Deadlock Detection in Distributed Databases

EDGAR KNAPP

Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712

The problem of deadlock detection in distributed systems has undergone extensive study. An important application relates to distributed database systems. A uniform model in which published algorithms can be cast is given, and the fundamental principles on which distributed deadlock detection schemes are based are presented. These principles represent mechanisms for developing distributed algorithms in general and deadlock detection schemes in particular. In addition, a hierarchy of deadlock models is presented; each model is characterized by the restrictions that are imposed upon the form resource requests can assume. The hierarchy includes the well-known models of resource and communication deadlock. Algorithms are classified according to both the underlying principles and the generality of resource requests they permit. A number of algorithms are discussed in detail, and their complexity in terms of the number of messages employed is compared. The point is made that correctness proofs for such algorithms using operational arguments are cumbersome and error prone and, therefore, that only completely formal proofs are sufficient for demonstrating correctness.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; *distributed databases*; *network operating systems*; D.4.1 [**Operating Systems**]: Process Management—*concurrency*; *deadlocks*; *synchronization*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*; H.2.4 [**Database Management**]: Systems—*distributed systems*; *transaction processing*

General Terms: Algorithms

Additional Key Words and Phrases: Deadlock detection, deadlock models, distributed deadlocks

INTRODUCTION

Deadlock detection is an important problem in database systems (DBSs), and much attention has been devoted to it in the research community. Generally speaking, a deadlock situation is the possible result of competition for resources, such as multiple database transactions requesting exclusive access to data items.

The deadlock problem has several interesting components. Among these are deadlock prevention, deadlock avoidance, and—in connection with deadlock detection—the selection of a so-called victim whose rollback or abortion breaks the deadlock, and finally, deadlock resolution itself. This paper is concerned only with the aspect of deadlock detection. Recent developments in the area of distributed deadlock detection algorithms are surveyed, with a special emphasis on their relation to distributed DBSs. The paper introduces a uniform framework for the discussion of these algorithms. The abstraction achieved this way allows us to talk about the algorithms in terms of the underlying theoretical concepts, instead of just giving a phenomeno-

CONTENTS

logical description of the workings of the algorithms (cf. Elmagarmid [1986]).

The paper is organized as follows. Section 1 focuses on the relationship between the deadlock problem and DBSs. For the benefit of those readers not familiar with the necessary database terminology, a brief outline of the relevant concepts is given. For a more thorough treatment of this material, the reader is referred to recent texts on concurrency control, for example, Bernstein et al. [1987] and Papadimitriou [1987]. Next, a database model is presented, and a specification of the deadlock detection problem in terms of this model is developed. Section 2 gives a systematic classification of the models of deadlock as they appear in database applications. A hierarchy of models that gives rise to one way of classifying most of the distributed deadlock detection procedures found in the literature is introduced. Another classification, focusing on the theoretical principles underlying the work in distributed deadlock detection schemes, is given in Section 3. A survey of a number of algorithms can be found in Section 4, with examples from each of the classes introduced in the two previous sections. In the final section the relative merits of the algorithms presented are discussed. The references contain an exhaustive list on the work done in the field of distributed deadlock detection after 1980. Earlier papers included constitute "classical articles" related to the subject.

1. THE DEADLOCK PROBLEM

1.1 A Brief Introduction to Concurrency Control

The deadlock problem in DBSs is part of the area of *concurrency control*.[1] Concurrency control deals with the problem of coordinating the actions of processes that operate in parallel, access shared data, and therefore potentially interfere with each other. The object of study is an abstraction (model) of many different types of information systems. The main component of this model is the *transaction*. Informally, a transaction is an execution of a program that accesses a shared database. In our model transactions are characterized by a sequence of operations, for example, $R(x)$ denoting the operation of reading some data item x from the database and $W(x)$ standing for the operation of assigning a new value to data item x in the database. When two or more transactions execute concurrently, their database operations execute in an interleaved fashion. That is, operations from one transaction may execute in between two operations of another transaction. This interleaving can cause

[1] Part of Section 1.1 follows the introductory chapter of Bernstein et al. [1987].

transactions to behave incorrectly or interfere, thereby leading to an inconsistent database state.

The part of the DBS that controls the relative order in which database operations requested by transactions execute is called the *scheduler*. The scheduler determines the interleaving of database operations such that the consistency of the database is preserved. A particular such interleaving of database operations is called a *schedule*.

As an example, consider the schedule given below involving two transactions t_1 and t_2 and two entities x and y (the notation follows Bernstein et al. [1987] and Papadimitriou [1987]):

t_1:	$R(x)$		$W(y)$	
t_2:		$R(y)$		$W(x)$

This schedule formalizes the following sequence of database operations: transaction t_1 first reads database item x, then t_2 reads y, followed by t_1 writing y, and finally t_2 writing x.

1.2 Deadlock in Centralized Systems

There are many different known strategies for schedulers for solving the problem of concurrent accesses without compromising database consistency. A detailed discussion of these strategies is beyond the scope of this paper. The interested reader is referred to Bernstein et al. [1987] and Papadimitriou [1987].

The most popular of these strategies is so-called *locking*. Locking is the strategy of reserving access rights (locks) that prevent other transactions from obtaining certain other (conflicting) locks.

As an example, consider a protocol called basic *two-phase locking* (2PL), which is widely in used in commercial systems. In this protocol[2] a transaction that has released a lock may not subsequently obtain any more locks. If this strategy is applied to the example schedule given above, the following scenario is bound to happen:

t_1 locks x,
t_2 locks y,
t_1 waits for t_2 to release the lock on y,
t_2 waits for t_1 to release the lock on x.

Hence both transactions are blocked, waiting for each other: a deadlock situation.

Informally, deadlock in a DBS can be defined as "a situation in which each transaction in a set of transactions is blocked waiting for another transaction in the set, and therefore none will become unblocked unless there is external intervention" (cf. Bernstein et al. [1987]).

Even though some concurrency control protocols are provably deadlock free (e.g., conservative 2PL,[3] tree locking), most known protocols are vulnerable to deadlock. We next look at a number of other ways in which deadlock can arise in a DBS.

Let us consider the case of multiversion schedulers, where each write operation on some data item produces a new version of this item, and each read operation of an item is mapped to some version of this item that was written previously. In the protocol for a multiversion–view-serializability (MV–VSR) scheduler, the last step of a transaction is treated in a special way to ensure that at most one uncommitted version exists for any data item in the database. If such a scheduler is given the schedule of the previous example, the following happens:

t_1 reads x,
t_2 reads y,
t_1 waits for t_2 to finish (commit),
t_2 waits for t_1 to finish (commit),

and, again, the result is deadlock.

Lock conversion is a concurrency control technique that allows upgrading of a lock to a stronger lock type, such as converting a read lock on a data item into a write lock on the same item. Schedulers that allow for lock conversion are prone to deadlock situations for yet another reason. To see this, consider the following example:

t_1:	$R(x)$		$W(x)$	
t_2:		$R(x)$		$W(x)$

After both reads have been performed, with t_1 and t_2 holding read locks on x, neither transaction can convert its read lock into a write lock; hence they are blocked forever.

[2] Also called dynamic 2PL in Papadimitriou [1987].

[3] Also called static 2PL in Papadimitriou [1987].

Multigranularity locking is a method whereby transactions can lock different granularities of data items, such as a record, a disk page, or an entire file. In multigranularity locking protocols, deadlock can occur for more than one reason. First, we have the problems due to the already mentioned two-phase rule and lock conversion, especially in conjunction with lock escalation, a technique in which a transaction that obtains too many locks on data items of small granularity can increase the granularity of its subsequent lock requests. Another problem arises when granules are structured hierarchically in a rooted directed acyclic graph. In this case a locking protocol may require a transaction that wants to lock some set of granules to lock a majority of parents of these granules first. If two transactions happen to try locking the same set of granules, they may get to a point at which they both hold locks on exactly half of the parents of the set so that neither of them will succeed. If more than two transactions are competing for an intersecting set of granules, then deadlock is even more likely to result.

Notice that there is a fundamental difference between deadlocks due to majority locking and the other schemes mentioned above. The former has been termed *communication* deadlock, since it was first studied in systems of communicating processes, where a process waits to communicate with *any one* from a set of neighbors. The same principle underlies majority locking, in which a transaction that is blocked can proceed after some other transaction releases its lock on a parent of the granule in question.

The other examples demonstrate what has been called a *resource* deadlock, which assumes that a process becomes unblocked only after it receives *all* the resources for which it is waiting. In the case of a database model in which a transaction is either active or waiting for exactly one resource, the distinction between resource and communication deadlock is irrelevant since they reduce to the same concept.

As we shall see in Section 2, there is a whole hierarchy of deadlock models that subsumes—among others—the traditional resource and communication models.

1.3 Deadlock in Distributed Databases

In general, a distributed DBS consists of a number of sites, each of which constitutes a centralized system. Hence all problems of the previous section plus additional ones due to the distributed nature of the database (e.g., replication of data, single transactions executing in parallel at different sites) are present. Also, distributed deadlock is harder to detect, since each site has only a local view of the whole system, and hence collaboration of the sites is required to detect deadlocks involving more than one site.

Both resource and communication deadlocks can be distributed. In distributed DBSs, transactions that access nonlocal data migrate to other sites by invoking subtransactions that may run concurrently with each other. So the originating transaction is blocked until *all* subtransactions terminate, an indication of the resource model.

Communication deadlock can occur if in a replicated database a transaction requests the value of some nonlocal data item and is blocked until one of the sites that hold a copy of this item responds. Furthermore, one can conceive of subtransactions running in parallel on a replicated database, resulting in situations in which resource and communication models are interwoven.

As an example of such an interplay between both models, consider a distributed DBS with replicated data. Gifford [1979] has shown that in order to preserve database consistency, a transaction that wants to read (write) a replicated data item, must read (write) r (w) copies out of the n copies of the data item such that $r + w > n$ and $2w > n$. This is to ensure that at most one writer has access to a replicated data item at a time. To read or write some copy of a data item, a transaction must request and obtain a lock on this copy. Therefore, the reading and writing of a data item generate so-called $\binom{n}{r}$ and $\binom{n}{w}$ resource requests, re-

spectively. These requests require essentially a combination of the resource and communication models.

A uniform model of a distributed DBS underlying most of the algorithms found in the literature is made precise in the following section.

1.4 The Database Model

We introduce a model due to Menasce and Muntz [1979] for studying deadlock detection algorithms for distributed DBSs. A distributed DBS consists of a collection of N *sites*, $S_1, S_2, \ldots, S_N$, connected by a communication network. The network is assumed to be reliable and fully connected. Each site is a centralized DBS that stores some portion of the database. There are M transactions, $T_1, T_2, \ldots, T_M$ running on the distributed database. A transaction presents *resource requests* to a transaction manager (TM), also called controller. There is one controller C_i per site S_i. A resource request may be a request to lock some data item or may have a more abstract meaning. A transaction is blocked from the time it presents a request to a TM until the TM grants the request and the transaction becomes active. A resource request can be local or can refer to a resource at another site, in which case the transaction is distributed. A distributed transaction T_i is implemented by *transaction agents* t_{ij}, each of which is the local agent for transaction T_i at site S_j. In case a transaction agent t_{ij} requests a nonlocal resource that is managed by some controller C_m, controller C_j transmits the request to agent t_{im} via controller C_m. When t_{im} acquires the requested resource from C_m, it sends a message to t_{ij} (via C_m and C_j) stating that the resource has been acquired. Hence intersite requests are always between two agents of the same transaction.

When agents in a transaction T_i no longer need a resource managed by controller C_m, they communicate with agent t_{im}, which is responsible for releasing the resource to C_m. We assume that messages sent by any controller C_i to C_j arrive sequentially and in finite time. We assume further that if a single transaction runs by itself in the distributed DBS, it will terminate in finite time and release all resources. When two or more transactions run in parallel, deadlock may arise.

A transaction agent is said to be *idle* if it is waiting to acquire a resource; it is said to be *executing* if it is not idle. Thus, if an agent never acquires a requested resource, it is permanently idle. For notational simplicity, we may assign a single identifying subscript (rather than a double subscript) to an agent. Hence t_i denotes the ith agent.

In subsequent sections we often refer to *processes* instead of transaction agents. Processes are more powerful than transaction agents. They are assumed to know the identities of all the processes they are waiting for, for example, by having access to their controller's tables. Besides sending request and release messages like transaction agents, they can also exchange other messages. This means that, with respect to deadlock computations, transactions are passive objects, whereas processes are active participants in deadlock detection. In our database model, processes can be thought of as belonging in part to the controller and in part to the transaction. Although there can be at most one transaction agent per transaction at each site, there is no such restriction for processes. As for controllers, message passing between processes is assumed to be first in, first out.

At any time a process is in one of two states: blocked or executing. A process is *blocked* from the time it issues a resource request until it receives a grant message for the requested resource. If a process never receives a grant message for which it is waiting, it is permanently blocked. While a process is blocked it may not send any request or grant messages. It may, however, send and receive other messages or perform other tasks (e.g., related to deadlock detection). Examples of this behavior are given in later sections.

Henceforth we refer to either transaction agents or processes, depending on which variant of the model is more appropriate for our discussion.

1.5 A Specification of the Deadlock Problem

A transaction wait-for-graph (WFG) is a mathematical model of resource requests. The vertices of the graph are associated with transaction agents (or processes, depending on the context). Directed edges in the graph represent blocking relations between transaction agents (processes). A vertex with outgoing edges corresponds to an idle transaction agent (blocked process). More precisely, there is an edge in the WFG from transaction agent t_{ij} to t_{kj} if controller C_j has a request from t_{ij} for resources held by t_{kj}; such an edge is called an *intracontroller* edge. There is an edge from t_{ij} to t_{im} if t_{ij} is waiting for a grant message from t_{im} (that it has acquired a resource managed by C_m); such an edge is called an *intercontroller* edge.

A *cyclic structure* in this graph indicates a deadlock. The precise definition of the term cyclic structure depends on the deadlock model we are considering. An example is given in Figure 1. There are five transactions T_1–T_5, implemented by eight transaction agents. The directed edge from node t_{11} to node t_{21} indicates that transaction agent t_{11} is blocked. This edge is an intracontroller edge, whereas edge (t_{21}, t_{22}) is an intercontroller edge. Node t_{22} has no outgoing edges and is therefore active (nonblocked). Node t_{11} has two outgoing edges, which means that it has two outstanding resource requests. Note the presence of the cycle $t_{11} \rightarrow t_{31} \rightarrow t_{33} \rightarrow t_{43} \rightarrow t_{41} \rightarrow t_{11}$ in the WFG. This cycle may or may not indicate a deadlock, depending on which deadlock model we adopt. The relationship between deadlocks and WFGs is made more precise in Section 2, when we talk about specific deadlock models.

The correctness of a deadlock algorithm depends on two conditions. First, every deadlock must be detected eventually. This constitutes the basic progress property any solution must have. Second, if a deadlock is detected, it must indeed exist (safety property). Incorrectly detected deadlocks due to message delays and out-of-date WFGs have been termed *phantom deadlocks*. In the presence of spontaneous aborts no deadlock scheme can guarantee to detect only genuine deadlocks. For global deadlock detection, Bernstein et al. [1987] show that as long as transactions follow a 2PL protocol, a phantom deadlock can occur only if some transaction spontaneously aborts. In accordance with most of the articles on the subject, for the purpose of our discussion we assume that the DBS is free of spontaneous abortions.

1.6 Centralized versus Distributed Deadlock Detection

There are a number of reasons why distributed deadlock detection seems more attractive than a centralized scheme, that is, one in which a single agent is responsible for deadlock detection. First, a centralized deadlock detection algorithm is vulnerable to failures of the central detector. Hence special provisions for this kind of faults have to be made, resulting in long delays until a new central agent is determined and supplied with up-to-date wait-for information. Distributed algorithms deal with these kinds of problems in a much more natural way. Furthermore, because of the heavy traffic to and from the central agent, this agent can constitute a performance bottleneck, limiting the overall performance of the DBS.

More evidence for the superiority of distributed schemes is supplied by the observation that for typical applications most WFG cycles are very short. Bernstein et al. [1987] give theoretical reasons for the predominance of short paths in WFGs. In particular, for most applications over 90% of WFG cycles can be expected to be of length 2. The same figure also appears in an empirical study [Gray et al. 1981].

The observation that deadlock cycles are short makes centralized deadlock detection an even less attractive choice. With a global algorithm there may be a significant time and message overhead in assembling all the local WFGs at the global detector. Thus, a distributed deadlock might go undetected for quite a while. Since most deadlocks involve only two sites, they can detect the deadlock more efficiently by communicating directly.

Mitchell and Merritt [1984] present a fully distributed deadlock detection algo-

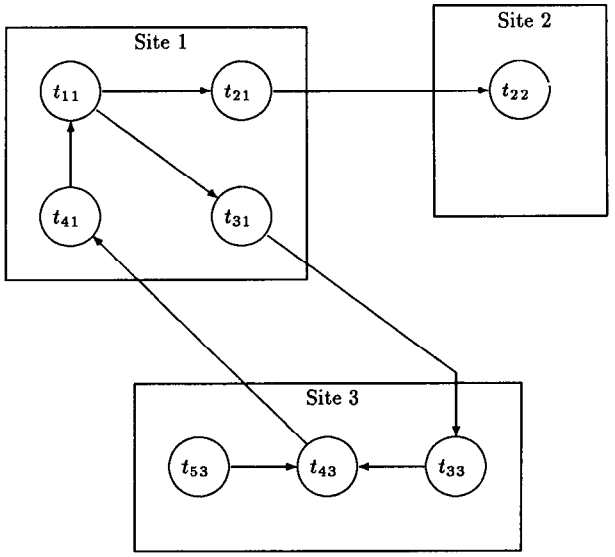

Figure 1. Example of a WFG.

rithm that has a very simple and appealing correctness proof and that, according to the authors, had been implemented in a DBS in less than an hour. This dissents from the widely held opinion that distributed algorithms are necessarily more complex and harder to prove and implement than centralized schemes.

2. MODELS OF DEADLOCK

Depending on the application, database systems allow a number of different kinds of resource requests. For example, a transaction might need to acquire a combination of resources like (resource *a and* resource *b*) *or* resource *c*. This section introduces a hierarchy of request models used in the literature, starting from very restricted forms and going to models with no restrictions whatsoever. This hierarchy can then be used to classify deadlock detection algorithms according to the complexity of the resource requests they permit.

2.1 One-Resource Model

The simplest possible model is one in which a transaction can have at most one outstanding resource request at a time. Hence the maximum outdegree of the WFG is 1. Finding deadlocks in this model corresponds to finding a *cycle* in the WFG. A formal justification for this correspondence is given in the next section. The model is widely used in theoretical studies of database systems (cf. Bernstein et al. [1987] and Papadimitriou [1987]). A very simple and elegant algorithm for deadlock detection in the one-resource model appears in Mitchell and Merritt [1984] and is described in Section 4.2.

2.2 AND Model

In the AND model, transactions are permitted to request a set of resources. A transaction is blocked until it is granted *all* the resources it has requested. Therefore, requests of this type are called AND requests. The AND model is identical to the resource model mentioned in Section 1.2. We prefer the term AND model for systematic reasons. The AND model has been the traditional view of resource requests in distributed DBSs. The nodes of the WFG are called AND nodes and may have outdegree greater than 1. The problem of detecting deadlocks again reduces to finding cycles in the WFG.

As an example, again consider the WFG given in Figure 1. Node t_{11} has two outstanding resource requests, and in the case of the AND model both must be satisfied before t_{11} becomes active. The example depicts a deadlock situation, corresponding to the cycle $t_{11} \rightarrow t_{31} \rightarrow t_{33} \rightarrow t_{43} \rightarrow t_{41} \rightarrow t_{11}$.

More precisely, we define deadlock in the AND model along the lines of Chandy and Misra [1982] as follows: A transaction agent t_i is said to be *dependent* on agent t_j if there is a sequence $seq = t_i, t_{i_1}, \ldots, t_{i_m}, t_j$ of transaction agents such that each agent in *seq* is idle and each agent except the first holds a resource for which the previous agent in *seq* is waiting. We define t_i to be locally dependent on t_j if all the agents in *seq* belong to the same controller. Observe that, if t_i is dependent on t_j, then t_i remains idle at least as long as t_j does. Furthermore, t_i is deadlocked if it is dependent on itself or an agent that is dependent on itself. In either case, deadlock exists only if there is a cycle of idle agents, each dependent on the next one in the cycle.

Deadlock detection algorithms for the AND model declare that deadlock exists if and only if such cycles exist. Note that this condition does not imply that, if an agent t_i is deadlocked, the detection algorithm will detect that t_i is deadlocked. In fact, if t_i is deadlocked but not part of a cycle of deadlocked agents, t_i might never be declared deadlocked. As an example, consider transaction agent t_{53} in Figure 1, which is deadlocked even though it is not part of a cycle.

Deadlock in the one-resource model is conveniently defined the same way, with the additional restriction that a transaction agent can have at most one outstanding request (i.e., one outgoing edge) at a time. From this it is immediate that the AND model is strictly more general than the one-resource model.

In the literature a number of algorithms have been proposed for the AND model [Chandy and Misra 1982; Chandy et al. 1983; Gligor and Shattuck 1980; Haas 1981; Haas and Mohan 1983; Menasce and Muntz 1979; Obermarck 1980, 1982]. We take a closer look at two of them [Obermarck 1982; Chandy and Misra 1982] in Sections 4.1 and 4.3, respectively.

2.3 OR Model

An alternative model of resource requests is the OR model. A request for numerous resources is satisfied by granting *any* requested resource, such as satisfying a read request for a replicated data item by reading any copy of it. This model was referred to as communication model in Section 1.2. In the OR model, discovery of a cycle is insufficient for deadlock detection. To see this, suppose all requests in Figure 1 are OR requests; the nodes are then called OR nodes. In this case, transaction T_1 is not deadlocked because t_{22} has no outgoing edges, and after T_2 releases the resources it holds, T_1 can continue.

In terms of the WFG, a *knot* will indicate a deadlock [Holt 1972]. By definition, a vertex v is in a knot if ($\forall w :: w$ is reachable from $v \Rightarrow v$ is reachable from w). Intuitively, no paths originating from a knot have "dead ends."

Formally, we define deadlock in the OR model in terms of processes as follows (cf. Chandy et al. [1983]): A process is *blocked* if it has an outstanding OR request. Associated with each blocked process is a set of processes, called its *dependent set*. A blocked process starts executing upon receiving any grant message from a process in its dependent set. Otherwise it does not change state or its dependent set. Intuitively, a set S of processes is deadlocked if all processes in S are permanently blocked. A process is permanently blocked if it never receives a grant message from any process in its dependent set. More precisely, a set S of processes is deadlocked if

(1) all processes in S are blocked,
(2) the dependent set of every process in S is a subset of S, and
(3) there are no grant messages in transit between processes in S.

A process is deadlocked if it belongs to some deadlocked set. A set S of processes satisfying the above three conditions remains permanently blocked because (1) a blocked process p_i in S can start executing only after receiving a grant message from some process p_j in its dependent set, (2) every process p_j in p_i's dependent set is also in S and cannot send a grant message while remaining blocked, and (3) there are no grant messages in transit from p_j to p_i, which implies that p_i will never receive a

message from any process in its dependent set.

Presence of a deadlocked set of processes is equivalent to the existence of a knot in the WFG. Hence, deadlock detection in the OR model can be reduced to finding knots in a graph. Note, however, that a process can be deadlocked without being in a knot. Rather, a necessary and sufficient criterion for deadlock of some process p is the following: A blocked process p is deadlocked if p is in a knot or p can reach only deadlocked processes. The algorithm for the OR model we discuss in Section 5.4 detects deadlock for *any* process belonging to some deadlocked set.

AND-model deadlock detection can be simulated by repeated applications of OR-model deadlock computations, where each invocation operates on a subgraph of the AND-model WFG. This method, however, is hopelessly inefficient, so it is only of theoretical interest. In this sense, OR-model deadlock is a more general notion than AND-model deadlock.

An algorithm for distributed knot detection appears in Misra and Chandy [1982a]. Termination detection of diffusing computations in the OR model, which is also the model of deadlock in CSP, is discussed in Misra and Chandy [1982b]. The algorithm presented in Section 5.4 is taken from Chandy et al. [1983]. Other algorithms for the OR model are given in Haas [1981], Natarajan [1986], and Räuchle and Toueg [1983].

2.4 AND–OR Model

The AND–OR model is a generalization of the two previous models. AND–OR requests may specify any combination of *and* and *or* in the resource request. For example, a request for (a *and* (b *or* c)) *or* d is possible, and a, b, c, and d may exist at different sites. There does not appear to be a familiar construct of graph theory to describe a deadlock situation in the AND–OR model in terms of the WFG. In principle, deadlock in the AND–OR model can be detected by repeated application of the test for OR-model deadlock, exploiting the fact that deadlock is a *stable* property; that is, it does not go away by itself. But this strategy is, in general, not very efficient. A more efficient algorithm that was developed in Hermann and Chandy [1983] is the topic of Section 5.5. This section also includes a formalization of deadlock in the AND–OR model. Since this definition does not capture the notion of deadlock exclusively, it has been omitted from the more general discussion here.

2.5 $\binom{n}{k}$ Model

The $\binom{n}{k}$ model allows the specification of requests to obtain any k available resources out of a pool of size n. The $\binom{n}{k}$ model is a generalization of the AND–OR model. Even though it turns out that both models are equivalent in expressive power, the length of an AND–OR formula corresponding to an $\binom{n}{k}$ request is $k\binom{n}{k}$, which is of exponential size for $n \approx 2k$, since

$$\binom{2n}{n} \stackrel{\text{def}}{=} \frac{2n(2n-1)\cdots(n+1)}{n(n-1)\cdots 1}$$

$$\geq \underbrace{\frac{2n}{n}\,\frac{2n-2}{n-1}\cdots\frac{2}{1}}_{n \text{ factors}}$$

$$= 2^n.$$

So every request in the $\binom{n}{k}$ model can be expressed in the AND–OR model. To see that the converse is also true, observe that any AND or OR requests for n resources can be stated as an $\binom{n}{n}$ or $\binom{n}{1}$ request, respectively. The only definition of deadlock in the $\binom{n}{k}$ model we know was given by Bracha and Toueg [1983] and suffers from the same deficiencies as that of the AND–OR model. It is, therefore, not discussed here. An algorithm for deadlock detection in the $\binom{n}{k}$ model was published in Bracha and Toueg [1983] and is presented in Section 5.6.

2.6 Unrestricted Model

In the most general model no underlying structure of resource requests is assumed. Instead, the stability of deadlock is the only assumption made. The advantage of looking at the deadlock problem in this way is

that it helps in the separation of concerns: Properties of the underlying database computations (e.g., degree of concurrency, i.e., single locus of control for each transaction versus parallelism of individual transactions, and message passing versus synchronous communication) are rigorously abstracted and separated from concerns about properties of the problem (stability of deadlock). Therefore, all the algorithms dealing with this general model can be used to detect other stable properties as well. However, in the context of deadlock detection in distributed databases, these algorithms seem to be of more theoretical value, since the very fact that no further assumptions are made about the underlying structure of the database computation leads to a great deal of overhead that can be avoided in algorithms for the simpler models. In Section 3.4 we present a general theory due to Chandy and Lamport [1985], which can be applied to both the previous and the unrestricted models. For more details on the subject, the interested reader is referred to Awerbuch and Micali [1986], Chandy and Lamport [1985], Chandy and Misra [1986], Chang [1982], Hélary et al. [1987], and Misra [1983].

3. CLASSES OF DISTRIBUTED DEADLOCK DETECTION ALGORITHMS

The distributed deadlock detection algorithms that are found in the literature developed basically from four different roots: path-pushing, edge-chasing, diffusing computations, and global state detection. This observation gives rise to another way of classification, which will be developed in the next four sections.

3.1 Path-Pushing Algorithms

The first distributed algorithms for the deadlock problem maintained the notion of an explicit global WFG, which had worked so well in the centralized case. One influential algorithm appeared in Menasce and Muntz [1979]. The basic idea underlying this class of algorithms is to build some simplified form of global WFG at each site. For this purpose each site sends its local WFG to a number of neighboring sites every time a deadlock computation is performed. After the local data structure of each site is updated, this updated WFG is then passed along, and the procedure is repeated until some site has a sufficiently complete picture of the global situation to announce deadlock or to establish that no deadlocks are present. The main feature of this scheme, namely, to send around paths of the global WFG, has led to the term *path-pushing algorithms.*

One noteworthy point about path-pushing algorithms is that many of them were found to be incorrect, either by not detecting true deadlocks, by discovering phantom deadlocks, or both. For example, Gligor and Shattuck [1980] show that the algorithm of Menasce and Muntz [1979] is defective; a counterexample to the algorithm of Ho and Ramamoorthy [1982] was presented by Jagannathan and Vasudevan [1982]; in Section 4.1 we give reasons why Obermarck's algorithm [Obermarck 1982] is incorrect. This is even more surprising as these algorithms had all been "proved" correct.

Looking back, the failure of many of these algorithms is not so astonishing as might first appear, since at that time the notion of snapshots and consistent global states in asynchronous systems was not well understood. Another consequence of this lack of understanding was the fact that most of the algorithms had to depend on "freezing" the underlying (database) computation for the time the deadlock detection was going on. This guaranteed in most cases that the picture of the assembled global WFG was consistent.

For historical reasons, an example of a path-pushing algorithm [Obermarck 1982], which has been implemented in System R, is presented in Section 4.1.

3.2 Edge-Chasing Algorithms

The presence of a cycle in a distributed graph structure can be verified by propagating special messages called *probes* along the edges of the graph. Probes are assumed to be distinct from resource request and grant messages. When the initiator of such

a probe computation receives a matching probe, it knows that it is on a cycle in the graph.

A nice feature of this approach in connection with deadlock detection is that executing processes can simply discard any probes they receive. Blocked processes propagate the probe along their outgoing edges. An interesting variation of this method can be found in Mitchell and Merritt [1984], where probes are sent upon request and in the opposite direction of the edge. We look at this algorithm more closely in Section 4.2.

Another example for this approach is Chandy and Misra's algorithm [Chandy and Misra 1982], which is discussed in Section 4.3.

3.3 Diffusing Computations

The second category of algorithms was inspired by the work of Chang [1982] and Dijkstra and Scholten [1980]. Here the basic idea is that a *diffusing computation* is activated, for example, by a transaction manager that suspects a deadlock. This computation is superimposed on the underlying database computation. If this computation terminates, the initiator declares deadlock. The characteristic feature of the superposed computation in the case of distributed deadlock detection is that the global WFG is implicitly reflected in the structure of the computation. The actual WFG, however, is never built explicitly. The diffusing computation grows by sending *query* messages and shrinks by receiving *replies*. In our case query and reply messages are concerned exclusively with deadlock detection and are distinct from resource request and grant messages. When a diffusing computation shrinks back to its root, it terminates.

More precisely, nodes different from the root are called *internal* nodes. Each node in the diffusing computation has an initial state called the *neutral state*. The root (also called initiator) sends queries to its successors to start a diffusing computation. After receipt of its first query, a node leaves the neutral state and becomes active. The first query received by node p_i is called the *engaging query* for p_i. The process that sent the engaging query is called the *engager* of p_i. The edge along which the engaging query was sent is called the *engagement edge* of p_i.

After receipt of the engaging query, an internal node is free to send queries to its successors. Besides its ability to receive queries from its predecessors and send queries to its successors, a node is also able to receive replies from its successors and send replies to its predecessors. Notice that queries always travel in the direction of the edges, whereas replies always travel the opposite way.

We require that the number of queries received along an edge always be at least the number of replies sent in the opposite direction. The difference between the number of queries and replies sent over an edge is called the *deficit* of this edge. Hence, from the above we have the following: The deficit of all edges is at least zero.

The neutral state of a node can now be defined to be the state in which the deficits of all incoming and outgoing edges are zero. The diffusing computation terminates if the root returns to its neutral state. When should a node reply to a query? We stipulate that an active node reply to all queries it receives immediately. The crucial question is: When should a node reply to its engaging query? This reply is called the *engaging reply*. We require that a node send back its engaging reply only after it has received replies for each query it sent.

With these stipulations it is not hard to show that (1) each engagement edge connects two active nodes, (2) engagement edges do not form cycles, and (3) each active internal node has exactly one incoming engagement edge. We say that the diffusing computation has terminated if and only if all internal nodes are in their neutral state. From what has been said above, it now follows that (cf. [Dijkstra and Scholten 1980])

(1) when the root returns to the neutral state, the diffusing computation has terminated;
(2) a bounded number of steps after the diffusing computation has terminated,

the root will have returned to the neutral state.

Algorithms using the paradigm of diffusing computations are presented in Sections 4.4 [Chandy and Misra 1982] and 4.5 [Hermann and Chandy 1983]. In general, this approach results in shorter messages and less deadlock detection overhead as compared with path-pushing algorithms. Besides the work mentioned above, there are other variations on this theme [Chandy and Misra 1986; Chang 1982; Dijkstra et al. 1983; Haas 1981; Haas and Mohan 1983; Misra 1983; Misra and Chandy 1982a; Misra and Chandy 1982b].

3.4 Global State Detection

The work that has been done in the area of global state detection is largely based on results by Chandy and Lamport [1985]. The key notion here is a *consistent global state* that can be determined without temporarily suspending ("freezing") the underlying (database) computation. Below we give a condensed presentation of the results of Chandy and Lamport, that are relevant to our context. The discussion follows Bracha and Toueg [1983].

The *underlying computation*, henceforth referred to as the *system*, is a collection of processes, that can be thought of as transaction managers and transaction agents. Processes communicate by sending messages (e.g., resource requests or grants) according to some underlying protocol (2PL, e.g.). *Events* in the system are the sending and receipt of messages. We denote the set of events in a system by E. The *local state* of a process p consists of the history of all events that occurred on p. Along the lines of Lamport [1978] we define a partial order $\preceq \subseteq E \times E$ as follows:

Definition 3.1

Let $e_1, e_2 \in E$. Then $e_1 \preceq e_2$ (e_1 happened before e_2) if either

(1) e_1 and e_2 are both on the same process p, and e_1 occurred earlier in p then e_2;
(2) e_1 is a send event and e_2 is the corresponding receive;
(3) $(\exists e' : e' \in E : e_1 \preceq e' \wedge e' \preceq e_2)$.

Part (1) of the definition says that the events of a single process are totally ordered. Part (2) expresses the fact that messages are received after they are sent. Part (3) essentially states that $\preceq$ is transitive.

We can represent the history of a system and its happened-before relation by a diagram like that in Figure 2. The dots represent events, the horizontal lines are the time axes of the processes, and the arrows link corresponding sends and receives.

The following formalization is due to Chandy and Lamport [1985]. A *cut* c of E is a partition of E into two sets P_c and F_c, standing for past and future, respectively. A cut is *consistent* if F_c is closed under $\preceq$. A consistent cut defines a consistent state. Hence we use consistent cut and consistent state interchangeably. Intuitively, consistent cuts are those that do not contain a send event in F_c with the corresponding receive event in P_c.

Looking again at the example in Figure 2, we see that $P_c = \{e_1, e_3, e_4, e_7, e_8, e_9, e_{10}\}$ and $F_c = \{e_2, e_5, e_6\}$. Furthermore, since F_c is closed under $\preceq$, c is a consistent cut.

A special type of consistent state is S_t, the global state at time t that is the collection of all the local states of the processes at time t. Note that S_t is a purely theoretical construct that cannot be observed, since this would require an outside observer to record the local states of the processes instantaneously, an impossible task in practice. In contrast, consistent states can be obtained from within the system. We now extend the relation $\preceq$ to consistent states.

Definition 3.2

Let S_1, S_2 be consistent states. Then $S_1 \preceq S_2$, if $P_{S_1} \subseteq P_{S_2}$.

We define a relation $\vdash$ between states, called reachability relation.

Definition 3.3

Let S be a consistent state and $e \in E$, such that $P_S \cup \{e\}$ defines a consistent state S'. Then $S \vdash^e S'$ (S' is *reachable* from S).

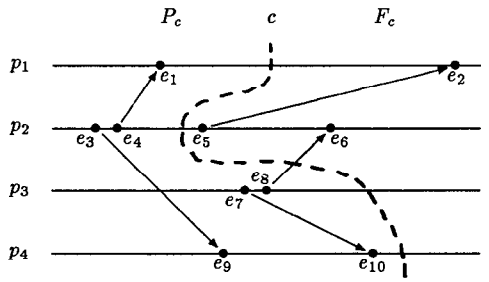

Figure 2. A cut of a distributed system.

A sequence of events $\sigma = (e_1, e_2, \ldots, e_n)$ is a *schedule*, if $S \vdash^{e_1} S_1 \vdash^{e_2} \cdots \vdash^{e_{n-1}} S_{n-1} \vdash^{e_n} S'$. We then write $S \vdash^{\sigma} S'$ for short. One can show [Chandy and Lamport 1985] that

Lemma 3.1

$S \preceq S'$ implies ($\exists$ schedule $\sigma :: S \vdash^{\sigma} S'$).

In the context of deadlock detection, the state of the system is a WFG, and schedules are sequences of WFG transformations. We say that a transaction is deadlocked if it is deadlocked in WFG_t, the WFG at time t. Given a definition of deadlock in terms of a WFG_t, the following lemma of Bracha and Toueg [1983] allows us to apply a distributed deadlock detection algorithm to consistent WFGs instead of WFG_ts:

Lemma 3.2

(WFG $\preceq$ WFG′ and v is deadlocked in WFG) implies v is deadlocked in WFG′.

This is the fundamental result on which deadlock detection algorithms can be based. Chandy and Lamport [1985] show how to obtain a consistent global state of a distributed system by propagating markers along the channels of the system. A consistent global state obtained in this fashion is also called a snapshot of the system. Such a snapshot can then be examined for deadlock off-line. Since this snapshot is by definition a static object, there are no problems in conjunction with message delays, and deadlock detection becomes much easier. In Section 4.6 we see an example of the application of this result.

4. A SURVEY OF SELECTED ALGORITHMS

4.1 Obermarck's Path-Pushing Algorithm

In this section we discuss an algorithm that appeared in Obermarck [1982]. The underlying deadlock model is the AND model; hence the algorithm looks for cycles in the global WFG. First, the author makes some simplifying assumptions:

(1) Transactions have a single locus of control; that is, at most one transaction agent of each transaction can be active at any time.
(2) Communication between transaction agents is logically synchronous.
(3) The transactions are totally ordered, which is useful in reducing deadlock detection overhead and ensuring that exactly one transaction in each cycle detects deadlock.
(4) The portion of the local WFG sent from one site to another does not change until the information has been received and processed by some final site. The final site is defined as
    (a) the site at which a deadlock cycle is completed, or
    (b) the most distant site at which global deadlock can be proved not to exist.

Points (1) and (2) imply that in each transaction only one agent may be active or in resource-wait. This agent is expected to send a message to other agents of the same transaction, which are waiting to receive a message.

Obermarck [1982] admits that point (4) is a fairly unrealistic assumption. It turns out that even with this assumption the algorithm is still vulnerable to detecting phantom deadlocks. He suggests that, if the occurrence of deadlock is rare, the assumption be dropped and cycles found by the algorithm be validated.

Each controller c_j at site S_j runs a copy of the deadlock detection algorithm. The basic structure of the algorithm is an iteration of the following steps:

(1) Receive deadlock information from some other sites that was produced

by the previous deadlock detection iteration.

(2) Build part of the global WFG using local wait-for information and the information received from other sites in step 1. A special node in the WFG called EXternal is used to represent intersite wait-for relations.

(3) Find all elementary cycles in the WFG. Break all cycles that do not contain EX by aborting suitable transactions.

(4) Consider each elementary cycle containing EX. Such a cycle constitutes a potential global deadlock. For each such cycle EX $\rightarrow T_1 \rightarrow \cdots \rightarrow T_n \rightarrow$ EX compare T_1 with T_n. If $T_1 > T_n$, send the cycle to each site, where an agent of T_n is waiting to receive a message from the agent of T_n at this site.

In Obermarck [1982, sect. 6], an attempt is made to prove the algorithm correct. That the algorithm and proof are incorrect (in the sense that false deadlocks may be detected) can easily be seen from the following observation [Elmagarmid 1986]: The portions of the WFG that are shipped around may not represent a consistent view of the global WFG, since each site takes its snapshot asynchronously.

As far as the performance of the algorithm is concerned, Obermarck shows that, if s sites are involved in a deadlock, at most $s(s-1)/2$ messages are sent, where each message may be of length $O(s)$. Under certain assumptions the expected case performance is shown to be roughly linear in s, with a small constant factor.

For more details, the reader may consult Obermarck [1982]. In our opinion, however, this algorithm has been rendered obsolete by more recent developments.

4.2 Mitchell and Merritt's Algorithm for the Single-Resource Model

The algorithm by Mitchell and Merritt [1984] presented in this section is as simple as the deadlock model for which it was defined. It is an edge-chasing algorithm in which probes are sent in the opposite directions of the edges of the WFG. In the simplest case, a probe consists of a single natural number that is unique to the nodes in the WFG. When the probe comes back to its initiator, the initiator declares deadlock.

The algorithm will be stated in terms of processes. It has a number of nice features:

(1) It is very simple, making the proofs elegant and fun to read (and write), and the task of implementing a matter of hours.

(2) Exactly one process in the cycle will detect deadlock, which simplifies deadlock resolution since this process could simply abort. By including priorities in the algorithm, the lowest priority process in a cycle detects deadlock and aborts.

(3) Spontaneous aborts are allowed, even though under this assumption phantom deadlocks cannot be excluded. It can be shown, however, that only genuine deadlocks will be detected in the absence of spontaneous aborts.

In this discussion, only the first version of the algorithm (without priorities) is given. The extension to priority handling can be found in Mitchell and Merritt [1984].

Each node of the (virtual) WFG has two local variables, called labels: a *private* label, which is unique to the node at all times, though not constant, and a *public* label, which can be read by other processes and need not be unique. A process is represented as $\binom{u}{v}$ where u and v are the public and private labels, respectively. Initially, private and public labels are equal for each process.

The state of the system is given by the global WFG. The WFG is maintained by the four state transitions shown in Figure 3, where $z = \text{inc}(u, v)$, and $\text{inc}(u, v)$ yields a unique label greater than both u and v. Labels not mentioned explicitly remain unchanged.

Block creates an edge in the WFG. Two messages are needed, one resource request and one message back to the blocked process to inform it of the public label of the

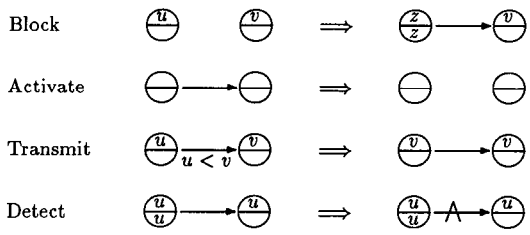

Figure 3. The four possible state transitions.

process it is waiting for. *Activate* means that a process acquired the resource from the process it was waiting for. *Transmit* propagates larger labels in the opposite direction of the edges by sending a probe message. A process that receives a probe with a number smaller than its own public label can simply ignore the probe. *Detect* means that the probe with the private label of some process has made a whole round in a circle, indicating a deadlock.

Note that the requirement for uniqueness of the public label does not cause any problems at all in the Block step of the algorithm. Assuming that we can assign a unique name to each process, labels can be represented as pairs of sequence numbers and process names, and $<$ can be chosen to be lexicographical ordering. Then to Block, only sequence numbers have to be compared; to Transmit, the whole pair is sent.

The proof of the correctness of the algorithm is quite simple. Mitchell and Merritt show that every deadlock is detected. Since they did not exclude spontaneous aborts, they did not worry about phantom deadlocks. Below we prove that in the absence of spontaneous aborts only genuine deadlocks are detected.

Assume for now that there are no spontaneous aborts. The following is an *invariant*:

For all processes $\ominus: v \leq u$.

Proof. Initially $u = v$ for all processes. The only transactions that change u or v are

(1) Block: u and v are set such that $u = v$.
(2) Transmit: u is increased. □

From the invariant the following lemma is immediate:

Lemma 4.1

For any process $\ominus$, *if* $u > v$, *then* u *was set by a Transmit step.*

Now we are ready to prove the following theorem:

Theorem 4.1

If a deadlock is detected, a cycle of blocked nodes exists.

Proof. Deadlock is detected if the following edge $p \rightarrow p'$ exists:

$$\ominus \rightarrow \ominus$$

We will prove the following claims:

(1) u has been propagated from p to p' via a sequence of Transmits.
(2) p has been continuously blocked, since it "transmitted" u (i.e., engaged in a Transmit event with some process q, $q \rightarrow p$).
(3) For all intermediate nodes q in the transmit path of (1), including p', q has been continuously blocked since it transmitted u.

The result then follows immediately.

Ad 1. By the invariant and the uniqueness of private labels, we have for the private label v of $p' : v < u$. By Lemma 4.1, u was set by a Transmit step. By the semantics of Transmit, there is some p'' with private label u, public label w.

If $w = u$, then $p'' = p$, and we are done. Otherwise, $w < u$, and we repeat the argument. Since there are only finitely many processes, one of them is p.

Ad 2. Assume that p was active since it transmitted u. It is blocked when it detects deadlock; hence upon Blocking it incremented its private label. But then private and public labels cannot be equal.

Ad 3. Assume that there is a process that has been active since it transmitted u. Its predecessor has been active since its transmission, too, because Transmits migrate in the opposite direction of the

edges. By repeating this argument, we find that p has been active since it transmitted u. □

The algorithm given can be easily extended to include priorities such that the lowest priority process in a deadlock cycle aborts itself. The extended algorithm has two phases. The first phase is almost identical to the simple algorithm. In the second phase the smallest priority is propagated around the circle, as the largest public label was propagated before. The propagation stops when one process recognizes the propagated priority as its own. The full algorithm is given in Mitchell and Merritt [1984].

The performance of this algorithm is not studied in Mitchell and Merritt [1984]. It is, however, not hard to obtain the following complexity results. Assuming that a deadlock persists long enough to be detected, the worst-case complexity of the simple algorithm is $s(s - 1)/2$ Transmit steps, where s is the number of processes in the cycle. After this many steps, every process in the cycle will have compared its public label with every other one. That this bound is tight can be seen from an example in which the public labels of the processes are ordered increasingly around the cycle, with the detecting process having the greatest label and the process for which the detecting process is waiting having the smallest label. A similar argument shows that for the priority algorithm the largest number of Transmit steps for detecting deadlock is twice as large: $s(s - 1)$. We conjecture that the expected-case complexity is linear for both algorithms.

Interestingly, the algorithm does not remain correct if public labels are transmitted in the same direction as the edges instead of the other way round. The reason for this is exactly the point we were making when we defined AND- and OR-model deadlock in Section 3: If a deadlocked process that is not part of a cycle has the largest public label among the deadlocked processes, this label might enter the cycle and circulate once without any process in the cycle detecting the deadlock. Also, there seems to be no straightforward extension of the algorithm for handling processes with more than one outgoing edge (AND requests).

4.3 Chandy and Misra's Algorithm for the AND Model

In the approach developed by Chandy and Misra [1982], each controller runs a copy of the deadlock detection algorithm. In order to determine whether an idle transaction agent is deadlocked, its controller initiates a probe computation. In a probe computation, controllers send probes to each other. Probe computations may be initiated for several transactions, and the same transaction agent may have several probe computations initiated for it in sequence. A probe consists of a triple (i, j, k), denoting that it belongs to a probe computation for t_i[4] and that this probe was sent along intercontroller edge (t_j, t_k). A controller sends a probe (i, j, k) if the following conditions hold: (1) t_j is idle, (2) t_j is waiting for t_k, and (3) t_i is dependent on t_j. We call an intercontroller edge (t_j, t_k) that meets these three conditions an *outgoing edge* of t_i.

Probes received by a controller may be discarded or accepted; probes that are accepted are called *meaningful.* Formally, a probe (i, j, k) is meaningful if (1) t_k is idle and (2) the controller of t_k did not know that t_i was dependent on t_k and can now deduce that t_i is dependent on t_k. It is immediate that, if the controller of t_i receives a meaningful probe (i, j, i) for any j, then t_i is deadlocked. The formulation of these observations in terms of an algorithm can be found in Figure 4.

Observe that in a subsequent refinement step, the test of whether a probe is meaningful can be implemented by a Boolean array $dependent_k$, where $dependent_k(i) \equiv$ t_k's controller knows that t_i is dependent on t_k. Local dependence and outgoing edges can be determined by a standard marking algorithm, like the one used for reachability problems in graphs.

The algorithm is proved correct by coloring the edges of the WFG in the following

[4] Note, that we make use of the fact that double subscripts can be replaced by a single subscript.

For initiation of a probe computation by the controller of t_i:

if t_i is locally dependent on itself
  then declare deadlock for t_i
  else send probes (i, j, k) on all outgoing edges (t_j, t_k) of t_i.

For a controller on receiving a probe (i, j, k):

if the probe is meaningful
  then if $k = i$
    then declare deadlock for t_i
    else send probes (i, p, q) on all outgoing edges (t_p, t_q) of t_k.

Figure 4. Algorithm for the AND model.

manner. An edge (t_i, t_j) is

- *gray* if t_i has sent a request to t_j that t_j has not yet received;
- *black* if t_j has received a request from t_i but has not yet sent a grant message to t_i;
- *white* if t_j has sent a grant message to t_i but t_i has not yet received it.

Hence edge colors represent the state of a "channel" between processes. This approach is also used in Bracha and Toueg [1984], Chandy et al. [1983], and Hermann and Chandy [1983].

Gray and black edges are called *dark* edges. It is easy to see that in our model of message passing, a dark cycle, that is, a cycle in which all edges are dark, will persist forever. Hence the existence of a dark cycle is equivalent to deadlock. In order to prove the algorithm correct, one must show the following:

(1) [Safety] If the initiator of a probe computation for t_i receives a meaningful probe (i, j, i), then t_i is on a black cycle when this probe is received.
(2) [Progress] If t_i is on a dark cycle at the time its controller initiates a probe computation for it, then the controller of t_i will eventually get a meaningful probe (i, j, i).

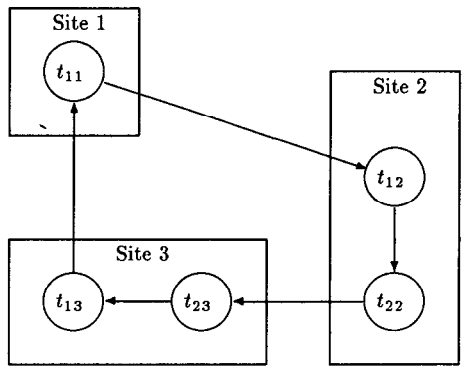

Figure 5. A counterexample to the algorithm in Section 6.6 of Chandy and Misra [1982].

Incidentally, the algorithm given in Chandy and Misra [1982, sect. 6.6] is not correct, as can be observed by applying it to the counterexample of Figure 5. If the controller at site 1 initiates a probe computation for t_{11}, t_{11} will not be marked in the process of finding a local deadlock at site 1 (because t_{11} is not part of a local deadlock). Since the set of outgoing edges is determined starting from marked transaction agents only, the set of outgoing edges will be found empty, and no probes will be sent. Therefore, deadlock will never be detected by the controller of site 1, even though t_{11} is part of a deadlock cycle. A subsequent version of the algorithm that

appeared in Chandy et al. [1983, sect. 3.1] introduced the notion of dependence and is (to our knowledge) correct. The formulation of this algorithm given above both retains the notational simplicity of the first—incorrect—solution and is free of errors.

Below, the performance analysis of the algorithm is summarized. Each probe sent is of fixed length. Deadlock detection overhead is introduced primarily when transaction agents are idle (i.e., have nothing to do and nothing to send). Furthermore, if transaction agents that are referred to in a probe are executing, the controller simply discards that probe. Every single deadlock detection computation involves no more than e probes, where e is the number of communicating pairs of controllers in the network. Hence in the worst case $e = N(N - 1)$. Normally, however, e will be much less, depending on the locality behavior of the transactions.

Some optimizations regarding questions of when and how often probe computations should be initiated are given in Chandy and Misra [1982].

4.4 Chandy, Misra, and Haas's Algorithm for the OR Model

The algorithm for the OR model [Chandy et al. 1983, sect. 4] is an application of the technique of diffusing computations. A blocked process can determine whether it is deadlocked by initiating a diffusing computation. Several processes may initiate diffusing computations at the same time. However, for the time being we restrict ourselves to the case in which each process initiates at most one diffusing computation. The extension to the same process initiating diffusing computations several times in a row is then quite straightforward.

The messages in the deadlock computation have the form $query(i, j, k)$ and $reply(i, j, k)$, denoting that these messages belong to the diffusing computation initiated by process p_i and are being sent from p_j to p_k; p_i, p_j, p_k are called the initiator, sender, receiver, respectively. There will be at most one message of the form $query(i, j, k)$; there will be at most one reply message of the form $reply(i, k, j)$ to the query message $query(i, j, k)$. A blocked process initiates a deadlock computation by sending queries to processes in its dependent set (cf. Section 2.3). The basic idea is that a blocked process, on receiving a query, should propagate the query to its dependent set if it has not done so already. Thus, if there is a sequence of permanently blocked processes $p_i, \ldots, p_j$ such that each process in the sequence (except the first) is in the dependent set of the previous process in the sequence, a query initiated by p_i will be propagated to p_j.

Next we discuss the action taken by a process p_k on receiving a query or reply with *fixed* initiator i and some sender j. If p_k is active, it ignores all queries and replies. If it is blocked, there are several possibilities: If p_k receives an engaging query, it propagates the query to all processes in its dependent set and remembers the number of queries sent in a local variable $num(i)$. Let the local variable $wait(i)$ denote the fact that p_k has been continuously blocked since it received its engaging query. If p_k receives subsequent queries, it replies to them immediately, if $wait(i)$ holds. If it has been executing since then, that is, $\neg wait(i)$ holds, it discards the query.

If p_k receives a reply and $wait(i)$ holds, it decrements $num(i)$. When should p_k reply to its engaging query? From our discussion in Section 3.3 it should be clear that p_k replies to its engager only if it has received a reply for each query it has propagated, that is, if $num(i) = 0$.

When p_k initiates a deadlock computation, it does so by sending $query(k, k, j)$ to each process j in its dependent set and setting $num(k)$ to the number of queries sent. If the initiator receives replies to all the queries sent, then the initiator is deadlocked.

These observations lead to the algorithm in Figure 6. S denotes the dependent set of p_k, $wait(i) \equiv false$ initially, for all i.

To guarantee that every deadlock will be detected by some deadlocked process, we now relax the restriction that only one deadlock computation can be initiated by some particular process. We require that a process initiate a diffusing computation

```
For a blocked process p_k to initiate a diffusing computation:
    send query(k, k, j) to all p_j in S.
    set num(i) := |S|; wait(i) := true.

For a blocked process p_k upon receiving query(i, j, k):
    if the query is the engaging query for initiator i
      then send query(i, k, m) to all p_m in S.
           set num(i) := |S|; wait(i) := true.
      else if wait(i) then send reply(i, k, j) to p_j.

Upon receiving reply(i, k, j):
    if wait(i)
      then num(i) := num(i) - 1.
           if num(i) = 0
             then if i = k then declare deadlock for p_k.
             else send reply(i, k, j) to the engager of p_k.

For a blocked process upon becoming executing:
    wait(i) := false, for all i.
```

Figure 6. Simplified algorithm for the OR model.

each time it becomes blocked. To distinguish several diffusing computations initiated by the same process p_i, queries and replies are endowed with an additional parameter denoting the sequence number of the diffusing computation initiated by p_i. The generalized algorithm can be found in Chandy et al. [1983], together with a correctness proof. In particular, the following theorems are proved:

Theorem 4.2

If the initiator of a diffusing computation is deadlocked when it initiates the computation, it will (eventually) declare itself deadlocked.

Theorem 4.3

If the initiator of a diffusing computation declares itself deadlocked, then it belongs to a deadlocked set.

Theorem 4.4

At least one process in every deadlocked set will report deadlock if every process initiates a new diffusing computation whenever it becomes blocked.

The analysis of the algorithm's performance is similar to that for the AND-model algorithm in the previous section. There is a maximum number of e queries and e replies per diffusing computation, where $e = N(N-1)$.

4.5 Hermann and Chandy's Algorithm for the AND–OR Model

The basis of the algorithm for the AND–OR model [Hermann and Chandy 1983] is a so-called *tree* computation. A tree computation consists of a hierarchy of diffusing computations along the lines of Section 3.3. Below we will make this idea more precise. Transaction agents are mapped to processes in the following manner: A process may have an AND request or an OR request; an AND–OR request issued by some transaction agent is mapped to a tree of processes. The mapping is a representation of the AND–OR request in a regular form, such as disjunctive normal form (DNF).

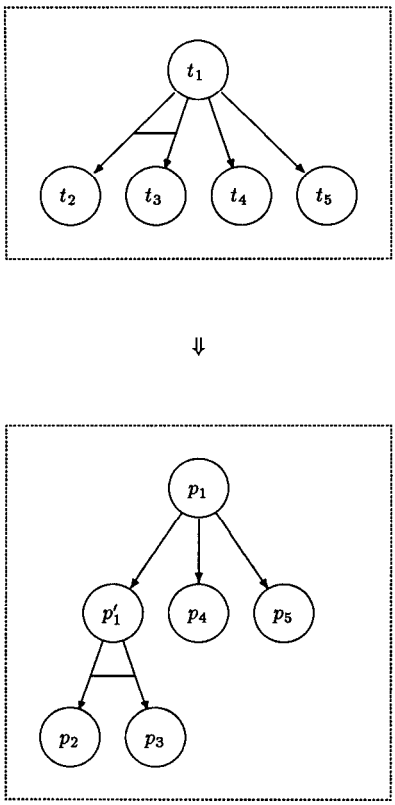

Figure 7. Mapping transaction agents to processes.

Figure 7 gives an example of this mapping. Transaction agent t_1 waits for (t_2 *and* t_3) *or* t_4 *or* t_5; a line connecting edges denotes an AND request. We call processes like p_1' AND processes; consequently, p_1 is called an OR process.

The behavior of individual processes with respect to the underlying computation is a refinement of the process behavior defined in Section 1.3. Upon receiving a grant message an edge in the WFG disappears, and there are several possibilities for the receiving blocked process:

(1) No outgoing edges remain, that is, the process is active.
(2) If outgoing edges remain, there are two cases:
   (a) An AND process remains blocked.
   (b) For an OR process, all outgoing edges disappear instantaneously and the process is active.

The central idea underlying the algorithm is that any time a diffusing computation reaches a blocked OR process, the diffusing computation is propagated to the dependent set of this process; if the engaged process is a blocked AND process, it initiates a separate tree computation for each outgoing edge. So a tree computation consists of either a diffusing computation or a set of tree computations. In order to start a deadlock computation, an initiating process sends a query to the process that is suspected of deadlock. From there queries are propagated according to the rules explained later. A tree computation terminates when its initiator receives a reply from the suspected process. Deadlock in the process model is defined in the following obvious way:

Definition 4.1

A blocked process p is deadlocked, if either

(1) p is an AND process and will never receive a grant for at least one of the resources requested, or
(2) p is an OR process, but will never receive a grant message.

Note, that in order to use this definition to define the correctness of a deadlock detection algorithm, we have to exclude permanent blocking of processes due to individual starvation or infinite loops. For this reason, Hermann and Chandy call this a *local* definition of deadlock.

Next we discuss process behavior with respect to the deadlock computation proper. Let us assume for now that only one deadlock computation is performed at a time. We shall see later how to extend the scheme to many concurrent deadlock computations.

Queries have the form *query*(seq, k), where $seq = \langle i_1, \ldots, i_n \rangle$ is a sequence of processes and k is the sender of the query. The initiator i of a deadlock computation sends *query*($\langle i \rangle$, i). A query is propagated in the following manner: If an engaging

query(*seq*, *m*) arrives at a blocked AND process p_k, a new set of tree computations is initiated by p_k. For this purpose, p_k sends out a query of the form *query*(*seq* ∘ *l*, *k*) to p_l for all outgoing edges (p_k, p_l) of the WFG (∘ denotes concatenation). If a blocked OR process receives an engaging *query*(*seq*, *m*), it propagates *query*(*seq*, *k*) to all processes in its dependent set. These actions are referred to as *extension*.

If *query*(*seq*, *m*) is not engaging and the receiving process p_k has been blocked continuously from the time it received its engaging query, a *reply*(*seq*, *k*), is sent to p_m. This action is called *reflection*.

If a *reply*(*seq*, *m*) is received by an AND process, it sends back its engaging reply if it has been continuously blocked from the time it received its engaging query. An OR process sends back its engaging reply only if it has received all the replies from processes in its dependent set and has not been executing from the time it was engaged. These actions are called *collation*. In all other cases messages received are discarded.

To distinguish among many different concurrent deadlock computations, processes use the information in *seq*. Two messages $M(s, k)$ and $M'(s', k')$ received by some process in this order relate to the same deadlock computation if and only if $s \leq s'$, where $\leq$ means "is prefix of." This observation and the rule for sequence extension by AND processes enables us to identify tree computations with sequences, which plays an important role in the proof of the algorithm. To keep track of the queries sent and replies received by each process, two lists of messages are used: an incoming query list *IQ-list* and an outgoing query list *OQ-list*. Those lists are updated in a straightforward manner. Care has to be taken only in the case in which a grant is received. For details the reader is referred to Hermann and Chandy's paper [1983].

A deadlock computation is started by some controller, creating a process called the initiator; the initiator then sends a query to the process that is checked for deadlock. Several deadlock computations can be initiated concurrently and for the same process. The only constraints are that each time a new initiator be created and that the names of the initiators be unique.

Verification of the algorithm proceeds by proving the following claims:

(1) [Safety] If an initiator i detects deadlock for some process p, then p is truly deadlocked.

(2) [Progress] If a process p is deadlocked when a deadlock computation is initiated by some initiator i, then i will detect deadlock for p in finite time.

The proof employs invariants and the so-called "tree computation termination lemma" given below:

Lemma 4.2

A tree computation T terminates iff for every i and j such that query(T, i) *is sent to* p_j, *reply*(T, j) *arrives at* p_i *with no intervening grants.*

The details of the very well-written proof can be found in the paper by Hermann and Chandy [1983]. A performance analysis is not provided by the authors, but it is not hard to see that in the worst case one single deadlock computation will take at most $e = N^2(N - 1)$ queries and e replies, where N is the number of processes. Messages are of variable length with a maximum size of N. However, N can be exponential in the number of transactions if a normal form like DNF is used for the transaction-to-process mapping, as suggested in the paper. On the other hand, we do not see the necessity for converting an arbitrary AND–OR request into normal form: AND–OR requests can be mapped directly to a tree of processes without an exponential blowup. Hints for efficiency improvements and implementation considerations are again given in Hermann and Chandy's paper.

4.6 Bracha and Toueg's Algorithm for the $\binom{n}{k}$ Model

The algorithm for the $\binom{n}{k}$ model presented in this section [Bracha and Toueg 1983] is an application of the global state detection technique described in Section 3.4. We shall discuss in some detail only the first of the three versions of the algorithm given

by Bracha and Toueg, which assumes synchronous communication between processes and a static WFG. The second algorithm is supposed to relax the constraint of synchrony, but this is the case only to a very limited extent. The state of an edge (channel) incident on a process must still be known to that process in order for the algorithm to be correct. Hence the second algorithm is basically synchronous as well; it is just one more scheme to simulate synchrony by sending status messages back and forth, which is a fairly standard scheme and not new at all. The third algorithm first determines a global snapshot of the WFG by using the technique introduced in Chandy and Lamport [1985]. This snapshot can then be used to run one of the first two algorithms on to detect whether there is a deadlock.

The underlying resource model is the $\binom{n}{k}$ model. A transaction can have as a request an arbitrary *and–or* combination of $\binom{n}{k}$ requests. This combination is mapped to a tree of processes by using a scheme similar to the one discussed in the previous section, with each process having a single $\binom{n}{k}$ request.

A process becomes blocked upon issuing an $\binom{n}{k}$ request. It does so by sending out n request messages. It becomes executing again when it receives k grant messages. In this case it sends *relinquish* messages to the remaining $n - k$ processes, informing them that the edge created by sending the request no longer exists. Relinquish messages are necessary because each process must know both its set of outgoing edges *and* its set of incoming edges.

Deadlock in this model is defined in terms of the WFG, using the terminology of Section 3.4.

Definition 4.2

A process p is deadlocked in a WFG G if there is a schedule σ such that ($G \vdash^{\sigma} G'$ and p is executing in G').

The same problem as with the definition of Hermann and Chandy [1983] arises here, even though Bracha and Toueg seem to fail to recognize this. If there is no extension of the computation due to individual starvation or infinite loops, then the definition of deadlock is not quite correct.

Bracha and Toueg's first algorithm is a nested invocation of diffusing computations, with a slight twist. The twist is that a process leaving its neutral state remembers that it did so. Only the first query it ever receives will be engaging. All subsequent queries are answered immediately with a reply, even if the process has returned to its neutral state. One consequence of this behavior is that the number of messages exchanged during an invocation of the algorithm is reduced. Another consequence, however, is that many of the nice properties of diffusing computations are lost and the proofs of correctness become messy and almost incomprehensible.

A novel feature of the algorithm in comparison with the others we have seen so far is that a diffusing computation always terminates, and when it terminates, every process knows whether or not it is deadlocked. Every process p employs a local variable $free_p$, whose value upon termination satisfies $free_p \equiv p$ is not deadlocked in the static WFG. The value of *free* is established by simulating the propagation of grant messages through the WFG.

With these remarks, the algorithm can be described as a nesting of two instances of an algorithm similar to a diffusing computation, which Bracha and Toueg call CLOSURE. So the deadlock detection algorithm looks like the following:

- [Outer invocation of CLOSURE] Find the set S of all reachable executing processes by propagating queries, starting at some initiator i.
  - —Each $p \in S$ simulates granting all the resources it holds and the other processes are waiting for. A separate instance of CLOSURE is invoked by each p.
  - —The grants are propagated and the number g of simulated grants received at each reachable process q is compared with the number r of resources needed by q to become executing again. If $g < r$, then q will never get enough grants; hence it is deadlocked.

Observe that all the inner instances of CLOSURE will terminate before the outer one does. That the algorithm always terminates and that at termination each process knows whether or not it is deadlocked are proved by Bracha and Toueg [1983]. The proof uses purely operational arguments, and no invariants are given. The algorithm itself is stated in a Pascal-like language and uses global side effects, which make both the understanding of the algorithm and its proof unusually hard.

Moreover, the second version of the algorithm cannot be correct. Bracha and Toueg claim that "if an initiator i starts the deadlock detection algorithm in a colored WFG G then the algorithm terminates.[5] Moreover, when i terminates, the local variable $free_i \equiv true$ if and only if i is not deadlocked in G." Deadlock is defined as in Definition 4.2, with G containing black edges only. A counterexample to the above claim is a simple cycle of gray edges in G. These edges will turn black after a finite amount of time. Hence, there is no schedule σ such that ($G \vdash^{\sigma} G'$ and any of the processes in the cycle are active in G'). Therefore, by Definition 4.2, all the processes in the cycle are deadlocked. But since Bracha and Toueg chose to consider gray edges as nonexistent in the WFG, if the initiator is among the processes in the cycle, it will erroneously claim that it is not deadlocked. It is true, however, that this deadlock will be detected eventually. But this does not save the algorithm from not meeting its specification.

The performance analysis is summarized below. In the synchronous case at most $4N(N - 1)$ message are needed. In the asynchronous version more messages have to be exchanged to determine the state of the edges between processes. In the worst case an additional overhead of $O(N^2)$ messages suffices. Since the algorithm proceeds by taking a snapshot first and then running deadlock detection, the time between the occurrence of deadlock and its detection may be significant. Also, situations that will inevitably lead to deadlock, but cannot be detected yet, lead to further delays. A number of optimizations are suggested and can be found in Bracha and Toueg's paper [1983].

Compared with many other deadlock detection schemes, this algorithm employs very little concurrency. First, since snapshooting is used and the WFG is processed off-line, there is no true concurrency between deadlock detection and underlying computation. Second, the algorithm proceeds in phases (up to three, when space-saving optimizations are used), which sequentialize the deadlock computation.

To conclude the discussion of Bracha and Toueg's paper, we want to point out that a study of the algorithms revealed a number of errors. One of the more severe ones was the fact that the instantiation of the CLOSURE algorithm for the first phase of detecting all reachable executing nodes is incorrect. Recently, Gafni [1986] suggested improvements to Bracha and Toueg's algorithm but without giving a correctness proof.

5. DISCUSSION

The large number of errors in published algorithms addressing the problem of distributed deadlock detection [Bracha and Toueg 1983; Chandy and Misra 1982; Ho and Ramamoorthy 1982; Menasce and Muntz 1979; Obermarck 1982] shows that only rigorous proofs, using as little operational argumentation as possible, suffice to show the correctness of these algorithms. By falling back on well-known and completely general principles like diffusing computations and global state detection, it seems possible to achieve both elegance and correctness, even for more advanced models of resource requests, without introducing unnecessary complexity.

We believe that recent developments in the area of distributed deadlock detection algorithms has rendered much of the older work obsolete. This paper has focused on a small number of concepts and does not claim completeness in the sense that all known approaches have been covered. Among recent publications that have not been considered here are Chandy and Misra

[5] The notion of a colored WFG used here is similar to the one we introduced in Section 4.4.

[1986], Elmagarmid [1985], Hélary et al. [1987], Natarajan [1986], and Sinha and Natarajan [1984]. An annotated bibliography on distributed deadlock detection algorithms appeared in Zobel [1983].

The differences in message complexity of the algorithms presented have turned out to be less significant than expected. The one-resource, AND-, and OR-model algorithms all had a worst-case complexity of $O(N^2)$, where N was the number of nodes in the WFG. The AND–OR algorithm needed at most $O(N^3)$ messages but was vulnerable to an exponential blowup if $\binom{n}{k}$ requests are cast into AND–OR form. The algorithm for the $\binom{n}{k}$ model itself used $O(N^2)$ messages in the worst case, but this could only be achieved by performing the deadlock detection proper off-line from the database computations. There are no clear winners among the algorithms: Mitchell and Merritt's algorithm excels by its elegance and simplicity, Chandy and Misra's AND-model algorithm by its streamlined proof, and Hermann and Chandy's technique by the idea of applying diffusing computations to several levels of hierarchy. Each of the algorithms presented has its merits, but many of them achieve simplicity by severely restricting the forms of resource requests permitted. In selecting a deadlock detection scheme to be embedded into a particular application, it is therefore advisable—in order to avoid unnecessary complexity—to choose the least general technique that is still general enough to solve the problem at hand.

ACKNOWLEDGMENTS

We are grateful to Hank Korth for his help and encouragement and to Salvatore March and the referees for their comments and suggestion. The work was partially supported by Office of Naval Research contract N00014-86-K-0182.

REFERENCES

AWERBUCH, B., AND MICALI, S. 1986. Dynamic deadlock resolution protocols. In *Proceedings of the Foundations of Computer Science* (Toronto, Canada). IEEE, New York, pp. 196–207.

BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. 1987. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, Reading, Mass.

BRACHA, G., AND TOUEG, S. 1983. A distributed algorithm for generalized deadlock detection. Tech. Rep. TR 83-558, Cornell Univ., Ithaca, N.Y.

BRACHA, G., AND TOUEG, S. 1984. A distributed algorithm for generalized deadlock detection. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Vancouver, Canada, Aug.). ACM, New York, pp. 285–301.

CHANDY, K. M., AND LAMPORT, L. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Program. Lang. Syst. 3*, 1 (Feb.), 63–75.

CHANDY, K. M., AND MISRA, J. 1982. A distributed algorithm for detecting resource deadlocks in distributed systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Ottawa, Canada, Aug.). ACM, New York, pp. 157–164.

CHANDY, K. M., AND MISRA, J. 1986. An example of stepwise refinement of distributed programs: Quiescence detection. *ACM Trans. Program. Lang. Syst. 8*, 3 (July), 326–343.

CHANDY, K. M., MISRA, J., AND HAAS, L. M. 1983. Distributed deadlock detection. *ACM Trans. Comput. Syst. 1*, 2 (May), 144–156.

CHANG, E. 1982. Echo algorithms: Depth parallel operations on general graphs. *IEEE Trans. Softw. Eng. SE-8*, 4 (July), 391–401.

DIJKSTRA, E. W., AND SCHOLTEN, C. S. 1980. Termination detection for diffusing computations. *Inf. Process. Lett. 11*, 1 (Aug.).

DIJKSTRA, E. W., FEIJEN, W., AND VAN GASTEREN, A. J. M. 1983. Derivation of a termination detection algorithm for distributed computations. *Inf. Process. Lett. 16*, 5 (June), 217–219.

ELMAGARMID, A. K. 1985. Deadlock detection and resolution in distributed processing systems. Ph.D. dissertation, Dept. of Electrical Engineering, Ohio State Univ., Columbus, Ohio.

ELMAGARMID, A. K. 1986. A survey of distributed deadlock detection algorithms. *ACM SIGMOD Rec. 15*, 3 (Sept.).

GAFNI, E. 1986. Perspectives on distributed network protocols: A case for building blocks. In *IEEE Military Communications Conference* (Monterey, Calif.). IEEE, New York, pp. 1.1.1–1.1.5.

GIFFORD, D. G. 1979. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles* (Pacific Grove, Calif., Dec.). ACM, New York, pp. 150–163.

GLIGOR, V., AND SHATTUCK, S. 1980. On deadlock detection in distributed databases. *IEEE Trans. Softw. Eng. SE-6*, 5 (Sept.).

GRAY, J. N., HOMAN, P., KORTH, H. F., AND OBERMARCK, R. L. 1981. A straw man analysis of the probability of waiting and deadlock in a database system. Tech. Rep. RJ 3066, IBM Research Laboratory, San Jose, Calif.

HAAS, L. M. 1981. Two approaches to deadlock detection in distributed systems. Ph.D. dissertation, Dept. of Computer Sciences, Univ. of Texas, Austin, Tex.

HAAS, L. M., AND MOHAN, C. 1983. A distributed deadlock detection algorithm for a resource-based system. Res. Rep. RJ 3765, IBM Research Laboratory, San Jose, Calif.

HÉLARY, J., JARD, C., PLOUZEAU, N., AND RAYNAL, M. 1987. Detection of stable properties in distributed applications. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Vancouver, Canada, Aug.). ACM, New York, pp. 125–136.

HERMANN, T., AND CHANDY, K. M. 1983. A distributed procedure to detect AND/OR deadlock. Tech. Rep. TR LCS-8301, Dept. of Computer Sciences, Univ. of Texas, Austin, Tex.

HO, G. S., AND RAMAMOORTHY, C. V. 1982. Protocols for deadlock detection in distributed database systems. *IEEE Trans. Softw. Eng. SE-8*, 6 (Nov.), 554–557.

HOLT, R. C. 1972. Some deadlock properties on computer systems. *ACM Comput. Surv. 4*, 3 (Sept.), 179–196.

JAGANNATHAN, J. R., AND VASUDEVAN, R. 1982. A distributed deadlock detection and resolution scheme; performance study. In *Proceedings of the Third International Conference on Distributed Computing Systems* (Miami, Fla.). IEEE, New York, pp. 496–501.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in distributed systems. *Commun. ACM 21*, 7 (July), 558–565.

MENASCE, D., AND MUNTZ, R. 1979. Locking and deadlock detection in distributed databases. *IEEE Trans. Softw. Eng. SE-5*, 3 (May).

MISRA, J. 1983. Detecting termination of distributed computations using markers. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Montreal, Canada, Aug.). ACM, New York, pp. 290–294.

MISRA, J., AND CHANDY, K. M. 1982a. A distributed graph algorithm: Knot detection. *ACM Trans. Program. Lang. Syst. 4*, 4 (Oct.), 678–686.

MISRA, J., AND CHANDY, K. M. 1982b. Termination detection of diffusing computations in communicating sequential processes. *ACM Trans. Program. Lang. Syst. 4*, 1 (Jan.), 37–43.

MITCHELL, D. P., AND MERRITT, M. J. 1984. A distributed algorithm for deadlock detection and resolution. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. ACM, New York, pp. 282–284.

NATARAJAN, N. 1986. A distributed scheme for detecting communication deadlock. *IEEE Trans. Softw. Eng. SE-12*, 4 (Apr.), 531–537.

OBERMARCK, R. 1980. Deadlock detection for all resource classes. Res. Rep. RJ2955, IBM Research Laboratory, San Jose, Calif.

OBERMARCK, R. 1982. Distributed deadlock detection algorithm. *ACM Trans. Database Syst. 7*, 2 (June), 187–208.

PAPADIMITRIOU, C. 1987. *The Theory of Database Concurrency Control*. Computer Science Press, Rockville, Md.

RÄUCHLE, T., AND TOUEG, S. 1983. Exposure to deadlock for communicating processes is hard to detect. Tech. Rep. TR 83-555, Cornell Univ., Ithaca, N.Y.

SINHA, M. K., AND NATARAJAN, N. 1984. A distributed deadlock detection algorithm based on timestamps. In *Proceedings of the 4th International Conference on Distributed Computing Systems*. IEEE, New York, pp. 546–556.

ZOBEL, D. 1983. The deadlock problem: A classifying bibliography. *Operat. Syst. Rev. 17*, 2 (Oct.), 6–15.

BIBLIOGRAPHY

BADAL, D. Z., AND GEHL, M. T. 1983. On deadlock detection in distributed computing systems. In *IEEE INFOCOM*. IEEE, New York.

BRACHA, G. 1985. Randomized agreement protocols and distributed deadlock detection algorithms. Ph.D. dissertation, Cornell Univ., Ithaca, N.Y.

COHEN, S., AND LEHMANN, D. 1982. Dynamic systems and their distributed termination. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Ottawa, Canada, Aug.). ACM, New York, pp. 29–33.

DIJKSTRA, E. W. 1982. Distributed termination detection revisited. EWD 828, Plataanstraat 5, 5671 Al Nuenen, The Netherlands.

FRANCEZ, N. 1980. Distributed termination. *ACM Trans. Program. Lang. Syst. 2*, 1 (Jan.), 42–55.

FRANCEZ, N., AND RODEH, M. 1982. Achieving distributed termination without freezing. *IEEE Trans. Softw. Eng. SE-8*, 3 (May), 287–292.

FRANCEZ, N., RODEH, M., AND SINTZOFF, M. 1981. Distributed termination with interval assertions. In *Proceedings of Formalization of Programming Concepts* (Peninsula, Spain). Springer Verlag, New York.

GOLDMAN, B. 1985. Deadlock detection in computer networks. Tech. Rep. LCS TR-185, Massachusetts Institute of Technology, Cambridge, Mass.

GOUDA, M. 1981. Distributed state exploration for protocol validation. Tech. Rep. TR-185, Dept. of Computer Sciences, Univ. of Texas, Austin, Tex.

ISLOOR, S. S., AND MARSLAND, T. A. 1980. The deadlock problem: An overview. *Computer* (Sept.), 58–70.

JAGANNATHAN, J. R., AND VASUDEVAN, R. 1982. Detection and resolution of deadlocks in distributed systems. Tech. Rep. 82-108-27, Univ. of Calgary, Calgary, Alta., Canada.

KORTH, H. F., KRISHNAMURTHY, R., NIGAM, A., AND ROBINSON, J. T. 1983. A framework for understanding distributed (deadlock detection) algorithms. In *Proceedings of the Second ACM Symposium on Principles of Database Systems* (Atlanta, Ga., Mar.). ACM, New York, pp. 192–201.

MARSLAND, T. A., AND ISLOOR, S. S. 1980. Detection of deadlocks in distributed database systems. *INFOR 18*, 1 (Feb.), 1–20.

TSAI, W. 1982. Distributed deadlock detection in distributed database systems. Ph.D. dissertation, Univ. of Illinois at Urbana-Champaign, Urbana, Ill.

TSAI, W., AND BELFORD, G. 1982. Detecting deadlock in distributed systems. In *IEEE INFOCOM*. IEEE, New York, pp. 89–95.

WUU, G. T., AND BERNSTEIN, A. J. 1985. False deadlock detection in distributed systems. *IEEE Trans. Softw. Eng. SE-11*, 8 (Aug.), 820–821.

Received May 1987; final revision accepted January 1988.