Distributed Deadlock Detection in Ada Runtime Environments*

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

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

Abstract

Distributed deadlock detection has been studied in distributed database systems and distributed timesharing operating systems, but has not been widely used in real-time systems such as Ada runtime environments. In this paper we are interested in explicitly tying the formal properties of deadlock algorithms to Ada and its runtime system. 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.

1 Introduction

The deadlock detection problem has a very rich formalism associated with it. Deadlock can be formally studied in isolation by using graph theory. However, in this paper we are interested in explicitly tying the formal properties of deadlock algorithms to Ada and its runtime system. 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. To bridge this gap we must understand Ada's concurrency, synchronization, and resource allocation models and show how they relate to the theory. In addition, the deadlock problem becomes further complicated when the underlying system is distributed and when tasks have timing constraints. In this paper, we also address how deadlock detection for distributed, real-time applications can be supported by Ada runtime environments.

The remainder of this paper is organized as follows. Section 2 provides some background material on deadlock problems. Section 3 discusses the various deadlock models from the simplest to the most complex and explains how they relate to Ada. Section 4 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 5 we present two algorithms and relate those algorithms to the levels of complexity associated with the theoretical models and the restrictions imposed on Ada. Finally, Section 6 discusses a related work and Section 7 summarizes the paper.

2 The Deadlock Problem

In a concurrent processing system deadlock situations may arise if and only if the following four resource competition conditions hold simultaneously: (1) mutual exclusion, (2) hold and wait, (3) no preemption, and (4) circular wait. In an Ada runtime environment, deadlocks may occur due to competition for underlying shared resources. This is independent of Ada. For example, a program with two tasks T_1 and T_2 executing on a system with two disk drives where each of T_1 and T_2 needs both disk drives together, will deadlock 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 as a result of Ada semantics itself, e.g., when tasks are waiting for each other in Ada's rendezvous. Both causes of deadlocks must be handled.

Principally, there are three strategies for dealing with the deadlock problem: (1) deadlock prevention, (2) deadlock avoidance, and (3) deadlock detection.

*This work was supported by the Charles Stark Draper Laboratory, Inc. and by NSF under grant IRI-8908693.

Appeared in the *TRI-Ada '90 Conference Proceedings*, Baltimore, December 3–7, 1990.

The first two strategies ensure that the system will *never* enter a deadlock state. These two methods have a number of drawbacks including inefficiency. Deadlock prevention is inefficient because it decreases system concurrency by restricting the execution of the tasks to avoid at least one of the deadlock conditions. Deadlock avoidance is inefficient because checking for a safe state is computationally expensive. In the third strategy, the system is allowed to enter a deadlock state and then it is detected and recovery is performed. The detection of deadlocks requires examination of task/resource interactions for the presence of cyclic waits. Once a deadlock is formed, it persists until it is detected and recovered from (so called "stable property" of the deadlock problem). The deadlock detection computation can proceed concurrently with the normal activities of a system.

In the following sections we use the concept of a GRG (General Resource Graph) because Ada's rendezvous can be formalized as consumable resources in the GRG model. The GRG was proposed by Holt[14]. Vertices in a GRG are of three types: *tasks*, *reusable resources*, and *consumable resources*. Edges are of three types: (1) *a request edge:* directed from a requesting task to the requested resource, (2) *an assignment edge:* directed from a reusable resource to its assigned holder, and (3) *a producer edge:* directed from a consumable resource to one of its producers. In Section5.1, we discuss how Ada's rendezvous can be described using this *producer edge* model.

3 A Hierarchy of Deadlock Models and Ada

Knapp[16] 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 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[21] 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 [3, 9, 23].

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[14], 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 [3, 15, 19, 22].

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[13] 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[1] 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[1] 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[10] and global state detection[4, 18].

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.

4 Real-Time Distributed Deadlock Detection and Resolution

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. Other aspects of real-time processing are supported by Ada's *selective wait*[17] statement. Using a combination of the **delay** statement and the **else** alternative in Ada's **select** statement, provides 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.

In Section 4.1, we describe the deadlock problems for real-time systems in which timing constraints are taken into consideration. In Section 4.2 we discuss the various levels of complexity for the hierarchy of deadlock models.

4.1 Deadlock Problems in Real-Time Systems

In real-time systems, such as Ada environments, due to timing constraints attached to a 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 (cycle/knot) and neither a 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 deadlocks. However, due to timing constraints, a nonempty subset of tasks in a cycle/knot may be timed out or aborted from the blocked state. The deadlock condition, 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 might be only a normal waiting condition, or an abnormal condition such as being livelocked or becoming 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 situation should be terminated 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 not assumed 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 the problem with non-deadlocked blocking.

The detection and resolution of temporal deadlock is especially important for the tasks with tight timing constraints. 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, the detection and resolution of the temporal deadlock is important to T. Task T may fail without knowing the existence of this temporal deadlock if no detection operation is attempted or the detection operation can not complete before a time in which it is still possible to meet the timing constraint of T.

4.2 The Levels of Complexity for the Deadlock Models

In Section 3 a hierarchy of six deadlock models were used to describe the characteristics of deadlocks. Except for the Unrestricted model, the 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 (simple cycles), (2) the AND model (nested cycles), (3) the OR model (knot), and (4) the AND-OR and the C(n,k) models (both cycle detection and knot detection are not sufficient).

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[21], each of the blocked tasks is assigned a 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 and 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 and may cause false detection of deadlocks even in this simplest model. If timing constraints are considered in the algorithm (for example, timing constraints can be associated with each deadlock detection probe), false detection of deadlocks can be reduced.

In the AND deadlock model, since multiple outgoing edges as well as multiple incoming edges in the GRG are possible, nested cycles are expected. *Foreign probes*, 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. For example, in an algorithm proposed by Chandy et al.[3], each task has to keep track of all the probes have been received since it became blocked. A probe will be discarded when it starts to travel in a cycle a second time. Also, 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, 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. Due to the existence of nested cycles, simple cycle detection algorithms, such as the ones used in 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 could be taken into consideration to minimize the detection of false deadlocks.

In the OR model, a knot is a sufficient condition for deadlock while a cycle is only a necessary condition. Hence, deadlock detection in the OR model can be reduced to finding knots in the GRG. 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 and backward directions along the edges in the 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. Dijkstra and Scholten introduced the notion of *diffusing computation*[5] in a distributed system of tasks, and suggested an algorithm to detect the termination of an arbitrary diffusing computation in any network environment. The generality of the solution makes it a candidate for a number of problems in the distributed systems. The knot detection algorithms proposed in the literature[3, 15, 19, 20] are based on the notion of *diffusing computation*. 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 in a similar way.

The complexity of the remaining models — the AND-OR model and the C(n,k) model — are roughly the same. Many systems are neither solely AND-OR nor solely C(n,k), but a mixture of the 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. Since there is no simple construct of graph theory in terms of the GRG to describe the deadlock condition in the C(n,k) model, Bracha and Toueg[1] proposed an idea of processing a system snapshot GRG to detect deadlocks. Bracha and Toueg's algorithm uses special *FREEZE* messages to notify each of the tasks in the system to take a *snapshot* of its local state for a later deadlock detection phase. The collection of these local snapshots does not describe any particular global system state; instead, they can constitute a *consistent global snapshot*[4]. In the deadlock detection phase, each of the active tasks in the snapshot GRG can be simulated to terminate and to release the resources it hold. The GRG thus can be "reduced" to a new state. This procedure proceeds until the GRG

reaches an "irreducible" state or is empty. The blocked tasks in the final irreducible GRG are said to be deadlocked. In real-time applications, the timing constraints associate with each task can be collected while taking snapshots. A temporal deadlock can be detected if a blocked task cannot be simulated to become active to ensure its timing constraints.

5 Proposed Real-Time Distributed Deadlock Detection and Resolution 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 *Distributed Runtime Tasking Supervisors*[24] (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 sensible 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. This feature provides the flexibility that a task could randomly accept one entry call from many possible tasks. However, a task blocked by an **accept** statement may become an orphan if the number of possible calling tasks is actually zero. Further, deadlocks occurring due to the Ada **accept** statement are impossible to detect if the potential calling tasks cannot be identified. 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 structures and program code (not explicitly programmed) for deadlock detection. The extra code provided by the 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.

5.1 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 from a deadlock due to multiple detection of a cycle. By using a technique similar to that used in the Mitchell and Merritt's algorithm[21], 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

```
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.

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 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). 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

```
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 pro-
  -- duced 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 re-
  -- source R.probe_bufd. A period of waiting time
  -- Δt which is chosen as a function of task T's dead-
  -- line 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.

```
procedure TASK_RCV_PROBE
            (T: in out TASK_TYPE;
             P: in PROBE_TYPE) is
-- This procedure is invoked whenever a task T re-
-- ceives 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.

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 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. Otherwise, the probe is discarded.

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 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

```
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.

the task accepting an entry call are ready to establish a rendezvous. A rendezvous is a consumable resource. Either one of the calling and called tasks arriving at the rendezvous first will wait, and, hence, becomes the "consumer" of the "rendezvous." The second task which establishes a rendezvous is always the "producer" of the "rendezvous." After a rendezvous is established, the calling task becomes BLOCKED while the called task is executing corresponding statements following the **accept** statement. The called task, therefore, is ACTIVE and acts as the producer during the rendezvous period.

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

A GRG can be implemented as a two dimensional matrix. One dimension represents tasks, and the other dimension represents resources. Each of the elements in a GRG matrix represents one of the following three states: (1) the task is waiting for the resource, (2) the resource is held by the task, or (3) there is no relationship between them. Another possible implementation of GRG is to store the information in each of the task tables and resource tables, for example, the resource_table in TASK_TYPE and the task waiting_queue in RESOURCE_TYPE used in our algorithm data structure.

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

5.2 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[2] and followed by Haas and Mohan[9]. 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 they are 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 its 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 can no longer 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 will not be detected if no new probes with a larger timestamp 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 whose 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

```
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 id
    initr_id: TASK_ID_TYPE:=0;
      -- task_id of its 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 pro-
  -- duced 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.

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. A period of time Δt which is chosen as a function of a task's deadline 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 searching 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 is removed from the chain. Similar to the Single-Resource algorithm, if no cycle is found,

```
procedure TASK_INIT_PROBE
          (T: in out TASK_TYPE) is
  -- This procedure is invoked when a ACTIVE task
  -- T requests resources which are not all FREE, or
  -- when a BLOCKED task is granted one of its pend-
  -- ing requests but remains BLOCKED. A period of
  -- waiting time Δt which is chosen as a function of
  -- task T's deadline 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.

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 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

```
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 of the chain 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
      else                -- the chain
        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 re-
  -- ceives 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.

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

6 Related Work

There has been very limited research in the dynamic detection of deadlock for Ada programs. One noteworthy exception is the work performed by German, Helmbold, and Luckham[6, 7, 11, 12]. Their work uses a runtime monitor to detect a class of Ada deadness errors including some limited forms of deadlock. There are four major differences between their work and the work presented here. First, the monitoring mechanism used in their work is a centralized mechanism, while our work focuses on distributed systems. Second, deadlock problems are analyzed in very different ways. In their work, deadness errors are classified into three types: Circular Deadlock, Global Blocking, and Local Blocking. Circular Deadlock is extremely restricted and involves only entry calls. It does not account for deadlock that may occur for any other reason such as circular conflicts over a disk resource. Their Circular Deadlocks model is equivalent to the Single-Resource deadlock model, but (again) is used only on entry calls. In addition to handling Circular Deadlocks for entry calls, they can also detect a Global Blocking situation. A Global Blocking means that all the tasks in the system are stopped, regardless of the reason. This implies that when they find all tasks stopped it may have been due to problems that would have to be modelled by the OR model, or the AND-OR model, or the C(n,k) model. However, to find such a problem requires either the unlikely situation that all tasks are stopped, or waiting long enough for all tasks which are still running to terminate successfully, or to eventually call a blocked task. This is unreasonable for a real-time system. A third type of deadness error that they deal with is called Local Blocking. Local Blocking means that a subset of interacting tasks are blocked. However, neither the Circular Deadlock nor the Local Blocking consider deadlocks which involve the OR logic wait-for dependency due to **accept** statements. Consequently, their monitor cannot detect any local deadlock with OR model or AND-OR model, or C(n,k) model complexity unless the situation evolves into a Global Blocking situation. Third, timing constraints are not considered in their work. This is due, in part, because it was not their intent, and in part, because they do not treat a task which is executing a **delay** of a selective wait statement as in the blocked state. Fourth, their monitor is built at the user application level rather than as part of the underlying runtime system. Most of their research effort was spent on the program transformations necessary to gain program visibility at the user level. In our approach, we assume the compiler and underlying system can provide the proper information, and then focus on the development of efficient, distributed, real-time deadlock detection algorithms for different levels of complexity of Ada programs.

7 Summary

Distributed deadlock detection has been studied in distributed database systems and distributed timesharing operating systems, but has not been widely used in real-time systems such as Ada runtime environments. In this paper we explicitly addressed the formal properties of deadlock algorithms and tie them directly to Ada and its runtime system. First, we analyzed the deadlock

properties in Ada environments by using Knapp's hierarchy of deadlock models. Based on the sufficient conditions for deadlock detection, we divided these models into four levels of problem complexity. 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 to 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 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 assumed and utilized this property. In real-time systems, timing constraints are attached to the tasks. A task may be timed 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 also addressed how the deadlock problems can be solved in each of the four levels of problem complexity and how the timing constraints are considered in the possible solutions. After discussing the issues and needed solutions in general, we then provide two complete 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.

References

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

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

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

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

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

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

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

[8] J. Gray, P. Homan, R. Obermarck, and H. Korth, "A Straw Man Analysis of Probability of Waiting and Deadlock," *Fifth International Conference on Distributed Data Management and Computer Networks*, 1981.

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

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

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

[12] D. Helmbold and D. Luckham, "Debugging Ada Tasking Programs," *IEEE Software*, Vol. 2, No. 2, pp. 47–57, March 1985.

[13] 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.

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

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

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

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

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

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

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

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

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

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

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