Deadlock Detection in Distributed Real-Time Systems and Its Application to Ada Environments*

CSI Invited Paper

Chia-Shiang Shih
ECE Department, University of Massachusetts
Amherst, MA 01003

John A. Stankovic
COINS Department, University of Massachusetts
Amherst, MA 01003

August 20, 1991

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. 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. Related problems, such as livelocks, orphan tasks, task termination problems, and global state detection, are considered when it is appropriate.

*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[11] 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 [35]:

- 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 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 summarize the

related background in terms of graph theory, the Ada system model, and a hierarchi of deadlock models, and propose extensions and solutions 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 Ada's underlying system model. In Section 4, we discuss a hierarchy of deadlock models. Section 5 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 6 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 7 summarizes the paper.

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. A wait-for graph is a *digraph* (directed graph). A digraph is a pair (V, E), where V is a nonempty set of *vertices* (which represent tasks or resources) and E is a set of *directed edges* (which represent "wait-for" dependencies). Each directed 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 a directed edge. If both task vertices and resource vertices coexist in a graph, a wait-for graph becomes a *bipartite* digraph. A bipartite graph is one in which all the vertices in V are partitioned into two disjoint subsets (a subset of tasks T and a subset of resources R in the case of wait-for graphs) such that there are no edges connecting vertices from the same subset.

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

TWFG: The TWFG (Task Wait-For Graph) is the simplest graph among three types of graph. A TWFG is a digraph in which vertices represent tasks; hence, only the task-task wait-for relations are depicted in the TWFG.

TRG: A TRG (Task-Resource Graph) is a bipartite digraph in which task-resource interactions (i.e., resource competitions) can be described by means of the directed edges between T and R.

GRG: A GRG (General Resource Graph) is a generalized TRG in which inter-task communications are represented by "consumable" resources.

In the following discussions we use the concept of GRG to merge the problems of the deadlocks due to both inter-task communication and resource competition. The GRG was proposed by Holt to describe his *General Resource System*(GRS)[22]. 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 units. 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). Therefore, edges in a GRG are of three types:

a request edge: directed from a requesting task to the requested resource,

an assignment edge: directed from a reusable resource to its assigned holder, and

a producer edge: directed from a consumable resource to one of its producers.

3 Model of Ada's Distributed Concurrent Programming Environment

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 environment. To bridge this gap we must understand Ada's concurrency, synchronization, and resource allocation models and show how they relate to the theory.

Ada's concurrent programming mechanisms are generalized from many aspects of Hoare's "Communicating Sequential Processes" (CSP)[21] and Brinch Hansen's "Distributed Processes" (DP)[4]. 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*[26] 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.

A rendezvous can be treated as a consumable resource in GRG. 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 will wait for the called task to execute corresponding statements following the **accept** statement. The called task, therefore, acts as the producer during the rendezvous period.

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 5.1, *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*[13] and *global state detection*[6] can be extended and applied to solve these problems.

4 A Hierarchy of Deadlock Models and Ada

Knapp[25] classified the deadlock problem into a hierarchy of six models to reflect the complexity of a particular deadlock problem. 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 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. The six models are summarized as follows.

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 the necessary and sufficient condition for deadlock. Examples of this model can be found in database systems where transactions are requesting data items one by one exclusively. In the Ada environment, if the resources are non-shareable and only one outstanding request is allowed, and at most two tasks may be involved in either a rendezvous or task termination, the system can be formalized as a Single-Resource model. Mitchell and Merritt[31] proposed a very simple and elegant algorithm based on this Single-Resource model for non real-time systems.

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 exclusive use until *all* its shared lock holders have released the lock. Applications of the AND model 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 a necessary and sufficient condition for deadlock in the AND model. The AND model is, therefore, strictly more general than the Single-Resource model. Many non real-time algorithms have been proposed based on this AND model such as [5, 7, 16, 18, 28, 33].

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. Also, in the Ada environment, the mechanism of the **accept** statement can be categorized as an OR model. In the OR model, detecting a *cycle* in the GRG is not a sufficient condition for deadlock. As pointed out by Holt[22], a *knot* is a sufficient condition for deadlock while a *cycle* is only a necessary condition. Algorithms proposed for the OR model can be found in [7, 24, 29, 32].

AND-OR Model: The AND-OR model is a generalization of the two previous models. A task in the 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. The Ada runtime environment is an AND-OR model since both AND and OR mechanisms exist as described above. There is no simple construct of graph theory 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 the 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[20] proposed a more efficient algorithm to detect deadlock in the AND-OR model.

C(n,k) Model: The C(n,k) model, which was first formulated by Bracha and Toueg[3] as k-out-of-n request, is a generalization of the AND-OR model. Although AND and OR can also express a k-out-of-n request, the length of the corresponding AND-OR formula is $k \cdot C(n,k)$. Again, the algorithm presented by Bracha and Toueg[3] 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 requesting k pages memory from a total of n pages.

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 the Ada environment. Many algorithms related to this model have been studied theoretically such as: stable properties detection[19] and global state detection[6, 27].

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, 10, 13, 12, 14, 15, 30]. 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.

In the Ada environment, to use the Single-Resource model, very severe program restrictions must be enforced. These restrictions are used to eliminate the AND and the OR wait-for mechanisms which may cause multiple outgoing edges in the GRG. The Ada task termination mechanism (an AND logic wait-for mechanism) should be either avoided or programmed very carefully so that each terminating task won't be involved in a more complicated AND deadlock. Further, the Ada **accept** statement, an OR logic mechanism, which allows one of many potential calling tasks to be in rendezvous with the accepting task, must be programmed in a one to one fashion that limits the number of potential calling tasks to precisely one. Obviously, these are very severe restrictions. Restrictions upon task termination can be removed for the AND model, while restrictions upon

the **accept** statement can be removed for the OR model. The AND-OR or C(n,k) model is general enough that all programming constraints on an Ada programmer can be removed.

5 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.

5.1 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.

5.2 The Complexity of Deadlock Models

In Section 4, 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[31], 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 6.2).

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, 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[7, 24, 29, 32] 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[3], 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.

5.3 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 5.1). 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 5.2) 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.

6 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 5.3 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 [36, 8, 9]). 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 5.3 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.

6.1 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*[34] (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.

6.2 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[31], our algorithm can reduce the number of probe messages and achieve a single point of detection of every deadlock cycle. Each 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 1, 2, and 3, respectively. In Figure 1 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 2 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 3 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 4 describes 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 5. This algorithm guarantees that only tasks in the BLOCKED state may receive probes

```
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 earliest
                                       -- timing constraint in its travelling path
    end record;
```

Figure 1: Data structure for probes in the Single-Resource algorithm.

```
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 2: 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 3: 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 4: Procedure for probe initiation in the Single-Resource algorithm.

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 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 6. 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

```
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 5: Procedure for tasks handling received probes in the Single-Resource algorithm.

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.

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." The second task which establishes a rendezvous is always the "producer." 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.

```
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 6: Procedure for resources handling received probes in the Single-Resource algorithm.

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.

6.3 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[5] and followed by Haas and Mohan[18]. 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 7, 8, and 9, respectively. In Figure 7, the fields probe_id and initr_id are defined in the same way as those in the 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 8 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 9 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.

```
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 7: Data structure for probes in the AND algorithm.

```
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 8: 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 9: Data structure for resources in the AND algorithm.

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 10 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 11. 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 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 12. 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

```
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 10: Procedure for probe initiation in the AND algorithm.

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.[17], 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

```
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 11: 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 12: Procedure for resources handling received probes in the AND model.

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.

7 Summary

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[23].

References

[1] K. R. Apt, "Correctness proofs of distributed termination algorithms," *ACM Transactions on Programming Languages and Systems*, vol. 8, 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, pp. 370–379, July 1984.

[3] 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*, (Vancouver, B.C., Canada), pp. 285–301, ACM SIGACT-SIGOPS, Aug. 1984.

[4] P. Brinch Hansen, "Distributed processes: A concurrent programming concept," *Communications of the ACM*, vol. 21, pp. 934–941, Nov. 1978.

[5] 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*, (Ottawa, Ontario, Canada), pp. 157–164, Aug. 1982.

[6] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, pp. 63–75, Feb. 1985.

[7] K. M. Chandy, J. Misra, and L. M. Haas, "Distributed deadlock detection," *ACM Transactions on Computer Systems*, vol. 1, pp. 144–156, May 1983.

[8] 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, pp. 10–17, Jan. 1989.

[9] 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, p. 1644, Dec. 1989.

[10] S. Cohen and D. Lehmann, "Dynamic systems and their distributed termination," in *Proceedings ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, (Ottawa, Ontario, Canada), pp. 29–33, Aug. 1982.

[11] E. W. Dijkstra, "Co-operating sequential processes," in *Programming Languages* (F. Genuys, ed.), pp. 43–112, New York: Academic Press, 1968.

[12] 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, pp. 217–219, June 1983.

[13] E. W. Dijkstra and C. S. Scholten, "Termination detection for diffusing computations," *Information Processing Letters*, vol. 11, pp. 1–4, Aug. 1980.

[14] N. Francez, "Distributed termination," *ACM Transactions on Programming Languages and Systems*, vol. 2, pp. 42–55, Jan. 1980.

[15] N. Francez and M. Rodeh, "Achieving distributed termination without freezeing," *IEEE Transactions on Software Engineering*, vol. SE-8, pp. 287–292, May 1982.

[16] V. D. Gligor and S. H. Shattuck, "On deadlock detection in distributed systems," *IEEE Transactions on Software Engineering*, vol. SE-6, pp. 435–440, Sept. 1980.

[17] 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, Feb. 1981. Also appeared in *Fifth International Conference on Distributed Data Management and Computer Networks*, 1981.

[18] 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, Jan. 1983.

[19] J.-M. 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*, (Vancouver, B.C., Canada), pp. 125–136, ACM SIGACT-SIGOPS, Aug. 1987.

[20] 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.

[21] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, pp. 666–677, Aug. 1978.

[22] R. C. Holt, "Some deadlock properties on computer systems," *ACM Computing Surveys*, vol. 4, pp. 179–196, Sept. 1972.

[23] 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*, (Santa Monica, California), pp. 144–153, IEEE-CS, Dec. 1989.

[24] S.-T. Huang, "A distributed deadlock detection algorithm for csp-like communication," *ACM Transactions on Programming Languages and Systems*, vol. 12, pp. 102–122, Jan. 1990.

[25] E. Knapp, "Deadlock detection in distributed databases," *ACM Computing Surveys*, vol. 19, pp. 303–328, Dec. 1987.

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

[27] H. F. Li, T. Radhakrishnan, and K. Venkatesh, "Global state detection in non-fifo networks," in *The 7th International Conference on Distributed Computing Systems*, (Berlin, West Germany), pp. 364–370, IEEE-CS, Sept. 1987.

[28] D. A. Menasce and R. R. Muntz, "Locking and deadlock detection in distributed data bases," *IEEE Transactions on Software Engineering*, vol. SE-5, pp. 195–202, May 1979.

[29] J. Misra and K. M. Chandy, "A distributed graph algorithm: Knot detection," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 678–686, Oct. 1982.

[30] J. Misra and K. M. Chandy, "Termination detection of diffusing computations in communicating sequential processes," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 37–43, Jan. 1982.

[31] 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*, (Vancouver, B.C., Canada), pp. 282–284, ACM SIGACT-SIGOPS, Aug. 1984.

[32] N. Natarajan, "A distributed scheme for detecting communication deadlocks," *IEEE Transactions on Software Engineering*, vol. SE-12, pp. 531–537, Apr. 1986.

[33] R. Obermarck, "Distributed deadlock detection algorithm," *ACM Transactions on Database Systems*, vol. 7, pp. 187–208, June 1982.

[34] D. S. Rosenblum, "An efficient communication kernel for distributed ada runtime tasking supervisors," *Ada LETTERS*, vol. VII, pp. 102–117, March-April 1987.

[35] M. Singhal, "Deadlock detection in distributed systems," *IEEE Computer*, vol. 22, pp. 37–48, Nov. 1989.

[36] M. K. Sinha and N. Natarajan, "A priority based distributed deadlock detection algorithm," *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 67–80, Jan. 1985.