Proceedings of the 1994 Winter Simulation Conference
ed. J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila

DEADLOCK DETECTION AND RESOLUTION IN SIMULATION MODELS

Murali Krishnamurthi
Amar Basavatia
Sanjeev Thallikar

Department of Industrial Engineering
Northern Illinois University
DeKalb, Illinois 60115, U.S.A.

ABSTRACT

Even though simulation models are validated and verified during the development process, a problem known as "deadlock" can still occur and go unnoticed in large, complex simulation models. Of all the commercial simulation languages currently in use, none can currently detect or prevent deadlocks and this can lead to incorrect results and decisions. Unfortunately, a deadlocking situation will not show up as a syntax or run time error since it is a modeling error. In this paper, the issues related to deadlock detection and resolution in discrete event simulation models will be analyzed and an algorithm for detecting deadlocks in simulation models will be presented and illustrated with examples. Issues related to deadlock resolution will also be discussed.

1 INTRODUCTION

Simulation involves the building and execution of models of systems and the analysis of their output statistics for the purpose of making decisions regarding the systems. The conceptualization, design, and development of simulation models is often a complex and time-consuming task. In a simulation study, the correctness of a simulation model is determined through two distinct steps, namely, Validation and Verification. *Verification* is concerned with determining that a simulation program performs as intended and this may involve debugging the program. *Validation* is concerned with determining whether a simulation model is an accurate representation of the system under study (Law and Kelton 1991) and this may involve comparing the behavior of the model with the real system. Even though these steps are generally taken to ensure the correctness of simulation models, a problem known as "deadlock" can still occur in simulation models, as well as in real systems.

Several definitions of deadlock exist and most of these definitions have come from operating systems and distributed database systems literature. Isloor and Marsland (1980) state that "a deadlock arises when members of a group of processes which hold resources are blocked indefinitely from access to resources held by other resources within the group." A deadlocking situation in discrete event simulation models can be best illustrated with the following example: Part A held by a Robot wants to be loaded on the Machine for processing, but there is a finished Part B on the Machine which wants to be unloaded off the Machine by the same Robot. Until Part B is unloaded from the Machine, Part A cannot be loaded on the Machine and until the Robot relinquishes Part A, Part B cannot be unloaded off the Machine by the Robot. Until this deadlock is resolved, all parts involved are blocked indefinitely. A simulation model of this situation developed in SIMAN (Pegden et al. 1990) for the deadlocked and undeadlocked versions is shown in Figure 1 along with the outputs generated. It can be easily seen from the simulation output of this simple example that the entities are indefinitely blocked. However, this type of a situation cannot be readily identified from the output of a large scale simulation model.

Deadlocks can not only occur in simulation models but also in real systems such as flexible manufacturing systems (Leung 1994). Deadlocks can be detected and resolved more easily in real systems since humans can observe, intervene, and resolve the deadlock. However, in large scale, complex simulation models the existence of a deadlock can be identified only after a simulation model has been run and the output has been inspected thoroughly. It is generally very difficult to predict how long it will take to detect a deadlock in the output of a large scale simulation model, if at all it can be done successfully.

WITH DEADLOCK

```
;MODEL FILE:
BEGIN;
   CREATE, 10;
   QUEUE,  Load Q;
   SEIZE:  Robot;
   QUEUE,  MachineQ;
   SEIZE:  Machine;
   DELAY:  1;
   RELEASE: Robot;
   DELAY:  5;
   QUEUE,  UnLoadQ;
   SEIZE:  Robot;
   RELEASE: Machine;
   DELAY:  1;
   RELEASE: Robot;
   COUNT:  Jobs: Dispose;
END;
```

```
;EXPERIMENT FILE:

BEGIN;
   PROJECT,With Deadlock, MK;
   QUEUES: UnLoadQ: LoadQ: MachineQ;
   RESOURCES: Robot: Machine;
   COUNTERS: Jobs;
   DSTATS: NR(Robot), Robot Utilization:
           NR(Machine), Machine Utilization:
           NQ(LoadQ), Robot LoadQ:
           NQ(UnloadQ), Robot UnLoadQ:
           NQ(Machine), Machine Q Length;
   REPLICATE, 1, 0, 100;
END;
```

PARTIAL OUTPUT FILE:

```
                 DISCRETE-CHANGE VARIABLES
Identifier        Avg.   Var.   Min.   Max.   Final
Robot Util.      1.0000 .00000 .00000 1.0000 1.0000
Machine Util.    1.0000 .00000 .00000 1.0000 1.0000
Robot UnLoadQ    .94000 .25265 .00000 1.0000 1.0000
Robot LoadQ      8.0100 .01242 .00000 9.0000 8.0000
MachinQ          .99000 .10050 .00000 1.0000 1.0000

              COUNTERS
Identifier        Count    Limit
Jobs                  0    Infinite
```

WITHOUT DEADLOCK

```
;MODEL FILE:
BEGIN;
   CREATE, 10;
   QUEUE,  Load Q;
   SEIZE:  Machine:Robot;
   DELAY:  1;
   RELEASE: Robot;
   DELAY:  5;
   QUEUE,  UnLoadQ;
   SEIZE:  Robot;
   RELEASE: Machine;
   DELAY:  1;
   RELEASE: Robot;
   COUNT:  Jobs: Dispose;
END;
```

```
;EXPERIMENT FILE:

BEGIN;
   PROJECT,Without Deadlock, MK;
   QUEUES: UnLoadQ: LoadQ;
   RESOURCES: Robot: Machine;
   COUNTERS: Jobs;
   DSTATS: NR(Robot), Robot Utilization:
           NR(Machine), Machine Utilization:
           NQ(LoadQ), Robot LoadQ:
           NQ(UnloadQ), Robot UnLoadQ;
   REPLICATE, 1, 0, 100;
END;
```

PARTIAL OUTPUT FILE:

```
                 DISCRETE-CHANGE VARIABLES
Identifier        Avg.   Var.   Min.   Max.   Final
Robot Util.      .20000 2.0000 .00000 1.0000 .0000
Machine Util.    .60000 .81650 .00000 1.0000 .0000
Robot UnLoadQ    .00000 --     .00000 1.0000 .0000
Robot LoadQ      3.1500 1.0053 .00000 9.0000 .0000

              COUNTERS
Identifier        Count    Limit
Jobs                 10    Infinite
```

Figure 1: An Example of a Simulation Model and Output With and Without Deadlock

Of all the commercial simulation languages currently in use, none can detect or prevent deadlocks and this can lead to incorrect results and wrongful conclusions. Manual detection of deadlocks is relatively easy in small simulation models. However, in large simulation models with thousands of programming statements it becomes a substantial task. It is also difficult to train modelers to avoid developing simulation models with deadlocks since the modeling process, due to its creative nature, is not readily conducive to formalization. This is especially true in the case of large models developed in modules at different locations by different modelers. Therefore, the fundamental concept of simulation modeling itself may have to be modified radically before modelers are trained to avoid programming in deadlocks (Fujimoto 1990). Furthermore, in large models with many entities seizing and relinquishing resources and entering and departing the system, deadlocks can still occur due to the randomness in the system and for the same reason modelers cannot anticipate when deadlocks can occur. Therefore, this warrants the design and development of an automated technique for detecting deadlocks in simulation models.

2 REVIEW OF RELEVANT RESEARCH

Past work relevant to this research has come primarily from the operating systems and database systems literature [Hunt 1986, Lee and Kim 1992, Coffman 1971]. In the existing literature, four conditions have been mentioned as necessary for a deadlock to exist and they are: mutual exclusion, hold and wait, no preemption, and circular wait [Peterson and Silberschatz 1983, Wojcik and Wojcik 1989]. Mutual exclusion means that a resource can be used only by one process at a time and a resource cannot be shared. If another process requests a busy resource, then the requesting process must be delayed until the resource has been released. The second condition, hold and wait, means that a process that is holding at least once resource is waiting to acquire additional resources that are currently being held by other processes. The third condition, no preemption, simply means that a resource cannot be preempted. A resource can only be released voluntarily by the process holding it, after the process has completed its task. The fourth condition, circular wait, means that there exist a set of waiting processes $\{p_0, p_1, \ldots, p_n\}$ such that p_0 is waiting for a resource held by p_1, p_1 is waiting for a resource held by $p_2, \ldots p_{n-1}$ is waiting for a resource held by p_n, and p_n is waiting for a resource held by p_0.

The existing deadlock detection algorithms analyze a system for the four mentioned necessary conditions. For the purpose of deadlock detection, a dynamic representation of the state of the process-resource interaction of the system is usually modeled as a graph. This graph is referred to as the system resource allocation graph or as the general resource graph. This graph consists of a pair G = (V,Z) where V is a set of vertices and Z is a set of edges. The set of vertices is partitioned into two types, $P=\{p_1,p_2, \ldots, p_n\}$, the set consisting of all processes in the system, and $R=\{r_1,r_2, \ldots, r_n\}$, the set consisting of all resource types in the system. Edges of the graph are represented by (u,v); that is, an edge starts at vertex u and ends at vertex v. The edges can be either request or assignment edges. A request edge is directed from the node of a requesting process to the node of a requested resource. An assignment edge is directed from the node of an assigned resource to the node of the process assigned. A system is deadlocked if this graph contains a directed cycle. A cycle is a path whose first and last nodes are the same. A sink is a node with no edges directed from it, and an isolated node is a node with no edges directed to or from it.

Another variation of the system resource allocation graph is simply the state graph. This is a directed graph whose nodes correspond to the resources and whose edges are defined so that if some process P has access to resource r_i and is waiting for access to resource r_j, then there exists an edge directed from node r_i to node r_j (Isloor and Marsland 1978). In this graph also, a directed cycle is necessary for a deadlock to exist. All the graphs found in the literature on the topic of deadlock are similar and refer to the same concept of cycles. Given one graph, a variation of it can be easily constructed and the use of a particular type of graph is dependent on the algorithm.

Several algorithms have been developed by computer scientists and their work can be classified under three categories: prevention, avoidance, and detection [Singhal 1989]. Prevention requires ensuring that deadlocks cannot take place and prevention methods have to be custom designed for each individual system. Several methods of prevention exist, including requesting all resources at once, preempting resources held, and linearly ordering resources (Isloor and Marsland 1980).

Deadlock avoidance techniques ensure that there is at least one way possible for all processes to complete execution after a resource is granted. The "banker's algorithm" is a common avoidance scheme (Isloor and Marsland 1980). This algorithm manages multiple units of a single resource by requiring that the processes specify their total resource needs at initiation time. In addition, each process acquires or returns resource units one at a time and denies the requests of processes whose needs exceed available resources

Detection requires only that a deadlock needs to be found, if there is one when the algorithm is invoked. Resolution breaks the deadlock or may provide information to help break the deadlock. Detection methods generally involve two steps: the construction and maintenance of the state graph and the searching of the state graph for cycles. To recover from the detected deadlock, resolution is required. Some possible methods for resolution include violating the mutual exclusion condition by allocating to several processes simultaneously, aborting one or more processes to break the circular wait condition, or preempting some resources from at least one of the deadlocked processes.

It can be seen from the existing literature that much has been done on deadlocks related to operating systems and distributed database systems. However, it can also be seen that very little has been done on deadlock detection and resolution in discrete event simulation models. Much of past research on deadlocks is directly applicable to discrete event simulation, but first the issues related to deadlock in simulation models and their specific requirements must be analyzed.

3 ISSUES RELATED TO DEADLOCKS IN SIMULATION MODELS

There following issues related to deadlock detection must first be analyzed before designing and developing an algorithm: (1) stage of deadlock detection in simulation model development, (2) possible scenarios of entity-resource interactions during a deadlock, (3) point at which deadlock detection must be enforced, and (4) other necessary features of deadlock detection.

3.1 Stage of Deadlock Detection

The first issue to address is where does deadlock detection fit in the stages of simulation model development. Verification and validation seem to be the obvious answer. In a sense, deadlock detection should be a part of both verification and validation. If a real system does not contain a deadlock, but the simulation modes, then the model is not true to the system simulated. However, when the deadlock is detected, the modeler may have to modify reality captured to prevent the deadlock from occurring.

Another possibility is that a real system can itself contain deadlocks and they could be captured in the model also. In this case, the simulation model will be true to the real system, but without a method for deadlock detection, the deadlocks may easily go undetected. The model may considered to be a valid one, but without a deadlock detection method, deadlocks may go undetected in both the model and the real system. If a method is available to detect deadlocks in the model, it will be beneficial to the real system also.

3.2 Entity-Resource Interactions During a Deadlock

The four conditions stated in the previous section for a deadlock to exist are applicable in simulation also. However, the possible scenarios of entity-resource interactions that can occur in simulations during a deadlock must be explored further. The possible scenarios are that: (1) each resource may have only one unit, and (2) each resource may have multiple units.

In the case of each resource having exactly one unit, a cycle found in the entity-resource interaction graph may imply that a deadlock has occurred. For example, if resources R_1 and R_2 and entities E_1 and E_2 are examined, and let it be assumed that R_1 is allocated to E_1, R_2 is allocated to E_2, and E_1 has requested R_2 as shown in Figure 2. Now if E_2 requests R_1, then there is a cycle as shown in Figure 3 and hence there is a deadlock. Thus, a cycle is a necessary and sufficient condition for the existence of a deadlock in this scenario.

If each resource can have multiple units, then the necessary and sufficient conditions change. For example, let it be assumed that three resources, R_1, R_2, and R_3 exist along with four entities, E_1, E_2, E_3, E_4. Let R_1 and R_2 have one unit of each and are allocated to E_2 and E_3, respectively. R_3 has two units available and they are allocated to E_1 and E_2. Further, E_1 has requested R_1 and E_2 also has requested R_1 as shown in Figure 4. Now suppose that E_3 requests a unit of R_3. A request edge (E_3, R_3) is added to the graph as shown in Figure 5. It can be seen from the figure that two cycles exist. Entities E_1, E_2, and E_3 are deadlocked. In this case, a cycle implies a deadlock and it is both necessary and sufficient.

In the scenario shown in Figure 6, if each resource is allowed to have more than one unit, then a cycle in the graph does not necessarily imply that a deadlock has occurred. For example, suppose that two resources R_1 and R_2 each have two units of resources. Then one unit of R_1 can be allocated to E_1 and the other unit can be

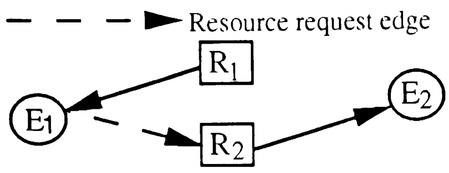

Figure 2: No cycles

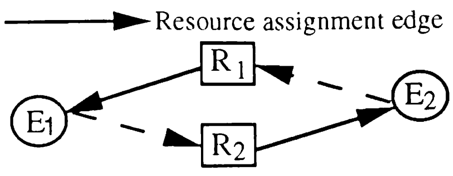

Figure 3: Cycle exists

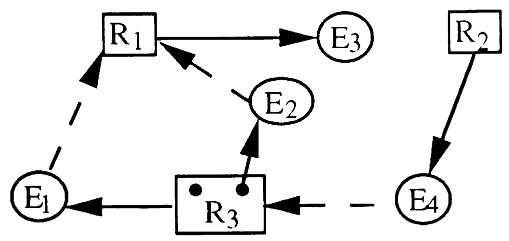

Figure 4: Resources with multiple units but no cycles

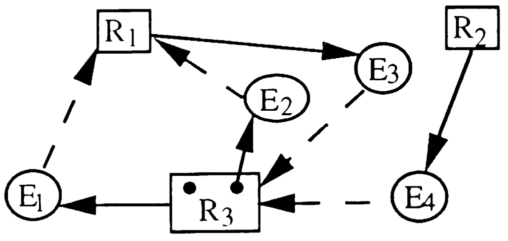

Figure 5: Resources with multiple units and 2 cycles

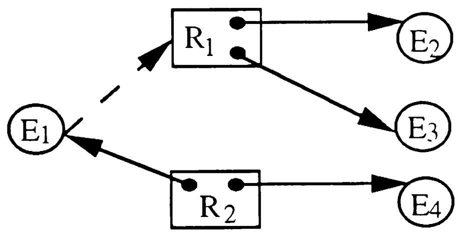

Figure 6: Resources with multiple units but no cycles

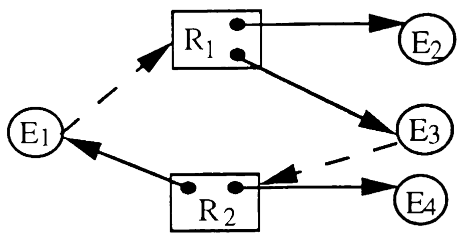

Figure 7: Cycle exists but no deadlock

allocated to E_2. Similarly, one unit of R_2 can be allocated to E_2 and the other unit can be allocated to E_3, thus eliminating the cycle and the possibility of a deadlock as shown in Figure 6. Now suppose that E_3 requests a unit of R_2 and E_1 requests a unit of R_1 as shown in Figure 7, since no resource unit is available, a request edge (E_3, R_2) is added to the graph. Now there is a cycle E_1-R_1-E_3-R_2-E_1 in the graph. However, no deadlock can occur in this example since entity E_4 may release its unit of resource R_2 at any time. Then, this unit of resource can be allocated to E_3 and break the cycle.

The above example introduces two types of deadlocks, namely, *temporary* and *permanent* deadlocks. An example of a temporary deadlock is illustrated by the temporary cycle exhibited in Figure 7. This type of deadlock, may temporarily exhibit the characteristics of a deadlock, but will be eliminated as soon as the necessary unit of a resource is available. An example of a permanent deadlock is illustrated in Figure 5 and this will remain indefinitely. From these examples, it is safe to conclude that if there are no cycles for multiple resources, then there will be no deadlocks. If however, there are cycles, then the model may be in a state of deadlock. Therefore, cycles are a necessary, but not a sufficient condition for deadlocks.

3.3 Point of Deadlock Detection in Simulation

Assuming for now that there is an algorithm available for deadlock detection, the next issue to address is at what point during a simulation run, deadlock detection must be enforced. There are several possible answers to this issue and they are discussed below.

If the deadlock detection algorithm is invoked at the beginning of a simulation run and remains in effect throughout the simulation, it can have a serious impact on the simulation run time. Checking for deadlocks every time a resource is requested and updating the status of the entity-resource interaction graph can be very inefficient and time consuming. It is also unnecessary to do this since once a deadlock has occurred it will remain in effect for the rest of the simulation run. The advantage of this approach is that the existence of a deadlock can be detected and displayed as soon as it has occurred.

Instead of invoking the deadlock detection algorithm at the beginning of the simulation, it can be invoked periodically to check the existence of deadlocks. This can be more efficient than the previous alternative and less time consuming. However, deciding how often to invoke the algorithm may be difficult and the detection of the existence of a deadlock may not be immediate as in the previous case.

A third alternative would be to invoke the deadlock detection algorithm at the end of the simulation run. and this would still result in the detection and display of deadlocks. If a deadlock is a permanent one, it will remain in effect for the entire simulation run, and therefore, the algorithm need not be invoked from the beginning or periodically during the simulation run. The disadvantage is that the entire simulation may have to be run before detecting a deadlock, but the detection time spent on this may be less compared to the other two alternatives. This alternative seems to be a better and an efficient choice when compared to the other two.

3.4 Features for Deadlock Resolution

The final issue that must be analyzed is related to the resolution of deadlocks detected. In operating systems and distributed database systems, deadlock resolution is much more crucial since they deal with actual and not simulated systems and aborting a process may be destructive. In simulation models, deadlock detection is more important than resolving them. Resolving a deadlock correctly in a model is much harder than it may appear since it requires a thorough knowledge of the entity-resource interactions and their constraints.

In a simulation model, an easier way to resolve a deadlock may be to preempt a deadlocked entity holding a resource. However, preemption requires attention to three issues (Peterson and Silberschatz 1983): selecting the proper entity to abort, rollback, and starvation. Rollback will involve turning back the simulation clock to a certain point so that entity-resource interaction can take place again without a deadlock. Starvation will involve denying an entity access to a requested resource for a long time. Once an entity has been selected, then the extent of rollback has to be determined. However, selecting a proper entity for preemption is easier said than done in a conceptual model and the cost of doing these tasks will be generally prohibitive compared detecting the deadlocks and leaving it to the modeler to resolve it by correcting the model. Further, in some cases such as manufacturing simulations, preemption may not be possible since it may result in a part to be scrapped. Therefore, a better solution to deadlock resolution in simulation models is not to resolve it at all since a deadlock cannot be correctly resolved without additional information about the entities deadlocked and if the modeler is required to provide this information, then it may be much easier for the modeler to resolve the deadlock.

4 THE DEADLOCK DETECTION ALGORITHM

An algorithm for deadlock detection should be based on the following requirements that resulted from the analysis of the issues presented in the previous section:

1. The algorithm should be invoked only at the end of the simulation run.
2. The algorithm should capable of detecting different types of deadlocks caused by resources with one unit or multiple units.
3. The algorithm should distinguish between temporary and permanent deadlocks.
4. The algorithm should identify deadlocks regardless of if the deadlock existed in the system modeled or not.

The implementation details of the algorithm are not presented here and only a simple, generic version that illustrates the concepts is demonstrated with examples.

4.1 Details of the Algorithm

This algorithm basically eliminates candidates that cannot be involved in a cycle and uses the concept of sink nodes. Sink nodes have only one edge connected to them and have either a directed edge to them and no edge leaving them, or have a directed edge leaving them, but no edge coming to them. Any node that is a sink node is eliminated as a candidate for a cycle and a deadlock. Only resources that do not have a request made for them and are assigned to an entity can be sink nodes. Entities can be sink nodes it they have either a request edge or an assignment edge.

The algorithm maintains in a linked list information about the resource allocations and requests by entities. During initialization of the algorithm, new nodes are inserted for each resource. New nodes are also inserted for every entity that seizes or requests a resource. An entity that has been allocated a resource will have new nodes for each request made for another resource. The linked list contains the following five elements:

1. Node identification (resource or entity)
2. Identification number of resource or entity
3. Unit number, in the case of a resource
4. In the case of a resource node, no. of the assigned entity, otherwise no. of the allocated resource
5. Number of the resource requested by entity

The algorithm searches the linked list for the first busy resource node, i.e, a resource that has been allocated to an entity. The entity number is stored in a

```
BEGIN:
A is the set of all resource units
B is a set of all busy resource units in A

For each resource R in B do
    Mark R as "checked"
    Find the entity E assigned to R and mark it as
        "checked"
    If entity E requests another resource then
        do while(($S \neq R$) or (S or E is not a sink node) or
                T is not marked "checked")
            Find the resource S requested by E and check
                whether $S \subset B$
            Find the entity F assigned to resource S
                If entity F requests resource T then
                    $E \leftarrow F$
                    $S \leftarrow T$
                    Mark S as "checked"
    endwhile
    Endif

    If S = R then
        List deadlocked nodes
        Unmark "checked" resources and entities
    Endif
END.
```

Figure 8: The Deadlock Detection Algorithm

variable VAR1 and resource number is stored in variable VARCYCLE. Then the search is made for the entity node requesting a resource and whose number is equal to VAR1. If the entity node is located, the requested resource number is stored in a new variable VAR2. If the entity node was not located, the search skips over to the next busy resource node. If an entity node has been found, the search continues and identifies the busy resource node whose number is equal to VAR2. The number of the entity that has been allocated the resource is stored in variable VAR3. Next, the entity requesting a resource and whose number is equal to VAR3 is identified. The requested resource number is stored in variable VAR4. Then VAR4 and VARCYCLE are compared to see if they are equal. If they are, a deadlock is detected and the corresponding nodes are displayed. If a deadlock is not detected, VAR2 is set to the value of VAR4 and the algorithm continues to check the other busy resource nodes in the same sequence as in the process of identification of variables VAR3 and VAR4, and the process is repeated to identify deadlocks. Figure 8 shows a generic version of the algorithm which has been implemented in C and tested with discrete event simulation models developed in C.

Table 1: Linked lists created for the first example

No Deadlock

R	1	1	1	
R	2	1	2	
E	1	1	1	
E	1	0	0	2
E	2	1	2	

Deadlock Exists

R	1	1	1	
R	2	1	2	
E	1	1	1	
E	1	0	0	2
E	2	1	2	
E	2	0	0	1

Table 2: Linked lists created for the second example

No Deadlock

R	1	1	3	
R	2	1	4	
R	3	1	1	
R	3	2	2	
E	1	1	3	
E	1	0	0	1
E	2	2	3	
E	2	0	0	1
E	3	1	1	
E	4	1	2	
E	4	0	0	3

Deadlock Exists

R	1	1	3	
R	2	1	4	
R	3	1	1	
R	3	2	2	
E	1	1	3	
E	1	0	0	1
E	2	2	3	
E	2	0	0	1
E	3	1	1	
E	3	0	0	3
E	4	1	2	
E	4	0	0	3

4.2 Example Applications of the Algorithm.

Two examples are presented in this section to illustrate the application of the algorithm discussed above. The first example, shown in Figure 2, contains an undeadlocked model of the robot machine model shown in Figure 1. In this example, there are 2 entities and 2 resources at the end of the simulation that need to be considered for deadlock detection. Resource R_1 is assigned to entity E_1, resource R_2 is assigned to entity E_2. E_1 requests for R_2. The search does not detect any deadlocks because E_2 and R_1 are sink nodes.

The deadlocked version of the above model is shown in Figure 3. In this example, the two entities vie for the resources held by the other, thus causing a cycle as shown in the figure. After the first pass of the algorithm, the deadlock is detected thus indicating that there is a cycle formed by entities E_1 and E_2, and resources R_1 and R_2. Only one pass is needed to detect the deadlock in this example.

The second example, shown in Figure 4, contains an undeadlocked model. There are 3 resources, R_1 and R_2 of 1 unit each and R_3 with 2 units. Resource R_1 is allocated to E_3, R_2 is allocated to E_4, one unit of R_3 is allocated to E_1 and the other unit assigned to E_2. It can be seen from Figure 4 that R_2 and E_3 are sink nodes since R_2 does not have a directed edge leading to it and E_3 does not have a directed edge leading towards another resource. The algorithm passes through all the resource nodes but no deadlock cycles are detected.

Figure 5 shows the deadlocked model of the above example with 4 resources and 3 entities. It can be seen that there are 2 cycles. One deadlock involves R_1, E_1, R_3, and E_3 and the other involves R_1, E_2, R_3, and E_3. The algorithm detects both the deadlock cycles and the corresponding nodes are displayed. Table 1 and 2 show the link lists created by the algorithm for the deadlock and undeadlocked cases of the two examples.

5 CONCLUSIONS AND RECOMMENDATIONS

In this paper, the issues related to deadlock detection and resolution were presented and discussed. From the issues presented, the requirements for a deadlock detection algorithm were decided. A generalized algorithm for detecting deadlocks in discrete event simulation models was presented and this algorithm is designed to be invoked at the end of a simulation run. The application of the algorithm was also illustrated with examples.

The issue of deadlock resolution was not included in the algorithm since resolution requires additional information about the simulation model itself. This is a difficult requirement for a generalized algorithm to handle and this issue must be explored further. One recommendation to handle this is to design this as an interactive, optional step for which the modeler can input some additional information about the deadlocked entities and resources.

REFERENCES

Coffman, E. G. Jr., Elphick, M. J. and Shoshani, A. 1971. "System Deadlocks," *ACM Computing Surveys*, Vol. 3, No. 2, pp. 10-17.

Fujimoto, R. M. 1990. "Parallel Discrete Event Simulation," *Communications of the ACM*, Vol. 33, No. 10, pp. 29-53.

Hunt, J. G. 1986. Detection of deadlocks in Multiprocessor Systems," *SIGPLAN Notices*, Vol. 21, No. 1, pp. 46-48.

Isloor, S. S., and Marsland, T. A. 1978. "An Effective On-Line Deadlock Detection Technique for Distributed Database Management Systems," *Proceedings of IEEE Compsac 78*, pp. 283-288.

Isloor, S. S., and Marsland, T.A. 1980. "The Deadlock Problem: An Overview," *Computer*, September, pp. 58-77.

Law, A. M. and Kelton, W. D. 1991. *Simulation Modeling and Analysis*, Second Edition, McGraw-Hill, New York.

Lee, D., and Kim, M. 1992. "A Distributed Scheme for Dynamic Deadlock Detection and Resolution," *Information Sciences*, Vol. 30, No. 5, pp. 149-164.

Leung, Y. T., and Sheen, G. 1994. "Resolving Deadlocks in Flexible Manufacturing Cells," *Journal of Manufacturing Systems*, Vol. 12, No. 4, pp. 291-304.

Pegden, C. D. et al. 1990. *Introduction to Simulation and SIMAN*, McGraw-Hill, New York.

Peterson, J. L., and Silberschatz, A. 1983. *Operating System Concepts*, Addison-Wesely, Reading, Massachusetts.

Singhal, M. 1989. "Deadlock Detection in Distributed Systems," *Computer*, Vol. 22, No. 11, pp. 37-48.

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

AUTHOR BIOGRAPHIES

MURALI KRISHNAMURTHI is an Assistant Professor in the Department of Industrial Engineering at Northern Illinois University. His research interests include simulation, manufacturing systems, AI/ES, operations research, and information systems. He is a member of IIE, SME, and AAAI.

AMAR BASAVATIA is a graduate student in the Department of Industrial Engineering at Northern Illinois University. His research interests include simulation and manufacturing systems.

SANJEEV THALLIKAR is a graduate student in the Department of Industrial Engineering at Northern Illinois University. His research interests include simulation, manufacturing, and information systems.