SURVEY OF DEADLOCK DETECTION IN DISTRIBUTED CONCURRENT PROGRAMMING ENVIRONMENTS AND ITS APPLICATION TO REAL-TIME SYSTEMS

Chia-Shiang Shih and John A. Stankovic

COINS Technical Report 91-43
Replaces COINS TR 90-69
May 1991

Survey of Deadlock Detection in Distributed Concurrent Programming Environments and Its Application to Real-Time Systems and Ada*

Invited Paper

Chia-Shiang Shih
ECE Department, University of Massachusetts
Amherst, MA 01003
shih@ecs.umass.edu

John A. Stankovic
COINS Department, University of Massachusetts
Amherst, MA 01003
stankovic@cs.umass.edu

Abstract

Deadlock is one of the most serious problems in multitasking concurrent programming systems. The deadlock problem becomes further complicated when the underlying system is distributed and when tasks have timing constraints. Distributed deadlock detection has been studied to some extent in distributed database systems and distributed timesharing operating systems but has not been widely used in real-time systems. In this paper, we investigate deadlock detection algorithms in distributed environments and extend the results to real-time systems by considering timing constraints in the algorithms. In particular, we direct our attention to Ada environment and try to apply our solutions to it. Related problems, such as livelocks, orphan tasks, task termination problems, and global state detection, are considered when it is appropriate.

This paper has two main parts. First, we complete a state-of-the-art survey of the distributed deadlock detection algorithms proposed in the literature. The survey work includes: (1) graph theory, (2) distributed concurrent programming systems, (3) deadlock models, and (4) distributed deadlock detection and resolution algorithms. Second, we extend the deadlock problems into real-time systems and develop solutions for them. In particular, the solutions are developed with the applications to the Ada environment in mind. Also, we analyze and categorize the deadlock problem in Ada environments into four levels of complexity by using Knapp's hierarchy of deadlock models. To fully support Ada semantics it is necessary to develop solutions for the most complex level. Many Ada applications, however, do not utilize all the features that Ada provides. Consequently, according to the characteristics of an application, the deadlock problem may be simplified by imposing certain restrictions on the use of Ada. We develop a series of solutions depending on the level of restriction imposed on the use of Ada and we relate those solutions to the levels of complexity associated with the theoretical models. Two algorithms related to the first two levels of complexity are presented in this paper.

*This work was supported, in part, by the Charles Stark Draper Laboratory, Inc., and by NSF under grants IRI-8908693 and CDA-8922572.

1 Introduction

Deadlock is one of the most serious problems in multitasking concurrent programming systems. As early as in the 60's the deadlock problem was recognized and analyzed (Dijkstra[17] described it as the *problem of the deadly embrace*). Deadlock occurs when one or more tasks in a system are blocked by each other forever and their requirements can never be satisfied. A deadlock situation may arise if and only if the following four resource competition conditions hold in a system simultaneously: (1) mutual exclusion, (2) hold and wait, (3) no preemption, and (4) circular wait. To some degree the last condition implies the other three. However, it is quite useful to consider each condition separately in analyzing and designing a deadlock free system.

Principally, there are three strategies for dealing with the deadlock problem:

1. Deadlock Prevention – by ensuring that at lease one of the deadlock conditions cannot hold,
2. Deadlock Avoidance – by providing *a priori* information so that the system can predict and avoid deadlock situations, and
3. Deadlock Detection – by detecting and recovering from deadlock states.

The first two strategies ensure that the system will *never* enter a deadlock state. Deadlock prevention is commonly achieved either by guaranteeing that tasks do not have to hold and wait on resources (e.g., by forcing all tasks to acquire resources a priori), or by allowing preemption (e.g., a task that holds the needed resource might be preempted by another task with a higher priority). For deadlock avoidance, a task proceeds if the resulting global state is checked and proved to be safe from deadlock. These methods carry the following drawbacks[57]:

- They are usually inefficient when applied in complex distributed systems. For deadlock prevention, it is inefficient because it decreases system concurrency by restricting the execution of the tasks to avoid at least one of the deadlock conditions. For deadlock avoidance, checking for a safe state is computationally expensive and inefficient. This inefficiency is especially significant in a complex distributed system due to the large numbers of tasks and resources.
- They are apt to fail in complex distributed systems. For example, if the tasks are required to acquire resources a priori, a group of tasks may get deadlocked in the resource-acquiring phase due to lack of a perfect global synchronization mechanism. Similarly, in the deadlock

avoidance case, due to inconsistent local views caused by the imperfect synchronization mechanism, different sites may all find the states safe and grant the requests concurrently, but the final global state may turn out to be deadlocked.

- The requirements for deadlock prevention or avoidance may not be fulfilled. For instance, in many systems future resource requests are unpredictable which makes "a priori resource acquiring" deadlock prevention impossible.

Alternatively, by applying the third strategy, the system is allowed to enter a deadlock state and then it is detected and recovered from. The detection of deadlocks requires the examination of the system state (principally, the task/resource interactions) for the presence of cyclic waits. Once a deadlock is formed, it persists until it is detected and broken (the so called *stable property* of the deadlock problem). The deadlock detection computation can be performed in parallel with the other normal system activities, therefore, it may not have a serious impact on the system performance. Also, since certain deadlock detection algorithms can be embedded in the underlying operating system, they are able to extend the fault tolerance of software design faults even if a deadlock prevention or avoidance approach is used in the user application.

Yet another potential benefit of the "detection" strategy for deadlocks is that it may be integrated with other related problems. For example, many problems appear in multitasking systems, such as livelock (a.k.a. effective deadlock or starvation), task termination, and orphan tasks, which must be detected dynamically at runtime. Some of these problems, e.g. task termination and orphan task problems, carry the the same stable property as the deadlock problem. The detection of these system faulty states requires examination of task/resource interactions which is similar to certain techniques used in deadlock detection. Certain deadlock detection algorithms, therefore, can be tailored for the detection of these problems and vice versa, without too much additional effort and overhead.

Considering distributed deadlock detection as part of *global state detection* is another situation where deadlock detection may be resolved with related problems. For example, if a global state detection algorithm is adopted to facilitate applications such as distributed debugging and distributed system monitoring, it can be extended for detection of distributed deadlocks with little overhead added.

In real-time systems, deadlock prevention and avoidance methods have received most of the attention and are the current "best" strategies. However, because of the drawbacks pointed out above these strategies might work successfully in relatively simple systems, but may be inefficient and

very difficult to design and verify in more complex systems such as multiprocessors or distributed systems. Distributed deadlock detection, which is the focus of this research, has been studied to some extent in distributed database systems and distributed timesharing operating systems but has not been widely used in real-time systems. In the rest of this paper, we will first survey the related research work in conventional non-real-time environments, and then propose extensions for distributed real-time systems.

The paper is organized as follows. Section 2 briefly summarizes a few terms and results from graph theory. Section 3 presents the underlying system model. In Section 4, we discuss several deadlock models. These deadlock models combine with the system model to serve both as a framework in the survey and as a basis for the analysis of the distributed deadlock detection and resolution algorithms which are given in Section 5. Section 6 describes some new concerns regarding deadlock in real-time systems, and shows how the deadlock problem can be divided into four levels of complexity. We also indicate that to fully support Ada semantics it is necessary to develop solutions for the most complex level. Many Ada applications do not utilize all the features that Ada provides. Consequently, according to the characteristics of an application, the deadlock problem may be simplified by imposing certain restrictions on the use of Ada. In Section 7 we present two algorithms and relate those algorithms to the levels of complexity associated with the theoretical models and the restrictions imposed on Ada. Section 8 summarizes the paper. Throughout the discussions, examples are given in the Ada programming language.

2 The Concepts from Graph Theory

A wait-for graph is a mathematical tool which has been used to model the system state in describing deadlock related problems. Depending on the complexity of the underlying model, several different types of wait-for graphs are used in the literature. We will discuss them in this section after a summary of some concepts from graph theory.

Some related definitions of graph theory are summarized based on [33]:

- A *directed graph* (digraph) is a pair (V, E), where V is a nonempty set of *vertices* and E is a set of *edges*. Each edge in E is an ordered pair (a, b), where a and b are vertices in V. Also, the notation "$a \rightarrow b$" may be used to represent an edge.

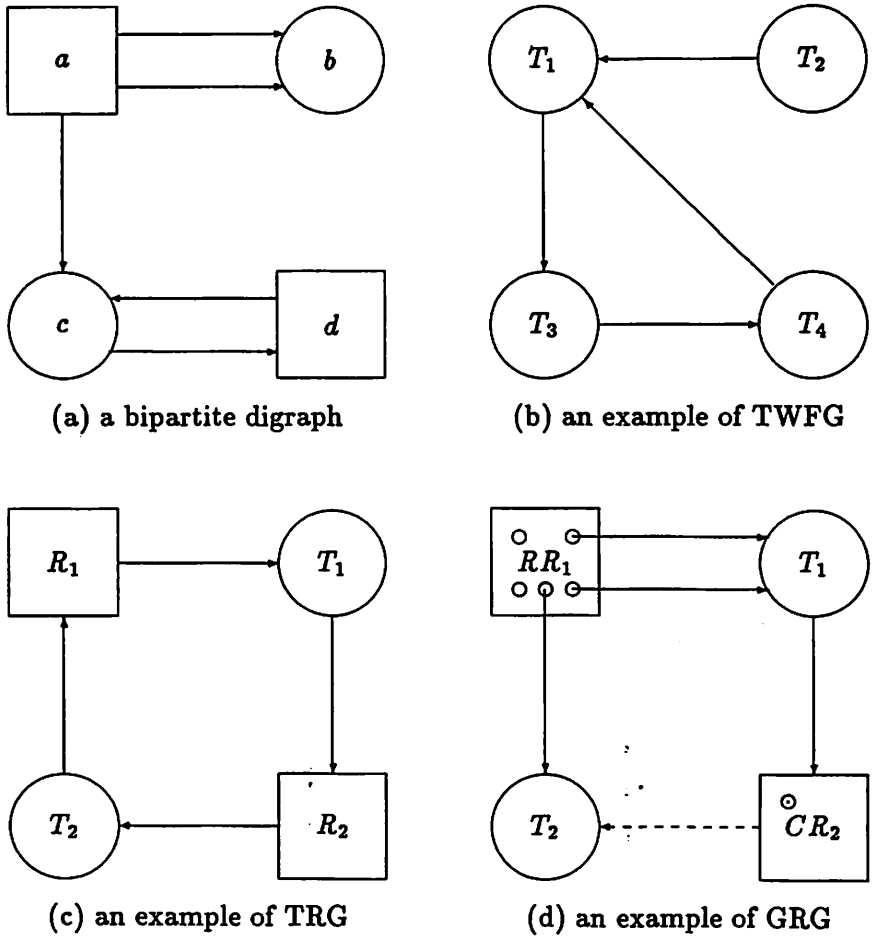
- A *bipartite graph* is one in which all the vertices in V are partitioned into two disjoint subsets such that there are no edges connecting vertices from the same subset.

- A vertex i is an *isolated vertex* if it has no incoming edges (a, i) and has no outgoing edges (i, b).
- A vertex s is a *sink* if it has no outgoing edges (s, b).
- A *path* is a sequence $(a, b, c, ..., y, z)$ of at least two vertices, where (a, b), (b, c), ..., (y, z) are edges. Also, the notation "$a \rightarrow b \rightarrow ... \rightarrow y \rightarrow z$" may be used to visualize a path.
- A *cycle* is a path with the same *start* and *end* vertex.
- Two cycles C_1 and C_2 are *nested* if $C_1 \cap C_2 \neq \emptyset$ and $C_1 \neq C_2$.
- The *reachable set* of a vertex v is the set of all vertices b such that there is a path directed from v to b. The reachable set of v may contain v itself if there is a cycle.
- A *knot* K is a nonempty set of vertices such that the reachable set of each vertex v in K is exactly the knot K.

Figure 1-(a) illustrates a bipartite digraph with vertices $V = \{a, b, c, d\}$ partitioned into two subsets $\{a, d\}$ and $\{b, c\}$, and with edges $E = \{(a, b), (a, b), (a, c), (c, d), (d, c)\}$ connecting the two subsets of vertices. Vertex b is a sink, path "$c \rightarrow d \rightarrow c$" is a cycle, and set $\{c, d\}$ is a knot.

In Section 4 we will see that in the simple deadlock models, such as the Single-Resource model and the AND model, a system is in deadlock state if and only if there are cycles in the system state digraph. In the OR model, the existence of a knot in a system state graph is a necessary and sufficient condition for deadlock. However, in more complex models, such as the AND-OR model and the C(n,k) model, there is no simple construct of graph theory to describe the condition of deadlock. Many deadlock detection algorithms are designed for simple models, therefore, their major functions are detecting cycles or knots. The following results relating to cycles and knots are listed in [33, 43] and are very helpful in understanding these deadlock detection algorithms:

- If there is a knot in a digraph, then there is a cycle in it.
- There is no sink in a knot.
- There is no path from a vertex in a knot to a sink.
- A digraph is free of knots if and only if, for each vertex v, v is a sink or there is a path from v to a sink.



(a) a bipartite digraph

(b) an example of TWFG

(c) an example of TRG

(d) an example of GRG

Figure 1: Examples of the system state graph

The state of a system is in general dynamic; that is, tasks (or processes) continuously acquire and release resources and communicate with each other. Characterization of deadlocks requires a representation of the system state in terms of task-resource and task-task interactions. Depending the complexity of the model, a system state can be depicted by one of three types of graph:

TWFG: The TWFG (Task Wait-For Graph) is the simplest graph among three types of graph. Only the task-task wait-for relations are depicted in the TWFG. A TWFG is a digraph in which vertices represent tasks. A directed edge (T_1, T_2) represents that a task T_1 is waiting for another task T_2 either due to a resource conflict or due to a synchronized communication attempt between them. In the database literature, a TWFG is referred to as a Transaction-Wait-For Graph in which vertices are transactions. A transaction can be viewed as a task that performs a sequence of database operations. In general, depending on the type of the underlying system, either tasks or transactions can represent the unit of computation in deadlock problems. The terms task and transaction are used interchangeably hereafter.

Figure 1-(b) is an example of TWFG with four tasks T_1, T_2, T_3, and T_4. Path $T_1 \rightarrow T_3 \rightarrow T_4 \rightarrow T_1$ is a cycle, and tasks T_1, T_2, T_3, and T_4 are deadlocked.

TRG: A TRG (Task-Resource Graph or, as referred to in the database literature, Transaction-Resource Graph) is a bipartite digraph in which vertices are partitioned into two subsets: a subset of all the task vertices in the system $T = \{T_1, T_2, ..., T_m\}$ and a subset of all the resource vertices in the system $R = \{R_1, R_2, ..., R_n\}$. Edges of a TRG depict assignments or pending resource requests. A pending request is represented by a *request edge* (T_i, R_j) directed from a requesting task T_i to the requested resource R_j. A resource assignment is represented by an *assignment edge* (R_k, T_l) directed from an assigned resource R_k to its holder task T_l.

Figure 1-(c) is an example of TRG with two tasks T_1 and T_2, and two resources R_1 and R_2. Notice that resources are depicted by squares "□" to distinguish resources from tasks which appear as circles "○". The edges $R_1 \rightarrow T_1$ and $R_2 \rightarrow T_2$ are assignment edges and edges $T_2 \rightarrow R_1$ and $T_1 \rightarrow R_2$ are request edges. Path $T_1 \rightarrow R_2 \rightarrow T_2 \rightarrow R_1 \rightarrow T_1$ is a cycle; T_1 and T_2 are deadlocked.

GRG: A GRG (General Resource Graph[33]) is a generalized TRG which is used to describe Holt's *General Resource System*[33]. The resource vertices in a GRG are partitioned into two disjoint subsets: a *reusable* resource subset $RR = \{RR_1, RR_2, ..., RR_x\}$ and a *consumable* resource subset $CR = \{CR_1, CR_2, ..., CR_y\}$. For each reusable resource RR_i, its *total units* is a strictly positive integer rt_i. For each consumable resource, its *producers* are a nonempty subset of the task set T. Edges are of three types:

Request edge: A request edge (T_i, RR_j) or (T_i, CR_k) is directed from a requesting task T_i to the requested resource RR_j or CR_k. It represents a pending request.

Assignment edge: An assignment edge (RR_i, T_j) is directed from a reusable resource RR_i to its assigned holder T_j. The total number of assignment edges directed from RR_i can not exceed rt_i.

Producer edge: A producer edge (CR_i, T_j) is directed from a consumable resource CR_i to one of its producers T_j. Edge (CR_i, T_j) exists if and only if T_j produces CR_i.

Figure 1-(d) is an example of GRG with two tasks T_1 and T_2, and two resources RR_1 and CR_2. Task T_1 holds two units of reusable resource RR_1 and requests one unit of consumable resource CR_2. Task T_2 also holds one unit of RR_1, and is the only producer of CR_2. The total inventory of RR_1 is $rt_1 = 5$. Notice that the graph is bipartite. Reusable resources are permanent and are depicted by small circles "○". Consumable resources are temporary and are depicted by dotted circles "⊙". Request and assignment edges may appear and disappear with state changes; they are depicted by solid arrows. Producer edges are permanent and are depicted by dashed arrows.

3 Models of Distributed Concurrent Programming Systems

Deadlock can be formally studied in isolation by using graph theory. However, in this work we are interested in explicitly tying the formal properties of deadlock algorithms directly to the actual languages and systems that need to use the theory. We believe that there is an important gap that exists between the theory and its application that has not been addressed very well to date. In particular, we are interested in Ada and its run time system. To bridge this gap we must understand Ada's concurrency, synchronization, and resource allocation models and show how they relate to the theory. Here we identify Ada's concurrency model, but since it is based on notions of Hoare's "Communicating Sequential Processes" (CSP)[32], and Brinch Hansen's "Distributed Processes" (DP)[7], we first say a few words about CSP and DP.

Hoare's CSP recognized that synchronous input and output can be the basic primitives of concurrent programming. In CSP, tasks synchronize and communicate by means of *input* and *output* commands. An input/output command is delayed in one task until a matching output/input command is executed by another task. Messages are transferred using input and output commands, therefore, CSP is based on message-passing synchronization requiring no shared memory (variables). Messages are not buffered and their one-to-one transfers are synchronously performed.

The notation

$$[G_1 \rightarrow CL_1 \ \square \ G_2 \rightarrow CL_2 \ \square \ \cdots \ \square \ G_n \rightarrow CL_n]$$

defines an *alternative* command, where $G_i \rightarrow CL_i$ is a *guarded* command with the guard G_i and its corresponding command list CL_i. The alternative command is executed by evaluating each guard G_i for *success* or *failure*. One of the command lists associated with a successful guard is chosen nondeterministically if more than one guard succeeds. If a guard contains an input/output command, it succeeds only when the preceeding elements of the guard indicate success and the corresponding output/input command is executed.

The alternative command given above specifies the execution of exactly one of its constituent guarded command. An error results if no guard succeeds. On the other hand, a *repetitive* command

$$*[G_1 \rightarrow CL_1 \ \square \ G_2 \rightarrow CL_2 \ \square \ \cdots \ \square \ G_n \rightarrow CL_n]$$

specifies as many iterations as possible of its constituent alternative command. In a repetitive command the failure of all guards terminates the repetition instead of resulting in an error.

Another mechanism based on synchronous message passing is Brinch Hansen's DP. In DP, a task performs two kind of *operations*: the *initial statement* defined within itself and the *external requests* made by other tasks. A task starts from executing its initial statement until the statement either terminates or waits for a condition to become true. Meanwhile, another pending operation (external request), if any, is started and continues until it either terminates or waits for a condition to become true. Therefore, this *interleaving* of the initial statement and the external requests is controlled by the program instead of the system clock interruption at the machine level.

The *common procedure* is the only form of task communication in DP. A task may call common procedures defined within itself or within other tasks. These procedures are the external requests to the called task. A task T_1 calls a procedure OP defined within another task T_2 by:

call $T_2.OP$(*expressions, variables*)

Before the procedure OP is executed, the *expressions* are evaluated and their values are assigned to the input parameters of OP. When the OP completes, the values of its output parameters are assigned to the *variables* of the call. Parameter passing between tasks may be implemented either by shared memory or by one-to-one message passing without shared memory. In addition to CSP's

guarded command, a *guarded region* in DP enables a task to wait until a choice among several guarded statements can be made.

Ada's concurrent programming mechanisms are generalized from many aspects of CSP and DP. A *task* is the unit of computation in Ada environments. A task is a program module that is executed asynchronously. Tasks may communicate and synchronize their actions through:

- the *entry calls* and *accept statements*, which are a combination of procedure calls and message transfer, and
- the *select statement*, which is a non-deterministic control structure similar to the alternative guarded command in CSP and DP.

Entry declarations and calls are syntactically similar to procedure declarations and calls. Entry declarations can occur only in the specification of a task. The corresponding **accept** statements are given in the body of the task, which have the following form[1]:

accept <*entry−name*> [*parameters*]
[**do** {*statement*} **end**];

The {*statement*} part of an **accept** statement can be executed only if another task invokes the <*entry−name*>. Invoking an <*entry−name*> (an entry call) is syntactically the same as a procedure call in DP. First, parameters are passed before the execution of the {*statements*}. After the execution reaches the **end** statement, parameters may be passed back. Both tasks are free to continue from this point. The **accept** statement and the corresponding entry call are executed synchronously similar to the input and output commands in CSP. This synchronization performed between two tasks is the Ada's *rendezvous* concept. Thus the entry call and accept statement serve both as a communication mechanism and a synchronization tool.

Choices among several entry calls is accomplished by the **select** statement, which is similar to the guarded region in DP. There are three kinds of select statements: the *selective wait*, the *conditional entry call*, and the *timed entry call*:

[1]Square brackets [ ] denote an optional part, while braces { } denote a repetition of zero or more times.

The *selective wait* statement allows a task to **accept** an entry call from more than one task non-deterministically. A *selective wait* statement has the form

select
    select−alternative
{**or**
    select−alternative}
[**else**
    {*statement*}]
end select;

in which *select−alternative* is of the form

[**when** <*boolean−expression*> ⇒] *selective−wait−alternative*

and a *selective−wait−alternative* can be one of

accept−statement [{*statement*}]
| *delay−statement* [{*statement*}]
| **terminate;**

A *select−alternative* statement is said to be *open* if there is no prefixed guard (**when** clause) before it or if the <*boolean−expressions*> in the prefixed guard is true; otherwise, it is said to be *closed.* A *selective wait* statement can have at most one **terminate** alternative. The *delay−statement* and **terminate** alternatives cannot coexist in a *selective wait* statement. The **else** part is not allowed when either a *delay−statement* or a **terminate** alternative is present.

According to the *Ada Reference Manual*[40] the following rules define the execution of a *select−alternative* statement:

1. Determine all the open alternatives and start counting time for the open *delay−statements* (if any).
2. If there are open alternatives that can be selected, the execution follows the steps:
    (a) An open *accept−statement* alternative may be selected for execution only if a corresponding rendezvous is possible. The subsequent statements, if any, are then executed.
    (b) The subsequent statements following an open *delay−statement* will be selected for execution if no other alternative is selected before the specified delay duration has elapsed.

(c) A **terminate** alternative may be selected if all the sibling tasks and their dependent tasks which belong to the same root creator have terminated or are waiting at a **terminate** alternative. A task terminates if it reaches the end of its code sequence or if a **terminate** alternative is selected. The termination of a task is subject to the condition that there are no calls pending to any entry of the task.

3. If the **else** part is present, it is executed under the condition that no open alternative can be selected immediately or all alternatives are closed. If no **else** part is present and open alternatives exist, the execution is delayed until an open alternative can be selected. If no **else** part is present and all alternatives are closed, an error exception is raised.

When attempting to perform an immediate rendezvous, a *conditional entry call* is used. A *conditional entry call* has the form

```
select
      entry-call [{statement}]
else
      [{statement}]
end select;
```

If an immediate rendezvous is possible, then rendezvous takes place and the subsequent statements following the *entry-call* are executed; otherwise, the alternative sequence of statements specified in the **else** alternative is executed.

When attempting to establish a rendezvous within some specified time period, a *timed entry call* is used. A *timed entry call* has the form

```
select
      entry-call [{statement}]
or
      delay-statement [{statement}]
end select;
```

If a rendezvous can be established within the specified period then rendezvous takes place and the statements following the *entry-call* are executed; otherwise, the statements following the specified *delay-statement* are executed.

As in most of the concurrent programming environments, deadlocks may occur due to tasks that are competing for shared resources in Ada's environment both at the underlying system level, and

at the language level. For example, consider a program with two tasks T_1 and T_2 executing on a system with two disk drives. Each of T_1 and T_2 needs both disk drives together, say for copying a file from one disk to the other. Deadlock will occur if each task is holding the permission to use one disk drive and is waiting for the permission to use the other drive. Deadlocks may also occur due to tasks that are waiting for each other in Ada's rendezvous. For example, two tasks T_1 and T_2 each want to call the other task before accepting an entry call from the other. Portions of the code of tasks T_1 and T_2 are illustrated as follows:

```
task body T1 is
⋮
begin
      ⋮
      T2.READ(···);
      accept READ ··· end READ;
      ⋮
end T1;
```

and

```
task body T2 is
⋮
begin
      ⋮
      T1.READ(···);
      accept READ ··· end READ;
      ⋮
end T2;
```

Tasks T_1 and T_2 are deadlocked at the entry calls $T_2.READ$ and $T_1.READ$, respectively. A more detailed discussion of deadlocks will be given in the Section 4.

Some aspects of real-time processing is supported by the select statement. The else alternative and the delay statement in the *selective wait* both provide ready escapes in the event that no open alternatives exist or that open alternatives are unduly delayed in their selection. Using the *conditional* or *timed* entry calls, the calling task can ensure that it will not be blocked due to the inability of the called task to complete a rendezvous. However, as discussed in Section 6.1, *temporal* deadlock is still a problem in such a real-time system.

Livelock occurs when interacting tasks cannot finish their work in a limited period of time. For example, if a task T_1/T_2 is unable to rendezvous immediately with another task T_2/T_1, it performs some secondary activity, so that it does not waste time being blocked for a rendezvous, and then tries to rendezvous again. Livelock may occur if every time T_1/T_2 attempts to rendezvous with T_2/T_1, T_2/T_1 is performing some secondary activity and is not ready for a rendezvous. Eventually T_1 and T_2 will miss their deadlines.

Also, task termination and orphan task problems may arise when using Ada's concurrent programming facilities in distributed environments:

- *Task termination* – In Ada, the termination of tasks, whether it is normal termination or abnormal termination, is well defined not to affect other executing tasks. It is not difficult to realize a task termination mechanism correctly in a single site system. However, task termination becomes complex and difficult when implemented in distributed environments due to intersite task interaction.
- *Orphan task* – In a distributed system, a task may create several subtasks at different sites. Due to site failure or network partitioning, a subtask might become an orphan which has lost connection to its parent task. Similarly, if site failure or network partitioning takes place when tasks from different sites are in a rendezvous, the tasks waiting for the rendezvous might become orphans.

The *stable* property of task termination and orphan tasks are similar to that of deadlock problem. These problems are all caused by task interaction and once they occur will remain until they are detected and resolved. Therefore, techniques such as *diffusing computations* and *global state detection* discussed in Section 5, can be extended and applied to solve these problems.

4 Deadlock Models

Depending on the particular deadlock situation sufficient conditions for detecting deadlock vary. Modelling deadlocks is a way of understanding the properties of deadlocks as well as a way of finding methods to detect and resolve the deadlocks. In this section we discuss three deadlock model approaches: (1) the resources/communication deadlock model[8] used in most algorithms, (2) the general resource system model presented by R. C. Holt[33], and (3) a hierarchy of deadlock models presented by E. Knapp[39]. They are discussed in the following subsections.

4.1 Resource and Communication Deadlock Models

Many deadlock detection and resolution algorithms are used either in solving "resource" deadlocks [3, 8, 10, 13, 15, 22, 21, 25, 27, 28, 31, 38, 42, 44, 45, 48, 51, 53, 55, 58, 60], or used in communication based systems [8, 29, 47, 50], or both [6, 30]. However, the distinction between these two models is not always clear since the messages could be treated as implied resources and may not be distinguished from physical or data resources.

In resource deadlocks, tasks access resources (for example, data items in database systems or memory buffers in operating systems). A task acquires a resource before accessing it and relinquishes it after using it. The accessed resource is held by the accessing tasks and edges are, therefore, formed in the TRG from the resource toward the accessing tasks. A task that requires resources which are not available (held by other tasks or not yet created) can not proceed until the requests are satisfied. The requesting task is waiting for the availability of the requested resources and, therefore, edges are formed in the TRG from the task toward the requested resources. A set of tasks is resource-deadlocked if and only if each task in the set requests at least one resource held by another task in the set.

In communication deadlocks, messages, tokens, or "synchronization between tasks" are the implied resources for which tasks wait. In Ada environments, tasks communicate via synchronization and communication points called *entries*. A synchronization or communication may involve two or more tasks. The waiting tasks can proceed only after all the tasks involved in the synchronization or communication are ready to synchronize or exchange information (i.e., the *rendezvous* concept). An edge is formed in the TWFG from a waiting task toward its waited task. Each idle task has a set of tasks for which it is waiting, called its *dependent set*. A nonempty set of tasks S is communication-deadlocked[8] if and only if (1) all tasks in S are idle, (2) the dependent set of every task in S is a nonempty subset of S, and (3) there are no messages, tokens, or synchronization in transit between tasks in S.

There are several differences between the resource model and the communication model. One major difference is that different request patterns are assumed in the two models. In the resource model, a blocked task cannot proceed until it is granted *all* the requested resources. A cycle detection algorithm is usually used in the resource deadlock models. On the other hand, in a communication environment, such as CSP, DP, or Ada, a task's communication request may be satisfied by *at least one* of its corresponding tasks. For instance, in the Ada environment, the accept statement requires one of many potential calling tasks to be in rendezvous with the accepting task.

A cycle is no longer a sufficient condition for communication deadlocks. Knot detection algorithms are usually used for the communication deadlock detection. A knot detection algorithm is more complicated than one which detects cycles. More detailed discussions are given in Section 4.3.

A second difference is that the agent of deadlock detection in the two environments is usually not the same. In the resource model, the wait-for dependency among tasks may not be known by the tasks directly. A "controller" (a "controller" could be a separate task or a shared data structure plus program code) at each site keeps track of its resources, and only the controllers can deduce that one task is waiting for another. On the other hand, in the early communication models, it is assumed that a task always can know the identity of the tasks it is waiting for. Thus, in the communication model the tasks have the necessary information to perform deadlock detection if they act collectively.

4.2 General Resource System Model

A generalized resource system model was proposed by R. C. Holt[33] which unifies the resource and communication deadlock models (see Section 4.1) into a *general resource system* (GRS) model. In Holt's GRS model, the term "resource" is used in a special sense to mean any object which may cause a task to become blocked. A resource is either reusable or consumable. "Reusable resources" are used to model competition for objects such as shared data and memory buffers. "Consumable resources" are used to model explicit interactions among tasks such as synchronization or exchange of signals or messages among tasks.

Both types of resources consist of a number of identical units which can be *requested* by tasks. The total number of units of a reusable resource is fixed, but it is unlimited for any of the consumable resources. A task requesting units is blocked until enough units are available to satisfy its request; then the task can *acquire* the requested unit. A task can *release* units only when it is not idle (waiting). The fundamental difference between reusable and consumable resources is that the units of a reusable resource are never created or destroyed, but only transferred (requested and acquired) from a pool of available units to a task and then transferred back (released) to the pool. On the contrary, units of a consumable resource are created ("produced," or released) and destroyed ("consumed," or requested and acquired).

The GRS model merges the communication deadlock model and the resource deadlock model by distinguishing the set of resources into two disjoint subsets, a set of reusable resources (resource deadlock model) and a set of consumable resources (communication deadlock model). A generalized

TRG called the *general resource graph* (see GRG in Section 2) is used to depict the system state. A GRG is a TRG with each resource vertex explicitly labeled with a number of reusable/consumable resource units.

Holt suggested a *graph reduction* technique[33] to test if tasks are deadlocked. A GRG can be *reduced* by any task vertex T_i which is neither isolated nor blocked in the following manner:

- For each reusable resource vertex RR_j, delete all request edges (T_i, RR_j) and assignment edges (RR_j, T_i). Increment its total inventory units rt_j by the number of deleted assignment edges.
- For each consumable resource vertex CR_j, delete all request edges (T_i, CR_j) and producer edges (CR_j, T_i). Decrement its total inventory units ct_j by the number of deleted request edges (T_i, CR_j) . Set $ct_j = \infty$, if there is a producer edge (CR_j, T_i) deleted.

A reduction of a GRG by a task vertex T_i may lead to unblocking of another task T_j, therefore, the task T_j may be chosen as a candidate for the next reduction. A GRG is said to be *completely reducible* if there exists a sequence of graph reductions that deletes all edges in the GRG. A task T_i is not deadlocked in state S if and only if there exists a sequence of reductions in the corresponding GRG that leaves T_i unblocked. If a GRG is completely reducible, then the state it represents is not deadlocked.

Many systems use an "expedient" resource allocation strategy in which all requests for available units are granted immediately. In such systems, a new allocation of resources can take place only immediately following a resource request or a release. Holt[33] has proven that in a GRG, (1) a *cycle* is a necessary condition for deadlock, and (2) if the graph is expedient, then a *knot* is a sufficient condition for deadlock.

4.3 A Hierarchy of Deadlock Models

Edgar Knapp[39] proposed a hierarchical set of deadlock models to describe the characteristics of deadlocks. Each model is characterized by the restrictions that are imposed upon the form resource requests can assume. For example, a task might need to acquire a combination of resources like (R_1 *and* R_2) *or* R_3. The hierarchical set of deadlock models ranges from very restricted request forms to models with no restrictions whatsoever. The hierarchy can expand the unified GRS model (see Section 4.2) to further explore the conditions for deadlock. This hierarchy can also be used

to classify deadlock detection algorithms according to the complexity of the resource requests they permit.

4.3.1 Single-Resource Model

The simplest possible model is one in which a task can have at most one outstanding resource request at a time and all the resources are not sharable. Hence the maximum number of edges from a task or a resource in a GRG is 1. A *cycle* in a GRG is a necessary and sufficient condition for deadlock. A blocked task T is said to be deadlocked if T is in a cycle or T can only reach deadlocked tasks. Examples of this model can be found in database systems[4, 52]. In the Ada environment, if the resources are non-shareable and only one outstanding request is allowed, and no more than two tasks may be involved in either a rendezvous or task termination, the system can be formalized as a Single-Resource model. The Mitchell-Merritt algorithm[48] is a very simple and elegant algorithm based on this Single-Resource model.

4.3.2 AND Model

In the AND model, tasks are permitted to request a set of resources or resources are sharable. A task is blocked until it is granted *all* the resources it has requested. A shared resource is not available for holding exclusively until all its shared lock holders have released the lock. The requests of this type, therefore, are called AND requests. This model was referred to as the resource deadlock model in the literature (see Section 4.1). Examples of the AND model system can be found in some distributed DBMS where subtransactions can be executed concurrently on different sites. In the Ada environment, the shared resources and the task termination mechanism can be formalized as the AND model. Again, a *cycle* in a GRG is the necessary and sufficient condition for the existence of deadlocks in the AND model. And a blocked task T is said to be deadlocked if T is either in a cycle or can only reach deadlocked tasks. The AND model is, therefore, strictly more general than the Single-Resource model.

A number of algorithms have been proposed based on this AND model: Chandy-Misra algorithm[10], Chandy-Misra-Haas algorithm[8], Gligor-Shattuck algorithm[25], Haas-Mohan algorithm[28], Menasce-Muntz algorithm[45], and the Obermarck algorithm[51].

4.3.3 OR Model

In contrast to the AND model, an alternative way for making resource requests is the OR model. In this model, a task is blocked until it is granted *any* of the resources it has requested. For example, in replicated distributed database systems, a read request for a replicated data item is satisfied by reading any copy of it. This model was first referred to as communication deadlock model (see Section 4.1) by Chandy et al. in [8]. In OR model, detecting a *cycle* in GRG is not a sufficient condition for deadlock. As pointed out by Holt[33], a *knot* is a sufficient condition for deadlock while a *cycle* is only a necessary condition. Hence, deadlock detection in the OR model can be reduced to finding knots in the GRG. However, a task can be deadlocked without being in a knot. Therefore, a necessary and sufficient condition for a deadlocked task is: a blocked task T is deadlocked if T is in a knot or T can only reach deadlocked tasks.

There are similarities between the detection of an OR model deadlock and the detection of the termination of a group of cooperating tasks in a distributed computation[1, 2, 16, 19, 18, 23, 24, 46]. In a distributed system tasks cooperate with each other in a computation by means of message exchange. A distributed computation is said to be globally terminated if it reaches a *final* state which, in turn, relies on its member tasks reaching their final states and being ready to terminate. The termination problem arises when tasks are ready to terminate locally, but they still agree to communicate with other cooperating tasks. The *global termination condition* is defined as the condition that each of the cooperating tasks in a distributed computation is either terminated or ready to terminate. A global termination condition is not satisfied if *any* of the cooperating tasks in a distributed computation is not waiting for termination. For example, in Ada environments a *selective wait* statement allows a task to **terminate** if all its sibling tasks and their dependent tasks which belong to the same root creator have terminated or are waiting at a **terminate** alternative. This is a pessimistic model of the termination problem in that it assumes all the active sibling tasks may want to make an entry call to the ones which are ready to terminate in a selective wait statement. The distributed termination problem can be viewed as a special case of an OR deadlock where all the cooperating tasks in a distributed computation are involved in a deadlock (waiting for others to terminate). Therefore, any knot detection algorithm for the OR deadlocks can also be tailored for solving the distributed termination problem and vice versa.

Francez[23] first brought the distributed termination problem into prominence by defining global termination conditions in CSP environments, and proposed a detection algorithm for them. Dijkstra and Scholten[19] introduced the notion of *diffusing computation* (See Section 5.2.3) and suggested

an algorithm to detect the termination of an arbitrary diffusing computation in any network environment. Cohen and Lehmann[16] extended Francez's CSP termination model and solution to distributed systems where new cooperating tasks are created and terminated dynamically. Misra and Chandy[46] discussed how a termination computation algorithm can detect deadlock for the OR model. And then in [47] Misra and Chandy presented an algorithm for distributed knot detection. Chandy, Misra and Haas[8] presented an algorithm for the communication deadlock model which is an OR model in GRG. Natarajan's algorithm[50] is also based on the OR model. Their work is followed and improved by Huang[34].

4.3.4 AND-OR Model

The AND-OR model is a generalization of the two previous models. A task in AND-OR model may specify resources in any combination of AND and OR requests. For example, a task may request resources R_1 *or* (R_2 *and* (R_3 *or* R_4)) where R_1, R_2, R_3, and R_4 may exist at different sites.

There is no simple construct of graph theory in terms of GRG to describe the deadlock condition in the AND-OR model. In principle, deadlock in the AND-OR model can be detected by applying the test for OR model deadlock repeatedly, where each invocation operates on a subgraph of the AND part of the model. However, this strategy is not very efficient. Hermann and Chandy[30] proposed a more efficient algorithm for AND-OR deadlock. Their algorithm is based on a hierarchy of diffusing computation which they called a *tree* computation. The central idea of their algorithm is that when a diffusing computation reaches a blocked task: (1) the diffusing computation is propagated to its dependent set if it is an OR task, or (2) it initiates a separate tree computation if it is an AND task.

4.3.5 C(n,k) Model

The C(n,k) model, which was first formulated by Bracha and Toueg[5] as a k-out-of-n request, is a generalization of the AND-OR model. Any AND-OR request can be transformed into C(n,k) fashion and vice versa, hence, both models can be used interchangeably. In most of the cases, to express an C(n,k) request in the AND-OR form is more complicated than the transformation the other way around. The length of the corresponding AND-OR formula for a C(n,k) request is $k \cdot C(n,k)$. Again, the algorithm presented by Bracha and Toueg[5] suffers from the same deficiencies as that of the AND-OR model. Although the AND-OR model can describe the interaction mechanisms among tasks defined in Ada, in general, we would like to categorize Ada runtime environment as

the C(n,k) model because it is easier to formalize specific situations such as a task requests k pages of memory from a total of n pages.

4.3.6 Unrestricted Model

In the most general model, there is no assumed underlying structure for resource requests. The only assumption made is the stable property of deadlocks. Since Ada real-time applications have timing constraints which, if violated, may actually break a deadlock thereby breaking the stable property, great care should be taken in applying the techniques developed for this unrestricted model to Ada environment.

The motivation of using the Unrestricted model is to separate the problem property (the stability of deadlocks) from the abstracted underlying system structures (e.g., the abstracted data structure and synchronization mechanisms provided in many high level languages such as Ada), hence, a generalized solution can be developed. For example, in a general purpose system, one of the user applications may be a database system written in the Pascal in which the user provided lock manager is of the complexity of the AND deadlock model. Yet another user written application may be an Ada concurrent program in which the rendezvous is the OR deadlock model. Both applications are written in high level languages with abstracted data structures. Therefore, in this example, a system provided deadlock detection mechanism should be unrestricted and general enough to solve deadlocks of at least the OR model, regardless of what programming languages are used. However, the deadlock detection algorithms developed for the unrestricted model carry additional overhead which can be avoided in the algorithms designed for previously stated simpler models. This is due to the very fact that the underlying system computation structure is not fully utilized in helping to factor out some unnecessary conditions for deadlock. For example, an algorithm for the Unrestricted model usually needs to search the whole system to construct a GRG for deadlock detection. In contrast, if the only application running in the system is a database system as described above, an algorithm which sends probes to search part of the GRG for cycles is enough for the deadlock detection, and it is even beneficial if most of the deadlocks are two cycles.

All the algorithms designed for this Unrestricted model can be used to detect other stable properties (a.k.a. *quiescence* problem[11], examples are livelock or starvation, task termination detection, and orphan detection problems) as well. Also, the algorithms which use a global state detection technique for the detection of deadlocks can also be applied to distributed system monitoring, distributed debugging, etc.

Many algorithms related to this model have been studied theoretically such as: (1) stable properties detection by Chandy and Misra[11] and Hélary et al.[35], (2) global state detection by Chandy and Lamport[9], Spezialetti and Kearns[59], and Li et al.[41], (3) termination detection by Mattern[42], and (4) orphan detection by Shrivastava[56].

5 Distributed Deadlock Detection and Resolution Algorithms

Deadlock detection involves two basic tasks: (1) maintenance of the system state (in terms of TWFG, TRG, or GRG) and (2) search of the system state graph for the presence of deadlock conditions (cycles or knots). In distributed concurrent programming environments, a deadlock may involve several sites and the search for deadlocks greatly depends on the way the system state graph is maintained across the system.

Deadlock detection algorithms can be classified according to (1) the way the system state information is maintained or (2) the methodology used in searching for deadlock conditions. Singhal[57] classified the distributed deadlock detection algorithms based on the former notion into three types: *centralized*, *distributed*, and *hierarchical* approaches. In contrast, Knapp[39] adopted the latter notion to classify the distributed algorithms for the distributed deadlock detection into four classes: *path-pushing algorithms*, *edge-chasing algorithms*, *diffusing computations*, and *global state detection*. In this section, the distributed deadlock detection and resolution algorithms are summarized and categorized in the following subsections according to both Singhal and Knapp's classifications.

5.1 Centralized Algorithms for Distributed Deadlock Detection

In the centralized algorithms for distributed deadlock detection, a designated site, often called the control site, has the responsibility of maintaining global system state information and searching it for deadlock conditions. The global system state information may be maintained continuously (either keeping track of the up-to-date system state information periodically, or whenever the system state changes) or gathered from each site whenever the deadlock detection computation is needed. Since the control site can be viewed as a global system monitor, the kind of deadlock which can be detected is unrestricted. However, for simplicity, some algorithms place restrictions on the information gathered and on the algorithms used for searching for deadlocks (for example, searching for cycles instead of searching for knots), which, in turn, will restrict the kind of deadlock that can be detected. Since the control site has a global view of the whole system, deadlock resolution policy is relatively easy to carry out if the system was found in a deadlock state.

Centralized algorithms are conceptually simple and easy to implement, but they are highly inefficient and unreliable. The control site has to be informed by all the operations (resources requests, releases, etc.) across the system. Long response time for user requests, large communication overhead, and congestion of communication links near the control site are the problems which we can expect. Also, the communication and computation overheads of the deadlock detection are unrelated to the frequency of occurrence and the structure of deadlocks. Because the status of resource allocation is centralized at a single site, the centralized algorithms for distributed deadlock detection are subject to a single point of failure.

Some problems (such as long response time and congestion of communication links near the control site) can be diminished by having each site maintain its local resource status table for all its resources. For each resource, the status table keeps track of the tasks which have acquired the resource or are waiting for the resource. The control site, therefore, can collect these local resource status tables periodically for construction of the global system state. However, as pointed out by Ho and Ramamoorthy[31], the global system state may be inconsistent and false deadlocks might be detected due to the inherent communication delay and the lack of perfectly synchronized clocks in a distributed system.

For example, suppose two resources R_1 and R_2 are stored at sites S_1 and S_2 respectively. Let the following two tasks T_1 and T_2 be started almost simultaneously at sites S_3 and S_4 respectively, and execute lock $L(\cdot)$ and unlock $U(\cdot)$ operations in the following total order sequence:

$$
\begin{array}{lccccccccc}
T_1: & L(R_1) & & & U(R_1) & & & L(R_2) & & U(R_2) \\
T_2: & & L(R_1) & & & U(R_1) & L(R_2) & & U(R_2) & \\
time: & \longrightarrow & & t_1 & & & & t_2 & &
\end{array}
$$

The scenario is that both tasks T_1 and T_2 request resources R_1 and R_2 at sites S_1 and S_2, respectively. At time t_1, T_1 holds R_1 and T_2 is waiting for R_1 at site S_1. Both T_1 and T_2 finish accessing resource R_1 and start to request R_2 between time t_1 and t_2. At time t_2, T_2 holds R_2 and T_1 is waiting for R_2 at site S_2. Due to the communication delay and imperfect synchronization, site S_1 reports its local resource status table as $T_2 \rightarrow R_1 \rightarrow T_1$ at time t_1 and site S_2 reports its local resource status table as $T_1 \rightarrow R_2 \rightarrow T_2$ at time t_2. After constructing the global system state at time t_2, the control site reports a false deadlock $T_1 \rightarrow R_2 \rightarrow T_2 \rightarrow R_1 \rightarrow T_1$.

Ho-Ramamoorthy Algorithm: Based on the observations of the above problems, Ho and Ramamoorthy[31] proposed two algorithms called the *two-phase* and *one-phase* deadlock detection protocols. Their protocols were intend to solve the deadlock problem in AND models because only cycles are searched for in attempting to detect deadlocks. However, this restriction is not

necessary. A more complex deadlock searching algorithm can be adopted in the designated control site to handle a more complex deadlock model.

In the two-phase deadlock detection protocol, each site maintains a status table for all the tasks that are initiated at that site. The status table of each task keeps track of the resources that the task has acquired and the resources that the task is waiting for. Periodically, a designated control site requests the status table report from all sites and constructs a global system state. If the global system state contains no cycle then there is no deadlock. Otherwise, the control site again requests the status table report from each site. Only the tasks common to both reports are used to construct a second phase global system state. The system is declared deadlocked if the same cycle is detected again in the second phase global system state.

Ho and Ramamoorthy claimed that a consistent view of the system can be guaranteed by using this two-phase deadlock detection protocol. However, Jagannathan and Vasudevan[37] have disproved the claim by a counterexample which shows false detection of deadlocks. Using two report phases reduces the probability of false detection but it does not eliminate it.

On the other hand, the one-phase deadlock detection protocol detects deadlocks in one communication phase. Two tables, the resource status table and the task status table, are maintained at each site and gathered by the designated control site periodically. Only the tasks that appear in both tables are used for the construction of the global system state. The system is deadlocked if and only if cycles can be found in the global system state.

The correctness of the one-phase deadlock detection protocol has been proven by Ho and Ramamoorthy. No false deadlocks are detected by the protocol because the inconsistency is eliminated by matching the information gathered from two status tables. In the bipartite GRG, the edges only appear between two disjoint groups of vertices, a group of tasks and a group of resources. An edge really exists if it is reported by both of its end vertices, that is, indicated both in a task status table and in a resource table. This pessimistic way of collecting global information might miss the detection of some real deadlocks formed while the status tables are being collected. However, all deadlocks will be detected eventually by the protocol due to the stable property of deadlocks.

The one-phase protocol is faster and requires fewer messages than the two-phase protocol. But it requires more storage for maintaining two status tables at each site as well as for transferring these two status tables in one communication phase.

5.2 Distributed Algorithms for Distributed Deadlock Detection

In distributed algorithms for deadlock detection all sites cooperate to detect deadlocks in the system. Only partial knowledge of the global system state, which is enough for the detection of deadlocks, is learned at each site. Deadlock detection computation can be initiated whenever there is an indication of a possible deadlock. In many of the distributed algorithms, the computation is initiated when a task suspects a deadlock[2], and the initiation site can be either the site where the task resides or the site where the awaited resource resides. The methodology used for searching for deadlock conditions can be roughly divided into four classes[39]: *path-pushing*, *edge-chasing*, *diffusing computations*, and *global state detection*, which are discussed in the following four subsections.

There are several reasons why distributed algorithms for deadlock detection are more attractive than centralized ones. First, unlike the centralized algorithms, the distributed algorithms are not subject to a single point of failure, and there is no single control site to be swamped with deadlock detection activities which might become a system bottleneck. Second, based on the observation of typical applications, the frequency of deadlocks (especially, the global ones) are low and the length of the deadlock cycles are short[26, 4]. In the distributed algorithms, deadlock detection can be initiated only if the system is suspected to be deadlocked and can be executed only at sites involved in the suspected deadlock, which makes it more efficient than the fixed cost centralized algorithms. Furthermore, distributed algorithms for deadlock detection could be used with the algorithms for deadlock prevention and deadlock avoidance to provide a better fault tolerance while keeping the total cost of the deadlock detection low. Third, a distributed algorithm is not necessarily more complex than a centralized one as most of the literature has pointed out. Both centralized and distributed algorithms have similar difficulties in the distributed environments due to the lack of global synchronized clocks and the inherent communication delay. A major factor that determines the complexity of a deadlock detection algorithm is what kind of deadlock model is being addressed by the algorithm. The Mitchell-Merritt algorithm[48] is a fully distributed Single-Resource model deadlock detection algorithm that has been claimed to be very simple and has been proved to be correct. According to the authors, their first algorithm has been implemented in a database system in under an hour by one of them. One drawback of the distributed algorithms is the resolution of the detected deadlock is usually more difficult (except for the strategy that simply aborts the task which detects the deadlock) than that in the centralized ones. This is basically due to the lack of complete global information at each site and the same deadlock might be seen (or detected) at more than one site.

[2]In many cases, a waiting task sets a timer and if the time expires then it suspects deadlock and initiates the algorithm.

5.2.1 Path-Pushing Algorithms

Closely following the ideas of the centralized algorithms, the early distributed algorithms for deadlock detection maintained the notion of an explicit global system state. The basic idea is to construct a simplified form of global system state graph at each site which is sufficient to detect deadlocks. When a deadlock computation is initiated, each site sends its local state information to a number of neighboring sites. Upon receiving the local state information from neighboring sites, the local state graph is updated and then passed along. The procedure is repeated until some site has a sufficiently complete picture of the global system state graph to announce deadlock or to ensure that no deadlock exists. The name *path-pushing algorithms* come from the main feature of this scheme, that is, to transmit partial paths for the construction of the global system state graph.

This class of algorithms appeared around 1979-1982 when distributed algorithms for deadlock detection were first explored. Many algorithms in this class have been "proved" correct. However, they were subsequently found to be incorrect because they may fail to detect true deadlocks or they may detect false deadlocks. For example, Gligor and Shattuck[25] have illustrated that the distributed scheme presented by Menasce and Muntz[45] is incorrect because message could be out of order, and hence, may fail to detect genuine deadlocks or may detect false deadlocks. Obermarck's algorithm and proof[51] have been identified incorrect by himself as well as Elmagarmid[20] and Knapp[39] in the sense that false deadlock may be detected. A plausible reason for these algorithm failures is that the notion of consistent global state in the distributed environment was not well understood. Consequently, most of the algorithms had to rely on the freeze of the underlying system computation while the deadlock detection is proceeding.

Though these algorithms are incorrect, the Menasce-Muntz algorithm[45] and the Obermarck algorithm[51] are briefly summarized to understand the difficulties of this class of algorithms. And the discussion is followed by the Badal algorithm[3] which is a correct extension of the Obermarck algorithm.

Menasce-Muntz algorithm: Menasce-Muntz algorithm[45] was the first one to use a simplified form of the *transaction-wait-for graph* (see TWFG in Section 2). In their algorithm, only portions of the complete global TWFG are maintained at the transaction's original site for the detection of deadlocks. The underlying deadlock model is the AND model; hence, the algorithm searches for cycles in a subset of the global TWFG.

A transaction can be either *blocked* (with outgoing edges in the TWFG) or *nonblocked* (without outgoing edges in the TWFG). A *blocking-set(T)* of a transaction T is the set of all nonblocked

transactions T' which can be reached from T in the TWFG via a directed path. When a transaction T is blocked, for each transaction T' in the *blocking-set(T)*, a blocking pair (T, T') is identified and this information is sent to the original site of T and T'. When the original site of T' receives the pair and it is not the original site of T, new blocking pairs (T, T'') are sent to the original site of T'' for each transaction T'' in the *blocking-set(T'')*. On receiving such pairs a site updates its TWFG and performs local deadlock detection in addition to transmitting new blocking pairs to other sites. A path (or cycle) thus constructed in a site's TWFG may be "condensed" in the sense that some intermediate transactions may be missing.

Gligor and Shattuck[25] have shown that this algorithm fails to detect genuine deadlocks or may detect false deadlocks. First, in the remote request case, the determination of whether a transaction is blocked or not is incorrect because the determination can not be made until the response come back from the remote site. Furthermore, when sufficient information is obtained to determine correctly whether a transaction is blocked (i.e., when a blocking pair arrives), the protocol fails to use that information. In providing a remedy to these problems, Gligor and Shattuck suggested that every site should determine precisely whether a transaction is active, blocked, or waiting for the response of a remote request. Further, when a transaction is blocked it, in addition to identifying its blocking set, should identify a "potential blocking set" (i.e., the set of all waiting transactions reachable from a blocked transaction). As the authors point out, the modified algorithm is impractical because the implementation complexity arising primarily due to the condensation of the TWFG. For example, consider a waiting chain $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$ in which no two transactions reside in the same site. The chain is condensed into $T_1 \rightarrow T_n$ at the original site of T_n. When T_n releases its resources, there is no straightforward way to inform all the intermediate sites in this chain to update their TWFG's.

Obermarck algorithm: Obermarck algorithm[51] combines Gray's centralized deadlock detection algorithm[27] and the Menasce-Muntz algorithm described above. The algorithm is developed for a distributed database system, in which transactions can have agents at multiple sites, but only have a single locus of control. The reliable underlying communication system assures that all messages are sent and received in the same sequence. The underlying deadlock model is the AND model; hence the algorithm searches for cycles to detect deadlocks.

In the algorithm, the local TWFG is built at each site by a deadlock controller, and both resource waits and communication waits are gathered from database and communication managers. The transactions are lexically ordered by their identifiers and the deadlock messages are only transmitted in one direction which reduces the overhead as well as assures the single detection point of a deadlock

cycle. A special vertex in a local TWFG called EX-ternal is used to represent intersite wait-for relations. The existence of the vertex EX in a local TWFG indicates potential global deadlocks. Only the portion(s) of the local TWFG related to potential global deadlocks are shipped as *String(s)* from one site to another to detect global deadlocks. Deadlock detection at each site is an iteration of the following steps executed by a deadlock controller. The deadlock controller:

1. waits for and receives deadlock related information (produced in the previous deadlock detection iteration) from other sites,
2. it combines the received information with local TWFG and searches for cycles,
3. it solves the local deadlocks by choosing a victim from each of the local cycles which does not have an EX vertex, and
4. for all cycles $EX \rightarrow T_1 \rightarrow \cdots \rightarrow T_n \rightarrow EX$ that contains vertex EX (those cycles constitute potential global deadlocks), it sends the String $EX \rightarrow T_1 \rightarrow \cdots \rightarrow T_n$ to each site where the transaction T_n has an agent if the transaction identifiers has the order $T_1 > T_n$.

It is assumed in the correctness proof that the portion(s) of the local TWFG shipped as String(s) from one site to another will be frozen until it has been received and processed at some *final* site. A final site is either (1) the site where the TWFG information completes a cycle, or (2) the first site where a global deadlock can be proved not to exist. The assumption attempts to get a snapshot of part of the system for the deadlock detection algorithm, and Obermarck admits that it is not valid in real-world distributed systems. With this assumption, Obermarck points out that the local TWFG still may not be a true picture of the wait-for relations in the single site. False deadlocks, therefore, may be detected even with this assumption.

This algorithm has the primary advantage that the TWFG information is transmitted only when there is a potential global deadlock. However, the algorithm suffers from the false deadlocks as pointed out above. Also, the building of the TWFG is expensive and the analysis for cycles is repeated every time a deadlock detection is initiated.

Badal algorithm: In an attempt to optimize the costly distributed deadlock detection algorithms in terms of detecting the most frequent deadlocks with maximum efficiency, Badal proposed a three level algorithm[3] extended from the Obermarck algorithm[51]. A deadlock cycle can have many different topologies. Badal partitioned the deadlocks into four types according to their topology. In general, a deadlock can either be *local* (all the involved transactions and resources reside in a single site) or *global* (some of the involved transactions or resources might be remote). It is not

necessary to use a costly algorithm designed for detecting complex global deadlocks in detecting simple local ones. Also, most of the deadlock cycles have only a length of two and the longer a deadlock cycle is the less frequently it occurs. This fact leads to an idea of simplifying the detection of global deadlocks by making the algorithm more efficient in detecting shorter global deadlocks. Based on these ideas, the performance is optimized in Badal algorithm by using three levels of deadlock detection to cope with different complexity classes of deadlocks. Activity at each level is more complex and costly than that at the preceding level.

The deadlock detection starts at the first level algorithm which detects global deadlocks of cycle length two. The level two algorithm starts whenever a requested resource is still not available after a period of delay time and no deadlock has been detected in level one. If all the involved activities are local, the level two algorithm can decide whether a deadlock occurs or not; otherwise, it will wait for another period of time then either ships the information to the third level algorithm if the resource is still not available or proceeds if the resource becomes available. The third level of this algorithm is basically the same as the Obermarck algorithm[51] (with improvements which can reduce one intersite message) and, hence, suffers similar inefficiencies.

Badal has proved that the algorithm can detect the most frequent deadlocks with a minimum of intersite messages. However, he also pointed out that the algorithm has a constant overhead due to (1) more information has to be kept track of (in the lock tables) and (2) frequent checking of deadlocks of length two. Consequently, this algorithm is most likely to be used in distributed environments where deadlocks occur frequently enough to justify the continuous overhead.

5.2.2 Edge-Chasing Algorithms

This class of deadlock detection algorithms received the name from the way they search for cycles in a distributed system state graph. A special message called a *probe* is initiated and propagated along the edges (called an edge-chasing algorithm by Moss[49]) of the system state graph. If the probe (or a matching one) is finally received by its initiator then a cycle is detected. Since this class of algorithms can only detect cycles, they can only be applied to solve deadlock models of complexity up to and including the AND model.

A nice feature of this approach in connection with deadlock detection is that the complexity and efficiency of an algorithm can vary according to the complexity of the underlying deadlock model. For example, Mitchell-Merritt algorithm[48] is a very simple and efficient algorithm based on the

Single-Resource model while Chandy-Misra algorithm[10] is somewhat more complex but can be used in a more general AND deadlock model.

A *priority* based probe algorithm is an extension of the edge-chasing algorithm. The priority is used to reduce the probe messages and provide deadlock resolution. Several priority based algorithms have been proposed: Mitchell-Merritt algorithm[48], Sinha-Natarajan algorithm[58] (remedied by Choudhary et al.[13, 14]), and Sanders-Heuberger algorithm[55]. They are all designed for the Single-Resource deadlock model.

Another variation of the probe based algorithm is called the SET-based algorithm proposed by Chandy and Misra[10], Haas and Mohan[28], and Choudhary et al.[12]. In SET-based algorithm, a "SET" of the system state information is propagated along with the probe. According to the Choudhary's simulation results, the SET-based algorithm is more efficient than the probe based algorithms.

For this class of algorithms, we only present the Mitchell-Merritt algorithm[48] as an example due to its simplicity and elegance. Mitchell-Merritt algorithm has two versions (one without priority and the other one with priority) which capture most of the important features of the edge-chasing algorithms.

Mitchell-Merritt algorithm: Mitchell-Merritt algorithm[48] is an edge-chasing algorithm in which probes are sent in the opposite directions on the edges of the TWFG. Each vertex of the TWFG has two labels: *private* and *public*. The private label of each vertex, though not constant, is unique and non-decreasing to the vertex all the times. The public label of each vertex can be read by other tasks and need not be unique. Initially, these labels for all vertices have the same value. The algorithm is executed in the following four steps:

1. *Block Step*: When a task begins to wait for another task upon some resources, it becomes *Blocked*. Both of the labels of the *Blocked* task are increased to a value greater than their previous values and greater than the public label of the blocking task.

2. *Active Step*: A task becomes *Active* when it gets a resource, times out or fails. Also, when a waiting task notices that the owner of a awaited resource changes, it must become *Active* and then *Blocked* again if it does not acquire the resource.

3. *Transmit Step*: When a *Blocked* task discovers that its public label is smaller than that of the blocking task, it updates its label to a value equal to that of the blocking task.

4. *Detect Step*: When a task receives its own public label back, a deadlock is detected.

When a task become *Blocked* it adds an edge to the TWFG by executing the *Block Step*. When a task becomes *Active*, it deletes an edge from the TWFG by executing the *Active Step*. The *Blocked* tasks execute the *Transmit Step* periodically. The effect is that the largest public label tends to migrate in the opposite direction along the edges of the TWFG. Only one task in a cycle will detect deadlock, which simplifies the problem of resolution. Only genuine deadlocks will be detected in the absence of spontaneous aborts. Spontaneous aborts are allowed, however, if false detection of deadlocks can be tolerated.

The simple algorithm given above can be easily extended to include priorities such that the lowest priority task in a deadlock cycle aborts itself. Each vertex in the TWFG now has two additional unique *priority numbers*, one private and one public. Initially, each task posts its unique private label and priority number. In the *Transmit Step*, vertices with equal public labels transfer the lower priority number. And in the *Detect Step*, a task detects deadlock and aborts itself when it receives its own priority number together with its own public number. Since the task with the highest public label may be waiting for the lowest priority task, the low priority number may not be propagated around the cycle until after the public label has gone full-cycle. Thus, this algorithm can take up to twice as long to detect deadlock as the above first algorithm.

As pointed out by Knapp[39], the algorithm does not remain correct if public labels are propagated in the same direction as the directed edges instead of the other way around. The reason for this is that if the largest public label is owned by a deadlocked task which is not part of a cycle, this label might enter the cycle and circulate once without detecting the deadlock. After all the public labels in the cycle have been updated to a larger value than any of their labels, none of the tasks in the cycle would detect the deadlock. Also, for a similar reason, there seems to be no straightforward way to extend the algorithm to handle deadlocks in the AND model.

5.2.3 Diffusing Computations

Dijkstra and Scholten[19] introduced the notion of *diffusing computation* in a distributed system of tasks, and suggested an algorithm to detect the termination of an arbitrary diffusing computation in any network environment. The generality of the solution makes it is a candidate for a number of problems in the distributed systems. Using this algorithm, Misra and Chandy presented algorithms for distributed deadlock detection for CSP-like environments[47, 46]. Their work is followed and improved by Huang[34]. Also, Chandy, Misra and Haas[8] presented an algorithm for a communication deadlock model which is an OR model in the GRS. Hermann and Chandy's algorithm[30] for the AND-OR model is a so called *tree* computation which is a hierarchy of diffusing computations.

First, we summarize the diffusing computation and its properties from [19]. In a distributed system, a diffusing computation can be mapped to a directed graph such that each vertex represents a task and each directed edge represents the relation between its two end tasks. If the graph contains an edge from vertex T_1 to vertex T_2, T_1 is called a *predecessor* of T_2 and T_2 is called a *successor* of T_1. It is assumed that there exists a root initiator vertex without incoming edges, from which a diffusing computation is initiated. The initiator is called *the environment* by Dijkstra and Scholten[19] due to its relation to the rest of the graph. Vertices different from the environment are called *internal* vertices. The *messages* are sent from a vertex to its successors, while the *signals* are propagated in the reverse direction along the edges. A diffusing computation grows by sending messages and shrinks by receiving signals. It is this feature that inspired the name *diffusing computations.*

Each vertex in a diffusing computation begins with a *neutral state.* The environment initiates the computation by sending out messages to its successors. In a diffusing computation, the first message sent to a vertex is called an *engaging message* while the last signal sent by a vertex, which is used to reply to its engaging message, is called an *engaging signal.* Upon reception of the first message (i.e., the engaging message), a vertex leaves its neutral state and becomes *engaged.* An internal engaged vertex (1) is free to propagate messages (including its engaging message) to its successors, and (2) is free to send signals to its predecessor for every message (except its engaging message) it receives. An internal engaged vertex is able to receive signals from its successors. An engaged vertex sends an engaging signal to its predecessor when it receives engaging signals from all its successors.

For each edge, its *deficit*, a non-negative value, is defined as the number of messages minus the number of signals transmitted over it. The neutral state of a vertex can now be redefined to be the state in which the deficits of all incoming and outgoing edges are zero. The diffusing computation terminates if the environment returns to its neutral state because it implies that all the internal vertices are in the neutral state.

If an engaged vertex T_1 sent an engaging message to its neutral state successor T_2, the edge (T_1, T_2) is called an *engagement edge.* The engagement edge (T_1, T_2) will be finally cancelled after the vertex T_2 sends an engaging signal to T_1 and returns to the neutral state. The lifetime of the engagement edge (T_1, T_2) is the same as the duration while T_2 is engaged. It follows that: (a) each engagement edge connects two engaged vertices, (b) engagement edges do not form cycles, and (c) each engaged internal vertex has exactly one incoming engagement edge. Consequently, the engagement edges form a rooted tree during a diffusing computation, and all engaged internal vertices are reachable from the environment via paths of these engagement edges.

The basic idea of using the diffusing computation for deadlock detection in the OR model is that the GRG can be implicitly reflected in the structure of the computation. When a task suspects a deadlock, it initiates a diffusing computation. Different from the original diffusing computation, the initiator is not necessarily the root of a tree in GRG. The initiator may have incoming edges which are the hints of the deadlocks. During the computation, the engagement edges always form a *engagement tree* spanning over the set of tasks which are waiting for each other. The engagement tree contains no cycle. Such a engagement tree eventually vanishes if all the outgoing edges in the set of waiting tasks find a cycle. Therefore, if the computation finally terminates, the initiator declares a deadlock. If no deadlock exists, the initiator will remain idle in the engaged state until its resource requests are fulfilled, and the transmission of all the messages and signals will simply stop. The algorithms summarized below will further clarify how one uses diffusing computations for deadlock detection.

Chandy-Misra-Haas Algorithm: By using the paradigm of diffusing computations, Chandy, Misra, and Haas[8] presented a distributed deadlock detection algorithm for the communication (OR) model.

In this algorithm, messages are called *queries* and signals are called *replies*. A blocked task initiates a deadlock computation by sending queries to tasks in its dependent set. An executing task ignores all queries and replies. On the other hand, if the diffusing computation reaches a blocked task then that blocked task becomes *engaged* and participates in the computation. A query is answered whenever it reaches an engaged task. A query will be eventually answered if it travels along the edges of a cycle, and all the involved engagement edges will be cancelled. It follows inductively that engagement edges do not form cycles. All the queries initiated found cycles if the diffusing computation terminates and the initiator returns to its neutral state. Consequently, the initiator is deadlocked if it returns to its neutral state.

As an example, a blocked task T_a initiates a deadlock detection by a diffusing computation and it requires resources B or C which are held by tasks T_b and T_c, respectively. Consider the situation that task T_b is executing and task T_c is waiting for T_a. The query received by task T_b will be ignored because it is not blocked. Upon receiving the query from T_a, task T_c becomes engaged and propagates the query to its dependent set which is $\{T_a\}$. Since T_a is in the engaged state, a reply will be sent back to T_c immediately. When T_c receives the reply from T_a, it sends back an engaging reply to T_a and returns to neutral state. No more queries and replies related to the computation initiated by task T_a will be send at this point. Task T_a remains idle and keeps waiting for the outcome of the executing task T_b. Consider another situation that both tasks T_b and T_c are waiting

for the task T_a. Both engagement edges (T_a, T_b) and (T_a, T_c) will eventually be cancelled. Finally task T_a terminates the computation and declares a deadlock after it receives replies from all the tasks in its dependent set $\{T_a, T_b\}$ and returns to its neutral state.

Hermann-Chandy Algorithm: Hermann and Chandy's algorithm[30] for the AND-OR model is a so called *tree* computation. A tree computation consists of a hierarchy of diffusing computations. First, the algorithm requires a general AND-OR request to be mapped to a regular form, such as disjunctive normal form. A task may have either an AND request or an OR request. An AND-OR request issued by a task is mapped to a tree of tasks with only AND or OR requests. A blocked task T is deadlocked, if either

- T is an AND task and will not be granted for at least one of its requests, or
- T is an OR task, but will not be granted for any request.

The basic idea of the algorithm is that whenever a diffusing computation reaches a blocked OR task, the computation is propagated to its dependent set; if the engaged task is an AND task, it initiates a separate tree computation for each of its outgoing edges. Therefore, a tree computation consists of either a diffusing computation or a set of tree computations. Since an engaging reply (the one that replies to an engaging query) means an engaged blocked task has been reached, a cycle is found along the path on which the reply returns. An AND task will send back an engaging reply when it receives an engaging reply from one of its dependent set. An OR task will send back an engaging reply when it receives all the replies from its dependent set. If the initiator is an AND task, it terminates when it receives an engaging reply. If the initiator is an OR task, it terminates when it receives all the engaging replies. A tree computation terminates and declares deadlock whenever the initiator terminates.

5.2.4 Global State Detection

A key notion in distributed global state detection algorithms is that a *consistent global state* can be determined without freezing the underlying system computations. Chandy and Lamport[9] proposed the notion of *distributed snapshots* which is a mechanism to get information about the global state of a distributed computation without synchronization. Spezialetti and Kearns[59] modified this algorithm to gain more efficiency in some special application cases. The algorithm is intended to provide a general purpose framework which can be adapted to specific implementation requirements.

The Chandy and Lamport's global snapshot algorithm runs in two independent phases. During the first phase, each task records its own state, and the two tasks connected by a communication channel cooperate in recording the channel state. Due to the lack of global clock, the recording procedure can only be carried out asynchronously, and the recorded system state is required to be "meaningful." The recorded task and channel states will then be collected and assembled in a second phase after the completion of the first phase.

The recorded task and channel states might be inconsistent in the global system state. For example, there are two tasks T_1 and T_2 connected by the channel C. Task T_1 sends a message along C to T_2. If T_1 records the state of channel C *before* sending the message and records its own state *after* sending the message, then an inconsistent global system state will erroneously show that a message is missing. Thus a "meaningful" global system state cannot be formed in this way.

Chandy and Lamport suggested that a special message, called a *marker*, being sent to avoid such inconsistent states. The outline of their algorithm is as follows:

Marker-Sending Rule for a Task T_1: For each outgoing channel C connected to T_1, a marker is sent along C after the state of T_1 is recorded and before any further message is sent along C.

Marker-Receiving Rule for a Task T_2: On receiving a marker along a channel C:

if T_2 has not recorded its state **then**
    T_2 records its own state;
    T_2 records the state of C as empty;
else
    T_2 records the state of C as the sequence of messages received along C between the event where it recorded its own state and the event where it received the marker;
end if;

The Distributed Snapshot algorithm is completely self-contained and makes no assumptions about the states it collects. The only assumption it utilizes is the *stable* system property. Therefore, this algorithm is suitable for a variety of stable system property applications such as termination detection, deadlock detection, and global state monitoring.

In using the snapshot algorithm for distributed deadlock detection, the collected global system state is a consistent GRG. Let GRG_t denote the GRG at time t. Bracha and Toueg[6] proved that if the time $t1$ is earlier than the time $t2$, and a task T is deadlocked in the GRG_{t1}, then the task T is deadlocked in the GRG_{t2}. This result allows a global snapshot of the GRG to be analyzed for deadlocks in parallel with the continued operation of the system.

Bracha-Toueg Algorithm: Bracha and Toueg[6] proposed an algorithm to process the system snapshot GRG to find deadlocks in the C(n,k) model. Since there is no simple construct of graph theory in terms of GRG to describe the deadlock condition in the C(n,k) model, the graph reduction techniques suggested by Holt[33] are used to determine the existence of deadlocks. Each of the active tasks in the snapshot GRG can be scheduled to terminate and to release the resources it holds. The GRG thus can be *reduced* to a new state (see Section 4.2). A GRG is said to be *completely reducible* if there exists a sequence of graph reductions that reduces the GRG to a set of isolated vertices. A task T_i is not deadlocked in state S if and only if there exists a sequence of reductions in the corresponding GRG that leaves T_i unblocked. If a GRG is completely reducible, then the state it represents is not deadlocked.

5.3 Hierarchical Algorithms for Distributed Deadlock Detection

In hierarchical algorithms for distributed deadlock detection, sites are logically organized in a hierarchy. Each site is responsible for detecting deadlocks involving itself and its children sites. In most of the hierarchical arranged systems, the resource access pattern is localized to a cluster of sites belonging to a same parent site. The performance, therefore, can be optimized if a hierarchical deadlock detection algorithm reflects the hierarchical structure of its underlying system.

Menasce-Muntz Algorithm: In the hierarchical deadlock detection algorithm presented by Menasce and Muntz[45], the database (DB) is partitioned into a set of subdatabases (sub-DB). The locking and deadlock controllers are arranged in a tree fashion. Each of the *leaf* controllers manage a sub-DB, while a *non-leaf* controller is responsible for distributed deadlock detection. A leaf controller maintains the part of the global GRG concerning the sub-DB managed at the controller. A non-leaf controller maintains the GRG including its children controllers and is responsible for detecting deadlocks involving only its children controllers. A non-leaf controller keeps track of the changes, such as the occurrences of allocation, wait, or release of the resources, in each of its children controllers. This job can be done continuously (that is, whenever a change occurs) or periodically. After each update of its own GRG, a non-leaf controller performs deadlock detection. This algorithm is developed for the AND model since it searches for cycles. However, the approach can be applied to a more complex deadlock model if an appropriate deadlock detection algorithm is used.

Ho-Ramamoorthy Algorithm: In the hierarchical deadlock detection algorithm presented by Ho and Ramamoorthy[31], sites which are close to each other are grouped into a cluster. A site in a cluster is chosen as the control site periodically. A control site collects status tables from all

the sites in its cluster, and applies a one-phase deadlock detection protocol (see their algorithm in Section 5.1) to detect intra-cluster deadlocks. Also, a central control site is chosen dynamically which then collects the inter-cluster information and constructs a system wide GRG for inter-cluster deadlock detection.

5.4 Summary of the Distributed Deadlock Detection Algorithms

In this section, we have surveyed, though not completely, a variety of distributed deadlock detection algorithms. To summarize, we want to emphasize two issues: correctness and performance of these algorithms.

During the survey of the distributed deadlock detection algorithms, we have observed that correctness proofs are very difficult in this area. The basic reason is that these algorithms are developed for the distributed environments which are usually complex and non-deterministic in nature. The execution patterns of an algorithm can be extremely large which eliminates the possibility of exhaustively studying all the possible situations. The lack of perfect global synchronization further complicates the correctness proof due to the timing sensitivity of the algorithms. Consequently, informal proof techniques based on intuitive arguments are widely used in most of the literature. These informal correctness proofs are relatively unreliable. Many algorithms were informally proved and claimed correct by their authors have been disproved later (many examples can be found in our survey). However, it is worth noting that the situation has been improved recently due to some important concepts, such as "global state consistency," "stability of the deadlocks," "complexity of the deadlock problems," and "diffusing computation," that have been introduced. These concepts makes the analysis of the problems and the resolutions more precise and, therefore, enhances the reliability of the algorithm correctness.

To evaluate the performance of a distributed deadlock detection algorithm is difficult because there are many correlated factors, such as the message traffic, the size of messages, the complexity of the problem to be addressed, the complexity of the algorithm, the frequency that the detection computation is invoked, the percentage of the invoked computations that detect deadlock, etc., which could affect the performance of an algorithm. Also, many algorithms were motivated by the performance improvement to a previously developed one, their achievements, however, are usually hard to judge. For example, many authors of these improved algorithms argue that their algorithms could perform better because they reduce the deadlock message traffic. This argument, however, may not be true due to the fact that the complicated new algorithm may increase the

overhead during the normal computations while they attempt to reduce the deadlock message traffic. Therefore, a precise performance comparison among different algorithms is very difficult.

In Table 1, we have listed the performance of the surveyed algorithms from a variety of aspects. The algorithms of the hierarchical approach are excluded because they are the modified versions that take the workload access pattern into consideration and their performance upper bound is exactly the same as their original versions. The values are gathered from the original articles, Singhal's survey[57], and Knapp's survey[39]. The algorithms listed are developed for different kind of system environments which may have different problem complexities. Consequently, the values listed in the Table 1 can only serve as a reference and cannot be used to judge which algorithm performs better.

In general, Knapp[39] compares the performance of the distributed algorithms in terms of the problem complexity and the algorithm complexity. He concludes that the Single-Resource, the AND, and the OR models all have a worst-case algorithm complexity of $O(N^2)$ messages, in which N is the number of the vertices in a GRG. The AND-OR model requires at most $O(N^3)$ messages which doesn't include the complexity of the mapping used to transform a general request, say a C(n,k) request, to the regular AND-OR form. The C(n,k) model needs $O(N^2)$ messages to construct a consistent global snapshot in which the complexity of searching a snapshot for deadlocks is not included. Therefore, Knapp suggests that in selecting a deadlock algorithm for a particular application, the least general technique which is still general enough to solve the problem is advised.

6 Distributed Deadlock Detection and Resolution in Real-Time Systems

In this section, some of the design issues concerning deadlock detection problems in a distributed real-time environments are discussed. In the discussion, the problem will first be defined. The complexity of the problem is categorized into four levels based on the sufficient conditions required to detect deadlocks for that problem. The deadlock models associated with each of the four levels are identified. Finally, design criteria for distributed deadlock detection and resolution algorithms for real-time systems are discussed.

Table 1: Performance of the Surveyed Algorithms

Algorithms	No. of Messages	Delay	Size of Messages	Problem Complexity
I. Centralized Approaches				
Ho-Ramamoorthy (two-phase)	$4N$	B: $4T$ W: $p+4NT$	V,L	AND
Ho-Ramamoorthy (one-phase)	$2N$	B: $2T$ W: $p+2NT$	V,L	AND
II. Distributed Approaches				
1. Path-Pushing Algorithms				
Menasce-Muntz	$m(n-1)$	nT	V,S	AND
Obermarck	$m(n-1)/2$	nT	V,M	AND
Badal (type I)	$\mathcal{M}-1$	$(\mathcal{M}-1)T$	V,M	AND
Badal (type II)	$n-1$	$(n-1)T$	V,M	AND
2. Edge-Chasing Algorithms				
Mitchell-Merritt (no priority)	$m(n-1)/2$	$(n-1)T/2$	C,S	Single-Resouce
Mitchell-Merritt (priority)	$m(n-1)$	$(n-1)T$	C,S	Single-Resouce
3. Diffusing Computation				
Chandy-Misra-Haas	W: $2m(n-1)$	$2dT$	C,S	OR
Hermann-Chandy	W: $2m^2(n-1)$	$2dT$	V,M	AND-OR
4. Global State Detection				
Bracha-Toueg	$4m(N-1)$	$4dT$	V,M	C(n,k)

N: number of sites; n: number of sites involved in deadlock;
m: number of tasks involved in deadlock; T: inter-site communication delay;
p: the period between two consequent GRG updates; d: diameter of GRG.

Message Sizes: V: variable; C: constant; L: large; M: medium; S: small.

B: best case; W: worst case.

Notes for Badal Algorithm:
type I: resource's intention lock can be determined before task migration;
type II: resource's intention lock can be determined after task migration;
for type I: $\mathcal{M} = \sum_{k=1}^{n}(n-k)$.

6.1 Deadlock Problems in Distributed Real-Time Systems

In real-time systems, due to timing constraints attached to each task, time-outs and abnormal aborts may occur when a task is blocked. If a task T is blocked in a real-time environment, it may be involved in the following situations:

Stable Deadlock: This is the situation that the reachable set $RS(T)$ of the blocked task T in the GRG forms a deadlock (may be a cycle or a knot) and neither any time-out nor abnormal abort is expected. These deadlock conditions are stable in that once they are formed, they will stay until they are detected and resolved.

Temporal Deadlock: This is the situation that the reachable set $RS(T)$ of the blocked task T in the GRG forms a deadlock. However, due to timing constraints, a nonempty subset of tasks in a set of deadlocked tasks may be timed out or aborted from the blocked state. The deadlock situation, therefore, may not exist forever and, hence, is temporal.

Non-deadlocked Blocking: This is the situation in which a task is blocked, but it is not involved in any deadlock. The situation exists for a normal wait, or an abnormal condition such as being livelocked or being an orphan task. In a real-time setting a task needs to make progress in a limited period of time. It is important that the waiting situation for whatever reason should be terminated in a timely manner to ensure the timing constraints.

The three situations stated above define three deadlock related problems in real-time systems. A stable deadlock in real-time systems is the same as a traditional deadlock in non-real-time systems. A temporal deadlock, on the other hand, is a special kind of deadlock which is not treated as a deadlock or is assumed not to exist in non-real-time systems. Such a deadlock is *temporal* and hence not *stable*. The *stable property* which is assumed in most of the traditional deadlock detection algorithms can no longer be used to detect temporal deadlocks in real-time systems. Timing constraints must be taken into consideration in detecting temporal deadlocks. The timing information collected for detecting temporal deadlocks can also be applied to resolve many of the the problems associated with non-deadlocked blocking.

The detection and resolution of temporal deadlock is important for tasks with timing constraints in real-time systems. For example, in the Ada environment, task T may be in a temporal deadlock state if there is a cycle (or a knot) in its $RS(T)$ which contains tasks blocked by timed statements. If T carries the nearest deadline and the highest criticalness in the deadlocked task set, it is important that a timely detection and resolution is completed before a time in which it is still possible to

meet the timing constraint of T. If no detection operation is attempted, task T may fail without knowing the existence of this temporal deadlock.

6.2 The Complexity of Deadlock Models

In Section 4.3, a hierarchical set of six deadlock models were used to describe the characteristics of deadlocks. Except for the Unrestricted model, the problem complexity of the other five models can be roughly divided into four levels based on sufficient conditions for detecting deadlocks:

1. the Single-Resource model (contains only simple cycles; simple cycle detection is sufficient),
2. the AND model (contains nested cycles; nested cycle detection is sufficient),
3. the OR (cycle detection is not sufficient; knot detection is sufficient), and
4. the AND-OR and the C(n,k) models (both cycle and knot detections are not sufficient; cycle detection may detect false deadlocks whereas knot detection is not sufficient to detect all deadlocks; a correct detection algorithm requires the recognition of AND, OR, AND-OR, and C(n,k) requests).

In the Single-Resource model no nested deadlock cycles can occur. This property gives rise to an interesting solution. If deadlock detection probes are propagated in the opposite direction along the edges of the GRG, only the *in-cycle probes* initiated by the tasks in a cycle will detect deadlock. It is possible that only one task in a cycle will detect deadlock if a probe propagation rule is enforced. For example, in the algorithm developed by Mitchell and Merritt[48], each of the blocked tasks is assigned an unique identifier and a probe is propagated in the reverse direction only when its initiator identifier is larger than that of the destination task. This algorithm guarantees that only the probe with the largest initiator identifier is able to travel through the whole cycle to detect the deadlock. Such an algorithm simplifies the problem of resolution as well as guarantees that only genuine deadlocks will be detected in the absence of spontaneous time-outs and aborts. In real-time systems, due to timing constraints attached to each task, spontaneous time-outs or aborts are possible which may cause false detection of deadlocks. To eliminate the false detection of deadlocks, we need to consider timing constraints so that the temporal waiting edges (due to timing constraints of the waiting tasks) in the GRG are well treated in the algorithm. For example, timing constraints can be associated with each deadlock detection probe to reflect the timing validity of the probe (see Section 7.2).

In the AND deadlock model since multiple outgoing edges as well as multiple incoming edges in the GRG are possible, nested cycles are expected in this model. Each cycle in a group of nested cycles is stable, but the whole group of nested cycles is not stable because new cycles may be forming and attaching to the existing nested cycles. Since there are joint parts between any two nested cycles, the resolution of a deadlock may actually break more than one cycle at their common part of the graph. Therefore, a detected cycle may not exist if it nested with another cycle which was detected and resolved earlier at a common part of these two cycles. To avoid false detection of deadlocks, we need to detect the whole group of the nested cycles as well as to prevent any new cycle attach to it before the current ones are resolved. This requires synchronization between deadlock detection and other system activities, for example, to "freeze" the system while a deadlock detection and resolution in ongoing. Unfortunately, a distributed system is too costly to be frozen and, therefore, false detection of deadlocks is inevitable. Also, due to the existence of nested cycles, simple cycle detection algorithms, such as the ones used in the Single-Resource model, can no longer guarantee that only genuine deadlocks will be detected even in the absence of spontaneous time-outs and aborts. Situations concerning the nested cycles have to be taken into consideration to minimize the detection of false deadlocks. In the probe based algorithms, *foreign probes*, which is initiated by tasks outside a cycle, may enter the cycle. A foreign probe may travel in the cycle more than once without detecting any deadlock if there is no mechanism to stop it. A foreign probe, which happens to meet the rule of the algorithm, may interfere with the in-cycle probes and, hence, may cause the algorithm to fail to detect the deadlock. For example, similar to the probe propagation rule introduced in the Mitchell-Merritt algorithm, if the in-cycle probe with the largest label is expected to travel through the cycle to detect the deadlock, a foreign probe with a even larger label may enter the cycle and compete with the in-cycle probes. The algorithm may fail if such situations are not carefully considered. Again, similar to the Single-Resource model, timing constraints can be carried by the probes to cope with the effect of spontaneous time-outs and aborts in real-time systems.

In OR model, a knot is a sufficient condition for deadlock while a cycle is only a necessary condition. Hence, deadlock detection in OR model can be reduced to finding knots in the GRG. A task T_i in a GRG is in a knot if for every task T_j reachable from T_i, T_i is reachable from T_j. To detect knots, probes are propagated in both forward (to search tasks which is reachable from T_i) and backward (to search tasks which can reach T_i) directions along the edges in GRG. After the GRG has been fully searched, the algorithm can decide the existence of knots. Whenever a sink in the GRG is reached, a non-deadlock condition is found. A knot detection algorithm should be able to terminate if it detects either a knot or a non-deadlock condition. As discussed in Section 4.3.3,

knot detection algorithms for the OR model deadlocks can be tailored to resolve the distributed termination problem, and vice versa. Many algorithms proposed in the literature[8, 34, 47, 50] are actually based on the notion of Dijkstra and Scholten's *diffusing computation* which is originally used for distributed termination detection. Similar to the previous two models, certain techniques can be used to reduce the number of probes travelling in the GRG. Also, the timing constraints can be addressed by applying deadlines to the probes.

The problem complexity of the remaining models — the AND-OR model and the C(n,k) model — are roughly the same. Many systems are neither solely the AND-OR model nor solely the C(n,k) model but a mixture of two. Such a mixed model system, however, can be mapped either to the AND-OR model or to the C(n,k) model. The mapping, in general, is easier toward the C(n,k) model than toward the AND-OR model. The main concern of this complexity level, therefore, is solving the problems that deal with the C(n,k) deadlock model. As suggested by Bracha and Toueg[5], to process a global snapshot is a way to find deadlocks in the C(n,k) model. Once again, since AND deadlocks are embedded in the AND-OR model and the C(n,k) model, nested deadlocks, similar to the nested cycles in the AND model, may occur. Therefore, we face a similar false deadlock detection problem as found in the AND model. In real-time applications, the timing constraints associate with each task can be collected while taking snapshots of the system. A temporal deadlock or a non-deadlocked blocking can be captured if a blocked task might not be scheduled to become active in the snapshot to ensure its timing constraints.

Different deadlock models are assumed in the four levels of problem complexity categorized above. The applications of the algorithms developed for each of these deadlock models, therefore, have different restraints. For the Single-Resource model, resources must be non-sharable and must be distinguishable. Low level system provided task synchronization mechanisms can be allowed if no multiple outgoing edges in the GRG may result. For example, a *semaphore* is a synchronization tool provided in many systems. A semaphore S is an integer variable that can be accessed only through two *atomic* operations P and V. The atomic operation P decreases the integer S by 1 if S is great than zero; otherwise, it waits. The atomic operation V, on the other hand, increases the integer S by 1. Such a semaphore is usually called a *counting* semaphore. A *binary* semaphore is a semaphore whose integer value can range only between 0 and 1. A counting semaphore may be "granted" to more than one task, which may cause multiple outgoing edges from the resource "semaphore," hence, is not permitted in this Single-Resource model. A binary semaphore may only be "granted" to at most one task, therefore, is allowed in the Single-Resource model. In the Ada environment, certain program restrictions must be enforced to ensure the single outgoing edge in GRG requirement of this model. For example, the Ada **accept** statement, which allows one of

many potential calling tasks to be in rendezvous with the accepting task, must be programmed in an one to one fashion that limits the number of potential calling tasks to exactly one.

For the AND or OR model, some of the constraints of the previous Single-Resource model can be relaxed. In the Ada environment, once using an algorithm powerful enough to solve the AND model, then all deadlocks with the AND logic mechanisms involved, such as task termination, can be detected. Also, resources may be sharable in this model. For the OR model, all deadlocks with OR logic such as task interactions and synchronizations due to accept statements, can be detected. The counting semaphores can be used in the OR model.

The AND-OR and C(n,k) models are general enough that all the constraints of programming to conform to the previous two models can be removed. Resources may be indistinguishable or sharable. All the Ada task interaction and synchronization mechanisms are supported by the deadlock detection in this model.

6.3 Criteria in Designing Distributed Deadlock Detection and Resolution Algorithms for Real-Time Systems

A deadlock detection algorithm is correct if and only if it satisfies the following criteria:

Correctness Criterion 1: The algorithm must be able to detect any deadlock in the system in a finite time.

Correctness Criterion 2: All the deadlocks detected by the algorithm must be genuine ones.

However, it is generally too expensive to completely achieve these two criteria when designing algorithms for distributed real-time systems. Some of the issues centered around the correctness criteria in distributed real-time systems are discussed as follows:

1. These criteria might be violated due to timing constraints in real-time systems. The timing constraints associated with tasks make deadlocks temporal (i.e., not stable, see Section 6.1). A temporal deadlock may break without ever being detected.

2. These criteria might be violated due to synchronization difficulties in distributed real-time systems. For example, if the deadlock detection computations are running concurrently with the other system activities, false deadlocks may be reported in the AND, AND-OR, and

C(n,k) (see Section 6.2) deadlock models which violates Correctness Criterion 2. To synchronize deadlock detection with the other system activities (e.g., to freeze the system while running a deadlock detection) is difficult and expensive, especially, in distributed real-time environments.

3. Another issue that arises is that sometimes one of the correctness criteria might be fulfilled at the sacrifice of the other one. Again, let's use the detection of AND model deadlocks as an example. It is required that the whole group of nested cycles should be detected together. However, most of the existing distributed deadlock detection algorithms for the AND model do not use such a complicated approach; instead, due to efficiency concerns, they simply detect and resolve individual deadlock cycles. When a cycle is detected, it lacks the information that the cycle might be nested with other cycles and that it might have been broken at the intersecting part of the graph. Therefore, the limited information indicates only the potential existence of deadlocks and the algorithm is responsible for the decision whether the situation is to be treated as a deadlock. Ignoring these potential deadlocks might violate Correctness Criterion 1. In contrast, treating these potential deadlocks as genuine ones might violate Correctness Criterion 2.

4. For soft real-time systems where violations will not cause any severe permanent faults these two criteria can be relaxed to an acceptable level. For example, to cope with timing and synchronization problems described above, a compromise may be to speed up the algorithm so the undetected and/or false deadlocks can be minimized. Also, only temporal deadlocks are allowed to go undetected since they will not stay in the system permanently. However, the effort of detecting temporal deadlocks before they disappear is still required in order to prevent a system from staying in a deadlocked state for too long. As for the decision to be made for the potential deadlocks, Correctness Criterion 1 has a higher priority than Correctness Criterion 2. The reason is that leaving the system in a deadlocked state is usually an uncontrollable fault whereas recovering from a false deadlock is a type of compensating action which impacts the system less severely. While this line of reasoning may not be true in some specific systems, our design of distributed deadlock detection algorithms for real-time systems will follow this criteria priority.

5. In many complex systems, especially in distributed real-time systems, structured (modular) and/or layered design approaches are used. Many deadlock detection or prevention algorithms only consider part of the system (i.e., a subset of the sites, the modules, and/or the layers of a system) as the problem domain and cannot detect or prevent deadlocks across related parts

of the system. By related parts of a system we mean the sites, the modules, and/or the layers of a system which constitute the environment in which a group of interacting tasks execute. For example, suppose an algorithm is designed for detecting deadlocks at the application layer with the assumption that a prevention strategy is used in the underlying system to provide a "deadlock free" environment. The prevention strategy used in the underlying system has no knowledge of the user application layer, but simply prevents tasks from circular waiting upon system resources. Suppose we have two concurrent tasks T_1 and T_2 in the system. At the application layer, T_1 is waiting for T_2 in Ada's rendezvous while at the system layer T_2 is waiting for T_1 upon a system resource. Both waiting situations are allowed to occur separately in the different layers of the system. From a global viewpoint, the blockings across these two layers actually form a deadlock cycle. Such deadlocks cannot be detected or prevented unless a "complete" strategy is adopted.

In addition to the correctness criteria discussed above, we must consider a number of performance issues relating to real-time requirements. One major question is whether deadlock detection is a feasible approach in a soft real-time system. After all, if a task is blocked in a deadlock, then it is likely to miss its deadline unless the deadlock algorithm is invoked soon enough to not only detect and resolve the deadlock, but to also leave enough time for this task (and possible the aborted tasks) to complete (even in the presence of subsequent "normal" blocking conditions). Consequently, deadlock detection for a given task should begin as a function of its deadline, D, remaining execution time, E, and the execution time cost for deadlock detection and resolution, DR. In other words, deadlock detection for this task should start *no later than* $D - E - DR$. It would also be advantageous if the aborted task(s) were able to be restarted and also *still* make their deadlines. For many real-time systems we can assume that D and E are known (or at least we have good approximations for them). On the other hand, the execution time cost of deadlock detection and resolution will not be known and will vary considerably depending on the graph representing the "waiting" states of the distributed system, the cost and delays involved in sending messages, and the application processing the nodes are performing in addition to the deadlock algorithm itself. Fortunately, experience has shown the most deadlock cycles are short, so it may be possible to develop a reasonable estimate for DR for the common deadlock case. Note that when the estimates are wrong, one or more tasks involved in the deadlock will miss its deadline and abort, and thereby deadlock will be broken. Real-time deadlock detection will be successful when it finds deadlock early enough, resolves deadlock and *more* and *higher value* tasks subsequently make their deadlines than otherwise would have without deadlock detection (i.e., using schemes such as simply using timeout and abort, or using an "always abort" a lower value task if it blocks

a high value task). Performance studies are required to determine which approach proves feasible in practice.

We must also consider resolution decisions based on real-time requirements. Sufficient information should be monitored and collected in order to make a good resolution decision to support real-time requirements. When collecting information to support real-time deadlock resolution, we need to consider: (1) the inter-dependency of the deadlocked tasks and their related tasks and (2) the timing dependency among deadlocked tasks. In (1), by related tasks we mean the tasks (not necessary involved in the deadlock) which rely on the success of a deadlocked task. If a deadlocked task is chosen as the victim to be aborted, it may result a cascading abort of its related tasks. The reason for (2) is that a deadlock is a situation of cyclic wait and breaking a deadlock may result an acyclic wait which, in turn, results in a timing dependency among the surviving tasks. Depending on where a deadlock is broken, different timing dependencies might be formed. For example, a cycle of deadlocked tasks $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$ is detected. Each of these tasks may reside at different sites. Their criticalness order is $T_1 < T_3 < T_2$. A simple resolution strategy may be to abort the least critical task, which is task T_1 in this example. This resolution creates a timing dependency that T_2 waits for T_3 to satisfy its request. Suppose T_3 cannot complete in time for T_2 to make its deadline. Consequently, both T_1 and T_2 will fail in this resolution. If the timing dependency among these deadlocked tasks has been considered in the resolution, selecting T_3 as the victim to resolve the deadlock might be a better decision in that both T_1 and T_2 might succeed. Therefore, in addition to the criticalness of each task, the timing dependency information is needed to make a better resolution decision (in the sense that it allows more of the surviving tasks to meet their timing constraints).

Reliability of the distributed deadlock detection algorithm is another major concern. The underlying communication subsystem usually can be assumed to be reliable, and a major concern of the reliability in distributed systems is how to deal with site failures. Site failures in a distributed system usually change the system state which, in turn, cause the GRG collected at each site to become inconsistent. A deadlock detection algorithm is not reliable if it cannot quickly recover the GRG from a inconsistent state due to site failures. To ensure that deadlock detection computations function properly after site failures, the GRG inconsistency must be corrected in finite time which basically can be achieved in one of the two ways: (1) inform the surviving sites of the failures to clean up inconsistent information or (2) make the inconsistent information obsolete in the view of newly initiated deadlock computations so that the inconsistency may fade away. In any case, time is needed to recover and correct the inconsistency, and the deadlock computations execute concurrently with a site failure recovery may not be able to function properly. Therefore, the reliability

criterion requires that the inconsistency be temporary and be corrected quickly so that only the deadlock detection computations in progress when the failures occur, may be affected.

7 Proposed Real-Time Distributed Deadlock Detection Algorithms

The two deadlock detection algorithms proposed in this section are our first attempt at dealing with timing constraints in distributed deadlock detection. Since there are complicated issues involved in the resolution of deadlocks in the distributed real-time systems, we simply assume the resolution of a detected deadlock is done by choosing a deadlocked task which declares the deadlock as the victim. Also, we only consider a deadline for each task as its real-time constraint to simplify the problem. More complex real-time constraints can be taken into account in a similar way.

These two algorithms can be used (1) in a general real-time system, where no other strategy is used, to deal with deadlocks, (2) in a deadlock free real-time system, where deadlock prevention strategy is used primarily, to increase system dependability, or (3) in the debugging phase of a distributed real-time system. As we pointed out in Section 6.3 that many "deadlock free" designs only deal with part of the system which cannot prevent deadlocks across related parts of a system. Therefore, the integrated approach is one of the major concerns in our design to increase the dependability of such "deadlock free" systems. Also, in such a "deadlock free" system, the occurrence of deadlocks are rare, therefore, efficiency is another important concern in our design. The probe based approach is used due to efficiency. A probe computation may be initiated only when there is a potential deadlock situation. If a task is waiting for its outstanding requests, there is a hint of potential deadlocks. A probe computation may be initiated after a task has waited for a period of time Δt. The Δt can be chosen as a function of a task's deadline and/or the average blocking time of a request. Therefore, in a system where deadlocks are rare, the frequency of deadlock detection probe computation can be kept as low as possible (it is limited by a function of deadlines in this case).

For simplicity and efficiency, our algorithms only initiate probe computation at most once for each idle task, and a probe is discarded whenever it finds no potential deadlocks (i.e., reaches a leaf vertex). For each idle task in a GRG, a probe computation may be initiated periodically or only once after it finds potential evidence of deadlocks. The periodic invocation of probe computations is not necessary in our algorithms since they ensure that at least one of the tasks in deadlock will declare deadlock in one invocation. Also, in a probe computation, a probe may be stored or

discarded whenever it reaches a leaf vertex in a GRG. A deadlock may be detected earlier if probes are not discarded but stored and forwarded later (when new edges are formed) at the leaf vertices. However, the stored probes may become obsolete and, hence, usually requires a more complicated mechanism to clean up them (e.g., the algorithms proposed in [58, 13, 14]). Also, this approach is relatively unreliable since site failures may cause the stored probes to become obsolete as well which, in turn, may cause the failure of deadlock detection computations if this situation is not taken into account carefully.

As stated in Section 6.3 it is very difficult in a distributed real-time system for a deadlock detection algorithm to fulfill the two correctness criteria. The two algorithms proposed in this paper only "attempt" to detect temporal deadlocks. It is assumed that most of the deadlocks are simple cycles. The two algorithms are designed to be efficient especially when detecting simple cycles. Therefore, the undetected temporal deadlocks can be minimized.

In these two algorithms optimizations are made based on efficiency concerns. The optimizations can reduce the probe overhead in terms of reducing the number of probe messages passed around as well as eliminating the possibility of repeated detection of a cycle (which means single point of detection for each deadlock). The deadlock resolution can also benefit from this single point of detection feature since no synchronization is necessary when resolving a deadlock.

Due to the concerns of simplicity and efficiency of the algorithms for these first two relatively simple and well defined deadlock models, we do not address the problems of livelocks, task termination, and orphan tasks, etc. In the solutions for the more complex deadlock models, such as the OR model and the C(n,k) model, it is necessary to pass around more information for deadlock detection. The solutions for the livelocks, task termination, and orphan tasks, etc., therefore, may be incorporated in a deadlock detection algorithm with little additional overhead.

In the following subsections, we first state our design assumptions before the presentation of the two algorithms.

7.1 Design Assumptions of the Algorithms

In developing the deadlock detection algorithms for distributed real-time systems, we made several assumptions. First, we assume that runtime tasking in the distributed environment is supported by a *Distributed Runtime Tasking Supervisors*[54] (DRTS's). Each of the nodes in a distributed system is equipped with a copy of DRTS. The DRTS's provide services by sending messages to each other. A DRTS could be a separate entity or embedded in the operating system (OS) or

kernel. Information concerning task interactions and synchronizations which are managed by the OS, kernel, or DRTS should be available for deadlock detection. For example, the local resource allocation status should be available in the local OS or kernel, the state of the inter-node task synchronization such as semaphore and wait/signal mechanisms should be provided by DRTS's.

Also, we assume that there is a deadlock detection agent at each node because (1) it is more efficient that local deadlock detection activities are performed in a single entity than in each of the involved tasks with message exchanges, (2) it is easier and more efficient that global deadlock detection activities are distinguished and only performed among agents, and (3) it is more secure and more knowledgeable to gather system wide information in a dedicated agent than in each of the user tasks. The agent may be a separate entity or embedded in the DRTS, OS, or kernel. Inter-node deadlock detection operations are performed by the agents which exchange information with each other.

Information concerning implicit task interactions and synchronizations should be supported both by the compiler and the runtime environment. For example, in Ada rendezvous semantics, the calling task is not provided in the accept statement. Without special compiler and runtime support, this feature makes the deadlock problem unsolvable. It is required that a correct and up-to-date GRG is built at runtime to support correct deadlock detection operations. One possible solution to accomplish this requirement is to ask the complier to provide extra data structure and program code (not explicitly programmed) for deadlock detection. The extra code provided by compiler is to be executed at runtime to maintain the deadlock detection related data structure. For example, two kinds of tables are to be built to support deadlock detection for the Ada accept statement. One *reachable entry calls* (REC) table for each task, and one *possible calling tasks* (PCT) table for each entry point declared in a task. Initial values for these tables should be entered by the compiler. Program code for maintaining the REC table should be inserted at the proper places in a task by the compiler. Each task, therefore, can update its REC table whenever it is necessary at runtime. When a task is blocked by an accept statement at an entry point, the deadlock detection agent is triggered to search every REC table in every possible calling task which is listed in the PCT table of that entry point. An edge is added to the GRG if there is a matched reachable entry call in a possible calling task.

In real-time systems a timing constraint is usually associated with a task. The runtime system should be able to detect a missed deadline and abort the task. This deadline information is not available from Ada. As summarized in Section 3 other aspects of real-time processing are supported by Ada's *selective wait* statement. Using a combination of the delay statement and the

else alternative in Ada's **select** statement, one can provide an escape in the event that no open alternatives exist or that open alternatives are unduly delayed in their selection. These delays and the ways to terminate them have an effect on whether the task makes its deadline. Similarly, using Ada's *timed* or *conditional* entry calls, a calling task can ensure that it will not be blocked forever impacting its ability to make its deadline. Primarily, we are concerned with task deadlines, and to meet a task's timing constraints, time-out durations are associated with the timed entry calls for the task calling an entry and the delay alternative in the selective wait statement for the task which is waiting for an entry call. How to pick up an appropriate time-out duration for each operation (a rendezvous attempt or a resource request) is beyond the scope of this paper. A simple choice which is assumed in the following discussion is to set the time-out of an operation by the task deadline which, of course, means that if it times out it will not make the deadline.

7.2 Algorithm for the Single-Resource Model

In this section, a simple probe algorithm that deals with the Single-Resource deadlock model in distributed systems is presented. This algorithm is able to detect all the stable deadlocks and attempts to detect temporal deadlocks. Spontaneous time-outs and aborts may occur and may cause false detection of temporal deadlocks. False detection of temporal deadlocks are minimized by attaching a deadline to each of the probes. If all the system clocks are perfectly synchronized and all the deadlines attached to the probes are absolutely accurate, the false detection of temporal deadlocks is eliminated. Also, a temporal deadlock may not be detected if the timing constraint in a cycle is so tight that none of the in-cycle probes can finish travelling through the cycle in time. An undetected temporal deadlock is resolved automatically when a task in the cycle times out or aborts. This spontaneous time-out or abort, however, may not be the best resolution of a temporal deadlock.

For the Single-Resource model, the basic idea of using probes in deadlock detection is to initiate a probe whenever a task is blocked by a pending request. Probes are propagated backward along the edges of a GRG and are discarded when they reach end vertices. If a probe comes back to its initiator, a deadlock is found. This method, although it guarantees the detection of all deadlocks, may detect a single deadlock cycle multiple times if different tasks in a cycle initiate probes almost simultaneously. This method has two drawbacks: (1) it is inefficient in terms of message overhead and (2) it is complicated to recover a deadlock due to multiple detection of a cycle. By using a technique similar to that used in the Mitchell and Merritt's algorithm[48], our algorithm can reduce the number of probe messages and achieve a single point of detection of every deadlock cycle. Each

```
type PROBE_ID_TYPE is range 0..INTEGER'LAST;
type TASK_ID_TYPE is range 0..INTEGER'LAST;
type PROBE_TYPE is
    record
        probe_id : PROBE_ID_TYPE := 0; -- probe id
        initr_id : TASK_ID_TYPE := 0;  -- the task_id of the probe initiator
        probe_dl : DURATION := 0.0;    -- probe deadline is determined by the earli-
                                       -- est timing constraint in its travelling path
    end record;
```

Figure 2: Data structure for probes in the Single-Resource algorithm.

probe is assigned a timestamp (the probe_id) which is a number greater than the largest timestamp that a task and its waiting resource have ever seen. Each vertex in a GRG memorizes the largest probe timestamp it has propagated. The probes which are allowed to pass through a vertex are in increasing timestamp order. Consequently, only the probe with the largest timestamp (it is likely to be the latest probe initiated by the task which closes the cycle) is allowed to go through the whole cycle and declares the deadlock. Each probe is associated with a deadline (the probe_dl). The probe deadline is defined as the earliest task deadline that a probe has ever seen. A probe misses its deadline if at least one of the tasks it visited misses the deadline. Therefore, a probe is discarded immediately if it is found to miss its deadline.

The data structures for the probes, tasks, and resources are defined in the Figures 2, 3, and 4, respectively. In Figure 2 the fields probe_id and initr_id give each probe an unique identification. A probe is said to be larger than another one if it carries a larger probe_id. The larger initr_id is used to distinguish between two probes with the same probe_id. The deadline of a probe is defined by the field probe_dl. Figure 3 shows the data structure for tasks, which may be part of a *task control block* or may be a separate data structure dedicated for deadlock detection. Two buffers are prepared for storing probes for each task: probe_init stores its own initiated probe and probe_bufd stores the largest probe ever received. Figure 4 defines the data structure for resources. Only one probe buffer is prepared for storing the largest probe ever received for each resource since no probe may be initiated by resources.

Probes are only initiated by the tasks when they become BLOCKED from the ACTIVE state (or after a period of time Δt which is chosen as a function of a task's deadline and/or the average blocking time of a request). The **procedure** TASK_INIT_PROBE depicted in Figure 5 describes

```
type TASK_STATE_TYPE is (ACTIVE, BLOCKED);
type TASK_TYPE is
    record
        task_id : TASK_ID_TYPE := 0;
        task_state : TASK_STATE_TYPE := ACTIVE;
        probe_init : PROBE_TYPE;          -- probe initiated
        probe_bufd : PROBE_TYPE;          -- probe buffered
        resource_table : RES_TABLE_TYPE;  -- resources held by the task
    end record;
```

Figure 3: Data structure for tasks in the Single-Resource algorithm.

```
type RESOURCE_ID_TYPE is range 0..INTEGER'LAST;
type RESOURCE_STATE_TYPE is (FREE, HELD);
    -- For a consumable resource, it is FREE if it is produced but is not consumed
    -- yet; on the other hand, it is HELD by its producer if it is requested but is
    -- not produced yet.
type RESOURCE_TYPE is
    record
        resource_id : RESOURCE_ID_TYPE := 0;
        resource_state : RESOURCE_STATE_TYPE := FREE;
        probe_bufd : PROBE_TYPE;     -- probe buffered
        waiting_queue : QUE_TYPE;    -- waiting queue for the resource
    end record;
```

Figure 4: Data structure for resources in the Single-Resource algorithm.

```
procedure TASK_INIT_PROBE (T: in out TASK_TYPE,
                           R: in RESOURCE_TYPE) is
    -- This procedure is invoked when a task T requested a resource R which is
    -- not FREE. The task T is in transition from ACTIVE state into BLOCKED
    -- state. It requests the probe from the waited resource R.probe_bufd. A
    -- period of waiting time Δt which is chosen as a function of task T's deadline
    -- and/or the average blocking time of a request might be inserted right before
    -- the calling of this procedure.
    resource : RESOURCE_TYPE;
begin
    -- prepare a new probe
    T.probe_init.probe_id := MAX(R.probe_bufd.probe_id,
                                 T.probe_bufd.probe_id) + 1;
        -- function MAX(a,b) returns the maximum value of a and b
    T.probe_init.initr_id := T.task_id;
    T.probe_init.probe_dl := (deadline of the operation);

    -- put the new probe in probe_bufd
    T.probe_bufd := T.probe_init;

    -- propagate the new probe to all the resources it holds
    for resource in T.resource_table loop
        SEND (T.probe_bufd, resource);
    end loop;
end TASK_INIT_PROBE;
```

Figure 5: Procedure for probe initiation in the Single-Resource algorithm.

how a probe is initiated. The newly created probe has the largest probe_id ever received by its creator. The deadline of the probe is initially set according to its creator's timing constraints. The newly created probe is, then, treated as the largest probe ever received and is propagated accordingly.

When a task receives a probe, it invokes the **procedure** TASK_RCV_PROBE described in Figure 6. This algorithm guarantees that only tasks in the BLOCKED state may receive probes since probes are propagated in the reverse direction along the edges in GRG. The received probe is, first, checked to see if it has missed its deadline. If so, it is discarded immediately because at least one task in the path that the probe traveled has timed out or was aborted at the time the probe is received. If the probe is still valid, it is checked whether it is initiated by the receiving task. If

```
procedure TASK_RCV_PROBE (T: in out TASK_TYPE; P: in PROBE_TYPE) is
    -- This procedure is invoked whenever a task T receives a probe P.
begin
    if (P.probe_dl <= current_time) then
        -- the received probe missed its deadline
        null; -- discard the received probe
    elsif ( (P.probe_id = T.probe_init.probe_id) and
            (P.initr_id = T.task_id) ) then
        a deadlock is found;
    elsif ( ( P.probe_id > T.probe_bufd.probe_id ) or
            ( ( P.probe_id = T.probe_bufd.probe_id ) and
              ( P.initr_id > T.probe_bufd.initr_id ) ) ) then
        -- update its deadline if necessary and put it in probe_bufd and propagate it
        if (P.probe_dl > deadline of T's operation) then
            P.probe_dl := (deadline of T's operation);
        end if;
        T.probe_bufd := P;
        for R in T.resource_table loop
            SEND (P, R);
        end loop;
    else
        null; -- discard the received probe
    end if;
end TASK_RCV_PROBE;
```

Figure 6: Procedure for tasks handling received probes in the Single-Resource algorithm.

so, a deadlock (may either be a stable deadlock or a temporal deadlock) is found. Otherwise, the probe is checked to see if it is the largest probe ever received. If so, the deadline of the probe is updated, if necessary, and then the probe is propagated to all the resources held by the task.

When a resource receives a probe, it invokes the **procedure** RESOURCE_RCV_PROBE shown in Figure 7. This algorithm guarantees that only HELD resources may receive probes since probes are propagated in the reverse direction from a BLOCKED task to all its HELD resources. In the Single-Resource model, a resource can only be held exclusively by one task and does not initiate any probes. It is not necessary to detect deadlocks at a resource vertex. The probe at the resource vertex, therefore, is only checked to see if it has missed its deadline. If so, the probe is discarded; otherwise, it is propagated to all the tasks waiting for that resource.

```
procedure RESOURCE_RCV_PROBE (R: in out RESOURCE_TYPE;
                              P: in PROBE_TYPE) is
    -- This procedure is invoked whenever a resource R receives a probe P.
begin
    if (P.probe_dl <= current_time) then
        -- the received probe missed its deadline
        null; -- discard the received probe
    else
        -- put it in probe_bufd and propagate it
        R.probe_bufd := P;
        for T in R.waiting_queue loop
            SEND (P,T);
        end loop;
    end if;
end RESOURCE_RCV_PROBE;
```

Figure 7: Procedure for resources handling received probes in the Single-Resource algorithm.

Initially, all tasks are ACTIVE, all reusable resources are FREE, and all consumable resources are HELD by the producers. In Ada, synchronization between two tasks occurs when the task issuing an entry call and the task accepting an entry call are ready to establish a rendezvous. A rendezvous is a consumable resource. Either one of the calling and called tasks arriving at the rendezvous first will wait, and, hence, becomes the "consumer" of the "rendezvous." The second task which establishes a rendezvous is always the "producer" of the "rendezvous." After a rendezvous is established, the calling task becomes BLOCKED while the called task is executing corresponding statements following the **accept** statement. The called task, therefore, is ACTIVE and acts as the producer during the rendezvous period.

When a task is in transition from the ACTIVE state to the BLOCKED state, it adds an edge to the GRG and executes the **procedure** TASK_INIT_PROBE to initiate a deadlock detection probe. When a task becomes ACTIVE from the BLOCKED state, it deletes the corresponding edges from the GRG. Resources are the passive entities in the GRG which will not initiate deadlock computation. If resources are eliminated and a TWFG is considered, the correctness of this algorithm still holds.

A GRG can be implemented as a two dimensional matrix. One dimension represents tasks, and the other dimension represents resources. Each of the elements in a GRG matrix represents one of the following three states: (1) the task is waiting for the resource, (2) the resource is held by the task, or (3) there is no relationship between them. Another possible implementation of GRG is to

store the information in each of the task tables and resource tables, for example, the resource_table in TASK_TYPE and the task waiting_queue in RESOURCE_TYPE used in our algorithm data structure.

An agent is assumed to handle the deadlock detection activities at each site. The probe SEND procedure, which is not described explicitly in the algorithm, is assumed to be handled by the agent. A simple copy operation can accomplish a local SEND operation, while a real message will be sent out for an inter-site SEND operation. The procedures TASK_INIT_PROBE, TASK_RCV_PROBE, and RESOURCE_RCV_PROBE are designed to be executed by the agent on behalf of each task or resource.

7.3 Algorithm for the AND Model

In this section, we propose a set-based probe algorithm that deals with the AND deadlock model in distributed real-time systems. This algorithm is able to detect all the stable deadlocks and attempts to detect temporal deadlocks. Spontaneous time-outs and aborts are allowed if they are needed for timing constraints. Since cycles may be nested, spontaneous aborts can also occur in stable deadlock cycles. Consequently, false detection of deadlocks are possible in stable deadlock cycles as well as in temporal deadlock cycles.

For the AND model, the basic idea of using probes in deadlock detection is to initiate a probe when a task becomes blocked if not all of its requests are granted, or when a task is granted one of its pending requests but remains blocked. Probes are propagated either forward or backward along the edges of a GRG and are discarded when they reach the end vertices. If a probe revisits a task, a deadlock is found. Again, this method is inefficient and may cause multiple detections of a single deadlock. In the algorithm we proposed for the Single-Resource model, we solve the problem by propagating probes backward and using the probe timestamps. These techniques can also be applied to the algorithms for the AND deadlock model. A GRG in the AND model is symmetric in the sense that multiple incoming and outgoing edges of a vertex are allowed. Therefore, the probe propagation direction does not make any difference. However, if we assume that most of the resources are not shared and most of the tasks do not make multiple requests, the backward propagation is preferred for the AND model algorithms. Unfortunately, the choice of the backward probe propagation conflicts with the optimization made with the probe timestamps when timing constraints are considered. We will discuss this issue later on.

Similar to our Single-Resource algorithm, each of the probes is assigned a timestamp which is an integer value greater than the largest timestamp that a task and its granted resources (or the

resources it is waiting for if probes are propagated backward) have ever seen. Again, the probes which are allowed to pass through a vertex are in increasing timestamp order. Since the foreign probes may enter a cycle in the AND model and interfere with the in-cycle probes, it is required that every probe (either in-cycle probes or foreign probes) should be able to detect deadlocks. More information, therefore, is needed for the foreign probes to determine the existence of a cycle. The notion of set-based probe was first proposed by Chandy and Misra[10] and followed by Haas and Mohan[28]. In Haas and Mohan's algorithm, a probe carries a *set* of permanent blocking edges that has been known to the probe. The probes are propagated in the forward direction along the edges of a GRG. Upon receiving a probe, each task searches for cycles that involve itself and deletes the edges related to the detected cycles from the set. If the remaining set is not empty, the task will append itself to the set and propagate it to the tasks it is waiting for. The set grows as it reaches more and more tasks and shrinks when cycles are detected.

In the algorithm proposed here, each probe includes a set of edges which only contains the path travelled by the probe. A set is an one-dimensional chain in our algorithm as opposed to a tree-like structure sub-GRG in the previous algorithms. The original motivation to propagate a tree-like set in each of the probes is to discover all cycles that involve a deadlocked task which can then act as a deadlock resolver. If deadlock resolution is taken into consideration, some of the detected cycles might have been broken (false deadlocks) due to the fact that cycles may be nested in the AND model. When a deadlocked task knows all cycles that it is involved with, this only reduces the false detection of deadlocks. On the contrary, if only chain-like set probes are propagated in detecting cycles, each deadlocked task will detect at most one cycle at a time. The remaining deadlock cycles, if they exist, will be detected as soon as all the involved tasks are searched by a probe. Unlike a tree-like set algorithm, in which a deadlock resolution is delayed until a task can determine it has detected all the cycles it involves, our algorithm attempts to resolve deadlocks as soon as it is detected. The probability of related false detection of deadlocks will be reduced since the detected deadlocks are resolved as soon as possible. Also, processing and propagation of the tree-like set probes are more costly compared to the chain-like set probes. Therefore, our algorithm can avoid some false detection of deadlocks comparable to the previous algorithms, while providing better efficiency. Consequently, in real-time applications where timing constraints are important, a chain-like set probe algorithm is more attractive than a tree-like set probe algorithm.

Different from the Single-Resource algorithm, only part of a probe's trace may form a cycle. The timing constraints can no longer be associated with the probes but should be attached to the tasks in the chain-like set transferred along with the probes. When searching for cycles in the set, the timing constraints attached to each of the tasks are also evaluated. A deadlock is found if none

of the tasks which form the cycle has missed its deadline. A chain may be broken at a task in the chain if the task is found missed its deadline. Also, the tasks in the chain dependent on the one that missed its deadline should be discarded. This is because the wait-for information may have been changed when the task which missed deadline aborted and released its resources.

If a probe is propagated backward, the chain grows by adding new dependents to the chain. A new dependent is impossible to be added to the chain if the chain is broken. Therefore, a backward propagated probe should be discarded if its chain is broken. Consequently, the algorithm may fail to detect the deadlock which is supposed to be searched and declared by the discarded probe. For example, consider a chain $T_i \rightarrow R_j \rightarrow T_k \rightarrow \cdots \rightarrow T_m \rightarrow \cdots \rightarrow R_x \rightarrow T_y$ is propagated along with a probe. If the task T_m is found to miss its deadline, the chain is broken at T_m and its left hand side of the chain $T_i \rightarrow R_j \rightarrow \cdots \rightarrow T_m$ is discarded. Since the probe is propagated backward, a new entry (a dependent of T_i) should be added to the left hand side of T_i which is no longer exists in the chain; the whole probe, therefore, should be thrown away. Suppose at the time the probe is discarded, a cycle $T_k \rightarrow R_l \rightarrow T_i \rightarrow R_j \rightarrow T_k$ exists and all the other probes are eliminated due to their smaller timestamps. This deadlock may not be detected if no new probes with a larger timestamp could possibly reach this cycle. Consequently, backward probes cannot be used when timing constraints are considered.

In contrast, if the probe is propagated forward along the directed edges, the new entries are added to the right hand side of T_y and the probe, after discarding its invalidated part of the chain, can continue to search the GRG until it reaches an end vertex (an active task) or finds a cycle. In other words, the forward propagated probe avoids the error that the backward probe exhibits.

A GRG is a bipartite graph that vertices are divided into two disjoint subsets, a set of resources vertices and a set of task vertices, such that there are no edges connecting vertices from the same subset. The graph may be simplified by eliminating one subset of the vertices. The subset of the resource vertices may be eliminated in the GRG by replacing an assignment edge (or a producer edge) and a request edge pair attached to a resource vertex with a single directed edge between two tasks. For example, $T_i \rightarrow R_j \rightarrow T_k$ can be simplified to $T_i \rightarrow T_k$ if the resource vertex R_j is not necessary in the graph. If all the resource vertices are eliminated, it becomes a task-wait-for graph (TWFG). In the algorithm presented, we ignore the resource vertices in the chain propagated along with the probes since we are not interested in detecting deadlocks at the resource vertices in a GRG. This simplification can reduce the size of the probe messages.

The data structures for the probes, tasks, and resources are defined in the Figures 8, 9, and 10, respectively. In Figure 8, the fields probe_id and initr_id are defined in the same way as those in the

```
type PROBE_ID_TYPE is range 0..INTEGER'LAST;
type TASK_ID_TYPE is range 0..INTEGER'LAST;
type TASK_PTR; -- point to a task in a chain
type PROBE_TYPE is
    record
        probe_id : PROBE_ID_TYPE := 0; -- probe identification
        initr_id : TASK_ID_TYPE := 0;    -- the task_id of the probe initiator
        chain_head : TASK_PTR := null;
            -- a chain of tasks which records the path of the probe;
            -- the chain_head points to the head of the path
    end record;
```

Figure 8: Data structure for probes in the AND algorithm.

algorithm for the Single-Resource model. A set of task_id's which record the path of the probe are chained together to propagate along with the probes. The field chain_head points to the head of such a path. Figure 9 shows the data structure for tasks. Two buffers are prepared for storing probes for each task: the probe_init stores its own initiated probe and the probe_bufd stores the largest probe ever received. Also, the data structure for chained task is defined as CHAINED_TASK. In the CHAINED_TASK, each task_id is attached with a task_dl (task deadline). Figure 10 defines data structure for the resources. Only one probe buffer is prepared for storing the largest probe ever received for each resource since no probe may be initiated by resources.

Probes are only initiated by the tasks when one becomes BLOCKED from the ACTIVE state or when a BLOCKED task is granted one of its pending requests and remains BLOCKED. A period of time Δt which is chosen as a function of a task's deadline and/or the average blocking time of a request may be inserted right before the initiation of a new probe. The **procedure** TASK_INIT_PROBE depicted in Figure 11 describes how a probe is initiated. A probe chain is created and is accessed through the chain_head in the new probe. The new probe is stored both in probe_init and probe_bufd, and is propagated to each resource in its pending_table (pending request table).

When a BLOCKED task receives a probe, it invokes the **procedure** TASK_RCV_PROBE described in Figure 12. The probes received by ACTIVE tasks are simply discarded. In the received probe, the head of the chain is treated separately because if it misses its deadline, the whole chain is thrown away and the probe will not be propagated. The cycle detection is done by search the current task id in the chain starting from the head of the chain. The deadlines are also checked for

```
type TASK_STATE_TYPE is (ACTIVE, BLOCKED);
type TASK_TYPE is
    record
        task_id : TASK_ID_TYPE := 0;
        task_state : TASK_STATE_TYPE := ACTIVE;
        probe_init : PROBE_TYPE;          -- probe initiated
        probe_bufd : PROBE_TYPE;          -- probe buffered
        holding_table : RES_TABLE_TYPE;   -- resources held by the task
        pending_table : RES_TABLE_TYPE;   -- pending requests of the task
    end record;
type CHAINED_TASK; -- a task in a chain
type TASK_PTR is access CHAINED_TASK;
type CHAINED_TASK is
    record
        task_id : TASK_ID_TYPE := 0; -- task id
        task_dl : DURATION := 0.0; -- task deadline
        next : TASK_PTR; -- pointer link to the next task in chain
    end record;
```

Figure 9: Data structure for tasks in the AND algorithm.

```
type RESOURCE_ID_TYPE is range 0..INTEGER'LAST;
type RESOURCE_STATE_TYPE is (FREE, HELD);
    -- For a consumable resource, it is FREE if it is produced but is not consumed
    -- yet; on the other hand, it is HELD by its producer if it is requested but is
    -- not produced yet.
type RESOURCE_TYPE is
    record
        resource_id: RESOURCE_ID_TYPE:=0;
        resource_state : RESOURCE_STATE_TYPE := FREE;
        probe_bufd : PROBE_TYPE;     -- probe buffered
        waiting_queue : QUE_TYPE;    -- waiting queue for the resource
        granted_table : TASK_TABLE_TYPE;  -- granted tasks of the resource
    end record;
```

Figure 10: Data structure for resources in the AND algorithm.

```
procedure TASK_INIT_PROBE (T: in out TASK_TYPE) is
    -- This procedure is invoked when an ACTIVE task T requests resources which
    -- are not all FREE, or when a BLOCKED task is granted one of its pending
    -- requests but remains BLOCKED. A period of waiting time Δt which is
    -- chosen as a function of task T's deadline and/or the average blocking time
    -- of a request might be inserted right before the calling of this procedure.
    R : RESOURCE_TYPE;
begin
    -- prepare a new probe
    T.probe_init.probe_id := T.probe_bufd.probe_id
    for R in T.holding_table loop
        T.probe_init.probe_id := MAX(R.probe_bufd.probe_id,
                                     T.probe_init.probe_id);
        -- function MAX(a,b) returns the maximum value of a and b
    end loop;
    T.probe_init.probe_id := T.probe_init.probe_id + 1;
    T.probe_init.initr_id := T.task_id;
    T.probe_init.chain_head := new TASK_PTR
        (T.task_id, <deadline of the operation>, null);
    -- put the new probe in probe_bufd
    T.probe_bufd := T.probe_init;
    -- propagate the new probe to all the resources it is waiting for
    for R in T.pending_table loop
        SEND (T.probe_bufd ,R);
    end loop;
end TASK_INIT_PROBE;
```

Figure 11: Procedure for probe initiation in the AND algorithm.

```
procedure TASK_RCV_PROBE (T: in out TASK_TYPE;
                          P: in out PROBE_TYPE) is
    -- This procedure is invoked when a BLOCKED task T receives a probe P.
    ptr : TASK_PTR;     found : BOOLEAN := FALSE;
begin
    ptr := P.chain_head;
    if ptr.task_dl <= current_time then -- the head missed its deadline
        null; -- discard the received probe
    elsif ptr.task_id = T.task_id then
        a deadlock is found;
    else
        -- search the current task in the chain
        while not found and then ptr.next /= null loop
            if (ptr.next.task_dl <= current_time) then
                ptr.next := null; -- discard the rest of the chain
            else
                ptr := ptr.next;
                if ptr.task_id = T.task_id then
                    found := TRUE;
                end if;
            end if;
        end loop;
        if found then
            a deadlock is found;
        elsif ((P.probe_id > T.probe_bufd.probe_id) or else
               ((P.probe_id = T.probe_bufd.probe_id) and
                (P.initr_id > T.probe_bufd.initr_id))) then
            -- append the task T to the head of the chain
            P.chain_head.next := new TASK_PTR
                (T.task_id, T.probe_init.probe_id, P.chain_head);
            T.probe_bufd := P; -- put it in probe_bufd
            for R in T.pending_table loop
                SEND (P,R); -- propagate the probe to each
            end loop;       -- resource in T's pending_table
        else
            null; -- ignore the received probe
        end if;
    end if;
end TASK_RCV_PROBE;
```

Figure 12: Procedure for tasks handling received probes in the AND algorithm.

```
procedure RESOURCE_RCV_PROBE (R: in out RESOURCE_TYPE;
                              P: in PROBE_TYPE) is
    -- This procedure is invoked when a resource R receives a probe P.
begin
    if ((P.probe_id > R.probe_bufd.probe_id) or else
       ((P.probe_id = R.probe_bufd.probe_id) and
        (P.initr_id > R.probe_bufd.initr_id))) then
         R.probe_bufd := P; -- put it in probe_bufd
         for T in R.granted_table loop
              SEND (P,T);
         end loop;
    else
         null; -- ignore the received probe
    end if;
end RESOURCE_RCV_PROBE;
```

Figure 13: Procedure for resources handling received probes in the AND model.

each task in the chain. If an expired task is found, the un-searched part of the chain is disconnected. Similar to the Single-Resource algorithm, if no cycle is found, the probe is checked to see if it is the largest probe ever received. If so, the current task is appended to the head of the chain, and the probe is propagated to all the resources which the task is waiting for.

When a resource receives a probe, it invokes the **procedure** RESOURCE_RCV_PROBE shown in Figure 13. The resource simply propagates the probe to all the tasks in its granted_table if it is the largest probe ever received.

Again, an agent is assumed to handle the deadlock detection activities at each site, therefore, the intra-site probe SEND can be achieved by a simple copy operation. The intra-site chain transfer operation can be done by simply copying its pointer. A real message will be sent out for an inter-site SEND operation.

There are two weak points of this algorithm. First, the detection of a deadlock may not be done in a limited period of time. This is because the foreign probes with increasing timestamps may keep on interfering with each other until one eventually travels through the whole cycle. The detection of an existing deadlock, therefore, may be infinitely delayed. This situation is more likely to happen in a complicated GRG. However, the statistical analyses, such as the one done by Gray et al.[26], have shown that most of the deadlocks are simple cycles with a length of two to three vertices

involved. This implies that the chance of infinitely delay of a deadlock detection is rare, and in many systems it may be justifiable to live with this rare occurrence in order to take advantage of the optimization made with the probe timestamps. Secondly, the resolution of a detected deadlock is limited to the abortion of the task which declares the deadlock. If more information is carried with each of the probes, such as the priorities of the tasks in the chain, the algorithm may be able to declare the deadlock at the task which is going to be chosen as the victim for the resolution. The priority may be defined to reflect any combination of the *criticalness, timing constraint, task processing time, laxity, amount of I/O completed*, etc. Also, if the priority considers the degree of forward and backward dependency of the task in GRG, the false detection of deadlocks due to the existence of nested cycles may be further minimized. However, more overhead in terms of the size and the number of the probe messages is required.

8 Summary

This paper has two main parts. First, we completed a state-of-the-art survey of the distributed deadlock detection algorithms proposed in the literature. Second, we extended the deadlock problems into real-time systems and developed solutions for them.

The survey work can be divided into four subsections:

1. *Graph Theory*: Based on the complexity of the underlying system three widely used wait-for graphs – TWFG (Task Wait-For Graph), TRG (Task-Resource Graph), and GRG (General Resource Graph) are summarized.

2. *Distributed Concurrent Programming Systems*: It is clear that deadlock can be formally studied in isolation by using graph theory. However, in this work we are interested in explicitly tying the formal properties of deadlock algorithms directly to the actual languages and systems that need to use the theory. To bridge the gap between the theory and its application we have to understand the property of the underlying system. Since Ada is based on notions of Hoare's "Communicating Sequential Processes", and Brinch Hansen's "Distributed Processes", we summarized the concurrency, synchronization, and resource allocation models of all these three concurrent programming environments.

3. *Deadlock Models*: With the help of deadlock models, we are able to understand and analyze the complexity of the deadlock problems. Three deadlock model approaches are summarized:

(i) the resources/communication deadlock model, (ii) the general resource system model proposed by Holt[33], and (iii) a hierarchy of deadlock models proposed by Knapp[39]. Among them, the approach (iii) is the most comprehensive one and is extensively used in our study. This hierarchical set of deadlock models consists of the Single-Resource model, the AND model, the OR model, the AND-OR model, the C(n,k) model, and the Unrestricted model. They are characterized by the restrictions that are imposed upon the form the resource requests can assume, and range from very restricted request forms to models with no restrictions whatsoever.

4. *Distributed Deadlock Detection and Resolution Algorithms*: According to Singhal[57], the deadlock detection algorithms are classified into three types: centralized, distributed, and hierarchical approaches. This classification is based on the way the system state information is maintained. Knapp[39] further classify the distributed algorithms into four classes: path-pushing algorithms, edge-chasing algorithms, diffusing computations, and global state detection, according to the methodology used in searching for deadlocks. Many important algorithms are studied and summarized in these categories.

In the literature, the "stable property" is an important notion of the deadlock problem. This property means a deadlock situation persists once it is formed. Many algorithms proposed in the literature assume and utilize this property. In real-time systems, timing constraints are attached to the tasks. A task may time out from a state in which it is waiting. Deadlocks may not be stable if timing constraints are considered. In this paper, we described the deadlock problems for real-time systems in which timing constraints are considered. Three types of problems: "Stable Deadlock," "Temporal Deadlock," and "Non-deadlocked Blocking" are identified and discussed.

We adopt Knapp's hierarchy of deadlock models for the analysis of the problem complexity. Based on the sufficient conditions for deadlock detection, we roughly divide the problem into four levels of complexity: (i) the Single-Resource model, (ii) the AND model, (iii) the OR model, and (iv) the AND-OR and the C(n,k) models. To fully support Ada semantics it is necessary to develop solutions for the most complex level. Since many Ada applications do not utilize all the features that Ada provides, the deadlock problem may be simplified by imposing certain restrictions on the use of Ada. We have indicated how Ada features are related to certain levels of deadlock problem complexity, and how the deadlock problem could be simplified if the use of certain Ada features are restricted. In each of the four complexity levels, we also address how the deadlock problems can be solved and how the timing constraints are considered in the possible solutions.

After discussing the issues and needed solutions in general, we then provide two algorithms; one for the Single-Resource model and one for the AND model. Both algorithms are able to detect stable deadlocks and attempt to detect temporal deadlocks. One unique aspect of these algorithms is their ability to address timing constraints of tasks. Both algorithms are based on probes to detect deadlock cycles. Probe message overheads are optimized by carefully choosing a probe propagation direction and imposing a probe propagation rule. In the algorithm developed for the Single-Resource model we have shown that the backward probe propagation is the best choice. Also, in the algorithm developed for the AND model, backward probe propagation is preferred if no timing constrains or no other optimizations cause conflicts with this choice. In both algorithms probes are assigned with timestamps. The probes which are allowed to pass through a vertex are in increasing timestamp order. This probe propagation rule can greatly reduce the probe overhead and ensure the single detection of deadlock cycles. This rule, however, conflicts with backward probe propagation in the AND algorithm if timing constraints are considered. Consequently, our algorithm for the AND model is forced to use forward probe propagation. Also, we point out that imposing this rule may cause an unlimited delay of the detection of certain deadlocks. However, this situation may be so rare that it is still justifiable to take advantage of the optimization made with the probe timestamps.

Our future work includes developing complete algorithms for the next two levels of complexity, formally proving all four algorithms correct, and implementing and evaluating the performance of the algorithms on our current real-time database testbed called RT-CARAT[36].

References

[1] K. R. Apt, "Correctness Proofs of Distributed Termination Algorithms," *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 3, pp. 388–405, July 1986.

[2] K. R. Apt and N. Francez, "Modeling the Distributed Termination Convention of CSP," *ACM Transactions on Programming Languages and Systems*, Vol. 6, No. 3, pp. 370–379, July 1984.

[3] D. Z. Badal, "The Distributed Deadlock Detection Algorithm," *ACM Transactions on Computer Systems*, Vol. 4, No. 4, pp. 320–377, November 1986.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[5] G. Bracha and S. Toueg, "A Distributed Algorithm for Generalized Deadlock Detection," In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, pp. 285–301, ACM SIGACT-SIGOPS, Vancouver, B.C., Canada, August 1984.

[6] G. Bracha and S. Toueg, "Distributed Deadlock Detection," *Distributed Computing*, Vol. 2, No. 3, pp. 127–138, December 1987.

[7] P. Brinch Hansen, "Distributed Processes: A Concurrent Programming Concept," *Communications of the ACM*, Vol. 21, No. 11, pp. 934–941, November 1978.

[8] K. M. Chandy, J. Misra, and L. M. Haas, "Distributed Deadlock Detection," *ACM Transactions on Computer Systems*, Vol. 1, No. 2, pp. 144–156, May 1983.

[9] K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems*, Vol. 3, No. 1, pp. 63–75, February 1985.

[10] K. M. Chandy and J. Misra, "A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems," In *Proceedings ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 157–164, Ottawa, Ontario, Canada, August 1982.

[11] M. Chandy and J. Misra, "An Example of Stepwise Refinement of Distributed Programs: Quiescence Detection," *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 3, pp. 326–343, July 1986.

[12] A. N. Choudhary, W. H. Kohler, J. A. Stankovic, and D. Towsley, *Performance Evaluation of Two Distributed Deadlock Detection Algorithms*, COINS Technical Report 89-13, University of Massachusetts at Amherst, November 1988.

[13] A. N. Choudhary, W. H. Kohler, J. A. Stankovic, and D. Towsley, "A Modified Priority Based Probe Algorithm for Distributed Deadlock Detection and Resolution," *IEEE Transactions on Software Engineering*, Vol. 15, No. 1, pp. 10–17, January 1989.

[14] A. N. Choudhary, W. H. Kohler, J. A. Stankovic, and D. Towsley, "Correction to "A Modified Priority Based Probe Algorithm for Distributed Deadlock Detection and Resolution"," *IEEE Transactions on Software Engineering*, Vol. 15, No. 12, p. 1644, December 1989.

[15] I. Cidon, J. M. Jaffe, and M. Sidi, "Local Distributed Deadlock Detection with Finite Buffers," In *IEEE INFORCOM '86*, pp. 478–487, Miami, Florida, April 1986.

[16] S. Cohen and D. Lehmann, "Dynamic Systems and Their Distributed Termination," In *Proceedings ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 29–33, Ottawa, Ontario, Canada, August 1982.

[17] E. W. Dijkstra, "Co-operating Sequential Processes," In F. Genuys, editor, *Programming Languages*, pp. 43–112, Academic Press, New York, 1968.

[18] E. W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren, "Derivation of a Termination Detection Algorithm for Distributed Computations," *Information Processing Letters*, Vol. 16, No. 5, pp. 217–219, June 1983.

[19] E. W. Dijkstra and C. S. Scholten, "Termination Detection for Diffusing Computations," *Information Processing Letters*, Vol. 11, No. 1, pp. 1–4, August 1980.

[20] A. K. Elmagarmid, "A Survey of Distributed Deadlock Detection Algorithms," *SIGMOD RECORD*, Vol. 15, No. 3, pp. 37–45, September 1986.

[21] A. K. Elmagarmid, N. Soundararajan, and M. T. Liu, "A Distributed Deadlock Detection and Resolution Algorithm and Its Correctness Proof," *IEEE Transactions on Software Engineering*, Vol. 14, No. 10, pp. 1443–1452, October 1988.

[22] A. K. Elmagarmid and A. K. Datta, "Two-Phase Deadlock Detection Algorithm," *IEEE Transactions on Computers*, Vol. 37, No. 11, pp. 1454–1458, November 1988.

[23] N. Francez, "Distributed Termination," *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 1, pp. 42–55, January 1980.

[24] N. Francez and M. Rodeh, "Achieving Distributed Termination Without Freezeing," *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 3, pp. 287–292, May 1982.

[25] V. D. Gligor and S. H. Shattuck, "On Deadlock Detection in Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 5, pp. 435–440, September 1980.

[26] J. Gray, P. Homan, R. Obermarck, and H. Korth, *A Straw Man Analysis of Probability of Waiting and Deadlock*, Research Report RJ3066, IBM Research Laboratory, San Jose, California, February 1981. Also appeared in *Fifth International Conference on Distributed Data Management and Computer Networks*, 1981.

[27] J. N. Gray, "Notes on Database Operating Systems," In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, pp. 393–481, Springer–Verlag, Berlin, 1978.

[28] L. M. Haas and C. Mohan, *A Distributed Deadlock Detection Algorithm for a Resource-Based System*, Research Report RJ 3765, IBM Research Laboratory, San Jose, California, January 1983.

[29] K. Hao and R. T. Yeh, "Detection of Inherent Deadlocks in Distributed Programs," In *The 3rd International Conference on Distributed Computing Systems*, pp. 518–523, IEEE-CS, Miami/Ft. Lauderdale, Florida, October 1982.

[30] T. Hermann and K. M. Chandy, *A Distributed Procedure to Detect AND/OR Deadlock*, Technical Report TR LCS-8301, Department of Computer Sciences, University of Texas, Austin, Texas, 1983.

[31] G. S. Ho and C. V. Ramamoorthy, "Protocols for Deadlock Detection in Distributed Database Systems," *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 6, pp. 554–557, November 1982.

[32] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, No. 8, pp. 666–677, August 1978.

[33] R. C. Holt, "Some Deadlock Properties on Computer Systems," *ACM Computing Surveys*, Vol. 4, No. 3, pp. 179–196, September 1972.

[34] S. Huang, "A Distributed Deadlock Detection Algorithm for CSP-Like Communication," *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 1, pp. 102–122, January 1990.

[35] J. Hélary, C. Jard, N. Plouzeau, and M. Raynal, "Detection of Stable Properties in Distributed Applications," In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pp. 125–136, ACM SIGACT-SIGOPS, Vancouver, B.C., Canada, August 1987.

[36] J. Huang, J. A. Stankovic, D. Towsley, and K. Ramamritham, "Experimental Evaluation of Real-Time Transaction Processing," In *Proceedings of the 10th Real-Time Systems Symposium*, pp. 144–153, IEEE-CS, Santa Monica, California, December 1989.

[37] J. R. Jagannathan and R. Vasudevan, "Comments on "Protocols for Deadlock Detection in Distributed Database Systems"," *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 3, p. 371, May 1983.

[38] S. Kawazu, S. Minami, K. Itoh, and K. Teranaka, "Two-Phase Deadlock Detection Algorithm in Distributed Databases," In *Proceedings 5th International Conference on Very Large Databases*, pp. 360–367, 1979.

[39] E. Knapp, "Deadlock Detection in Distributed Databases," *ACM Computing Surveys*, Vol. 19, No. 4, pp. 303–328, December 1987.

[40] H. Ledgard, *ADA: An Introduction/Ada Reference Manual*, Springer-Verlag, New York, 1981/1980. The *Part II — Ada Reference Manual*, July 1980, was also published by the United States Government.

[41] H. F. Li, T. Radhakrishnan, and K. Venkatesh, "Global State Detection in NON-FIFO Networks," In *The 7th International Conference on Distributed Computing Systems*, pp. 364–370, IEEE-CS, Berlin, West Germany, September 1987.

[42] D. B. Lomet, "Coping with Deadlock in Distributed Systems," In *Proceedings of IFIP Working Conference on Data Base Architecture*, pp. 95–105, North-Holland Publishing Company, Venice, Italy, June 1979.

[43] M. Maekawa, A. E. Oldehoeft, and R. R. Oldehoeft, *Operating Systems - Advanced Concepts*, The Benjamin/Cummings Publishing Company, Inc., 1987.

[44] T. A. Marsland and S. S. Isloor, "Detection of Deadlocks in Distributed Database Systems," *INFOR*, Vol. 18, No. 1, pp. 1–20, February 1980.

[45] D. A. Menasce and R. R. Muntz, "Locking and Deadlock Detection in Distributed Data Bases," *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 3, pp. 195–202, May 1979.

[46] J. Misra and K. M. Chandy, "Termination Detection of Diffusing Computations in Communicating Sequential Processes," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 1, pp. 37–43, January 1982.

[47] J. Misra and K. M. Chandy, "A Distributed Graph Algorithm: Knot Detection," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 4, pp. 678–686, October 1982.

[48] D. P. Mitchell and M. J. Merritt, "A Distributed Algorithm for Deadlock Detection and Resolution," In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, pp. 282–284, ACM SIGACT-SIGOPS, Vancouver, B.C., Canada, August 1984.

[49] J. E. B. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*, Report MIT/LCS/TR-260, Laboratory for Computer Science, MIT, April 1981.

[50] N. Natarajan, "A Distributed Scheme for Detecting Communication Deadlocks," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 4, pp. 531–537, April 1986.

[51] R. Obermarck, "Distributed Deadlock Detection Algorithm," *ACM Transactions on Database Systems*, Vol. 7, No. 2, pp. 187–208, June 1982.

[52] C. Papadimitriou, *The Theory of Database Concurrency Control, Principles of Computer Science Series*, Computer Science Press, Rockville, Maryland, 1986.

[53] M. Roesler and W. A. Burkhard, "Resolution of Deadlocks in Object-Oriented Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. 38, No. 8, pp. 1212–1224, August 1989.

[54] D. S. Rosenblum, "An Efficient Communication Kernel for Distributed Ada Runtime Tasking Supervisors," *Ada LETTERS*, Vol. VII, No. 2, pp. 102–117, March-April 1987.

[55] B. A. Sanders and P. A. Heuberger, "Distributed Deadlock Detection and Resolution with Probes," In *Proc. of the 3rd International Workshop on Distributed Algorithms*, pp. 207–218, Nice, France, September 1989.

[56] S. K. Shrivastava, "On the Treatment of Orphans in a Distributed System," In *3rd Symposium on Reliability in Distributed Software and Database Systems*, pp. 155–162, IEEE-CS/ACM, Clearwater Beach, Florida, October 1983.

[57] M. Singhal, "Deadlock Detection in Distributed Systems," *IEEE Computer*, Vol. 22, No. 11, pp. 37–48, November 1989.

[58] M. K. Sinha and N. Natarajan, "A Priority Based Distributed Deadlock Detection Algorithm," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 1, pp. 67–80, January 1985.

[59] M. Spezialetti and P. Kearns, "Efficient Distributed Snapshots," In *The 6th International Conference on Distributed Computing Systems*, pp. 382–388, IEEE-CS, Cambridge, Massachusetts, May 1986.

[60] W. Tsai, A. K. Elmagarmid, and A. R. Hurson, "Deadlock Detection and Resolution in Distributed Database Systems," In *IEEE INFORCOM '87*, pp. 77–86, San Francisco, California, March 1987.

Bibliography

1. Y. Afek and M. Saks, "Detecting Global Termination Conditions in the Face of Uncertainty," In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pp. 109–124, ACM SIGACT-SIGOPS, Vancouver, B.C., Canada, August 1987.

2. R. Agrawal, M. J. Carey, and L. W. McVoy, "The Performance of Alternative Strategies for Dealing with Deadlocks in Database Management Systems," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 12, pp. 1348–1363, December 1987.

3. G. R. Andrews and G. M. Levin, "On-the-fly Deadlock Prevention," In *Proceedings ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 165–172, Ottawa, Ontario, Canada, August 1982.

4. G. R. Andrews and F. B. Schneider, "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys*, Vol. 15, No. 1, pp. 3–43, March 1983.

5. B. Awerbuch and S. Micali, "Dynamic Deadlock Resolution Protocols," In *Proceedings of the Foundations of Computer Science*, pp. 196–207, IEEE, Toronto, Canada, 1986.

6. H. E. Bal, J. G. Steiner, and A. S. Tanenbaum, "Programming Languages for Distributed Computing Systems," *ACM Computing Surveys*, Vol. 21, No. 3, pp. 261–322, September 1989.

7. R. Balter, P. Berard, and P. Decitre, "Why Control of Concurrency Level in Distributed Systems is More Fundamental than Deadlock Management," In *Proceedings ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 183–193, Ottawa, Ontario, Canada, August 1982.

8. V. C. Barbosa, "Strategies for the Prevention of Communication Deadlocks in Distributed Parallel Programs," *IEEE Transactions on Software Engineering*, Vol. 16, No. 11, pp. 1311–1316, November 1990.

9. A. Burns, *Concurrent Programming in Ada*, Cambridge University Press, 1985.

10. A. Burns, A. M. Lister, and A. J. Wellings, *A Review of Ada Tasking*, Volume 262 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin Heidelberg, Germany, 1987.

11. N. Carriero and D. Gelernter, "How to Write Parallel Programs: A Guide to the Perplexed," *ACM Computing Surveys*, Vol. 21, No. 3, pp. 323–357, September 1989.

12. S. Ceri and G. Pelagatti, *Distributed Databases: Principles and Systems*, McGraw-Hill, 1984.

13. K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, Vol. 24, No. 11, pp. 198–206, April 1981.

14. K. M. Chandy and J. Misra, "The Drinking Philosophers Problem," *ACM Transactions on Programming Languages and Systems*, Vol. 6, No. 4, pp. 632–646, October 1984.

15. K. M. Chandy and J. Misra, "How Processes Learn," *Distributed Computing*, Vol. 1, No. 1, pp. 40–52, January 1986.

16. E. J. H. Chang, "Echo Algorithms: Depth Parallel Operations on General Graphs," *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 4, pp. 391–401, July 1982.

17. J. Cheng, "A Classification of Tasking Deadlocks," *Ada Letters*, Vol. X, No. 5, pp. 110–127, May/June 1990.

18. J. Cheng, "Task-Wait-For Graphs and Their Application to Handling Tasking Deadlocks," In *TRI-Ada '90 Proceedings*, pp. 376–390, ACM/SIGAda, Baltimore, MD, December 1990.

19. A. N. Choudhary, *Two Distributed Deadlock Detection Algorithms and Their Performance*, Master's thesis, University of Massachusetts at Amherst, February 1986.

20. A. N. Choudhary, W. H. Kohler, J. A. Stankovic, and D. Towsley, "A Priority Based Probe Algorithm for Distributed Deadlock Detection and Resolution," In *The 7th International Conference on Distributed Computing Systems*, pp. 162–168, IEEE-CS, Berlin, West Germany, September 1987.

21. I. Cidon, J. M. Jaffe, and M. Sidi, "Local Distributed Deadlock Detection by Cycle Detection and Clustering," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 1, pp. 3–14, January 1987.

22. A. K. Elmagarmid, A. P. Sheth, and M. T. Liu, "Deadlock Detection Algorithms in Distributed Database Systems," In *International Conference on Data Engineering*, pp. 556–564, IEEE-CS, Los Angeles, California, February 1986.

23. K. P. Eswaren, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, Vol. 19, No. 11, pp. 624–633, November 1976.

24. N. Francez, "Corrigendum: Distributed Termination," *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 3, p. 463, July 1980.

25. N. Gehani, *Ada: Concurrent Programming*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1984.

26. S. M. German, "Monitoring for Deadlock and Blocking in Ada Tasking," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 6, pp. 764–777, November 1984.

27. S. M. German, D. P. Helmbold, and D. C. Luckham, "Monitoring for Deadlocks in Ada Tasking," In *Proceedings of the AdaTEC Conference on Ada*, pp. 10–25, ACM, Arlington, Virginia, October 1982.

28. D. Helmbold and D. Luckham, *Debugging Ada Tasking Programs*, Technical Report No. 84-262 (Program Analysis and Verification Group Report No. 25), Stanford University, July 1984.

29. D. Helmbold and D. C. Luckham, *Runtime Detection and Description of Deadness Errors in Ada Tasking*, Technical Report No. 83-249 (Program Analysis and Verification Group Report No. 22), Stanford University, November 1983.

30. J. D. Ichbiah, J. G. P. Barnes, R. J. Firth, and M. Woodger, *Rationale for the Design of the Ada Programming Language*, United States Government, 1986. The Ada Rationale was developed by Alsys and Honeywell under a contract from the United States Government (Ada Joint Program Office).

31. S. S. Isloor and T. A. Marsland, "The Deadlock Problem: An Overview," *IEEE Computer*, Vol. 13, No. 9, pp. 58–78, September 1980.

32. J. R. Jagannathan and R. Vasudevan, "A Distributed Deadlock Detection and Resolution Scheme: Performance Study," In *The 3rd International Conference on Distributed Computing Systems*, pp. 496–501, IEEE-CS, Miami/Ft. Lauderdale, Florida, October 1982.

33. S. Katz and O. Shmueli, "Cooperative Distributed Algorithms for Dynamic Cycle Prevention," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 5, pp. 540–552, May 1987.

34. H. F. Korth, "A Deadlock-Free, Variable Granularity Locking Protocol," In *Proceedings of the 5th Berkeley Conference on Distributed Data Management and Computer Networks*, pp. 105–121, February 1981.

35. H. F. Korth, "Edge Locks and Deadlock Avoidance in Distributed Systems," In *Proceedings ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 173–182, Ottawa, Ontario, Canada, August 1982.

36. H. F. Korth, "Locking Primitives in a Database System," *Journal of the ACM*, Vol. 30, No. 1, pp. 55–79, January 1983.

37. H. F. Korth, R. Krishnamurthy, A. Nigam, and J. T. Robinson, "A Framework for Understanding Distributed (Deadlock Detection) Algorithms," In *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pp. 192–202, Atlanta, Ga., March 1983.

38. L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, Vol. 21, No. 7, pp. 558–565, July 1978.

39. W. K. Lin and J. Nolte, "Communication Delay and Two Phase Locking," In *The 3rd International Conference on Distributed Computing Systems*, pp. 502–507, IEEE-CS, Miami/Ft. Lauderdale, Florida, October 1982.

40. L. Liu, "Comments on "A Distributed Scheme for Detecting Communication Deadlocks"," *IEEE Transactions on Software Engineering*, Vol. 15, No. 7, p. 926, July 1989.

41. N. A. Lynch and M. R. Tuttle, "Hierarchical Correctness Proofs for Distributed Algorithms," In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pp. 137–151, ACM SIGACT-SIGOPS, Vancouver, B.C., Canada, August 1987.

42. F. Mattern, "Algorithms for Distributed Termination Detection," *Distributed Computing*, Vol. 2, No. 3, pp. 161–175, December 1987.

43. J. Misra, "Detecting Termination of Distributed Computations Using Markers," In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pp. 290–294, ACM SIGACT-SIGOPS, Montreal, Canada, August 1983.

44. J. Misra, K. M. Chandy, and T. Smith, "Proving Safety and Liveness of Communicating Processes with Examples," In *Proceedings ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 201–208, Ottawa, Ontario, Canada, August 1982.

45. T. Murata, B. Shenker, and S. M. Shatz, "Detection of Ada Static Deadlocks Using Petri Net Invariants," *IEEE Transactions on Software Engineering*, Vol. 15, No. 3, pp. 314–326, March 1989.

46. S. L. Murphy and A. U. Shankar, "A Note on the Drinking Philosophers Problem," *ACM Transactions on Programming Languages and Systems*, Vol. 10, No. 1, pp. 178–188, January 1988.

47. S. Owicki and L. Lamport, "Proving Liveness Properties of Concurrent Programs," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, pp. 455–495, July 1982.

48. S. Rajagopalan, *Deadlock Detection Techniques in Distributed Databases*, Master's thesis, University of Massachusetts at Amherst, May 1985.

49. S. P. Rana, "A Distributed Solution of the Distributed Termination Problem," *Information Processing Letters*, Vol. 17, No. 1, pp. 43–46, July 1983.

50. J. H. Reif and P. G. Spirakis, "Real-Time Synchronization of Interprocess Communications," *ACM Transactions on Programming Languages and Systems*, Vol. 6, No. 2, pp. 215–238, April 1984.

51. G. A. Riccardi and T. P. Baker, "A Runtime Supervisor to Support Ada Tasking: Rendezvous and Delays," In *Ada in use – Proceedings of the Ada International Conference*, pp. 329–342, ACM and Ada-Europe, Cambridge University Press, May 1985.

52. M. Roesler, W. A. Burkhard, and K. B. Cooper, "Efficient Deadlock Resolution for Lock-Based Concurrency Control Schemes," In *The 8th International Conference on Distributed Computing Systems*, pp. 224–233, IEEE-CS, San Jose, California, June 1988.

53. M. Roesler and W. A. Burkhard, "Deadlock Resolution and Semantic Lock Models in Object Oriented Distributed Systems," In *Proceedings International Conference on Management Data (SIGMOD Vol. 17, No. 3)*, pp. 361–370, ACM, Chicago, Illinois, June 1988.

54. D. S. Rosenblum, *Design and Verification of Distributed Tasking Supervisors for Concurrent Programming Languages*, PhD thesis, Stanford University, March 1988. Also appear as Technical Report No. 88-375 (Program Analysis and Verification Group Report No. 38).

55. C. Shih and J. A. Stankovic, *Survey of Deadlock Detection in Distributed Concurrent Programming Environments and Its Application to Real-time Systems*, COINS Technical Report 90-69, University of Massachusetts at Amherst, August 1990.

56. C. Shih and J. A. Stankovic, "Distributed Deadlock Detection in Ada Runtime Environments," In *TRI-Ada '90 Proceedings*, pp. 362–375, ACM/SIGAda, Baltimore, MD, December 1990.

57. M. K. Sinha and N. Natarajan, "A Distributed Deadlock Detection Algorithm Based on Timestamps," In *The 4th International Conference on Distributed Computing Systems*, pp. 546–556, IEEE Computer Society, San Francisco, California, May 1984.

58. N. Soundararajan, "Axiomatic Semantics of Communicating Sequential Processes," *ACM Transactions on Programming Languages and Systems*, Vol. 6, No. 4, pp. 647–662, October 1984.

59. K. Sugihara, T. Kikuno, N. Yoshida, and M. Ogata, "A Distributed Algorithm for Deadlock Detection and Resolution," In *4th Symposium on Reliability in Distributed Software and Database Systems*, IEEE-CS, Silver Spring, Maryland, October 1984.

60. Y. C. Tay, *Locking Performance in Centralized Databases*, Volume 14 of *Perspectives in Computing*, Academic Press, 1987.

61. W. Tsai and G. G. Belford, "Detecting Deadlock in a Distributed System," In *IEEE INFOR-COM '82*, pp. 89–95, Las Vegas, Nevada, March 1982.

62. J. D. Ullman, *Principles of Database Systems*, Pitman Publishing Limited, second edition, 1982.

63. R. A. Volz, T. N. Mudge, A. W. Naylor, and J. H. Mayer, "Some Problems in Distributed Real-time Ada Programs Across Machines," In *Ada in use – Proceedings of the Ada International Conference*, pp. 72–84, ACM and Ada-Europe, Cambridge University Press, May 1985.

64. R. A. Volz and T. N. Mudge, "Timing Issues in the Distributed Execution of Ada Programs," *IEEE Transactions on Computers*, Vol. C-36, No. 4, pp. 449–459, April 1987.

65. G. T. Wuu and A. J. Bernstein, "False Deadlock Detection in Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 8, pp. 820–821, August 1985.

66. B. E. Wójcik and Z. M. Wójcik, "Sufficient Condition for a Communication Deadlock and Distributed Deadlock Detection," *IEEE Transactions on Software Engineering*, Vol. 15, No. 12, pp. 1587–1595, December 1989.

67. B. Zhou, R. T. Yeh, and P. A. Ng, "An Algebraic System for Deadlock Detection and Its Applications," In *4th Symposium on Reliability in Distributed Software and Database Systems*, IEEE-CS, Silver Spring, Maryland, October 1984.

68. D. Zöbel, "The Deadlock Problem: A Classifying Bibliography," *Operating Systems Review*, Vol. 17, No. 4, pp. 6–15, October 1983.