \ldots, P_{n-1}$ are *deadlocked* at $p_0, \ldots, p_{n-1}$ over resources $R_0, \ldots, R_{n-1}$ if and only if they are quasi-deadlocked and for all $i, j \in \mathbf{Z}_n$, $\neg abort(P_j, p)$ for all $p \preceq p_i$.

In the above definitions, condition (a) says that each process P_i observes — at local time p_i — the existence of two edges $R_{i+1} \vec{P}_i(T_i, u_i)$ and $\vec{P}_i R_i(T_i', T_i')$, while condition (b) says that each resource R_i was granted to P_{i-1} at x_{i-1}, and the request from P_i has either not arrived, or has arrived but not been granted by time x_{i-1}. If processes do not abort, then a quasi-deadlock is a deadlock; otherwise, one must check the extra condition that differentiates the two. Intuitively, this extra condition says that each P_i could not have heard about an abort by any of the other processes at time p_i.

The following result justifies these two definitions.

Theorem 4

Suppose $P_0, \ldots, P_{n-1}$ are *quasi-deadlocked* at $p_0, \ldots, p_{n-1}$ over resources $R_0, \ldots, R_{n-1}$. Then $n \geq 2$ (so there are at least two processes) and

(i) if $\vec{P}_k R_k(T'_k, T'_k, v'_k)$, then $abort(P_i, p)$ for some $i \in \mathbf{Z}_n$ and some $p \preceq v'_k$ (thus, unless one of the processes aborts, all of them will wait forever for the resources they requested); and

(ii) if $P_0, \ldots, P_{n-1}$ are deadlocked, then $P_0, \ldots, P_{n-1}$ and $R_0, \ldots, R_{n-1}$ are distinct up to cyclic repetitions.

Proof

Suppose $n = 1$, so $p_0 : R_0\vec{P}_0(T_0, u_0)$ and $p_0 : \vec{P}_0 R_0(T'_0, T'_0)$. If $T'_0 < T_0 (\prec u_0 \leq p_0)$, then (Lemma 1(vi)) $T_0 : \vec{P}_0 R_0(T'_0, T'_0)$; but $R_0\vec{P}_0(T_0, u_0)$ implies (Lemma 1(i)) $\vec{P}_0 R_0(T_0, T_0)$, thus contradicting (A23) $T'_0 < T_0$. If $u_0 \leq T'_0 (\leq p_0)$, then (Lemma 1(vi)) $T'_0 : R_0\vec{P}_0(T_0, u_0)$, contradicting (A22) $\vec{P}_0 R_0(T'_0, T'_0)$.

Therefore $T_0 \leq T'_0 < u_0$, so (Lemma 2(i)) $T'_0 : \vec{P}_0 R_0(T_0, T_0)$, and hence (A23) $T_0 = T'_0$. Now (Lemma 3(iii)) $u_0 \leq T'_0 = T_0$, and we get a contradiction.

(i) Suppose $\vec{P}_k R_k(T'_k, T'_k, v'_k)$ and $\neg abort(P_i, p)$ for all $i \in \mathbf{Z}_n$ and all $p \preceq v'_k$. Then $\neg abort(P_k, v'_k)$ implies (A10, A11) $\vec{R}_k P_k(T'_k, y'_k)$ for some $y'_k \prec v'_k$, so (A18) $P_k\vec{R}_k(T'_k, x'_k, y'_k)$ for some $x'_k < y'_k$, and (from (b)) $x_{k-1} < y'_k$.

If $x'_k \leq x_{k-1} (< y'_k)$, then (Lemma 1(v)) $x_{k-1} : P_k\vec{R}_k(T'_k, x'_k)$, so $x_{k-1} : \vec{R}_k P_{k-1}(T_{k-1}, x_{k-1})$ and $P_k\vec{R}_k(T'_k, x'_k, y'_k)$ imply (Lemma 3(vi)) $\vec{R}_k P_{k-1}(T_{k-1}, x_{k-1}, y_{k-1})$ for some $y_{k-1} \leq y'_k$.

If $x_{k-1} < x'_k (< y'_k)$, then $\vec{R}_k P_{k-1}(T_{k-1}, x_{k-1})$ and $\vec{R}_k P_k(T'_k, y'_k)$ again imply (A20) $\vec{R}_k P_{k-1}(T_{k-1}, x_{k-1}, y_{k-1})$ for some $y_{k-1} \leq y'_k$.

Either way, $\vec{R}_k P_{k-1}(T_{k-1}, x_{k-1}, y_{k-1})$ now implies (A19) $R_k\vec{P}_{k-1}(T_{k-1}, u_{k-1}, v_{k-1})$ for some $v_{k-1} \prec y_{k-1}$. It follows (A24) that $\vec{P}_{k-1} R_{k-1}(T'_{k-1}, T'_{k-1})$ does not exist at v_{k-1}.

Now $v_{k-1}, T'_{k-1} \in \mathcal{T}_{P_{k-1}}$, so either $v_{k-1} < T'_{k-1}$, or $T'_{k-1} \leq v_{k-1}$ and $\vec{P}_{k-1} R_{k-1}(T'_{k-1}, T'_{k-1}, v'_{k-1})$ for some $v'_{k-1} \leq v_{k-1}$. But $v_{k-1} < T'_{k-1}$ and $p_{k-1} : \vec{P}_{k-1} R_{k-1}(T'_{k-1}, T'_{k-1})$ imply $v_{k-1} < p_{k-1}$, so $R_k\vec{P}_{k-1}(T_{k-1}, u_{k-1})$ does not exist at p_{k-1}, contradicting $p_{k-1} : R_k\vec{P}_{k-1}(T_{k-1}, u_{k-1})$.

Thus $\vec{P}_k R_k(T'_k, T'_k, v'_k)$ implies $\vec{P}_{k-1} R_{k-1}(T'_{k-1}, T'_{k-1}, v'_{k-1})$ for some $v'_{k-1} \leq v_{k-1} \prec y_{k-1} \leq y'_k \prec v'_k$. Inductively, $\vec{P}_k R_k(T'_k, T'_k, v'_k)$ and the cycle $\mathbf{Z}_n$ imply $\vec{P}_k R_k(T'_k, T'_k, v)$ for some $v < v'_k$, contradicting Lemma 1(iv). This contradiction proves the claim.

(ii) Suppose $P_i = P_j$ for some $i, j \in \mathbf{Z}_n$, $i \neq j$, so we have $p_i : \vec{P}_i R_i(T'_i, T'_i)$ and $p_j : \vec{P}_i R_j(T'_j, T'_j)$. We may assume $T'_i \leq T'_j$. Suppose $T'_i < T'_j$. Then (A23) $\vec{P}_i R_i(T'_i, T'_i)$ does not exist at T'_j, so $\vec{P}_i R_i(T'_i, T'_i, v'_i)$ for some $v'_i \leq T'_j$. By (ii), we have $abort(P_m, p)$ for some $m \in \mathbf{Z}_n$ and some $p \preceq v'_i \leq T'_j \leq p_j$, contradicting the hypothesis. Hence $T'_i = T'_j$, and (A23) $R_i = R_j$.

Suppose $R_i = R_j$ for some $i, j \in \mathbf{Z}_n$, $i \neq j$, so we have $p_{i-1} : R_i\vec{P}_{i-1}(T_{i-1}, u_{i-1})$ and $p_{j-1} : R_i\vec{P}_{j-1}(T_{j-1}, u_{j-1})$. Also (A11) $\vec{R}_i P_{i-1}(T_{i-1}, x_{i-1})$ and $\vec{R}_i P_{j-1}(T_{j-1}, x_{j-1})$, where

$x_{i-1} \prec u_{i-1}$ and $x_{j-1} \prec u_{j-1}$. Since $x_{i-1}, x_{j-1} \in \mathcal{T}_{R_i}$, we may assume $x_{i-1} \leq x_{j-1}$. Suppose $x_{i-1} < x_{j-1}$. Then (A20) $\vec{R}_i P_{i-1}(T_{i-1}, x_{i-1}, y_{i-1})$ for some $y_{i-1} \leq x_{j-1}$, so (A19, Lemma 1(iii)) $R_i \vec{P}_{i-1}(T_{i-1}, u_{i-1}, v_{i-1})$ for some $v_{i-1} \prec y_{i-1}$. That implies (Lemma 3(v)) $\vec{P}_{i-1} R_{i-1}(T'_{i-1}, T'_{i-1}, v'_{i-1})$ for some $v'_{i-1} \leq v_{i-1}$. By (ii), we have $abort(P_m, p)$ for some $m \in \mathbf{Z}_n$ and $p \preceq v'_{i-1} \leq v_{i-1} \prec y_{i-1} \leq x_{j-1} \prec u_{j-1} \leq p_{j-1}$, contradicting the hypothesis. Hence $x_{i-1} = x_{j-1}$, so (A21) $T_{i-1} = T_{j-1}$ and $P_{i-1} = P_{j-1}$, and (Lemma 1(ii)) $u_{i-1} = u_{j-1}$.

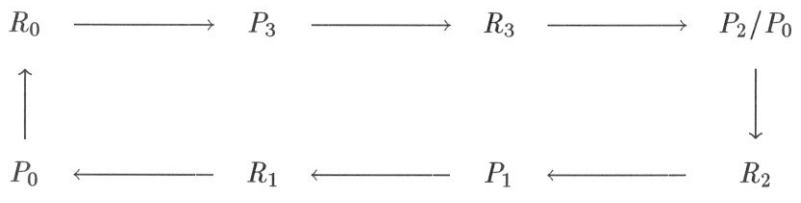
Therefore $P_0, \ldots, P_{n-1}$ are distinct if and only if $R_0, \ldots, R_{n-1}$ are distinct. If $P_0, \ldots, P_{n-1}$ are not distinct, say $P_i = P_j$ for some $i, j \in \mathbf{Z}_n$, $i < j$ and $j - i$ minimal, then inductively over $\mathbf{Z}_n$, we have $\alpha_{i-d} = \alpha_{j-d}$ for $\alpha = P, R, T, T', u$ and $d = 0, 1, 2, \ldots$; in particular, $\alpha_0 = \alpha_c$ where $c = j - i$, so $\alpha_k = \alpha_{k+c}$ for $k \in \mathbf{Z}_n$.

Let $n = qc + r$, where q and r are the quotient and remainder when n is divided by c (so $0 \leq r < c$). Then $\alpha_0 = \alpha_r$; since c is minimal, we have $r = 0$, so $n = qc$.

Finally, by the minimality of c, $P_0, \ldots, P_{c-1}$ are distinct and $R_0, \ldots, R_{c-1}$ are also distinct; furthermore, (a) and (b) are satisfied for $i \in \mathbf{Z}_c$, so $P_0, \ldots, P_{c-1}$ are quasi-deadlocked and $c \geq 2$. Thus, $P_0, \ldots, P_{n-1}$ and $R_0, \ldots, R_{n-1}$ are distinct up to cyclic repetitions. □

Conditions (a) and (b) form the predicate $\mathcal{D}$, and Theorem 4(i) says our definition of a quasi-deadlock has the property (viz. perpetually waiting processes) we required of $\mathcal{D}$ earlier. Surprisingly, there is no proof in the literature of this property, except for an early version (twenty-odd years ago) by Shoshani and Coffman [SC] for a centralized system. (Bracha and Toueg used a related property to *define* a deadlock [BT].) Any previous attempt to prove some version of Theorem 4(i) from existing definitions of a deadlock would have revealed the unsatisfactory nature of these definitions.

Why *quasi*? First, it is possible that $P_0, \ldots, P_{n-1}$ are quasi-deadlocked, but one of them has "already" aborted; second, $P_0, \ldots, P_{n-1}$ may not be distinct. We illustrate these with the example in Figure 1, which has real time as a common clock.



$\vec{P}_3 R_0(0,0)$	$\vec{R}_1 P_0(5,7)$	$\vec{P}_2 R_2(14,14)$	$P_1 \vec{R}_1(20,21)$
$P_3 \vec{R}_0(0,1)$	$R_1 \vec{P}_0(5,8)$	$\vec{P}_1 R_2(15,15)$	$abort(P_1, 23)$, $\vec{P}_1 R_1(20,20,23)$, $R_2 \vec{P}_1(15,19,23)$
$\vec{R}_0 P_3(0,2)$	$\vec{P}_2 R_3(9,9)$	$P_1 \vec{R}_2(15,16)$	$\vec{R}_2 P_1(15,17,24)$
$R_0 \vec{P}_3(0,3)$	$P_2 \vec{R}_3(9,10)$	$\vec{R}_2 P_1(15,17)$	$\vec{R}_2 P_2(14,25)$
$\vec{P}_3 R_3(4,4)$	$\vec{R}_3 P_2(9,11)$	$P_2 \vec{R}_2(14,18)$	$R_2 \vec{P}_2(14,26)$
$\vec{P}_0 R_1(5,5)$	$P_3 \vec{R}_3(4,12)$	$R_2 \vec{P}_1(15,19)$	$\vec{P}_0 R_0(26,27)$
$P_0 \vec{R}_1(5,6)$	$R_3 \vec{P}_2(9,13)$	$\vec{P}_1 R_1(20,20)$	$P_0 \vec{R}_0(26,28)$

Figure 1 P_0, P_1, P_2, P_3 quasi-deadlocked at 27, 21, 14, 9 over R_0, R_1, R_2, R_3.

In Figure 1, P_0, P_1, P_2, P_3 are quasi-deadlocked at 27, 21, 14, 9 over R_0, R_1, R_2, R_3, but P_1 has already aborted by time 27. Moreover, it is possible that $P_2 = P_0$ (see their actions at 8, 9, 26 and 27).

These observations motivated our definition for a deadlock: In our definition, a deadlock is a quasi-deadlock with a restriction on the timing of aborting processes; this restriction eliminates the first problem in the example, and consequently (Theorem 4(ii)) removes the second problem as well.

On the second problem, if the processes are quasi-deadlocked and $P_0, \ldots, P_{n-1}$ and $R_0, \ldots, R_{n-1}$ are distinct up to cyclic repetitions, does that mean there is a deadlock? No, and the scenario in Figure 1 is a counterexample.

In the definition of a deadlock, condition (b) is used to "hold" the cycle together — it is easy to construct an example to show that, without this condition, we may not have a deadlock. However, condition (a) suffices to define a deadlock if the cycle exists "simultaneously".

3.3 Simultaneity

We now consider the second requirement we set down for our definition, which is to relate it to our intuition of a deadlock as a set of processes waiting simultaneously for one another. Formally, we say a set of edges exist *simultaneously* at time $t \in \mathcal{T}$ if and only if they all exist at time t. Thus, if $t : R_{i+1}\vec{P}_i(T_i, u_i)$ and $t : \vec{P}_i R_i(T_i', T_i')$ where $i \in \mathbf{Z}_n$, then the process n-cycle exists simultaneously at t.

Theorem 5

(i) If a process n-cycle exists simultaneously at time t, then the processes in the cycle are deadlocked at some $p_0, \ldots, p_{n-1}$, where $p_i \preceq t$ for all $i \in \mathbf{Z}_n$.

(ii) Suppose $P_0, \ldots, P_{n-1}$ are deadlocked at $p_0, \ldots, p_{n-1}$ over $R_0, \ldots, R_{n-1}$, and for each i, $p_i \preceq t$ and $\neg abort(P_i, p)$ for all $p \preceq t$. Then the process n-cycle exists simultaneously at t.

Proof

(i) Assume $t : R_{i+1}\vec{P}_i(T_i, u_i)$ and $t : \vec{P}_i R_i(T_i', T_i')$, where $i \in \mathbf{Z}_n$. Note that $u_i, T_i' \in \mathcal{T}_{P_i}$. If $T_i' \leq u_i$, then $T_i' \leq u_i \preceq t$ (from $t : R_{i+1}\vec{P}_i(T_i, u_i)$) and $t : \vec{P}_i R_i(T_i', T_i')$ imply (Lemma 1(vi)) $p_i : \vec{P}_i R_i(T_i', T_i')$ and $p_i : R_{i+1}\vec{P}_i(T_i, u_i)$, where $p_i = u_i$. If $u_i < T_i'$, the same is similarly true for $p_i = T_i'$. Thus, for each $i \in \mathbf{Z}_n$, there is $p_i \in \mathcal{T}_{P_i}$ such that $p_i : \vec{P}_i R_i(T_i', T_i')$ and $p_i : R_{i+1}\vec{P}_i(T_i, u_i)$. Note that for all $i \in \mathbf{Z}_n$, $R_{i+1}\vec{P}_i(T_i, u_i)$ implies (A11) $\vec{R}_{i+1} P_i(T_i, x_i)$ for some $x_i \prec u_i$.

Suppose $P_k \vec{R}_k(T_k', x_k', y_k')$ for some $x_k' < y_k' \leq x_{k-1}$. If $abort\_msg(P_k, R_k, p', y_k')$ for some p', then $abort(P_k, p')$ and $p' \prec y_k'$. Now (A5) $T_k' < p' \prec y_k' \leq x_{k-1} \prec u_{k-1} \preceq t$ and $t : \vec{P}_k R_k(T_k', T_k')$ imply (Lemma 1(vi)) $p' : \vec{P}_k R_k(T_k', T_k')$. That implies $\neg \vec{P}_k R_k(T_k', T_k', z)$ for all $z \leq p'$, and thus (A8) $\vec{P}_k R_k(T_k', T_k', p')$ — a contradiction.

Therefore $\neg abort\_msg(P_k, R_k, p', y'_k)$ for all p'. Then (A17, A18) $\vec{R}_k P_k(T'_k, y'_k)$. If $y'_k = x_{k-1}$, then $\vec{R}_k P_k(T'_k, y'_k)$ and $\vec{R}_k P_{k-1}(T_{k-1}, x_{k-1})$ imply (A21) $P_{k-1} = P_k$ and $T_{k-1} = T'_k$, so $R_k \vec{P}_{k-1}(T_{k-1}, u_{k-1})$ becomes $R_k \vec{P}_k(T'_k, u_{k-1})$ and thus (A10) $\vec{P}_k R_k(T'_k, T'_k, u_{k-1})$, which contradicts $u_{k-1} \preceq t$ and $t : \vec{P}_k R_k(T'_k, T'_k)$.

Therefore $y'_k < x_{k-1}$. Now $t : R_k \vec{P}_{k-1}(T_{k-1}, u_{k-1})$ implies (Lemma 2(vi)) $t : \vec{R}_k P_{k-1}(T_{k-1}, x_{k-1})$. Since $y'_k < x_{k-1}$, we have (A20) $\vec{R}_k P_k(T'_k, y'_k, z'_k)$ for some $z'_k \leq x_{k-1}$. But $\vec{R}_k P_k(T'_k, y'_k, z'_k)$ implies (A19) $R_k \vec{P}_k(T'_k, u, v)$ for some $u < v \prec z'_k$ and hence (Lemma 1(iii), A10) $\vec{P}_k R_k(T'_k, T'_k, u)$, contradicting $u < v \prec z'_k \leq x_{k-1} \prec u_{k-1} \preceq t$ and $t : \vec{P}_k R_k(T'_k, T'_k)$.

We conclude that $\neg P_k \vec{R}_k(T'_k, x'_k, y'_k)$ for all $x'_k, y'_k \leq x_{k-1}$, so the processes are quasi-deadlocked. Next, suppose $abort(P_i, p)$ and $p \preceq p_j$ for some $i, j \in \mathbf{Z}_n$. Then (A8) $\vec{P}_i R_i(T'_i, T'_i, z)$ for some $z \preceq p \preceq p_j \preceq t$, contradicting $t : \vec{P}_i R_i(T'_i, T'_i)$. Therefore $P_0, \ldots, P_{n-1}$ are deadlocked.

(ii) The deadlock implies that for each $i \in \mathbf{Z}_n$, $p_i : R_{i+1} \vec{P}_i(T_i, u_i)$ and $p_i : \vec{P}_i R_i(T'_i, T'_i)$ for some T_i, T'_i, u_i. Since $p_i \preceq t$, these edges formed before t.

Suppose $\vec{P}_k R_k(T'_k, T'_k)$ does not exist at t, so $\vec{P}_k R_k(T'_k, T'_k, v'_k)$ for some $v'_k \preceq t$. Then (Theorem 4(ii)) $abort(P_j, p)$ for some $j \in \mathbf{Z}_n$ and $p \preceq v'_k \preceq t$, contradicting the hypothesis. Therefore $t : \vec{P}_i R_i(T'_i, T'_i)$ for all i.

Next, suppose $R_{k+1} \vec{P}_k(T_k, u_k)$ does not exist at t, so $R_{k+1} \vec{P}_k(T_k, u_k, v_k)$ for some $v_k \preceq t$. Then (Lemma 3(v)) $\vec{P}_k R_k(T'_k, T'_k, v'_k)$ for some $v'_k \leq v_k \preceq t$, contradicting $t : \vec{P}_k R_k(T'_k, T'_k)$. Therefore $t : R_{i+1} \vec{P}_i(T_i, u_i)$ for all i. □

We should stress that the time of simultaneous existence t in Theorem 5 is a local time and not necessarily a real time. In contrast, all other definitions of deadlocks that we know of, save one [BT], are in terms of simultaneous existence with respect to real time.

If we consider each pair $t : R_{i+1} \vec{P}_i(T_i, u_i)$ and $t : \vec{P}_i R_i(T'_i, T'_i)$ to mean the existence at t of an edge from R_{i+1} to R_i, then the characterization is in terms of a cycle of resource-to-resource edges, contrary to the common intuition that a deadlock arises from the simultaneous existence of a cycle of process-to-process edges (Section 3.1).

There are essentially four ways to define a simultaneous cycle of edges. One of them — a process n-cycle — leads to a deadlock (Theorem 5(i)), but two others — including a resource n-cycle — do not. (See Section 3.1; the asymmetry between processes and resources is further elaborated upon by the results in Section 4.) We consider the fourth in the following corollary.

Corollary 6
Suppose $t : R_{i+1} \vec{P}_i(T_i, u_i)$ and $t : P_i \vec{R}_i(T'_i, x'_i)$ for $i \in \mathbf{Z}_n$. Then $P_0, \ldots, P_{n-1}$ are deadlocked at some $p_0, \ldots, p_{n-1}$ over $R_0, \ldots, R_{n-1}$.

Proof
$t : P_i \vec{R}_i(T_i', x_i')$ implies (Lemma 2(v)) $t : \vec{P}_i R_i(T_i', T_i')$ or $abort(P_i, p_i)$ for some $p_i \preceq t$, where (A4) $p_i \in \mathcal{T}_{P_i}$. Suppose $abort(P_i, p_i)$. Then (Lemma 1(i), A5) $T_i < p_i$.

If $(T_i <) p_i < u_i$, then (Lemma 2(i)) $\neg \vec{P}_i R_{i+1}(T_i, T_i, z)$ for all $z < p_i$, so (A8) $\vec{P}_i R_{i+1}(T_i, T_i, p_i)$. On the other hand (Lemma 2(i)), $p_i : \vec{P}_i R_{i+1}(T_i, T_i)$ — a contradiction. Therefore $u_i \leq p_i$. Then (A13) $R_{i+1} \vec{P}_i(T_i, u_i, v_i)$ for some $v_i \leq p_i \preceq t$, contradicting $t : R_{i+1} \vec{P}_i(T_i, u_i)$.

We conclude that $abort(P_i, p_i)$ is impossible, so $t : \vec{P}_i R_i(T_i', T_i')$ and (Theorem 5(i)) the processes are deadlocked. □

By this corollary, we see that if there is a cycle in Sanders and Heuberger's implementation of their waits-for graph (Section 3.1), then there is indeed a deadlock.

3.4 The Definition Revisited

We expect the reader to ask: Is the idea of a quasi-deadlock necessary? And isn't our definition of a deadlock rather complicated for such a simple concept?

To answer the first question, we should point out that many of the algorithms in the literature detect, in fact, quasi-deadlocks (Section 5.2). For the theory constructed with our definitions to be useful in the analysis and synthesis of detection algorithms, it is therefore necessary to capture this natural concept of a quasi-deadlock. Furthermore, in the context of understanding the errors in the literature, the importance of quasi-deadlocks is evident from the fact that some of the errors are in fact quasi-deadlocks; the reason for this can be surmised from the example in Figure 1. The concept of a quasi-deadlock has not arisen before only because many papers do not take into account the effect of aborting processes.

For the second question, we have previously explained why the definition should have the form $\mathcal{D}(p_1, \ldots, p_n)$. It remains to verify that our definition of a deadlock agrees with our intuition in the presence of real time. Indeed, if there is real time, then we can let $\mathcal{T}$ be the set of real times; now Theorem 5 says that $P_0, \ldots, P_{n-1}$ are deadlocked at $p_0, \ldots, p_{n-1}$ if and only if the process n-cycle exists at time $t = \max\{p_0, \ldots, p_{n-1}\}$. One could therefore view our definition as a generalization of a definition that uses real time.

But we do not seek generality for its own sake. The primary reason for adopting the complicated definition is that we want the theory to be applicable, and we therefore base it on what a deadlock detector can observe. As validation, note that our definition can be used in Chandy and Lamport's snapshot algorithm for deadlock detection [CL] (see Section 5.2). Even if real time exists, the deductions that the detector can make about what is happening in real time must still be based on its separate observations of what is true at various sites; Theorem 5(ii) illustrates such a deduction. The complicated form of the definition is simply because it is an ensemble of local facts.

With our definitions, one could call the ensuing theory (Section 4) an *operational* theory, in the sense that it is based on what is observable.

4 Theory

In this section, we formalize the concepts used in various detection algorithms and relate them through a body of results. These results underlie the existing algorithms, and are illustrated in Figure 2. Their proof do not rely on operational arguments; rather, they are simply logical consequences of the axioms that codify the operational assumptions.

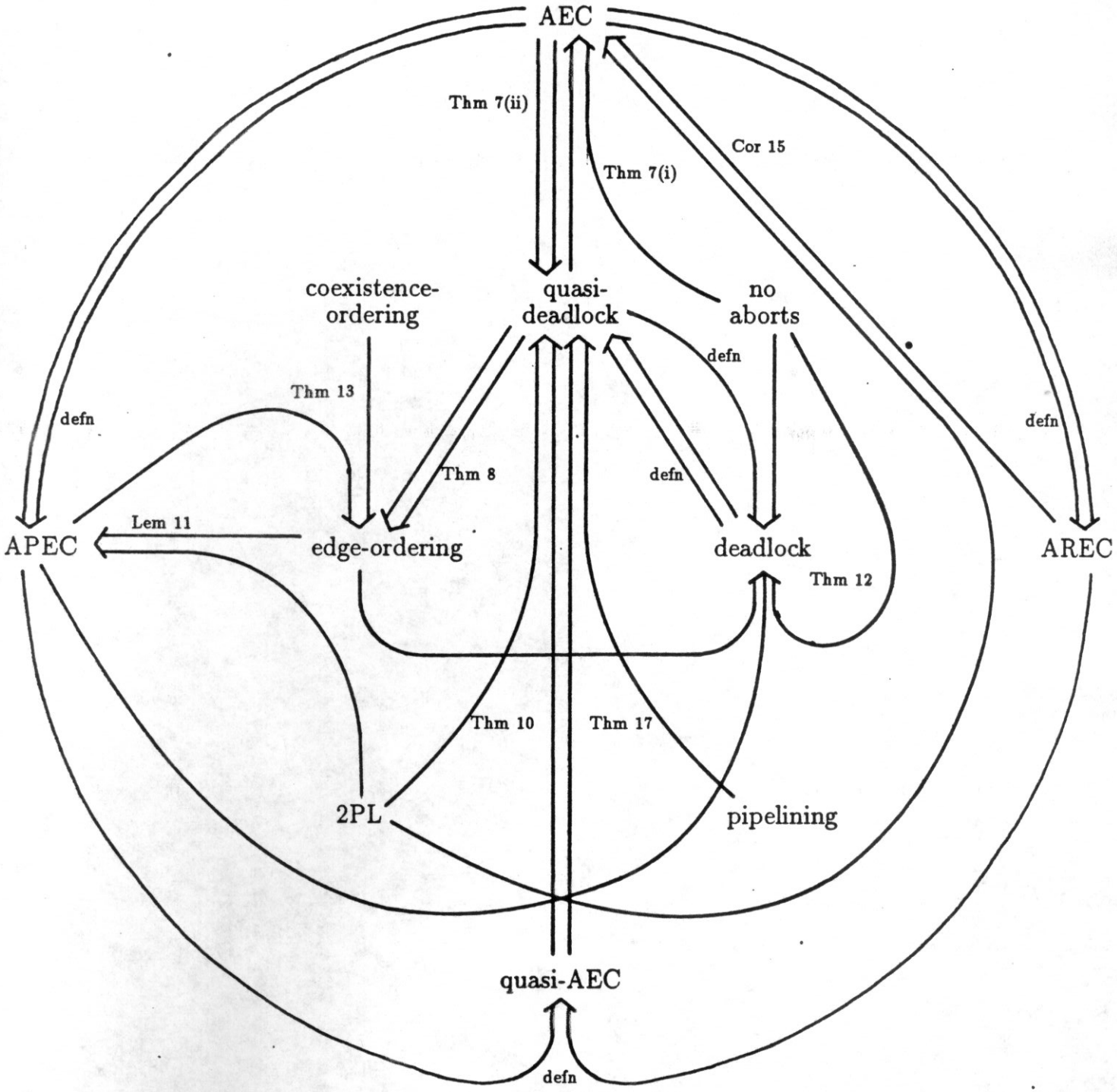


Figure 2 Overview of results.

The theory reveals the following structure in a typical deadlock detector: the algorithm starts by observing the simultaneous existence of certain pairs of adjacent edges, and deduces the existence of a deadlock from an (observed or imposed) ordering in the timing of events. We examine the former in Sections 4.1, 4.4 and 4.6 (on AEC, APEC and AREC), and the latter in Sections 4.3, 4.5 and 4.7 (on two-phase locking, coexistence ordering and pipelining). Section 4.2 is on edge-ordering, which is a central concept in the theory. Although the theory is developed for deadlock detection, it can also be applied to deadlock avoidance; this is illustrated in Section 4.8 (on resource ordering).

We make one definition before proceeding: A *double n-cycle* is a set of edges $R_{i+1}\vec{P}_i(T_i, u_i)$, $\vec{P}_iR_i(T_i', T_i')$, $P_i\vec{R}_i(T_i', x_i')$ and $\vec{R}_iP_{i-1}(T_{i-1}, x_{i-1})$, where $i \in \mathbf{Z}_n$.

Note that in a process n-cycle, each $R_{i+1}\vec{P}_i(T_i, u_i)$ implies (A11) $\vec{R}_{i+1}P_i(T_i, x_i)$ for some $x_i \prec u_i$ and each $\vec{P}_iR_i(T_i', T_i')$ implies (A14) $P_i\vec{R}_i(T_i', x_i')$ for some x_i' ($T_i' \prec x_i'$), unless M_{R_i} received an abort message from P_i before it receives the request message. Thus if the processes do not abort, then every process n-cycle implies a double n-cycle, and so does every resource n-cycle.

4.1 AEC: Adjacent Edges Coexist

We begin the theory with a partially equivalent condition for quasi-deadlocks.

Two edges $e(T, t)$ and $e'(T', t')$ are *adjacent* if and only if $e = R\vec{P}$ and $e' = \vec{P}R'$, or $e = P\vec{R}$ and $e' = \vec{R}P'$. Two adjacent edges *coexist* at time t if and only if they both exist at time t. For example, $P\vec{R}(T, x)$ and $\vec{R}P'(T', x')$ coexist at time t if $t : P\vec{R}(T, x)$ and $t : \vec{R}P'(T', x')$. We say there is *AEC* (for *adjacent edges coexist*) in a double n-cycle if and only if there exist P_i-times p_i and R_i-times r_i such that $p_i : R_{i+1}\vec{P}_i(T_i, u_i)$, $p_i : \vec{P}_iR_i(T_i', T_i')$, $r_i : P_i\vec{R}_i(T_i', x_i')$ and $r_i : \vec{R}_iP_{i-1}(T_{i-1}, x_{i-1})$ for all $i \in \mathbf{Z}_n$.

Theorem 7

(i) Suppose processes $P_0, \ldots, P_{n-1}$ are quasi-deadlocked. If none of them aborts, then they are in a double n-cycle with AEC, and whose every edge $e(T, t)$ exists at all local times $s \geq t$.

(ii) If there is AEC in a double n-cycle, then the processes are quasi-deadlocked.

Proof

(i) Suppose $P_0, \ldots, P_{n-1}$ are quasi-deadlocked at $p_0, \ldots, p_{n-1}$ over $R_0, \ldots, R_{n-1}$, and for each $i \in \mathbf{Z}_n$, $p_i : R_{i+1}\vec{P}_i(T_i, u_i)$, $p_i : \vec{P}_iR_i(T_i', T_i')$, $\vec{R}_{i+1}P_i(T_i, x_i)$, and (A6, A1, A14) $P_i\vec{R}_i(T_i', x_i')$ for some x_i'.

Consider any $i \in \mathbf{Z}_n$. By Theorem 4(ii), $p : \vec{P}_iR_i(T_i', T_i')$ for all $p \in \mathcal{T}_{P_i}$, $T_i' \leq p$; moreover, as in the proof of Theorem 5(ii), $q : R_{i+1}\vec{P}_i(T_i, u_i)$ for all $q \in \mathcal{T}_{P_i}$, $u_i \leq q$.

Next, if $\vec{R}_{i+1}P_i(T_i, x_i, y_i)$ for some y_i, then (A19) $R_{i+1}\vec{P}_i(T_i, u_i, v_i)$ for some v_i, contradicting the previous observation. Therefore $r : \vec{R}_iP_{i-1}(T_{i-1}, x_{i-1})$ for all $r \in \mathcal{T}_{R_i}$, $x_{i-1} \leq r$.

Finally, suppose $P_i\vec{R}_i(T_i', x_i', y_i')$ for some y_i'. Since P_i does not abort, we have (A17, A18, A1) $grant\_msg(R_i, P_i, T_i', y_i', z')$ for some z', so (A9) $\vec{P}_iR_i(T_i', T_i', z)$ for some $z \leq z'$, contradicting Theorem 4(ii). We conclude that $s : P_i\vec{R}_i(T_i', x_i')$ for all $s \in \mathcal{T}_{R_i}$, $x_i' \leq s$.

Since $x_{i-1}, x_i' \in \mathcal{T}_{R_i}$, we have $r_i : \vec{R}_iP_{i-1}(T_{i-1}, x_{i-1})$ and $r_i : P_i\vec{R}_i(T_i', x_i')$ for $r_i = \max\{x_{i-1}, x_i'\}$, so there is AEC.

(ii) Suppose $p_i : R_{i+1}\vec{P}_i(T_i, u_i)$, $p_i : \vec{P}_iR_i(T_i', T_i')$, $r_i : P_i\vec{R}_i(T_i', x_i')$ and $r_i : \vec{R}_iP_{i-1}(T_{i-1}, x_{i-1})$ for all i, so there is AEC in a double n-cycle.

If $P_k\vec{R}_k(T_k', x_k', y_k')$ for some $x_k', y_k' \leq x_{k-1}$, then $P_k\vec{R}_k(T_k', x_k')$ does not exist at $x_{k-1} \leq r_k$, contradicting $r_k : P_k\vec{R}_k(T_k', x_k')$. Hence for each i, $\neg P_i\vec{R}_i(T_i', x_i', y_i')$ for all $x_i', y_i' \leq x_{i-1}$, so the processes are quasi-deadlocked at $p_0, \ldots, p_{n-1}$. □

In Theorem 7(i), it is necessary that the processes do not abort. To see this, suppose there is real time and the last process to close the cycle in a quasi-deadlock aborts; the double n-cycle may never form if the abort message overtakes the request message in reaching the resource (see A14). This argument holds *per force* in the case of deadlocks.

Since Theorem 7 establishes an equivalence between quasi-deadlocks and AEC, why not use AEC as the definition of a quasi-deadlock? After all, AEC is simpler than conditions (a) and (b) in our definition, and it is also locally observable. The reason for not doing so lies in the fact that the equivalence in Theorem 7 is only partial, and this can be illustrated with an example.

Consider a scenario where there is real time and all processes in a deadlock cycle have made their requests except P_0, and all resource managers have received the requests except M_{R_0}. Now, at time p_0, P_0 sends a request for R_0, and the request is received by M_{R_0} at time r_0. Between times p_0 and r_0, the processes are already deadlocked — by our definition — but the double n-cycle has not yet formed, so we do not have AEC yet. (In Chandy and Misra's terminology, the processes are in a *dark* cycle [CM].) This example also explains why we do not state a converse to Corollary 6, in contrast to Theorem 5.

4.2 Edge-ordering

We next give a necessary condition for quasi-deadlocks. We say the pair of adjacent edges $R\vec{P}(T, u)$ and $\vec{P}R'(T', T')$ obeys *edge-ordering* if and only if $u \leq T'$, i.e. P acquires R before requesting R'. By Lemma 3(iii), if $s : R\vec{P}(T, u)$ and $s : \vec{P}R'(T', T')$ for some s, then this pair obeys edge-ordering. Of course, the converse is false, that is if $R\vec{P}(T, u)$, $\vec{P}R'(T', T')$ and $u \leq T'$, that does not imply that the two edges coexist at some time, since we may have $R\vec{P}(T, u, v)$ for some $v < T'$.

We say the pair of adjacent edges $P'\vec{R}(T', x')$ and $\vec{R}P(T, x)$ obeys *edge-ordering* if and only if $P'\vec{R}(T', x', y')$ and $\neg abort(P', p)$ for all $p \prec y'$ imply $\vec{R}P(T, x, y)$ for some $y \leq y'$, so R is retrieved from P before being granted to P', unless P' aborted. By Lemma 3(vi), if $s : P'\vec{R}(T', x')$ and $s : \vec{R}P(T, x)$ for some s, then this pair obeys edge-ordering. Again, the converse is false, i.e. edge-ordering does not imply coexistence.

We say a process, resource or double n-cycle obeys edge-ordering if and only if every pair of adjacent edges in the cycle obeys edge-ordering.

Theorem 8
Consider a double n-cycle. If the processes are quasi-deadlocked, then the cycle obeys edge-ordering.

Proof
Suppose the processes are quasi-deadlocked. Then (Lemma 3(iii)) the process n-cycle obeys edge-ordering. Now consider any i, and suppose $P_i\vec{R}_i(T_i', x_i', y_i')$ and $\neg abort(P_i, p_i)$ for all $p_i \prec y_i'$. Since the processes are quasi-deadlocked, we have $x_{i-1} < y_i'$ where $\vec{R}_i P_{i-1}(T_{i-1}, x_{i-1})$ is in the resource n-cycle.

If $x_{i-1} < x_i'$, then either $\vec{R}_i P_{i-1}(T_{i-1}, x_{i-1}, y_{i-1})$ for some $y_{i-1} < x_i' < y_i'$, or $x_i' : \vec{R}_i P_{i-1}(T_{i-1}, x_{i-1})$. For the latter, $x_i' : P_i\vec{R}_i(T_i', x_i')$ and $\neg abort(P_i, p_i)$ for all $p_i \prec y_i'$ imply (Lemma 3(vi)) $\vec{R}_i P_{i-1}(T_{i-1}, x_{i-1}, y_{i-1})$ for some $y_{i-1} \leq y_i'$.

If $x_i' \leq x_{i-1} (< y_i')$, then $x_{i-1} : P_i\vec{R}_i(T_i', x_i')$; this and $x_{i-1} : \vec{R}_i P_{i-1}(T_{i-1}, x_{i-1})$ again imply (Lemma 3(vi)) $\vec{R}_i P_{i-1}(T_{i-1}, x_{i-1}, y_{i-1})$ for some $y_{i-1} \leq y_i'$.

We conclude that the resource n-cycle obeys edge-ordering. □

From Theorem 7(i), we know that if the processes in a quasi-deadlock do not abort, then the edges are never deleted, so the resource n-cycle obeys edge-ordering vacuously. If some of the processes do abort, then edges are deleted and the resource n-cycle disintegrates, but this disintegration will, by Theorem 8, obey edge-ordering.

4.3 Two-phase Locking

Several algorithms for deadlock detection are specifically designed for databases, whose processes (i.e. transactions) must usually satisfy serializability [BHG]. The following condition guarantees that processes are serializable:

2PL $\quad R\vec{P}(T, u, v) \wedge \vec{P}R'(T', T') \rightarrow T' < v$

A process is *2PL* (*two-phase locked* [EGLT]) if and only if it satisfies the above condition, which says that it cannot make another request once it has released a resource. A process that is 2PL thus have two phases: a resource acquisition phase followed by a resource release phase.

Lemma 9
Suppose P is 2PL. Then

(i) $\vec{P}R(T, T)$ and $\vec{P}R(T', T')$ imply $T = T'$;

(ii) $R\vec{P}(T, u)$ and $R\vec{P}(T', u')$ imply $T = T'$ and $u = u'$;

(iii) $R\vec{P}(T, u, v)$ and $\vec{P}R'(T', T')$ imply $\vec{P}R'(T', T', v')$ for some $v' \leq v$.

Proof

(i) Assume $T \neq T'$. Since these are P-times, we may assume $T < T'$, so $\vec{P}R(T,T)$ formed before T'. $\vec{P}R(T',T')$ implies (A23) $\vec{P}R(T,T)$ does not exist at T', so $\vec{P}R(T,T,v)$ for some $v \leq T'$. Now $abort(P,v)$ would contradict (A5) $v \leq T'$ and $\vec{P}R(T',T')$, so (A10) $R\vec{P}(T,v)$. This, $\vec{P}R(T',T')$ and $v \leq T'$ then imply (Lemma 2(vii)) $R\vec{P}(T,v,v')$ for some $v' \leq T'$, which makes $\vec{P}R(T',T')$ impossible (2PL). We conclude that $T = T'$.

(ii) $R\vec{P}(T,u)$ and $R\vec{P}(T',u')$ imply (A10) $\vec{P}R(T,T,u)$ and $\vec{P}R(T',T',u')$. It follows from (i) and Lemma 1(iv) that $T = T'$ and $u = u'$.

(iii) $R\vec{P}(T,u,v)$ and $\vec{P}R'(T',T')$ imply (2PL) $T' < v$, so $\vec{P}R'(T',T')$ formed before time v. But $R\vec{P}(T,u,v)$ implies (A24) $\vec{P}R'(T',T')$ does not exist at v, so $\vec{P}R'(T',T',v')$ for some $v' \leq v$. □

A *quasi-double n-cycle* is a set of edges $R_{i+1}\vec{P}_i(T_i,u_i)$, $\vec{P}_iR_i(T_i',T_i')$, $\vec{R}_iP_{i-1}(T_{i-1}'',x_{i-1})$ and $P_i\vec{R}_i(T_i''',x_i')$ for $i \in \mathbf{Z}_n$, i.e. it is a union of a process n-cycle and a resource n-cycle. Thus a double n-cycle is a quasi-double n-cycle with $T_i = T_i''$ and $T_i' = T_i'''$ for all i. The next result shows that Theorem 7(ii) is true for quasi-double n-cycles, if processes are 2PL.

Theorem 10

Suppose processes are 2PL. If there is AEC in a quasi-double n-cycle, then the processes are quasi-deadlocked.

Proof

Let the edges be as in the definition of a quasi-double n-cycle. Note that T_i' and T_i'' are P_i-times. Assume $T_i' < T_i''$. Since (Lemma 3(iii)) $u_i \leq T_i' < T_i''$, $R_{i+1}\vec{P}_i(T_i,u_i)$ formed before T_i''. Now (A22) $R_{i+1}\vec{P}_i(T_i,u_i)$ does not exist at T_i'', so $R_{i+1}\vec{P}_i(T_i,u_i,v_i)$ for some $v_i \leq T_i''$, which is impossible (2PL). Assume $T_i' = T_i''$. Then (A23) $R_i = R_{i+1}$, so (AEC) $p : R_i\vec{P}_i(T_i,u_i)$ and $p : \vec{P}_iR_i(T_i',T_i')$ for some $p \in \mathcal{T}_{P_i}$. But (Lemma 3(iii)) $u_i \leq T_i' \leq p$, so (Lemma 1(vi)) $T_i' : R_i\vec{P}_i(T_i,u_i)$, contradicting (A22) $\vec{P}_iR_i(T_i',T_i')$.

We conclude that $T_i' > T_i''$. Now $\vec{R}_{i+1}P_i(T_i'',x_i)$ implies (A18, A1) $grant\_msg(R_{i+1},P_i,T_i'',x_i,u_i'')$ for some u_i''. Since $\vec{R}_{i+1}P_i(T_i'',x_i)$ implies (Lemma 1(i)) $\vec{P}_iR_{i+1}(T_i'',T_i'')$ — and $T_i'' \prec x_i \prec u_i''$ — we get (A9) $\vec{P}_iR_{i+1}(T_i'',T_i'',p)$ for some $p \leq u_i''$. Thus (A10), either $abort(P_i,p)$ or $R_{i+1}\vec{P}_i(T_i'',p)$.

Now suppose $abort(P_i,p)$, so (A5) $T_i' < p$. Since $T_i'' < T_i'$, we have (A23) $\vec{P}_iR_{i+1}(T_i'',T_i'',v_i'')$ for some $v_i'' \leq T_i'$. Since (A5) $\neg abort(P_i,v_i'')$, we get (A10) $R_{i+1}\vec{P}_i(T_i'',v_i'')$, so (A11) $grant\_msg(R_{i+1},P_i,T_i'',r,v_i'')$ for some r. Hence (Lemma 1(ii), A3) $x_i = r$ and $u_i'' = v_i'' \leq T_i' < p$, contradicting $p \leq u_i''$.

It follows that $R_{i+1}\vec{P}_i(T_i'',p)$, so (A11) $grant\_msg(R_{i+1},P_i,T_i'',r,p)$ for some r. Hence $x_i = r$ and $u_i'' = p$, so $R_{i+1}\vec{P}_i(T_i'',u_i'')$. By Lemma 9(ii), $T_i = T_i''$. Moreover, $P_i\vec{R}_i(T_i''',x_i')$ implies (Lemma 1(i)) $\vec{P}_iR_i(T_i''',T_i''')$, so (Lemma 9(i)) $T_i' = T_i'''$.

We thus get a double n-cycle, and since there is AEC, the claim follows from Theorem 7(ii). □

In Figure 2, we abbreviate this result as "quasi-AEC plus 2PL imply quasi-deadlock".

4.4 APEC: Adjacent Process Edges Coexist

Consider now a double n-cycle of edges $R_{i+1}\vec{P}_i(T_i, u_i)$, $\vec{P}_i R_i(T_i', T_i')$, $P_i\vec{R}_i(T_i', x_i')$ and $\vec{R}_i P_{i-1}(T_{i-1}, x_{i-1})$, where $i \in \mathbf{Z}_n$. We say there is *APEC* (for *adjacent process edges coexist*) in the cycle if and only if $p_i : R_{i+1}\vec{P}_i(T_i, u_i)$ and $p_i : \vec{P}_i R_i(T_i', T_i')$, where $p_i \in \mathcal{T}_{P_i}$, for all i; i.e. adjacent edges coexist in the process n-cycle, and the times of coexistence are local. APEC is, in a sense, half of AEC.

By Lemma 3(iii), if there is APEC in a double n-cycle, then the process n-cycle obeys edge-ordering. Lemma 11 says the converse is true if processes are 2PL.

Lemma 11
Suppose processes are 2PL. If a process n-cycle obeys edge-ordering, then there is APEC in the double n-cycle.

Proof
Assume we have $R\vec{P}(T, u)$ and $\vec{P}R'(T', T')$, where $u \leq T'$. Since $R\vec{P}(T, u)$ formed before T', either $R\vec{P}(T, u, v)$ for some $v \leq T'$ or $T' : R\vec{P}(T, u)$. But $R\vec{P}(T, u, v)$ would imply (2PL) $T' < v$, a contradiction. Therefore $R\vec{P}(T, u)$ and $\vec{P}R'(T', T')$ coexist at time T'. The claim follows. □

APEC and deadlocks are closely related; indeed, condition (a) in our definition of a quasi-deadlock is precisely APEC. The next theorem states some sufficient conditions which would yield a deadlock when added to APEC.

Theorem 12
Suppose a double n-cycle obeys edge-ordering. If there is APEC, and for all $i, j \in \mathbf{Z}_n$, $\neg abort(P_i, p)$ for all $p \preceq p_j$, then the processes are deadlocked.

Proof
Suppose there is edge-ordering, and $p_i : R_{i+1}\vec{P}_i(T_i, u_i)$, $p_i : \vec{P}_i R_i(T_i', T_i')$ and $\vec{R}_i P_{i-1}(T_{i-1}, x_{i-1})$ for all $i \in \mathbf{Z}_n$. Assume we have $P_k\vec{R}_k(T_k', x_k', y_k')$ for some k and $x_k', y_k' \leq x_{k-1} \prec u_{k-1}$.

Now $\neg abort(P_k, p)$ for all $p \prec y_k'$, since $y_k' \leq x_{k-1} \prec u_{k-1} \leq p_{k-1}$, so edge-ordering implies $\vec{R}_k P_{k-1}(T_{k-1}, x_{k-1}, y_{k-1})$ for some $y_{k-1} \leq y_k'$. But (Lemma 1(ii)) $x_{k-1} < y_{k-1}$, so $y_k' \leq x_{k-1} < y_{k-1} \leq y_k'$, a contradiction.

We conclude that for each $i \in \mathbf{Z}_n$, $\neg P_i\vec{R}_i(T_i', x_i', y_i')$ for all $x_i', y_i' \leq x_{i-1}$, so there is a deadlock. □

4.5 Coexistence Ordering

Deadlock detection depends fundamentally on the order in which events occur and facts are observed. This is most obvious in the role and definition of the next concept.

Suppose there is APEC in a double n-cycle, where $p_i : R_{i+1}\vec{P}_i(T_i, u_i)$ and $p_i : \vec{P}_i R_i(T'_i, T'_i)$ for $i \in \mathbf{Z}_n$ and $p'_0 : R_1\vec{P}_0(T_0, u_0)$ and $p'_0 : \vec{P}_0 R_0(T'_0, T'_0)$, where $p'_0 \in \mathcal{T}_{P_0}$. We say there is *coexistence ordering* if and only if $p_0 \prec p_1 \prec \cdots \prec p_{n-1} \prec p'_0$.

Theorem 13
If there is APEC and coexistence ordering in a double n-cycle, then the cycle obeys edge-ordering.

Proof
APEC implies (Lemma 3(iii)) the process n-cycle obeys edge-ordering, so it suffices to prove that the resource n-cycle obeys edge-ordering.

Suppose $P_i\vec{R}_i(T'_i, x'_i, y'_i)$, $\neg abort(P_i, p)$ for all $p \prec y'_i$, and $\neg\vec{R}_i P_{i-1}(T_{i-1}, x_{i-1}, y)$ for all $y \leq y'_i$. Then (A17, A18) $\vec{R}_i P_i(T'_i, y'_i)$. It follows (Lemma 3(iv)) that $\vec{R}_i P_{i-1}(T_{i-1}, x_{i-1})$ does not exist at y'_i, so $y'_i < x_{i-1}$, or $P_{i-1} = P_i$, $T_{i-1} = T'_i$ and $x_{i-1} = y'_{i-1}$.

For the latter, $p_{i-1} : R_i\vec{P}_{i-1}(T_{i-1}, u_{i-1})$ becomes $p_{i-1} : R_i\vec{P}_i(T'_i, u_{i-1})$, so (A10) $\vec{P}_i R_i(T'_i, T'_i, u_{i-1})$. But $T'_i = T_{i-1} \leq p_{i-1} \prec p_i$ and $p_i : \vec{P}_i R_i(T'_i, T'_i)$ imply (Lemma 1(vi)) $p_{i-1} : \vec{P}_i R_i(T'_i, T'_i)$, contradicting $\vec{P}_i R_i(T'_i, T'_i, u_{i-1})$ and $u_{i-1} \leq p_{i-1}$.

Therefore $y'_i < x_{i-1}$. Now $\vec{R}_i P_{i-1}(T_{i-1}, x_{i-1})$ implies (A18, A1) $grant\_msg(R_i, P_{i-1}, T_{i-1}, x_{i-1}, y)$ for some y. But $R_i\vec{P}_{i-1}(T_{i-1}, u_{i-1})$ implies (A11) $grant\_msg(R_i, P_{i-1}, T_{i-1}, x, u_{i-1})$ for some x. Hence (Lemma 1(ii) and A3) $x = x_{i-1}$ and $y = u_{i-1}$, so $x_{i-1} \prec u_{i-1}$.

Now $y'_i < x_{i-1} \prec u_{i-1} \preceq p_{i-1} \prec p_i$, so $p_i : \vec{P}_i R_i(T'_i, T'_i)$ implies (Lemma 2(viii)) $p_i : \vec{R}_i P_i(T'_i, y'_i)$. However, $\vec{R}_i P_i(T'_i, y'_i)$ and $\vec{R}_i P_{i-1}(T_{i-1}, x_{i-1})$ imply (A20) $\vec{R}_i P_i(T'_i, y'_i, z)$ for some $z \leq x_{i-1} \prec p_i$, contradicting $p_i : \vec{R}_i P_i(T'_i, y'_i)$.

We conclude that if $P_i\vec{R}_i(T'_i, x'_i, y'_i)$ and $\neg abort(P_i, p)$ for all $p \prec y'_i$, then $\vec{R}_i P_{i-1}(T_{i-1}, x_{i-1}, y_{i-1})$ for some $y_{i-1} \leq y'_i$, so the resource n-cycle obeys edge-ordering. □

In Theorem 13, can APEC itself imply edge-ordering in the double n-cycle? No; otherwise, Theorem 12 would mean that APEC implies a deadlock if the processes do not abort, thus making condition (b) in the definition of a deadlock redundant.

4.6 AREC: Adjacent Resource Edges Coexist

Consider again a double n-cycle of edges $R_{i+1}\vec{P}_i(T_i, u_i)$, $\vec{P}_i R_i(T'_i, T'_i)$, $P_i\vec{R}_i(T'_i, x'_i)$ and $\vec{R}_i P_{i-1}(T_{i-1}, x_{i-1})$. We say there is *AREC* (for *adjacent resource edges coexist*) in the cycle if and only if $r_i : \vec{R}_i P_{i-1}(T_{i-1}, u_{i-1})$ and $r_i : P_i\vec{R}_i(T'_i, x'_i)$, where $r_i \in \mathcal{T}_{R_i}$, for all $i \in \mathbf{Z}_n$; in other words, adjacent edges coexist in the resource n-cycle, and the times of coexistence are local. If APEC is one half of AEC, then AREC is the other half. We want to show that this half is equivalent to AEC if processes are 2PL.

Theorem 14

Suppose processes are 2PL. If there is AREC in a double n-cycle, then the cycle obeys edge-ordering.

Proof

Suppose $r_i : \vec{R}_i P_{i-1}(T_{i-1}, x_{i-1})$, $r_i : P_i \vec{R}_i(T'_i, x'_i)$, $R_{i+1}\vec{P}_i(T_i, u_i)$ and $\vec{P}_i R_i(T'_i, T'_i)$, where $i \in \mathbf{Z}_n$.

Assume edge-ordering is violated. Since there is AREC, the resource n-cycle obeys edge-ordering (Lemma 3(vi)), so the violation is in the process n-cycle, say $T'_k < u_k$. Then, $\vec{P}_k R_k(T'_k, T'_k)$ formed before u_k, so either $\vec{P}_k R_k(T'_k, T'_k, v'_k)$ for some $v'_k \leq u_k$, or $u_k : \vec{P}_k R_k(T'_k, T'_k)$.

Suppose $u_k : \vec{P}_k R_k(T'_k, T'_k)$. Note that $R_{k+1}\vec{P}_k(T_k, u_k)$ implies (A10) $\vec{P}_k R_{k+1}(T_k, T_k, u_k)$. If $T_k < T'_k (< u_k)$, then (Lemma 2(i)), $T'_k : \vec{P}_k R_{k+1}(T_k, T_k)$, contradicting (A23) $\vec{P}_k R_k(T'_k, T'_k)$. If $T'_k < T_k (< u_k)$, then (Lemma 1(vi)) $T_k : \vec{P}_k R_k(T'_k, T'_k)$, contradicting (A23) $\vec{P}_k R_{k+1}(T_k, T_k)$. If $T_k = T'_k$, then $\vec{P}_k R_k(T'_k, T'_k)$ and $\vec{P}_k R_{k+1}(T_k, T_k)$ imply (A23) $R_k = R_{k+1}$. It follows (A10) from $R_{k+1}\vec{P}_k(T_k, u_k)$ that $\vec{P}_k R_k(T'_k, T'_k, u_k)$, contradicting $u_k : \vec{P}_k R_k(T'_k, T'_k)$.

We conclude that $\vec{P}_k R_k(T'_k, T'_k, v'_k)$ for some $v'_k \leq u_k$. This implies (Lemma 2(iv)) that $abort(P_k, v'_k)$ or $P_k \vec{R}_k(T'_k, x'_k, y'_k)$ for some $y'_k \prec v'_k$. Assume $abort(P_k, v'_k)$, so (A5) $T_k < v'_k$. Then (A8) $\vec{P}_k R_{k+1}(T_k, T_k, z)$ for some $z \leq v'_k$. If $v'_k < u_k$, then $R_{k+1}\vec{P}_k(T_k, u_k)$ implies (Lemma 2(i)) $v'_k : \vec{P}_k R_{k+1}(T_k, T_k)$ — a contradiction. If $v'_k = u_k$, then $R_{k+1}\vec{P}_k(T_k, u_k)$ implies (A10) $\neg abort(P_k, v'_k)$ — another contradiction.

Therefore $P_k \vec{R}_k(T'_k, x'_k, y'_k)$ for some $y'_k \prec v'_k$. If $abort(P_k, p)$ for some $p \prec y'_k$, then (A5) $T'_k < p$ — so $\vec{P}_k R_k(T'_k, T'_k)$ formed before p — and (A8) $\vec{P}_k R_k(T'_k, T'_k, z)$ for some $z \leq p$; this and $\vec{P}_k R_k(T'_k, T'_k, v'_k)$ imply $v'_k = z \leq p \prec y'_k$, contradicting $y'_k \prec v'_k$. Therefore $\neg abort(P_k, p)$ for all $p \prec y'_k$.

By edge-ordering of the resource n-cycle, we have $\vec{R}_k P_{k-1}(T_{k-1}, x_{k-1}, y_{k-1})$ for some $y_{k-1} \leq y'_k$, so (A19) $R_k \vec{P}_{k-1}(T_{k-1}, u_{k-1}, v_{k-1})$ for some $v_{k-1} \prec y_{k-1}$ and (Lemma 9(iii)) $\vec{P}_{k-1} R_{k-1}(T'_{k-1}, T'_{k-1}, v'_{k-1})$ for some $v'_{k-1} \leq v_{k-1}$.

Thus $\vec{P}_k R_k(T'_k, T'_k, v'_k)$ implies $\vec{P}_{k-1} R_{k-1}(T'_{k-1}, T'_{k-1}, v'_{k-1})$ for some $v'_{k-1} \leq v_{k-1} \prec y_{k-1} \leq y'_k \prec v'_k$. Inductively, we get a contradiction from the cycle.

We conclude that the process n-cycle also obeys edge-ordering. □

Corollary 15

Suppose processes are 2PL. Then there is AREC in a double n-cycle if and only if there is AEC in the cycle.

Proof

By definition, AEC implies AREC. For the converse, AREC implies (Theorem 14) edge-ordering since the processes are 2PL. The claim now follows from Lemma 11. □

4.7 Pipelining

A major difficulty in analyzing distributed systems is that messages may not be pipelined, i.e. they are not delivered in the order they are sent. Formally, we say messages are *pipelined* if

and only if they satisfy the following condition:

Pipelining $receive(Q, Q', s, s') \wedge receive(Q, Q', t, t') \wedge (s < t) \rightarrow s' < t'$

Lemma 16

Assume messages are pipelined.

(i) $receive(P, R, p, r)$, $p : \vec{P}R(T, T)$ and $r : P\vec{R}(T', x')$ imply $T = T'$.

(ii) $receive(R, P, r, p)$, $r : \vec{R}P(T, x)$ and $p : R\vec{P}(T', u')$ imply $T = T'$.

Proof

Note from the messages that $p \in \mathcal{T}_P$ and $r \in \mathcal{T}_R$ in both cases.

(i) Assume $T < T'$. If $(T <)T' \leq p$, then $p : \vec{P}R(T, T)$ implies (Lemma 1(vi)) $T' : \vec{P}R(T, T)$, contradicting (A23) $\vec{P}R(T', T')$. If $p < T'$, then $\vec{P}R(T', T')$ implies (A6, A1, A3, A15) $receive(P, R, T', x')$, so $r < x'$ (pipelining), contradicting $r : P\vec{R}(T', x')$.

Assume $T' < T$. $P\vec{R}(T', x')$ implies (Lemma 1(i)) $\vec{P}R(T', T')$, so (A23) $\vec{P}R(T', T', v')$ for some $v' \leq T$. $\vec{P}R(T, T)$ implies (A5) $\neg abort(P, v')$, so (A10, A11) $\vec{R}P(T', u')$ for some $u' \prec v'$ and hence (A18) $P\vec{R}(T', x', u')$, where $u' \prec v' \leq T$; this contradicts $T \leq p \prec r < u'$ (from $r : P\vec{R}(T', x')$).

We conclude that $T = T'$.

(ii) $R\vec{P}(T', u')$ implies (A11) $\vec{R}P(T', x')$ and $receive(R, P, x', u')$ for some x'. Since $u' \leq p$ (from $p : R\vec{P}(T', u')$) and $receive(R, P, r, p)$, we have $x' \leq r$ (pipelining).

If $x < x'(\leq r)$, then $x' : \vec{R}P(T', x')$ and (Lemma 1(vii)) $x' : \vec{R}P(T, x)$, so $x = x'$ (Lemma 3(iv)), contradicting $x < x'$.

Therefore $x' \leq x(\leq r)$. Since $p : R\vec{P}(T', u')$ implies (Lemma 2(vi)) $p : \vec{R}P(T', x')$, and $r \prec p$ (from $receive(R, P, r, p)$), we have (Lemma 1(vii)) $x : \vec{R}P(T', x')$. Now $x : \vec{R}P(T, x)$ implies (Lemma 3(iv)) $T = T'$. □

Pipelining can replace the 2PL restriction on processes in Theorem 10, in the following sense:

Theorem 17

Assume messages are pipelined. Suppose there is AEC in a quasi-double n-cycle, and messages are received as follows: For all $i \in \mathbf{Z}_n$,
$receive(R_i, P_{i-1}, r_i, p'_{i-1})$, $r_i : P_i\vec{R}_i(T_i''', x'_i)$, $r_i : \vec{R}_iP_{i-1}(T''_{i-1}, x_{i-1})$, $p'_{i-1} : R_i\vec{P}_{i-1}(T_{i-1}, u_{i-1})$ and
$receive(P_i, R_i, p_i, r'_i)$, $p_i : R_{i+1}\vec{P}_i(T_i, u_i)$, $p_i : \vec{P}_iR_i(T'_i, T'_i)$, $r'_i : P_i\vec{R}_i(T_i''', x'_i)$.
Then the processes are quasi-deadlocked.

Proof

By Lemma 16, $receive(R_i, P_{i-1}, r_i, p'_{i-1})$ implies $T_{i-1} = T''_{i-1}$ and $receive(P_i, R_i, p_i, r'_i)$ implies $T'_i = T_i'''$. Thus, $p_i : R_{i+1}\vec{P}_i(T_i, u_i)$, $p_i : \vec{P}_iR_i(T'_i, T'_i)$, $r_i : P_i\vec{R}_i(T'_i, x'_i)$, and $r_i : \vec{R}_iP_{i-1}(T_{i-1}, x_{i-1})$ for all $i \in \mathbf{Z}_n$, i.e. AEC in a double n-cycle. By Theorem 7(ii), the processes are quasi-deadlocked. □

In Figure 2, we abbreviate this result as "pipelining plus quasi-AEC imply quasi-deadlock".

4.8 Resource Ordering

To round off, we prove one result on the avoidance of deadlocks, rather than their detection.

In an operating system, an hierarchical ordering of resources may exist or can be imposed [H, Mi]. To see how such an ordering can be exploited, suppose there is a partial order $\lhd$ on the set of resources $\mathcal{R}$. We say processes obey *resource ordering* if and only if they satisfy the following condition:

Resource Ordering $T' : R\vec{P}(T,u) \wedge \vec{P}R'(T',T') \rightarrow R \lhd R'$

This condition says that if process P is holding resource R when it requests resource R', then R and R' must follow the partial order.

Theorem 18
There are no quasi-deadlocks if processes obey resource ordering.

Proof
Suppose $P_0, \ldots, P_{n-1}$ are quasi-deadlocked at $p_0, \ldots, p_{n-1}$ over $R_0, \ldots, R_{n-1}$, and for each $i \in \mathbf{Z}_n$, $p_i : R_{i+1}\vec{P}_i(T_i, u_i)$ and $p_i : \vec{P}_i R_i(T_i', T_i')$. Then (Lemma 3(iii)) $u_i \leq T_i' \leq p_i$, so (Lemma 1(vi)) $T_i' : R_{i+1}\vec{P}_i(T_i, u_i)$ and $R_{i+1} \lhd R_i$ by resource ordering. Induction on $\mathbf{Z}_n$ gives $R_{n-1} \lhd \cdots \lhd R_1 \lhd R_0 \lhd R_{n-1}$, contradicting the partial order. □

In particular, if processes follow the partial ordering of resources in their requests ($\vec{P}R(T,T) \wedge \vec{P}R'(T',T') \wedge T < T' \rightarrow R \lhd R'$), then there will be no deadlocks. This is a well-known folk theorem in deadlock avoidance.

5 Application

We first discuss the conditions that the axioms impose on a system in terms of implementation (Section 5.1), then we use our results to survey the literature (Section 5.2).

5.1 Physically Speaking

Many papers on distributed systems assume (some implicitly) the existence of *real time*. There are two aspects to this concept. Physically, it is an absolute Newtonian time, a totally ordered set, and every event in the system is associated with a point in this time (e.g. see [NT]). Logically, it is a total ordering of all the events in the system (e.g. see [HF]). The latter is a stronger requirement: the existence of a physical real time does not induce a total ordering of events, since there may be simultaneous events. (Note that the simultaneous existence in Section 3.3 does not mean there are simultaneous events.)

If one chooses to accept existence of real time (either version), then it can be modeled by a common clock, i.e. a totally ordered $\mathcal{T}$. Note that the values on this clock would not be observable in general. An example of an observable common clock would be where the system is centralized; for example, if all processes and resource managers share the same processor, then the processor physically imposes a total ordering of all events. Another example of an observable common clock would be Lamport's implementation of a logical clock [L].

We do not assume the existence of real time. The assumptions that we do make about time (in Section 2) can be realized as follows: Each site has its own physical clock, which processes and resource managers resident on that site can read. Thus, for any $Q, Q' \in \mathcal{Q}$, $\mathcal{T}_Q$ and $\mathcal{T}_{Q'}$ are either equal or disjoint. For Q and Q' at the same site, a message from Q to Q' could be interpreted as either a transfer of processor control, or a write-and-read of a shared variable. Nonetheless, the partial ordering requires that $t \prec t'$ if $receive(Q, Q', t, t')$. We could do away with the physical clock all together, so each $\mathcal{T}_Q$ is simply a total ordering of events in Q. This models the case where Q moves from site to site.

Note that the distributed nature of the system (e.g. communication topology, process migration, etc.) is modeled entirely through the partial ordering on time. As in Lamport's clock, the existence of real time is irrelevant to the above realization of our model. (The existence of real time does not compel a model to explicitly represent it.)

Lemma 1(iii) claims that $e(T, t, t')$ implies $t < t'$. One would reasonably expect this to be so for $e = \vec{R}P$, but what about $e = P\vec{R}$, say? If $P\vec{R}(T, t, t')$ implies $t < t'$, that excludes the possibility that a request is granted as soon as it is received by M_R. We did in fact explore the theory for the case where $t = t'$ is possible, but the coincidence in edge formation and deletion times introduces numerous special cases that throw no new light on the theory. Given that the events involved require multiple instructions in any case (e.g. a lock-unlock pair in a database system may take a hundred instructions [G]), we believe the small loss in generality ($t < t'$ instead of $t \leq t'$) is justified in return for the removal of those special cases.

Similarly, if P and R share the same clock, one could argue that M_R can grant R to P, and

P acquire it at the same moment, i.e. $\vec{R}P(T,t)$, $R\vec{P}(T,t')$ and $t = t'$. However, this is ruled out by A11, which implies that $t \prec t'$. This follows from the previous observation that even for P and R at the same site, a message cannot be received at the instant it is sent.

In general, events such as sending and receiving messages, or requesting and granting resources, are not atomic in terms of machine instructions. (Indeed, Obermarck has observed that deadlock detection in centralized systems is also prone to error, if care is not taken to ensure that certain actions by the detector are atomic [O].) The axioms, however, assume that a single logical time can be assigned to these events. This means that each event should be implemented in a critical section, wherein the event will not be interrupted, so that it is *effectively* atomic. Some axioms in fact require atomicity of multiple events. For example, A8 requires that the time when a process aborts and when it cancels an outstanding request must be the same. The obvious way to ensure this would be to put in critical section the instructions for both events.

More on the physical aspects of messages: By requiring acknowledgment of a received message, lost messages can be resent, so A1 can, in principle, be satisfied. (In any realistic system, there will always be lost messages.) We do not concern ourselves with fraudulent messages, so A2 is acceptable. With the help of message identifiers, we can ignore all but one among copies — cf. A1 — of the same message, thus satisfying A3 effectively. Message identifiers can also be used to enforce pipelining: received messages are processed in the order that they are sent by the sender. Note that requests should be timestamped, and grant and release messages must be tagged with the timestamp (e.g. T in $grant\_msg(R,P,T,t,t')$) of the relevant request. This timestamp is unnecessary if processes are 2PL (Lemma 1(i) and Lemma 9).

What do logical edges correspond to physically? One possibility would be to materialize these edges exactly. For instance, P could have a data structure for physical edges, and a physical edge for $R\vec{P}(T,u)$ would be created when P receives notification at P-time u from M_R that it has been granted R (where P requested at P-time T), and deleted when P releases R at P-time v, corresponding to $R\vec{P}(T,u,v)$.

In practice, logical edges may not correspond exactly to data structures. It is common to define a *waits-for graph* with process-to-process edges; such an edge should imply some coexistent logical edges $r : P'\vec{R}(T',x')$ and $r : \vec{R}P(T,u)$, where $r \in \mathcal{T}_R$. As suggested by the logical edges, the physical process-to-process edge should be maintained by resource managers, not by the processes (as is usually done); otherwise, errors are likely to occur. An appropriate physical edge to keep with processes would be a resource-to-resource edge defined by $p : R\vec{P}(T,u)$ and $p : \vec{P}R'(T',T')$, where $p \in \mathcal{T}_P$. Such edges would be useful to the detection of deadlocks (as our very definition of a deadlock suggests) but, curiously, we know of no algorithm that uses such edges.

A process can abort because of a software or a hardware failure, or because it is asked to do so, say by the concurrency control or the load manager. If it restarts later, A5 requires it to assume a different identity (becomes a different P within $\mathcal{P}$). Similarly, for two-phase locking to make sense, a terminated transaction must assume a new identity before running again. The axioms do not cover resource failures. We assume that such failures will be transparent, in that their recovery will be in a manner consistent with the axioms.

Note that the axioms do not require that a resource manager M_R must grant R to waiting processes in any particular order, nor that it must grant an outstanding request if no other process has acquired R.

Finally, although A23 and A24 require that processes have single locus, that does not mean that a process must be idle while waiting to acquire a resource: the axiom does not exclude the possibility that a process continues computation after sending a request; it only requires that the process cannot send another request or release a resource. Specifically, the axiom allows a process to send and receive messages while it is waiting for a request to be granted. This is necessary since the process must, for instance, continue to participate in a deadlock detection algorithm.

We should emphasize here that the axioms are of no central importance — they are only a means to an end (namely, the theory). We view them simply as a rigorous formulation of our operational assumptions.

5.2 Survey

In Section 3.1, we reviewed the definition of deadlocks in the literature. We now review the algorithms that have been proposed. This survey is not meant to be comprehensive — only some segments of some papers are mentioned here. Our selection is based on whether the theory in Sections 3 and 4 has something to say about the selected material.

We only describe the algorithms in the abstract, omitting details (e.g. data structures) that are not relevant to an understanding of the algorithms. For expository reasons, we adopt an operational description of the algorithms, so the reader should refer to the relevant definitions for our precise meaning. Where an algorithm is more general in its assumptions (e.g. several allow sharing of resources), our comments apply only to the special cases where our restrictions apply. We assume the reader is familiar with the algorithms.

The terms *phantom deadlock* and *undetected deadlock*, widely used in the literature, implicitly assume there is real time — a phantom deadlock is one declared by a detector at real time t when in fact the system has no deadlock at that time, and the system has an undetected deadlock at real time t if the detector has not yet found the deadlock at that time. Accordingly, our comments on phantom and undetected deadlocks will be in the context of real time.

The first paper to present a deadlock detection algorithm for a distributed system is by **Menasce and Muntz** [MeMu]. We consider here one of the two algorithms they proposed. In this algorithm, they use process-to-process edges, where an edge $P'P$ exists if P' is "a (process) requesting resource R" and "the resource R cannot be granted to P' because it is being held by P". Following this description, we say there is an edge $P'P$ if and only if for some R and some T, T', x, x', $P'\vec{R}(T', x')$ and $x' : \vec{R}P(T, x)$, i.e M_R receives a request from P' while R is yet to be retrieved from P. For any P' and P'', there is a transitive edge $P' * P''$ if and only if there is an edge $P'P''$, or there are transitive edges $P' * P$ and $P * P''$ for some P. The algorithm declares a deadlock if it finds a transitive edge $P * P$.

Now $P * P$ implies a cycle of edges $PP_1, P_1P_2, \ldots, P_{n-1}P$. Corresponding to these edges are

resources $R_1, R_2, \ldots, R_n$, and there is AREC in the resource n-cycle implied by the edges. Since Menasce and Muntz assume processes are 2PL, it follows from Corollary 15 that there is AEC in the double n-cycle, and therefore a quasi-deadlock, by Theorem 7(ii). The example of Figure 1 shows that this quasi-deadlock detected by the algorithm can be a phantom.

Gligor and Shattuck point out that message delays can cause undetected deadlocks in Menasce and Muntz's algorithm [GS], and list two possible remedies, both of which try to make sure a cycle of edges is always detected. However, this does not suffice to eliminate undetected deadlocks from the algorithm. To see this, note that the definition of $P'P$ does not cover the case where the transaction holding resource R when the request from P' arrives at M_R is some P'', $P'' \neq P$. For example, P may acquire the resource after P'' releases it, while P' continues to wait.

The idea of process-to-process edges originates from waits-for graphs that are used in the case of centralized systems. **Obermarck**'s algorithm is based on an extension of the waits-for graph to distributed systems [O]. In his algorithm, sites construct portions of a waits-for graph, and send parts of those graphs to other sites. Several authors, including Obermarck himself, pointed out that the algorithm suffers from phantoms. However, the counterexamples are based on non-2PL processes. Indeed, as in Menasce and Muntz's algorithm, the edges in Obermarck's graph imply AREC, which is equivalent to AEC if processes are 2PL (Corollary 15). Therefore, if the algorithm is used by processes that are 2PL, then the declared deadlocks are quasi-deadlocks.

An algorithm that uses both processes and resources as nodes for its edges is **Ho and Ramamoorthy**'s two-phase algorithm [HR], which has a deadlock detector D that is a distinguished process. All sites where processes initiate must report to D the edges that they are aware of. For example, P's site must report every pair $R\vec{P}(T,u)$ and $\vec{P}R'(T',T')$ that exist at the time the site reports. It follows that any cycle detected would have APEC. By Theorems 8 and 12, it is necessary and sufficient to ensure there is edge-ordering for the algorithm to correctly detect deadlocks, if processes do not abort. The algorithm neglects this extra condition, and **Jagannathan and Vasudevan** has given an example [JV] to illustrate the gap.

Ho and Ramamoorthy also give a one-phase algorithm that requires all processes *and* resource managers to report edges they know about to the detector. The detector verifies that any quasi-double n-cycle in the reports is in fact a double n-cycle before declaring a deadlock. Since there is AEC, Theorem 7(ii) says that the declared deadlock is at least a quasi-deadlock. By Theorem 10, the verification is redundant if processes are 2PL.

The correctness of the converse — if there is a deadlock, then the reports will contain a double n-cycle with AEC — depends on the definition of a deadlock. As illustrated in Section 4.1, in the window of (real) time between when the last request in the cycle is sent and when it is received, there is (in our definition) a deadlock without a double n-cycle, so undetected deadlocks are possible within this window. Ho and Ramamoorthy requires that the one-phase algorithm be run periodically, so the deadlock will eventually be detected anyway.

Wuu and Bernstein has a variation [WB] to complement Ho and Ramamoorthy's algorithms: all resource managers are to report periodically to D all edges they are aware of individually. If there is a resource n-cycle in this collection of edges, a deadlock is declared. Wuu and Bernstein

proved that this scheme has no phantoms if processes are 2PL. There is another proof of this result using serializability theory [BHG].

What the algorithm detects is a quasi-deadlock: Observe that there is AREC in the implied double n-cycle, so there is AEC if processes are 2PL (Corollary 15), and hence a quasi-deadlock (Theorem 7(ii)). If none of the processes aborts, then this is a deadlock. However, one should be careful about asserting that if none of the processes *spontaneously* aborts — i.e. for reasons other than breaking a deadlock — then there is a deadlock (see [BHG]). Refer again to the example in Figure 1, with $P_2 = P_0$. Suppose at time 22, M_{R_1} and M_{R_2} report edges to D, where the deadlock between P_0 and P_1 is found, and P_1 is asked to abort at time 23. Now the other edges form, and M_{R_0} and M_{R_3} send their edges to D. Unless the detector at D does some preprocessing of the edges it now has, it may declare a deadlock involving P_0, P_3, and the already aborted P_1.

Wuu and Bernstein also stated the following result: If all edges of a cycle in the transaction wait-for graph have coexisted, then there must be a deadlock. Two problems with their definition of this graph (in terms of process-to-process edges) are mentioned in Section 3.1.

Kshemkalyani and Singhal propose a correction to Ho and Ramamoorthy's two-phase algorithm, as follows [KS]: In the first phase, each process P reports all existing edges $R\vec{P}(T,u)$ and $\vec{P}R'(T',T')$ to the detector D. If these reports together contain a process n-cycle, the detector starts the second phase by requesting another round of reports. For every $\vec{P}R'(T',T')$ and $R\vec{P}(T,u)$ on the process n-cycle in the first-phase, if $\vec{P}R'(T',T')$ reappears and there is $R\vec{P}(T'',u)$ in the second phase reports, then the detector declares a deadlock.

In their proof, the authors assume that time is *dense*, i.e. for every $r,s \in \mathcal{T}_Q$, $r < s$, there is $t \in \mathcal{T}_Q$ such that $r < t < s$. This assumption is not necessary, as is evident in the following analysis of their algorithm. Suppose the detector declares a deadlock, where the relevant process n-cycle in the first-phase reports is $\vec{P}_i R_i(T_i', T_i')$ and $R_{i+1}\vec{P}_i(T_i, u_i)$, p_i and p_i' are the P_i-times when P_i sends its reports in the first and second phase (respectively), and t the D-time when the detector initiates the second phase. Then $p_i \preceq t$, and thus $T_i' \preceq t$ and $u_i \preceq t$ (since $p_i : \vec{P}_i R_i(T_i', T_i')$ and $p_i : R_{i+1}\vec{P}_i(T_i, u_i)$) for all $i \in \mathbf{Z}_n$. Further, $t \preceq p_i'$ for all i.

Note that $R\vec{P}(T,u)$ and $R\vec{P}(T'',u)$ imply $T = T''$. (Suppose not, say $T < T''$. Then $T < T'' \prec u$ implies (Lemma 2(i)) $T'' : \vec{P}R(T,T)$, contradicting (A23) $\vec{P}R(T'',T'')$.) Therefore, if $R_{i+1}\vec{P}_i(T_i'', u_i)$ appears in the second phase, then $T_i = T_i''$. Since $p_i' : \vec{P}_i R_i(T_i', T_i')$ and $p_i' : R_{i+1}\vec{P}_i(T_i, u_i)$, $T_i' \preceq t \preceq p_i'$ and $u_i \preceq t \preceq p_i'$, we have (Lemma 1(vi)) $t : \vec{P}_i R_i(T_i', T_i')$ and $t : R_{i+1}\vec{P}_i(T_i, u_i)$ for all i so by Theorem 5(i), $P_0, \ldots, P_{n-1}$ are deadlocked.

Conversely, if $P_0, \ldots, P_{n-1}$ are deadlocked at $p_0, \ldots, p_{n-1}$, each P_i sends a first-phase report after p_i, and none of them aborts, then (Theorem 7(i)) the detector will detect the deadlock after a second phase.

Kshemkalyani and Singhal also give the following one-phase algorithm: As before, the detector D requests a report of edges from each process. Assume there are altogether N processes in the system, and P_i sends its report at P_i-time p_i. Here, each process has its own clock (which is a vector [Ma]), so $\mathcal{T}_{P_i} \cap \mathcal{T}_{P_j} = \phi$ for $i \neq j$, and each P must send no message while waiting for a resource, i.e. $t : \vec{P}R(T,T)$ and $send(P,Q,p)$ imply $t \neq p$. (Presumably, the report to D does not

count as a message.) Suppose there is a process n-cycle $p_i : R_{i+1}\vec{P}_i(T_i, u_i)$ and $p_i : \vec{P}_i R_i(T_i', T_i')$ where (without loss of generality) $i \in \mathbf{Z}_n$, $n \leq N$. If $T_i' \not\prec p_j$ for all $i \in \mathbf{Z}_n$, $j \in \mathbf{Z}_N$ and $i \neq j$, then D declares a deadlock.

Now assume a deadlock is declared at D-time t; if $P_i = D$, then $p_i = t$. (Kshemkalyani and Singhal implicitly assume that $P_i \neq D$ for all $i \in \mathbf{Z}_N$.) Note from the receipt of the reports that $p_i \preceq t$. Assume for some k and $v_k' \preceq t$, $\vec{P}_k R_k(T_k', T_k', v_k')$. Then $P_k \neq D$ (otherwise, $v_k' \preceq t = p_k$, contradicting $p_k : \vec{P}_k R_k(T_k', T_k')$). Assuming D gets no messages aside from the reports requested from processes, $v_k' \preceq t$ implies there is p_j, $j \in \mathbf{Z}_N$, such that $v_k' \preceq p_j \prec t$; but $T_k' < v_k'$, which now contradicts $T_k' \not\prec p_j$. We thus have, in this analysis, $t : \vec{P}_i R_i(T_i', T_i')$ for all $i \in \mathbf{Z}_n$, and therefore (Lemma 3(v)) $t : R_{i+1}\vec{P}_i(T_i, u_i)$, so $P_0, \ldots, P_{n-1}$ are deadlocked (Theorem 5(i)).

Conversely, suppose D receives all reports by D-time t and $t : R_{i+1}\vec{P}_i(T_i, u_i)$ and $t : \vec{P}_i R_i(T_i', T_i')$ for $i \in \mathbf{Z}_n$ (the authors use real time), so $P_0, \ldots, P_{n-1}$ are deadlocked. If $T_i' \prec p_j$ for some $i \in \mathbf{Z}_n$, $j \in \mathbf{Z}_N$, $i \neq j$, then $T_i' \prec p_j \preceq t$ implies (Lemma 1(vi)) $p : \vec{P}_i R_i(T_i', T_i')$ for all $T_i' \leq p \preceq p_j$, so P_i sends no messages for all such p, i.e. $\neg send(P_i, Q, p)$ for all p, $T_i' \leq p \preceq p_j$. This contradicts $T_i' \prec p_j$, because $\mathcal{T}_{P_i} \cap \mathcal{T}_{P_j} = \phi$, so the ordering $T_i' \prec p_j$ must be via messages. We conclude that $T_i' \not\prec p_j$ for all $i \in \mathbf{Z}_n$, $j \in \mathbf{Z}_N$, $i \neq j$, so the deadlock is detected.

In a departure from previous methods, which try to construct some sort of a waits-for graph, **Chandy and Misra**'s algorithm works by traversing such a graph with a *probe* [CM]: When a resource manager M_R receives a request from P while R is already granted to some process P', it initiates a probe in a message to P'; this probe is forwarded via coexistent adjacent edges. If M_R receives the probe back, it declares a deadlock.

Now each forwarding of the probe implies a pair of coexistent adjacent edges, so the return of the probe implies AEC in a quasi-double n-cycle. Moreover, the times at which the probe is received and forwarded are also the times at which the coexistence is locally observed. Since Chandy and Misra assume messages are pipelined, their algorithm is an application of Theorem 17. Specifically, $p_i = p_i'$ for all i and $r_i = r_i'$ for all $i \neq 0$, assuming M_{R_0} initiates the probe; if M_{R_0} receives the probe back at r_0', then the processes in the cycle are quasi-deadlocked.

It is possible that the deadlock is a phantom. Consider the following scenario for Figure 1: M_{R_3} and M_{R_2} both initiate probes (at times 12 and 18, say), with M_{R_2}'s probe leading the way round the cycle. When M_{R_3}'s probe is forwarded by P_1, M_{R_2}'s probe already returns; a deadlock is detected and P_1 aborted. Now M_{R_3}'s probe returns, and the deadlock is again declared, but now it has become a phantom, since P_1 has aborted. The possibility of a phantom is not considered by the authors since they do not consider aborting processes (even for deadlock resolution).

In their proofs, Chandy and Misra use logical edges that are assigned colors. While their black edge corresponds to $P\vec{R}(T, t)$, their white and grey edges are defined in terms of real time. For example, they define a grey edge from P to R to exist at time t, say, if "(P) has sent a request to (M_R) which (M_R) has not received yet" — this corresponds to $t : \vec{P}R(T, T) \wedge \forall x\ \neg t : P\vec{R}(T, x)$. This definition is ambiguous unless the location of t is specified (i.e. for which Q is $t \in \mathcal{T}_Q$?), or there is real time.

This probe algorithm was extended by **Sinha and Natarajan**, who allow forwarding of probes to be postponed and preservation of information from probes forwarded by aborted processes [SN]. Both extensions aim at minimizing the number of probes sent, while the latter also fills in the omission by Chandy and Misra on the release of resources. To reduce further the number of messages, Sinha and Natarajan assume that processes have priorities, which are used to decide the initiation and forwarding of probes.

The idea behind their algorithm is clear from Theorem 17: It is not necessary to have $p_i = p'_i$ and $r_i = r'_i$; rather, a probe received by an active process can be held in reserve, and forwarded when the process is blocked, so $p'_i \leq p_i$. Similarly, we can have $r'_i \leq r_i$. If the conditions for Theorem 17 are satisfied, then the algorithm would successfully detect a quasi-deadlock.

Choudhary et al give four examples [CKST] to illustrate gaps in this algorithm. In the first counterexample, the crucial error lies in having the probes sent from process to process, thus making Theorem 17 inapplicable. The probes should have been saved by resource managers as well as processes, as in the original Chandy and Misra algorithm. The second and third counterexamples produce phantom deadlocks that are in fact quasi-deadlocks. The fourth counterexample involves *clean* messages that serve to remove probes from aborted processes, but such messages are outside the scope of our theory.

Mitchell and Merritt's labeling algorithm [MiMe] is similar to probe algorithms in that it passes a *label* from process to process, much like the forwarding of probes but in the opposite direction in the deadlock cycle. (See also Section 3.1.) The idea behind their algorithm is as follows: When a process makes a request ($\vec{P}R(T,T)$), it sends a label to every process P' waiting for a resource R' it is holding (say, $T : R'\vec{P}(T',u')$ and $T : \vec{P}'R'(T'',T'')$); a process that receives a label and has an outstanding request passes on the label in the same manner; a process that receives a label originating from it declares a deadlock.

A returning label indicates a process n-cycle (which implies a double n-cycle), and the label passing ensures there is APEC and coexistence ordering. By Theorems 12 and 13, there is a deadlock if the processes have not aborted. Depending on how P' is determined, undetected deadlocks are possible.

Mitchell and Merritt's labels do some updating as they pass through each process, but this plays no role in ensuring that the detected cycle is a deadlock — it is just a device for them to prove their algorithm correct, and at the same time single out a victim to break the deadlock.

Finally, we should mention **Chandy and Lamport**'s shapshot algorithm [CL], which can be applied to deadlock detection. To do so, they require a *stable property* y, whose domain is the set of possible $\mathcal{S}$'s, where each $\mathcal{S}$ is a set of process and channel states. The authors do not specify what y is for deadlock detection (nor define what a deadlock is), but Theorem 7 offers one possibility: it says that AEC is a stable property of quasi-deadlocks. Now AEC is defined in terms of locally observable facts, so it can serve as y (with $\mathcal{S}$ appropriately defined). The restriction on the timing of aborting processes can also be verified from the local records of message history, and trivially stable, so this restriction and AEC can serve as y for detecting deadlocks. Alternatively, we can use our definition of a deadlock itself as property y.

6 Conclusion

We pointed out in the introduction that the errors in deadlock detection algorithms are caused by a carelessness in defining a deadlock and the absence of a theory. Having addressed both issues in the preceding sections, we now summarize our conclusions (Section 6.1) and indicate some directions for future work (Section 6.2).

6.1 Summary

Perhaps because deadlocks in a centralized system are so elementary and believed to be well-understood, some authors seem to consider a formal definition of distributed deadlocks unnecessary [CKST, HR, K, KS, SN], or else a straightforward extension sufficient; for the latter, witness the many definitions [GS, KMIT, MeMu, MiMe, O, SKYO, TB, WB] that follow the usual practice — for centralized systems — of defining the waits-for graph with process-to-process edges. One of Sinha and Natarajan's errors arises because their probes bypass resources and are sent from process to process.

There are two complications that cause errors in the definition of process-to-process edges. One complication is the release of resources [MeMu, WB]; the other is the confusion between the "true state" of a system and a local view (by a process or resource manager) of this state, a confusion that leads to ambiguities in the definition of such edges [GS, KMIT, MiMe, SKYO, TB]. These complications are evident when we try to understand a statement like "P is waiting for P'", a variant of which is usually used to define process-to-process edges. This statement also illustrates the weakness in operational arguments: their language is imprecise and implicitly assumes the existence of real time. One difficulty we faced in our survey is in (formally) interpreting the operational definitions and descriptions in the literature.

Aside from the definition of edges, another source of error is the intuition that a deadlock is a cycle of processes waiting simultaneously. Even if we take care to define edges as between processes and resources, there are four possible ways of defining a cycle of simultaneously existing edges. Only two of these four possibilities lead to deadlocks (Theorem 5(i), Corollary 6), while the other two do not (Section 3.1).

Among the errors that are not related to the definition of a deadlock, some would have been obvious in the light of a theory: Ho and Ramamoorthy's two-phase algorithm neglected edge-ordering (Theorem 12) and some of Obermarck's phantoms can be exorcized by two-phase locking (Corollary 15, Theorem 7(ii)). If one realizes that what a particular algorithm detects is a quasi-deadlock, then its vulnerability to phantoms would have been clear [CM, SN]. There is also a series of errors (see [CKST, RBC, SH]) arising from ad hoc techniques for optimizing the probe algorithm and which fall outside the scope of our theory.

We turn now to the theory's contribution towards our understanding of deadlocks and their detection.

With one exception, every algorithm we examined starts with either APEC or AREC in a double n-cycle, and augment it with some constraint on the timing of events. (The exception

is Chandy and Lamport's snapshot detector [CL], which can work directly with our definition of a deadlock.) The algorithms that start with APEC are those that have the processes (or their managers) participating in the deadlock detection [HR, KS, MiMe], while those that start with AREC have the resource managers doing the detection [MeMu, O, WB]; where both processes and resource managers participate, the algorithm starts with AEC in a quasi-double n-cycle [CM, HR, SN].

Two-phase locking is a constraint on the order in which a process may request and release resources, while the passing of probes (with pipelining) and labels is witness to a constraint on the timing of certain events (Theorem 13 and Lemma 16). The constraint imposed by two-phase locking augments AREC in a double n-cycle [MeMu, O, WB] or AEC in a quasi-double n-cycle [HR] for the detection of a quasi-deadlock (Theorem 10, Corollary 15 and Theorem 7(ii)). Similarly, the constraint witnessed by probe-passing augments AEC in a quasi-double n-cycle [CM, SN] (Theorem 17), and the constraint (coexistence-ordering) witnessed by label-passing augments APEC in a double n-cycle [MiMe] (Theorems 12 and 13).

The role played by constraints on the timing of events in detection algorithms illustrates our claim (in the Introduction) that logical time underlies their proofs of correctness. This role is most evident in the new concept of edge-ordering, which manifests itself in one of the errors [HR], and which is central to several results in the theory (Theorem 8, Lemma 11 and Theorems 12, 13, 14). Indeed, from the point of view of deadlock detection, the purpose of coexistence ordering and two-phase locking is to impose edge-ordering when there is, respectively, APEC (Theorem 13) and AREC (Theorem 14).

Edge-ordering also captures an asymmetry between processes and resources: the order of formation is important for edges adjacent to a process, but the order of deletion is important for those adjacent to a resource. We noted such an asymmtry previously in relation to Corollary 6, and in the remark that resource-to-resource edges are better suited to deadlock detection than process-to-process edges, but the difference in behavior between processes and resources, in general, is obvious from an overview of the results (Figure 2). This difference indicates that the approach — favored by some authors [BT, SH] — of not distinguishing between processes and resources will be limited in scope. For instance, it is not clear how two-phase locking and aborting processes can be modeled if we do not distinguish between processes and resources.

Besides revealing the structure of deadlock detectors and clarifying the difference between processes and resources, the theory also determined how deadlocks are affected by the timing of aborting processes (Theorems 4(i), 5(ii) and 13). The latter is an issue sidestepped in several papers (by not considering aborts), and which is important because an aborting process can confuse a deadlock detector, as in the example of Figure 1. Most algorithms detect quasi-deadlocks, which is fine if processes do not abort, since a quasi-deadlock is a deadlock in that case (Theorem 4(i)). If processes can abort, however, then the detector must be careful to avoid being confused (Theorem 4(ii)).

Finally, we note that the easy applicability of the theory to very different algorithms supports our claim that the analytic framework is unified, and justifies our choice of focusing on locally observable facts.

6.2 Further Development

The theory can be further developed in several ways. First, if we view the acquisition of a resource as locking that resource, then the theory can be refined to differentiate between exclusive and shared locks. It can also be extended to model systems which assign priorities to processes (an analog of resource ordering in Section 4.8) and abort processes as a way of resolving conflicts [RSL]. One may also wish to consider the effect of resource failures — the counterpart of process abortion — and their recovery [BHG].

One important direction of development begins with relaxing the assumption of single locus, so processes can make requests and release resources while they are waiting for a resource. This includes Chandy et al's AND model [CMH]. Note that allowing a waiting process to release resources adds to the confusion because now deadlocks can be broken spontaneously [SC]. Continuing from there to the OR model would include communication deadlocks [CJS], and lead to Bracha and Toueg's N-out-of-M model [BT].

Acknowledgment

We thank King Y. Tan for suggesting condition (b) in the definition of a deadlock, and Ted Johnson for many detailed comments that significantly improved our presentation. Section 3.4 was prompted by Vassos Hadzilacos, who described our definition of a deadlock as *formidable*.

Appendix: Proofs of Lemmas

Lemma 1

(i) $e(T,t)$ implies $\vec{P}R(T,T)$ and $T \preceq t$ ($T \prec t$ if $e \neq \vec{P}R$).
[All edges originate from a request for a resource.]

(ii) $e(T,t)$ and $e(T,t')$ imply $t = t'$.
[Time of formation of an edge is unique.]

(iii) $e(T,t,t')$ implies $e(T,t)$ and $t < t'$
[An edge can be deleted only if it formed earlier.]

(iv) $e(T,t_1,t_2)$ and $e(T,t_1',t_2')$ imply $t_1 = t_1'$ and $t_2 = t_2'$.
[Time of deletion of an edge is unique.]

(v) $e(T,t,t')$ implies $s : e(T,t)$ for all $t \preceq s \prec t'$.
[An edge exists at all times after its formation and before its deletion.]

(vi) $s : e(T,t)$ implies $s' : e(T,t)$ for all $t \preceq s' \preceq s$.
[If an edge exists at time s, it exists at all earlier times after its formation.]

Proof

(i) (a) $\vec{P}R(T,t)$ implies (A6) $T = t$.

(b) $P\vec{R}(T,t)$ implies (A15) $\vec{P}R(T,T)$ and *receive*(P,R,T,t); the latter implies $T \prec t$.

(c) $\vec{R}P(T,t)$ implies (A18) $P\vec{R}(T,x,t)$ for some x. $P\vec{R}(T,x,t)$ implies (A16) $x < t$ and $P\vec{R}(T,x)$, and thus $\vec{P}R(T,T)$ and $T \prec x < t$, by (b).

(d) $R\vec{P}(T,t)$ implies (A11) $\vec{R}P(T,r)$ and *receive*(R,P,r,t) for some r. From (c), we get $\vec{P}R(T,T)$ and $T \prec r \prec t$.

(ii) (a) $\vec{P}R(T,t)$ and $\vec{P}R(T,t')$ imply (A6) $t = T = t'$.

(b) $P\vec{R}(T,t)$ and $P\vec{R}(T,t')$ imply (A15) *receive*(P,R,T,t) and *receive*(P,R,T,t'); thus (A3) $t = t'$.

(c) $\vec{R}P(T,t)$ and $\vec{R}P(T,t')$ imply (A18) $P\vec{R}(T,x,t)$ and $P\vec{R}(T,x',t')$ for some x and x'. Therefore (A16, A15) *receive*(P,R,T,x) and *receive*(P,R,T,x'), so (A3) $x = x'$. Suppose $t \neq t'$, say $t < t'$. Then $P\vec{R}(T,x,t)$ contradicts (A16) $P\vec{R}(T,x,t')$.

(d) $R\vec{P}(T,t)$ and $R\vec{P}(T,t')$ imply (A11) $\vec{R}P(T,r)$ and *receive*(R,P,r,t) for some r, and $\vec{R}P(T,r')$ and *receive*(R,P,r',t') for some r'. From (c), $r = r'$, so (A3) $t = t'$.

(iii) $e(T,t,t')$ implies (A7, A12, A16, A19) $e(T,t)$. For $e = P\vec{R}$ and $R\vec{P}$, we have (A12, A16) $t < t'$. For $e = \vec{P}R$, if *abort*(P,t'), then (A5) $T < t'$; if $\neg$*abort*(P,t'), then (A10) $R\vec{P}(T,t')$, so

$T < t'$ by (i); either way $t = T < t'$. For $e = \vec{R}P$, we have (A19) $\vec{R}P(T,t)$, $R\vec{P}(T,u,v)$ and $receive(P,R,v,t')$ for some u. Now $R\vec{P}(T,u,v)$ implies (A12) $R\vec{P}(T,u)$, so (A11) $\vec{R}P(T,z)$ and $receive(R,P,z,u)$ for some z. By (ii), $t = z$. Also, $R\vec{P}(T,u,v)$ implies (A12) $u < v$, so $t = z \prec u < v \prec t'$.

(iv) By (ii) and (iii), $e(T,t_1,t_2)$ and $e(T,t_1',t_2')$ imply $t_1 = t_1' = t$, say. For $e = \vec{P}R$, $P\vec{R}$ and $R\vec{P}$, we have (A7, A12, A16) $t_2 \leq t_2'$ and $t_2' \leq t_2$, so $t_2 = t_2'$. For $e = \vec{R}P$, we have (A19) $R\vec{P}(T,x,y)$ and $receive(P,R,y,t_2)$ for some x, y, and $R\vec{P}(T,x',y')$ and $receive(P,R,y',t_2')$ for some x', y'; now we have $x = x'$ and $y = y'$, so (A3) $t_2 = t_2'$.

(v) Let $t \preceq s \prec t'$. By (iii), $e(T,t,t')$ implies $e(T,t)$. If $e(T,t)$ does not exist at s, there is $s' \preceq s$ such that $e(T,t,s')$. By (iv), $s' = t'$, so $t' \preceq s$; this contradicts $s \prec t'$. Therefore $s : e(T,t)$.

(vi) Let $t \preceq s' \preceq s$. If $e(T,t)$ does not exist at s', there is $t' \preceq s'$ such that $e(T,t,t')$. But $t' \preceq s' \preceq s$, so $e(T,t)$ does not exist at s, a contradiction. Therefore $s' : e(T,t)$. □

Lemma 2

(i) $R\vec{P}(T,u)$ implies $t : \vec{P}R(T,T)$ for all $T \preceq t \prec u$.
[After requesting a resource and before acquiring it, the process must wait.]

(ii) $\vec{R}P(T,x)$ implies for all $T \preceq t \preceq x$, either $t : \vec{P}R(T,T)$ or $abort(P,p)$ for some $p \preceq t$.
[After a request is issued and before it is granted, the process is waiting for the resource.]

(iii) $P\vec{R}(T,x)$ implies for all $T \preceq t \preceq x$, either $t : \vec{P}R(T,T)$ or $abort(P,p)$ for some $p \preceq t$.
[After a request is issued and before it is received, the process is waiting for the resource.]

(iv) $\vec{P}R(T,T,v)$ implies $P\vec{R}(T,x,y)$ for some x and y, $T \prec x < y \prec v$, or $abort(P,v)$.
[P stops waiting for R when P aborts or P is granted R.]

(v) $s : P\vec{R}(T,x)$ implies $s : \vec{P}R(T,T)$ or $abort(P,p)$ for some $p \preceq s$.
[If M_R has not yet granted P's request, then P — if unaborted — must be waiting for R.]

(vi) $s : R\vec{P}(T,u)$ implies $s : \vec{R}P(T,x)$ for some $x \prec u$.
[If a process is holding a resource R, then M_R has not yet retrieved R.]

(vii) Suppose $R\vec{P}(T,u)$ and $\vec{P}R(T',T')$. If $u \leq T'$, then $R\vec{P}(T,u,v)$ for some $v \leq T'$.
[If P is waiting for R, which it has acquired before, then P must have released R earlier.]

(viii) Suppose $\vec{R}P(T,x)$ and $t : \vec{P}R(T,T)$. If $x \preceq t$, then $s : \vec{R}P(T,x)$ for all $x \preceq s \preceq t$.
[M_R could not have retrieved a resource it has granted to P if P is still waiting for it.]

Proof

(i) $R\vec{P}(T,u)$ implies (A10) $\vec{P}R(T,T,u)$, so (Lemma 1(v)) $t : \vec{P}R(T,T)$ for all $T \preceq t \prec u$.

(ii) Let $T \preceq t \preceq x$. $\vec{R}P(T,x)$ implies (Lemma 1(i)) $\vec{P}R(T,T)$. Suppose $\neg t : \vec{P}R(T,T)$ and $\neg abort(P,p)$ for all $p \preceq t$. First, $\vec{P}R(T,T,y)$ for some $y \preceq t$. Then (A10) $R\vec{P}(T,y)$, so (A11) $\vec{R}P(T,r)$ and $receive(R,P,r,y)$ for some r. By Lemma 1(ii), $r = x$, so $x \prec y$, which contradicts $y \preceq t \preceq x$. We conclude that either $t : \vec{P}R(T,T)$ or $abort(P,p)$ for some $p \preceq t$.

(iii) Let $T \preceq t \preceq x$. Suppose $\neg t : \vec{P}R(T,T)$ and $\neg abort(P,p)$ for all $p \preceq t$. As in (ii), we have $R\vec{P}(T,y)$ for some $y \preceq t$, and $\vec{R}P(T,r)$ and $receive(R,P,r,y)$ for some r. Hence (A18), $P\vec{R}(T,x,r)$, so $x < r \prec y \preceq t$, contradicting $t \preceq x$. The contradiction proves the claim.

(iv) Suppose $\neg abort(P,v)$. Then (A10) $R\vec{P}(T,v)$, so (A11) $\vec{R}P(T,y)$ and $receive(R,P,y,v)$ for some y. Therefore (A18) $P\vec{R}(T,x,y)$ for some x, $T \prec x < y \prec v$ (Lemma 1(i, iii)).

(v) $s : P\vec{R}(T,x)$ implies (Lemma 1(i)) $\vec{P}R(T,T)$ and $T \prec x \preceq s$. Suppose $\neg s : \vec{P}R(T,T)$ and $\neg abort(P,p)$ for all $p \preceq s$. As in (iii), we have $P\vec{R}(T,x,r)$ for some $r \prec s$, contradicting the existence of $P\vec{R}(T,x)$ at time s.

(vi) $s : R\vec{P}(T,u)$ implies (Lemma 1(i)) $T \prec u \preceq s$ and (A11) $\vec{R}P(T,x)$ and $receive(R,P,x,u)$ for some $x \prec u$. Since $x \prec u \preceq s$, either $s : \vec{R}P(T,x)$ or $\vec{R}P(T,x,y)$ for some $y \preceq s$. The latter implies (A19) $R\vec{P}(T,u,v)$ and $receive(P,R,v,y)$ for some $v \prec y \preceq s$, contradicting $s : R\vec{P}(T,u)$. We conclude that $s : \vec{R}P(T,x)$.

(vii) Since $R\vec{P}(T,u)$ and $u \leq T'$, either $T' : R\vec{P}(T,u)$ or there is $v \preceq T'$ such that $R\vec{P}(T,u,v)$. Now, $\vec{P}R(T',T')$ implies (A6) $T' \in \mathcal{T}_P$, so (A22) we cannot have $T' : R\vec{P}(T,u)$. We conclude that $R\vec{P}(T,u,v)$ for some $v \preceq T'$. Since $v, T' \in \mathcal{T}_P$ (A12, A6), we get $v \leq T'$.

(viii) Suppose $x \preceq s \preceq t$. Then either $s : \vec{R}P(T,x)$ or $\vec{R}P(T,x,y)$ for some $y \preceq s$. For the latter, we have (A19) $R\vec{P}(T,u,v)$ for some $u < v \prec y$, and thus (Lemma 1(iii)) $R\vec{P}(T,u)$ and (A10) $\vec{P}R(T,T,u)$, which contradicts $t : \vec{P}R(T,T)$, since $u < v \prec y \preceq s \preceq t$. We conclude that $s : \vec{R}P(T,x)$. □

Lemma 3

(i) $s : R\vec{P}(T,u)$ and $s : R\vec{P}(T',u')$ imply $T = T'$ and $u = u'$.
[A process cannot have two requests for the same resource satisfied simultaneously.]

(ii) $s : \vec{P}R(T,T)$ and $s : \vec{P}R'(T',T')$ imply $R = R'$ and $T = T'$.
[A process cannot simultaneously wait for two different resources.]

(iii) $s : R\vec{P}(T,u)$ and $s : \vec{P}R'(T',T')$ imply $u \leq T'$.
[If P is holding R and waiting for R', then R was acquired before the latter request.]

(iv) $s : \vec{R}P(T,x)$ and $s : \vec{R}P'(T',x')$ imply $P = P'$, $T = T'$ and $x = x'$.
[R cannot be simultaneously granted to two different processes.]

(v) Suppose $s : R\vec{P}(T,u)$ and $s : \vec{P}R'(T',T')$. If $R\vec{P}(T,u,v)$, then $\vec{P}R'(T',T',v')$ for some $v' \leq v$.
[If P is holding R and waiting for R', then it must acquire R' before releasing R.]

(vi) Suppose $s : P'\vec{R}(T',x')$ and $s : \vec{R}P(T,x)$. If $P'\vec{R}(T',x',y')$, then either $abort\_msg(P',R,p',y')$ for some p' or $\vec{R}P(T,x,y)$ for some $y \leq y'$.
[If P' is waiting for R, which has been granted to P, then R must be retrieved from P before being granted to P'.]

Proof

(i) Suppose $T \neq T'$. Since $T,T' \in \mathcal{T}_P$, we may assume $T < T'$. We have (Lemma 1(i)) $\vec{P}R(T,T)$ and $\vec{P}R(T',T')$, so (A23) $\vec{P}R(T,T)$ does not exist at T'. But $T < T'$, so $\vec{P}R(T,T,v)$ for some $v \leq T'$. It follows (A10, Lemma 1(iv)) that $u = v \leq T'$, so either $T' : R\vec{P}(T,u)$ or $R\vec{P}(T,u,y)$ for some $y \leq T' \prec u' \preceq s$. The former is impossible (A22); the latter contradicts $s : R\vec{P}(T,u)$. We conclude that $T = T'$, and thus $u = u'$ (Lemma 1(ii)).

(ii) As in (i), suppose $T < T'$. But $s : \vec{P}R'(T',T')$ implies $T' \preceq s$, so $s : \vec{P}R(T,T)$ implies (Lemma 1(vi)) $T' : \vec{P}R(T,T)$, contradicting (A23) $\vec{P}R'(T',T')$. We conclude that $T = T'$. Now $T : \vec{P}R(T,T)$ and $\vec{P}R'(T,T)$ imply (A23) $R = R'$.

(iii) Note first that $u,T,T' \in \mathcal{T}_P$. Suppose $T' < u$. If $T < T'$, then $T < T' < u$ implies (Lemma 2(i)) $T' : \vec{P}R(T,T)$, contradicting (A23) $\vec{P}R'(T',T')$ and $T \neq T'$. If $T' < T$, then $T' < T < u \preceq s$ and $s : \vec{P}R'(T',T')$ imply (Lemma 1(vi)) $T : \vec{P}R'(T',T')$, similarly contradicting $\vec{P}R(T,T)$. If $T = T'$, then $T : \vec{P}R(T,T)$ and $\vec{P}R'(T,T)$ imply (A23) $R = R'$; moreover, $s : R\vec{P}(T,u)$ implies (A10) $\vec{P}R(T,T,u)$ and $u \preceq s$, contradicting $s : \vec{P}R(T,T)$. We conclude that $u \leq T'$.

(iv) If $x = x'$, then (A21) $P = P'$ and $T = T'$, so suppose $x \neq x'$. Since $x,x' \in \mathcal{T}_R$, we may assume $x < x'$. Then (A20) $\vec{R}P(T,x,y)$ for some $y \leq x' \preceq s$, contradicting $s : \vec{R}P(T,x)$.

(v) From (iii) and $s : \vec{P}R'(T',T')$, we get $u \leq T' \preceq s$, so (Lemma 1(vi)) $T' : R\vec{P}(T,u)$ and thus $T' < v$. Therefore, either $\vec{P}R'(T',T',v')$ for some $v' \leq v$, or $v : \vec{P}R'(T',T')$, but the latter contradicts (A24) $R\vec{P}(T,u,v)$.

(vi) Suppose $\neg abort\_msg(P',R,p',y')$ for all p'. Then (A17, A18) $P'\vec{R}(T',x',y')$ implies $\vec{R}P'(T',y')$. Note that $x,y' \in \mathcal{T}_R$ and $x \preceq s$ (from $s : \vec{R}P(T,x)$). If $y' \leq x (\preceq s)$, then $\neg s : P'\vec{R}(T',x')$, contradicting the hypothesis. Therefore $x < y'$. Now $y' : \vec{R}P(T,x)$ would imply (from $y' : \vec{R}P'(T',y')$ and (iv)) $x = y'$, a contradiction. Hence $\neg y' : \vec{R}P(T,x)$, i.e. $\vec{R}P(T,x,y)$ for some $y \leq y'$. □

References

[B] D.Z. Badal, *The distributed deadlock detection algorithm*, ACM Trans. Computer Systems 4, 4(Nov. 1986), 320–337.

[BHG] P.A. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, Massachusetts (1987).

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

[CJS] I. Cidon, J.M. Jaffe and M. Sidi, *Local distributed deadlock detection by cycle detection and clustering*, IEEE Trans. Software Engineering SE-13, 1(Jan. 1987), 3–14.

[CKST] A.N. Choudhary, W.H. Kohler, J.A. Stankovic and D. Towsley, *A modified priority based probe algorithm for distributed deadlock detection and resolution*, IEEE Trans. Software Engineering SE-15, 1(Jan. 1989), 10–17.

[CL] K.M. Chandy and L. Lamport, *Distributed snapshots: determining global states of distributed systems*, ACM Trans. Computer Systems 3, 1(Feb. 1985), 63–75.

[CM] K.M. Chandy and J. Misra, *A distributed algorithm for detecting resource deadlocks in distributed systems*, Proc. ACM Symp. Principles of Distributed Computing, Ottawa, Ontario (Aug. 1982), 157–164.

[CMH] K.M. Chandy, J. Misra and L.M. Haas, *Distributed deadlock detection*, ACM Trans. Computer Systems 1, 2(May 1983), 144–156.

[E] A.K. Elmagarmid, *A survey of distributed deadlock detection algorithms*, ACM Sigmod Record 15, 3(Sept. 1986), 37–45.

[EGLT] K.P. Eswaran, J.N. Gray, R.A. Lorie and I.L. Traiger, *The notions of consistency and predicate locks in a database system*, Commun. ACM 19, 11(Nov. 1976), 624–633.

[G] J. Gray, *Notes on data base operating systems*, in Operating Systems — An Advanced Course, R. Bayer et al (eds.), Springer-Verlag (1978), 393–481.

[GS] V. Gligor and S. Shattuck, *On deadlock detection in distributed databases*, IEEE Trans. Software Engineering SE-6, 5(Sept. 1980), 435–440.

[H] V.W. Havender, *Avoiding deadlock in multitasking*, IBM Systems Journal 2(1968), 74–84.

[HF] J.Y. Halpern and R. Fagin, *Modelling knowledge and action in distributed systems*, Distributed Computing 3(1989), 159–177.

[HR] G.S. Ho and C.V. Ramamoorthy, *Protocols for deadlock detection in distributed database systems*, IEEE Trans. Software Engineering SE-8, 6(Nov. 1982), 554–557.

[JV] J.R. Jagannathan and R. Vasudevan, *Comments on 'Protocols for deadlock detection in distributed database systems'*, IEEE Trans. Software Engineering SE-9, 3(May 1983), 371.

[K] E. Knapp, *Deadlock detection in distributed databases*, ACM Computing Surveys 19, 4(Dec. 1987), 303–328.

[KKNR] H.F. Korth, R. Krishnamurthy, A. Nigam and J.T. Robinson, Proc. ACM Symp. Principles of Database Systems, Atlanta, Georgia (Mar. 1983), 192–202.

[KMIT] S. Kawazu, S. Minami, K. Itoh and K. Teranaka, *Two-phase deadlock detection algorithm in distributed databases*, Proc. Int. Conf. Very Large Data Bases, Rio de Janeiro, Brazil (Oct 1979), 360–367.

[KS] A.D. Kshemkalyani and M. Singhal, *Correct two-phase and one-phase deadlock detection algorithms for distributed systems*, Proc. IEEE Symp. Parallel and Distributed Processing, Dallas, Texas (Dec. 1990), 126–129.

[L] L. Lamport, *Time, clocks, and the ordering of events in a distributed system*, Commun. ACM 21, 7(July 1978), 558–565.

[Ma] F. Mattern, *Virtual time and global states of distributed systems*, in Parallel and Distributed Algorithms, M. Cosnard et al (eds.), North-Holland (1989), 215–226.

[MeMu] D.A. Menasce and R.R. Muntz, *Locking and deadlock detection in distributed data bases*, IEEE Trans. Software Engineering SE-5, 3(May 1979), 195–202.

[Mi] T. Minoura, *Deadlock avoidance revisited*, J. ACM 29, 4(Oct. 1982), 1023–1048.

[MiMe] D.P. Mitchell and M.J. Merritt, *A distributed algorithm for deadlock detection and resolution*, Proc. ACM Symp. Principles of Distributed Computing, Vancouver, British Columbia (Aug. 1984), 282–284.

[NT] G. Neiger and S. Toueg, *Substituting for real time and common knowledge in asynchronous distributed systems*, Proc. ACM Symp. Principles of Distributed Computing, Vancouver, British Columbia (Aug. 1984), 281–293.

[O] R. Obermarck, *Distributed deadlock detection algorithm*, ACM Trans. Database Systems 7, 2(June 1982), 187–208.

[P] V. Pratt, *Modeling concurrency with partial orders*, Int. J. Parallel Programming 15, 1(1986), 33–71.

[RBC] M. Roesler, W.A. Burkhard and K.B. Cooper, *Efficient deadlock resolution for lock-based concurrency control schemes*, Proc. Int. Conf. Distributed Computing Systems, San Jose, California (June 88), 224–233.

[RSL] D.J. Rosenkrantz, R.E. Stearns and P.M. Lewis II, *System level concurrency control for distributed database systems*, ACM Trans. Database Systems 3, 2(June 1978), 178–198.

[SC] A. Shoshani and E.G. Coffman, *Detection and prevention of deadlocks*, Proc. Princeton Conf. Information Sciences and Systems, Princeton, New Jersey (Mar. 70), 355–360.

[SH] B.A. Sanders and P.A. Heuberger, *Distributed deadlock detection and resolution with probes*, Proc. Int. Conf. Distributed Algorithms (Sept. 1989), LNCS 392, 207–218.

[SKYO] K. Sugihara, T. Kikuno, N. Yoshida and M. Ogata,]*A distributed algorithm for deadlock detection and resolution*, Proc. IEEE Symp. Reliability in Distributed Software and Database Systems, Silver Spring, Maryland (Oct. 1984), 169–176.

[SN] M.K. Sinha and N. Natarajan, *A priority based distributed deadlock detection algorithm*, IEEE Trans. Software Engineering SE-11, 1(Jan. 1985), 67–80.

[TB] W.C. Tsai and G.G. Belford, *Detecting deadlock in a distributed system*, Proc. IEEE Infocom, Las Vegas, Nevada (Mar. 1982), 89–95.

[WB] G.T. Wuu and A.J. Bernstein, *False deadlock detection in distributed systems*, IEEE Trans. Software Engineering SE-11, 8(Aug. 1985), 820–821.