Deadlock Detection and Resolution for Discrete Event Simulation: Multiple-Unit Seizes

S. Venkatesh, J. Smith, B. Deuermeyer, and G. Curry

Department of Industrial Engineering
Texas A&M University
College Station, Texas 77843-3131

Abstract

This paper develops an automatic scheme to detect and resolve deadlocks in discrete-event simulation systems with entities capable of requesting multiple units of a resource. The research extends earlier deadlock work on discrete simulation systems with unit resource requests. The purpose of the deadlock handling scheme is to provide for additional capabilities in discrete simulation systems. This is accomplished by endowing the simulation system with appropriate data structures and algorithms. The algorithms presented are based on a graph model of deadlocks in the simulation system. The proposed algorithms identify different categories of permanent and transient deadlocks in the simulation system. A deadlock resolution scheme is also developed in the case of group-processing for permanent deadlocks.

1. Introduction

Simulation modeling and analysis has long been a major tool in the operational analysis of manufacturing systems. A subtle and often problematic aspect of simulation that has not been adequately addressed is deadlocking (Jun and Perros [11], Pierce and Drevna [17], Krishnaswamy and Ghiya [13]). Currently available commercial simulation systems provide no tools for detecting or resolving deadlocks. In response, Deuermeyer *et al.* [5] present procedures to detect and resolve simulation deadlocks under conditions of single-unit resource requests, and provide implementation and computational experience with the MOR/DS (Curry *et al*. [4]) simulation package. The intent of this paper is to present an extension to the original detection and resolution procedures that will facilitate its use under more general multiple-unit request conditions. A preliminary version of the conceptual approach of this paper is given in Venkatesh *et al*. [21].

The effects of a deadlock are usually noticeable in the simulation output, but the effects can be difficult to detect in certain circumstances. Typically, the effects of a deadlock show up as highly under-utilized resources associated with excessively long waiting lines and/or high resource utilization without a correspondingly high quantity of completed products. Informally, deadlock occurs when a set of two or more entities request resources held by other entities in the same set, and cannot return the resources they hold until they are granted the requested resources. For example, consider the simulation of a manufacturing system where two jobs, each controlling a different machine, request access to the machine held by the other. This situation leads to an interminable wait since neither job can relinquish control of its machine without being granted the access to the requested machine. As a consequence, both machines are highly utilized without a corresponding number of products being completed. Wysk *et al.* [23] have identified this problem in the context of real-time control of unmanned manufacturing systems. Similar deadlock situations have been studied in computer operating systems, databases, and communication systems (Coffman *et al.* [3], Knapp [12], Natarajan [15]).

Deadlock occurs in simulation systems due to the dynamic interaction of entities and resources in the course of the simulation. Deuermeyer *et al.* [5] use a graph formalism that represents this dynamic interaction to define deadlock characteristics in discrete simulation systems. The procedure presented detects and resolves deadlocks under the condition that entities are restricted to single-unit resource requests. A distinction is made between self-resolvable and non self-resolvable deadlocks in simulation systems. Self-resolvable deadlocks are called *transient deadlocks,* whereas non self-resolvable deadlocks are called *permanent deadlocks.*

Transient deadlocks have the potential to resolve themselves over time. Such a situation can occur when an entity currently using a multi-unit resource returns the resource unit(s) in its possession. The re-introduction of additional resource units into the simulation can allow other deadlocked entities to use the freed resource and, consequently, relieve the blockage situation in the simulation. Although transient deadlocks have the potential for self-resolution, they cause a loss of processing time and consequent degradation in the simulation generated statistics. In the case of permanent deadlocks, external intervention is required in order for the deadlocked resources to be freed and the entities to continue through the simulation.

In the multiple-unit resource request case, there is a similar distinction between self-resolvable and non self-resolvable deadlocks. However, unlike the single-unit request case, we must also distinguish between two categories of permanent deadlocks. These are called *pending deadlock* and *total deadlock*. Pending deadlock occurs when some of the deadlocked resources are currently processing entities and can have additional waiting entities that can be processed without external intervention. However, the deadlock will never resolve itself and will eventually degrade into total deadlock.

A transition diagram of the simulation deadlock states under a multiple-unit request condition is shown in Figure 1. The vertices in the figure represent possible deadlock states in the natural evolution of a simulation realization and the edges define feasible state transitions. The simulation starts in a deadlock-free state and can move through the different deadlock states as resource requests and assignments are made during the course of the simulation. Note that once the simulation enters the nonself-resolvable states of either pending or total deadlock, external intervention is required for the deadlocked resources and entities to be freed.

The objective of this paper is the development of algorithms that are capable of detecting, categorizing, and resolving deadlocks under the multiple-unit request condition. In the next section, we present a brief literature review and motivational examples of deadlock. These are followed by the necessary graph-theoretic notions and the evolution of deadlock in simulation. Finally, the paper develops the characterization of the occurrence, detection, and resolution of deadlocks

2. Literature Review

Deadlock studies have been reported in various areas of computer science where conflicts arise due to the allocation of resources, such as data objects in database systems (Knapp [12], Singhal [20]), printers and other devices in operating systems (Habermann [8], Coffman *et al*. [3], Holt [10]), and messages in communication systems (Natarajan [15]). Wysk *et al*. [23] also identify resource contentions due to conflicting part-routings in the context of automated manufacturing systems. The typical approach for modeling deadlocks involves the use of a graph-based representation to model the relationship between contending entities, and to control and manage deadlocks. Throughout this section we will use the term *entity* to represent the agent that requests a resource, such as a transaction in a database, a process in an operating system, or a part in a manufacturing system.

Researchers have identified three standard approaches to manage deadlock situations: prevention, avoidance, and detection and resolution. While the first two approaches ensure that deadlock will never occur, the last approach allows deadlock to occur and resolves it appropriately. Deadlock prevention averts deadlock by imposing constraints on how the resource requests are made, ensuring that the necessary conditions for deadlock can not occur. Examples of prevention strategies include allocating all of the resource requests before each entity begins execution or a waiting entity returns resources requested by an active entity (Singhal [20]). Examples of preventive measures in manufacturing systems control include providing a large number of in-process buffers and uni-directional batching of jobs (Wysk *et al*. [23]). In the case of simulation, such an approach could potentially restrict the level of realism that can be implemented in the simulation model. For example, in a simulation of a manufacturing situation, this would mandate acquiring both the operator and the machine during a routine setup, even if the machine is not required for the setup.

Deadlock avoidance works by ensuring that the sufficient conditions for deadlock are not met. In deadlock avoidance, every resource request is analyzed prior to execution, and only allowed to occur if it does not cause a deadlock. This analysis requires knowledge of future events of the current system realization. Habermann [8] presents a deadlock avoidance algorithm in which each entity declares the maximum possible number of units of each resource it may need. Given this information, as each resource request is made, an assignment is authorized only if there exists at least one sequence of executions that does not evolve to a deadlock. Wysk *et al*. [23], and Cho *et al*. [2] present related approaches for avoiding deadlocks in the context of automated manufacturing system control. The stochastic nature of simulation, however, makes it impossible to analyze the effects of future events. Furthermore, the general deadlock avoidance problem has been shown to be NP-Complete even when times are known with certainty [6].

In the deadlock detection and resolution strategy, resource requests are granted without considering the potential for deadlock. At appropriately chosen times, a deadlock detection procedure is invoked. If the procedure identifies a deadlock, the deadlock is resolved. Given that deadlock prevention can be restrictive and deadlock avoidance is infeasible in most simulations, we adopt the detection and resolution strategy for handling deadlocks. The deadlock detection problem has been studied extensively by previous researchers (Cho *et al*. [2], Coffman *et al*. [3], Holt [9,10], Natarajan [15], Wysk [23]). Knapp [12] and Singhal [20] present comprehensive surveys of deadlock detection procedures in distributed systems. In the context of the real-time control of manufacturing systems with no alternate part-routings, deadlock detection reduces to the detection of cycles in the graph model (Cho *et al*. [2], Wysk *et al*. [23]).

Many algorithms presented to detect/prevent deadlocks reorder the waiting entities into a non-decreasing list of entities by their request size. The algorithms then attempt to find at least one possible sequence of executions that does not result in deadlock (Habermann [8], Coffman *et al*. [3], Holt [10]). In the case of simulation systems, re-ordering of the waiting entities may violate the reallocation rule of the resource. The reallocation rule decides whether the resource queue allows (or disallows) late arrivals in the queue to pass (or not pass) stalled entities. In the case of multiple-unit requests, such a policy can alter the outcome of the deadlock detection by erroneously designating a deadlocked simulation to be free from deadlock. This will be demonstrated by examples in the following section. In this paper, we present a mechanism by which we can model different reallocation policies in a general simulation system. This transformation allows the use of existing procedures for deadlock detection and, further, allows for developing a generic deadlock detection procedure ignoring nuances due to different resource allocation rules.

Resolution is the second half of the detection/resolution strategy. The problem of *optimally* resolving deadlocks is known to be NP-hard (Gray [7], Leung and Lai [14]). In the case of real-time control of manufacturing systems, resolution consists of removing a deadlocked entity from the resource (machine) it holds, placing it in a temporary buffer, and reallocating the released resource to a waiting entity (Cho *et al*.[2], Wysk *et al*. [23]). The resolution procedure in computer applications typically chooses a set of entities to be aborted and restarted or partially rolled-back in order to break the deadlock. For example, an operating system would resubmit a job to start all over again or a database system would abort a set of transactions to be restarted.

Due to the varied nature of deadlocks in a general simulation system, such removal or roll-back of entities are not directly applicable in the context of modeling manufacturing systems. Deuermeyer *et al*. [5] present two situation-based deadlock resolution schemes that enable the simulation to automatically recover from deadlock situations without any additional burden on the simulation modeler. Later in this paper we present a resolution scheme for multiple-unit requests.

3. Assumptions and Deadlock Examples

We are primarily concerned with entity and resource relationships that are modeled using the time-dependent graph described by Deuermeyer *et al*. [5]. Attention is restricted to the **seize** and **release** statements that generate these relationships in a general simulation model. Other aspects of the simulation model are not germane to this analysis and are therefore not considered. In this section we will present three examples of deadlocks in simulation. The first example demonstrates the evolution of a simulation that leads to a total deadlock with entities restricted to single-unit resource requests. The details of the detection and resolution procedures under these conditions are described in Deuermeyer *et al*. [5]. The next two examples demonstrate the effects of multiple unit resource requests and the reallocation rule used by the resource to refill unassigned units on the classification of the deadlock.

The MOR/DS simulation paradigm (Curry *et al*. [4]) that we employ is similar to the approaches taken in both SIMAN (Pegden *et al*. [16]) and GPSS (Schriber [18, 19]). Under this paradigm, entities acquire necessary units of a resource through a simulation structure called the **seize** statement, and return the acquired resource units through a simulation structure called the **release** statement. Entities can request one resource type at a time in the simulation and, thus, entities requiring more than one type of resource simultaneously must acquire them sequentially. Resources can have a capacity greater than one and there is a unique queue associated with each resource. Furthermore, we assume that the total number of units simultaneously held and/or requested by a single entity never exceeds the total capacity of the resource.

The **seize** statement maintains the number of available units and the prioritized queue of entities waiting for a resource. If a sufficient number of resource units are available when an entity makes a request, the **seize** statement assigns the appropriate number of resource units to the requesting entity and updates the available capacity. If the requested number of units is not available, the requesting entity is inserted in the queue according to the user-specified queue discipline (e.g., first-in, first-out (FIFO), last-in, first-out (LIFO), priority number, etc.). The **release** statement manages the number of available resource units and the resource queue as entities relinquish control of the resource. As resource units are returned by an entity, the **release** statement reallocates the resource to waiting entities using a *resource fill rule*.

The fill rule, designated as either *pass* or *no-pass*, determines whether entities waiting in a resource queue can pass other entities in the same queue. Passing of entities can be a significant factor in the multiple-unit seize case where the first entity in the queue requests more units than are currently unassigned. Obviously, under these conditions the first entity will be forced to wait in the queue. However, in some cases, requests from entities further back in the queue could be satisfied if those entities were allowed to "pass" the stalled entity. The SIMAN language uses a no-pass fill rule, while GPSS uses the pass fill rule. In the MOR/DS simulation language, the fill rule for each queue is specified by the analyst when developing the model.

3.1 Examples

In the illustrations that follow, we use rectangles to represent resources and circles to represent entities. Notationally we represent the i^{th} entity by e_i and the j^{th} resource by r_j. A resource assignment of k units to an entity is depicted with a directed edge labeled k from the resource to the entity (Figure 2a), while a resource request of k units is depicted with a directed edge labeled k from the entity to the resource (Figure 2b).

Note that we drop the labeling on the edges with unit requests or unit assignments without ambiguity. We assume that a request for a resource that can be satisfied changes immediately to an assignment edge. Hence, there is no need to represent a request from an entity to a resource that can be satisfied. In text, we use the notation (e,r,k) to represent a request of k units of resource r by entity e, and (r,e,k) to represent an assignment of k units of resource r to entity e.

Example 1. Consider a manufacturing process with a unit capacity machine (r_2) that requires an operator (r_1) for loading and unloading of jobs. Figure 3 shows the evolution of a simulation that leads to a deadlock. The simulation of the manufacturing process assumes that a part in control of either of the resources will not relinquish control of the resource it is holding until its request has been granted.

In Figure 3a, job 1 (e_1) has control of the operator (r_1) so that it can be loaded on the machine (r_2), while job 2 (e_2) is being processed on the machine. In Figure 3b, job 1 has requested the machine (r_2), which creates a new edge from e_1 to r_2. In Figure 3c, job 2 has finished processing on the machine and requests the operator to unload the machine. This event creates an edge from to e_2 to r_1. Since neither entity will relinquish the resource that it controls until its respective request has been granted, the simulation is deadlocked, and the two jobs will wait indefinitely. Furthermore, any other entities requesting either resource will also wait indefinitely. The simulation system as shown in Figure 3c is said to be in total deadlock.

Example 2. Consider the situation depicted in Figure 4. The system consists of two resources (r_1 and r_2) each having two units of capacity. The current state of the system is shown in Figure 4a, which gives the ordering of the different entities in the resource queues. Figure 4b displays the entity-resource graph for the situation depicted in Figure 4a.

In Figure 4b entities e_1 and e_2 hold one unit of resources r_1 and r_2, respectively, and are currently requesting two units of resources r_2 and r_1, respectively. By assumption, entities e_1 and e_2 will not relinquish the resources they hold until they acquire the requested resource units. Additionally, entities e_3 and e_4 control one unit of resources r_1 and r_2, respectively. Entities e_5 and e_6 are waiting for a single unit of resources r_1 and r_2, respectively.

After entities e_3 and e_4 have completed processing, entities e_5 and e_6 will be processed. However, entities e_1 and e_2 are deadlocked, and cannot be processed since they each are requesting two resource units. The simulation system in the state as shown in Figure 4b is said to be in *pending deadlock.* The distinction from the previous example is that some of the resource units can continue to process entities. In the present instance, entities e_3 and e_4 are currently being processed, and entities e_5 and e_6 will be processed after these entities are completed. Furthermore, if either of the resource queues has a pass-fill rule, then that resource can continue to process newly arriving entities, so long as arriving entities request only a single unit of the resource. In large simulation models, such a situation may go undetected, and, thus, lead to a misinterpretation of the simulation output. The key factor is that, for the no-pass filling rule, if and when entities e_1 and e_2 reach the front of their respective queues, the simulation will be in total deadlock.

Example 3. Reconsider the situation as shown in Figure 4a but with the associated entity-resource relationship graph as shown in Figure 4c. Here, entities e_1 and e_2 are currently requesting one unit of resource r_2 and r_1, respectively, and resources r_1 and r_2 allow passing of entities. Also, entities e_5 and e_6 are waiting for two units of resources r_1 and r_2, respectively. Now, because of the pass filling rule, all of the entities in the different resource queues will eventually gain control of their requested resources. Consequently, the situation depicted in Figure 4c is in transient deadlock. It should be noted that if neither of the resources allows passing of entities, the simulation system would be in pending deadlock.

In the above examples deadlocks occurred as a result of the requesting entities being unable to relinquish the resources currently in their possession. This feature is the primary reason for the occurrence of deadlocks in simulation systems. The examples clearly demonstrate the complications inherent in detecting deadlocks in simulation systems. These complications are due to the dynamic nature of the simulation, the multiplicity of request sizes by entities, and the filling rules of the resources.

4. Entity and Resource Relationship Graph

A directed graph $G = (V,E)$ has a vertex set V and an edge set E, where $(u,v,k) \in E$ implies there is an edge directed from $u \in V$ to $v \in V$ with a label $k \in N$, where N is the set of positive integers. A vertex $v \in V$ is called a *root vertex* (source) of V, if there is no $u \in V$ for which $(u,v,k) \in E$ (i.e., the in-degree of v is zero). A vertex $v \in V$ is called a *leaf vertex* (sink) of V, if there is no $u \in V$ such that $(v,u,k) \in E$ (i.e., the out-degree of v is zero). A vertex is called an *interior vertex* if both the in-degree and out-degree are non-zero. A vertex is called an *isolated vertex* if both the in-degree and out-degree are zero.

A directed graph G' is said to be a subgraph of directed graph G, if $G' = (V',E')$, where $V' \subseteq V$ and $E' \subseteq E$. Let $V' \subseteq V$, and G' be constructed by letting $E' \subseteq E$, where $(u,v,k) \in E'$ for all $u,v \in V'$ and $(u,v,k) \in E$. The set V' is called a *strongly connected component* (SCC) of V if for all vertices u and $v \in V'$, there is a directed path in G' connecting u to v, and another path connecting v to u. In this case, the subgraph G' is *induced* by V'. Let V' be an SCC and G' its induced subgraph; then we say V' is a *closed strongly connected component* (CSCC) if there is no edge in G directed out of V'. In the formalism developed below, strongly connected components represent the circular wait relationships indicative of deadlock and are therefore critical to succeeding developments.

We represent the entity-resource (ER) relationship by the directed graph $G_t = (V_t,E_t)$, for $t \geq 0$. Here, G_t denotes the ER graph at instant t, with vertex set V_t and edge set E_t. The vertex set contains all the resources of the system as well as those entities who own at least one resource unit and/or are waiting for a resource at instant t. Let G_t, $t \geq 0$, denote the ER graph over time, beginning with the configuration G_0, where V_0 consists of the set of all resources and E_0 is empty. Since set G_t changes over time due to the execution of a series of **seize** or **release** statements, only those entities that are waiting for a resource or possess a resource at time t will be included in the graph G_t. Idle resources are represented as isolated vertices. Hence the graph can change configuration only upon execution of a **seize** or a **release** statement and we can completely describe the dynamic behavior of the ER graph by appropriately updating G_t after the execution of each **seize** or **release** statement. We take the convention that this update process is right continuous (i.e., G_t denotes the ER graph immediately after the execution of a **seize** or a **release** statement).

In the algorithms to follow, we use terminology and pseudo-code notation resembling common programming languages. Bold-face type is used to indicate reserved words within this pseudo-code language. We assume the language provides capabilities to work with arbitrary lists of objects, and provides facilities to modify these lists via the commands **insert**, **delete**, and **get**. The algorithms presented do not rely on the implementation details of these commands and, therefore, we do not explicitly describe an implementation mechanism. The **insert** command places a specified element at a specified position in the list. The **delete** command removes and destroys a specified element from the list. The **get** command removes and returns a specified element from the list, so as to be able to manipulate the element in the algorithm. We use **get**(l,i) to access the i^{th} element from list l.

We use the following quantities in our descriptions: $\text{QUEUE}_t[r]$ represents the set of entities in the queue at resource r, $\text{USES}_t\ [r]$ represents the set of resources owned by entity e, $\text{OWNS}_t\ [e]$ represents the set of entities that own units of resource r, and $\text{WAITS}_t\ [e]$ represents the resource that entity e is waiting for. In addition, CAP[r] denotes the capacity of resource r, $\text{REM}_t[r]$ denotes the unassigned capacity (remaining capacity) of resource r, and $\text{REQ}_t[e,r]$ denotes the number of requested units of resource r by entity e. We drop the square brackets to represent a vector and the index (t) wherever appropriate.

Figure 5 illustrates the **seize** statement operation in the notation described above. Lines 1-2 add entity e to the vertex set of the graph if entity e currently owns no resource. Lines 3-8 check to see if there is available capacity in resource r to accommodate the request by entity e; if so, entity e is added to the ER graph with edge (r,e,k), else entity e is added with edge (e,r,k).

Figure 6 defines the operation of the **release** statement. Lines 1-3 remove the assignment edge (r,e,k) from the graph and update the set USES[r] as entity e returns the units of the resource r it is holding. Line 3 updates the available units of resource r. Line 5 deletes entity e from the graph, if it no longer owns any other resource. Lines 6-14 perform the filling function to reallocate the available resource units to waiting entities. The function **firstfit** returns an entity that satisfies the capacity constraints of resource r, or returns a null (Ø) if no such entity exists. Details of these functions are provided in Venkatesh *et al.* [22].

5. Conditions for Deadlock and Deadlock Detection

5.1 Preliminaries

The notion of an SCC (or CSCC) is important in the ER graph G_t, since the existence of an SCC indicates the occurrence of a deadlock in the simulation. It was shown by Deuermeyer *et al*. [5] that with the simulation restricted to single-unit resource requests, total deadlock corresponds to the presence of at least one CSCC in the ER graph. In the multiple-unit request case, we define total deadlock similarly. However, for the multiple-unit request case, some resource units of the deadlocked resource can be idle. The presence of a total deadlock in the simulation indicates the need for external intervention for the deadlocked entities and resources to be freed. Further, the presence of an SCC in the multiple-unit resource requests case can indicate pending deadlock, which also requires external intervention for the deadlock to be resolved. Here, the user can choose not to intervene at the instant of discovery to avoid penalizing non-deadlocked entities during resolution, since some units of the deadlocked resources continue to process entities.

Beginning with G_0, which is free from deadlock, we track the evolution of the simulation after each execution of **seize** or **release** statements to identify the occurrence of deadlocks. The existence of an SCC at instant t in the ER graph implies that a deadlock is present. If the SCC is closed, this deadlock can only be resolved by external intervention (total deadlock). If the SCC is not closed then some entities will be delayed and some resource capacities not operational. This situation may be temporary however, and it may be acceptable to let the simulation proceed for a time. If it is temporary and will eventually resolve itself, the deadlock is a transient deadlock. Otherwise, it is a pending deadlock.

Suppose at instant t an SCC that is not closed has been found in the simulation ER graph. At this point, entities in the SCC are blocked because *every* entity in the SCC is waiting for resources held by other entities in the SCC, and *some* of the entities in the SCC are waiting for resources held by entities not in the SCC. However, it may be possible that some entity not in the SCC might return a resource that could initiate a sequence of events that would cause the SCC to collapse. An example would be the return to operation of a machine under repair or off-shift. This sequence of events might be realized in the near future, might be realized over the long term, or might never be realized, depending on the specific sequence of events within the simulation run.

We use the concept of *reduction* (similar to that given in Coffman *et al*. [3]) as a method to characterize the fortuitous events that may clear an SCC in the ER graph. Assume for current discussion purposes that the ER graph contains a single SCC. The reduction process is designed to distinguish between pending deadlock and transient deadlock. Basically, reduction attempts to return resource units from entities not waiting for any resource in the ER graph (i.e., sink vertices in G_t). As these entities are reduced, additional resource units will be freed, that in turn may allow the reduction of more waiting entities, and so on. The reduction process will eventually reduce entities that are not waiting for any resource in the SCC but holding resource units contained in the SCC. The reduction of such entities return units of resources contained in the SCC, thereby either causing the SCC to clear (indicating transient deadlock), or become closed (indicating pending deadlock).

As an example, consider the situation depicted in Figure 4b. The reduction process would begin with the set of entities (e_1, e_2, e_3, and e_4). Initially, entity e_3 will return the unit of resource r_1 in its possession. Next, entity e_4 will return the unit of resource r_2 in its possession. Now, since neither entity e_1 nor entity e_2 can obtain their requested resource units, and, further, since neither entity will relinquish the resource that it holds until it obtains the requested resource, the reduction process will terminate. The reduction will terminate with both entities e_1 and e_2 unable to satisfy their request, indicating a pending deadlock.

Figure 7 presents the details of reducing entity *e*. Here, HOLDS[*e*,*r*] denotes an entity attribute that indicates the number of units of resource *r* held by entity *e*. The EFF_CAP[*r*] indicates the *effective available capacity* of resource *r*, which is the sum of the returned units of resource *r* due to reduction and the unassigned capacity of resource *r*, REM[*r*]. Note EFF_CAP denotes a vector of size *R*, where *R* is the number of resources in the simulation system, and maintains the available resource units during the reduction process.

The reduction process has two possible concluding states: either the SCC in the ER graph has been cleared or there remains a CSCC. In the former case the SCC is said to be reducible, and the simulation is in transient deadlock (Figure 1). In the latter case, the original SCC in the ER graph can never unravel by natural simulation events, and the SCC is said to be irreducible, so the deadlock is pending (Figure 1). This leads to the following definitions.

Definition 1 (Total Deadlock): A simulation is said to be in total deadlock at any instant *t* if the simulation ER graph contains a CSCC.

Definition 2 (Pending Deadlock): Given a non-closed SCC at instant *t*, if the reduction of this graph is a CSCC then the simulation is said to be in pending deadlock.

Definition 3 (Transient Deadlock): Given a non-closed SCC at instant *t*, if the reduction of this graph is not a CSCC then the simulation is said to be in transient deadlock.

Definition 4 (Deadlock free): A simulation is said to be deadlock-free if there does not exist an SCC in the ER graph.

It should be noted that if there exists a pending deadlock in the simulation, then the SCC that could not be cleared in the reduction process will eventually evolve into a CSCC (total deadlock). Likewise, an unresolved CSCC (total deadlock) can revert to an SCC (pending deadlock) if an entity request of a later arrival can be satisfied by the idle resource units of a deadlocked resource. Consequently, except when a deadlocked resource allows passing of entities, total deadlock is an absorbing state, and pending deadlock is a transient state in the simulation state transition diagram.

Initially, we check for the existence of a CSCC in the ER graph using the **search** function. The function **search** is a depth-first search based algorithm to detect SCCs and CSCCs in a graph and has been adapted from the work of Deuermeyer *et al*. [5]. A key property of the algorithm is that it runs in linear time of the maximum of the number of edges and vertices in the graph O(max {|*V*|, |*E*|}). If such a closed component is discovered in the ER graph, we know that the simulation is in total deadlock. If we find an SCC that is not closed in the ER graph, we reduce the graph (using the function **reduction)** to determine if the simulation is in pending deadlock. Of course, if the outcome of the **reduction** function indicates an absence of pending deadlock then the simulation is in transient deadlock. The function **reduction** is adapted from the work of Coffman *et al*. [3].

5.2 Management of Pending and Total Deadlock

In light of the above discussion there are two deadlock cases (total and pending) that require external intervention for the deadlock to be resolved. However, we may choose not to intervene in the case of a pending deadlock at the instant of its discovery in the ER graph. This leads to two possible deadlock handling principles: either we maintain an ER graph that is free from pending and total deadlocks at any instant, or we maintain an ER graph that is free from total deadlocks. In the following discussion we will develop the theory required to handle these two cases.

Assuming that pending and total deadlocks in the ER graph are detected and resolved continuously, we know that the ER graph is free from these deadlocks before the execution of **seize** or **release** statement (although transient deadlocks may exist). In order to retain a simulation free from pending or total deadlocks, it is sufficient to invoke the deadlock detection scheme (and resolve any pending or total deadlocks) after every unsuccessful resource request. A successful resource request at a **seize** statement necessarily creates a new sink vertex in the ER graph, and hence does not create a new SCC. Further, the execution of a **release** statement also cannot create a pending or total deadlock, since, if it could, the ER graph would be in pending or total deadlock before the **release** statement. It should be realized that the ER graph may contain a number of SCCs, however, by the assumption of continuous detection and resolution of pending or total deadlocks, we are assured that none of these SCCs are in pending or total deadlock.

Before proceeding further we present a useful lemma. The lemma establishes that all the edges that emanate from an SCC are (*r*,*e*,*k*) edges, i.e., $r \in$ SCC and $e \notin$ SCC. Since we distinguish between transient and pending deadlocks in the ER graph, the lemma assists in appropriately updating the effective capacity of a resource (EFF_CAP[*r*]) in the deadlocked component ($r \in SCC$) during the

search procedure. This is accomplished by identifying the appropriate edges as **search** traverses the ER graph during the depth-first search procedure.

Lemma 1: Let $(u,v,k)\in E_t$, where $u,v\in V_t$ at instant $t>0$ in the ER graph G_t. Assume there is an SCC in G_t such that $u\in$ SCC and $v\notin$ SCC. Then u is a resource vertex and v is an entity vertex in G_t.

Proof: We have $u\in$ SCC and $v\notin$ SCC. If u is an entity, then either an edge needs to be incident on more than two vertices or the outdegree of an entity should be greater than one (a contradiction). Thus every edge (u,v,k) in the ER graph such that $u\in$ SCC and $v\notin$ SCC must necessarily be a (r,e,k) edge.

To detect pending deadlocks we need to execute the **reduction** function once we discover an SCC that is not closed in the ER graph. An unsuccessful resource request introduces an edge (e,r,k) in the ER graph and starts deadlock detection. Now the **search** function is called as a result of the invocation of deadlock detection, which creates a depth-first search tree rooted at resource r. By the assumption of continuous detection and resolution, any pending deadlocks that occur are necessarily contained in the SCC created due to the unsuccessful resource request. Consequently, the SCC that contains the pending deadlock is rooted at resource r, and by the definition of an SCC necessarily contains resource r. We call the SCC in the ER graph that contains resource r as the *principal strongly connected component* (PSCC). Thus, it is sufficient to execute the reduction process exclusively on the PSCC, rather than on every existing SCC in the ER graph.

Again by the assumption of continuous deadlock detection and resolution, it can be argued that any CSCC that is discovered in the ER graph necessarily has to be rooted at resource r, and hence be the PSCC. As a result, it is sufficient to look for pending or total deadlocks exclusively in the PSCC, which is established in the following theorem.

Theorem 1: Let G_S and G_t be the simulation ER graph at instants $s>0$ and $t>0$, respectively, such that $s<t$. Also, let $(e,r,k)\notin E_S$ and $(e,r,k)\in E_t$ such that $E_S \cup (e,r,k) = E_t$. If G_S is not in pending or total deadlock and G_t is in pending or total deadlock, then resource r is contained in the SCC $\in G_t$ that causes the total or pending deadlock (i.e., $r\in$ PSCC).

Proof: Either entity e or resource r in edge (e,r,k) has to be contained in the SCC$\in G_t$, otherwise either the edge is incident on more than two vertices in the ER graph G_t, or the ER graph G_S is in total or pending deadlock before the introduction of the edge (e,r,k), which is a contradiction. Further, Lemma 1 establishes that it is not possible to have $e \in$ SCC and $r \notin$ SCC in any ER graph.

Informally, **search** identifies SCCs farther away from the root of the depth-first search tree before identifying the SCCs closer to the root. By construction, the PSCC is the last component identified by **search** since the depth-first search tree is rooted at resource r. Further, notice that $r \in$ PSCC and $e \notin$ PSCC occurs when the resource queue for r has a no-pass filling rule. If the resource queue has a pass filling rule then entity e can be eventually processed. This is because the ER graph without the request by entity e is not in pending deadlock, and hence all the entities in the resource queue can be processed. Now entity e can be processed unless the requested units (k) is greater than the capacity of resource r (CAP[r]), which is a contradiction. Finally, note that if there exists a total deadlock then **search** identifies the CSCC, and by Theorem 1, this necessarily has to be the PSCC.

By the result of the preceding theorem we can afford the luxury of ignoring all but the PSCC for executing the **reduction** function. This implies that had we executed the **reduction** function on the entire ER graph, any pending deadlock that we discover necessarily has to be in the PSCC. We now establish the conditions necessary to execute the reduction process exclusively on the PSCC.

Entities in the PSCC may be in pending deadlock due to the fact that they may be waiting for resource(s) in the PSCC. Lemma 1 assures us that entities in the PSCC cannot be waiting for resource(s) outside of the PSCC. However, it is possible for some units of resources in the PSCC to be held by entities outside of the PSCC (Lemma 1). Also, Theorem 1 establishes that these resource units would be eventually returned during a reduction process, for any pending or total deadlocks are necessarily contained in the PSCC. Thus we need to find the effective available capacity of the resources in the PSCC as we invoke **reduction** on the PSCC. The effective available capacity of resource $r \in$ PSCC is obtained by incrementing the remaining capacity REM[r] by HOLDS[e,r] for all entities $e \notin$ PSCC such that $(r,e,k) \in E$.

To compute the effective available capacity, we will show that it is sufficient to consider exclusively $(r,e,k)\in E$ edges such that $r\in$ PSCC and $e\notin$ PSCC, among all component (singleton or otherwise) interactions in the ER graph. It should be noted that all edges $(u,v,k)\in E$ such that $u\notin$ PSCC and $v\in$ PSCC that are incident on the PSCC are irrelevant to the classification of the deadlock situation in the PSCC. An exception to this would be when a newly introduced edge (e,r,k) causes a pending deadlock, such that $e\notin$ PSCC and $r\in$ PSCC. Thus, we have established that it is sufficient to check for pending deadlock in the PSCC, implying that the classification of the pending deadlock situation in the simulation is the same as that of the PSCC.

Given that the depth-first search tree is rooted at resource r, all edges incident on the PSCC necessarily have to be from a disconnected part of the ER graph. This follows from the fact that every vertex in a connected graph has to be a descendant of the root of the depth-first search tree. Furthermore, any vertex that is a descendant of the root with a path to any vertex in the PSCC is also contained in the PSCC. In the following lemma we establish that all edges that are incident on the PSCC, with the exception of the one noted above, cannot alter the classification of the pending deadlock situation. Otherwise, this would imply that some entity $e\in$ PSCC is waiting for a resource outside of the PSCC or that the ER graph before the introduction of the edge (e,r,k) was already in pending deadlock.

Lemma 2: Let $(e,r,k)\in E_t$ create a PSCC in pending deadlock in the ER graph G_t at instant $t>0$. If $(u,v,k)\in E_t$ such that $u\notin$ PSCC and $v\in$ PSCC, where $u\in V_t - \{e\}$ and $v\in V_t$ in G_t, then $(u,v,k)\in E_t$ cannot alter the pending deadlock classification of the PSCC.

Proof: Clearly, u is not a descendant of either $e\in V_t$ or $r\in V_t$ in the depth-first search tree. If so, this would imply either that $u\in$ PSCC or the outdegree of entity e is greater than 1. Now, either u is a resource or u is an entity.

> *Case 1:* Suppose u is a resource. Then the outcome of the pending deadlock classification is unaltered since entities in PSCC are not waiting for resource u. If any of the entities in PSCC were waiting for resource u, then resource u would be part of PSCC, and would not be in a disconnected part of the ER graph.

> *Case 2:* Suppose u is an entity. Then by the fact that the ER graph was not in pending deadlock before edge (e,r,k) was introduced shows that u cannot influence the deadlock classification. Otherwise, if any of the entities are waiting for resources held by entity u, then as before we have that u would not be in a disconnected part of the ER graph.

Based on Lemma 2 we may ignore the edges incident on the PSCC without affecting the outcome of the reduction procedure. Furthermore, Lemma 1 established that all edges that emanate from the PSCC are (r,e,k) edges such that $r\in$ PSCC and $e\notin$ PSCC. This property is used to enhance **search** so that it may maintain the effective available capacity of the resources while processing (r,e,k) edges in the ER graph. The motivation to enhance **search** is to be able to provide the user the capability to switch the **reduction** procedure on or off, depending on single-unit or multiple-unit resource requests, thereby improving the efficiency of the execution of the simulation.

Depth-first search traverses a directed graph in the forward direction as long as possible. The edge set is partitioned into tree edges, back edges, forward edges, and cross edges. Note: If u and v are nodes in a graph with the depth-first number given by DFNUMBER[u] and DFNUMBER[v], respectively, then (Aho *et al*. [1])

(a) (u,v,k) is a *tree edge* if u is partially explored, v is newly visited, and DFNUMBER[u]<DFNUMBER[v];

(b) (u,v,k) is a *back edge* if u and v are partially explored and DFNUMBER[u]>DFNUMBER[v];

(c) (u,v,k) is a *forward edge* if u is partially explored, v is fully explored, and DFNUMBER[u]<DFNUMBER[v]; and

(d) (u,v,k) is a *cross edge* if u is partially explored, v is fully explored, and DFNUMBER[u]>DFNUMBER[v].

In order to process the information properly we need to be able to distinguish between different types of $(r,e,k)\in E$ edges, with $r\in$ PSCC and $e\notin$ PSCC, in **search**. By definition, any edge $(r,e,k)\in E$ cannot be either a back edge or a cross edge in the ER graph, otherwise we would have entity $e \in$ PSCC. Thus we are only concerned with tree and forward (r,e,k) edges, with $r \in$ PSCC and $e \notin$ PSCC, to calculate the effective available capacity of the resources.

This information is updated appropriately for trees and forward edges while invoking the **search** function. The total effective capacity is retained in EFF_CAP. Also, **search** naturally returns the entities in the PSCC in the process of detecting SCC (or CSCC) in the ER graph. Details of these functions are provided in Venkatesh *et al*. [22]. Thus, at the conclusion of the **search** function we have the set of entities in the PSCC and the effective available capacities of the resources to execute the **reduction** function, if **search** does not identify a CSCC in the ER graph.

Finally, if we choose to ignore the occurrence of pending deadlocks, but detect and resolve the occurrence of total deadlocks continuously, then we do not need to execute the **reduction** function on the ER graph. Now the deadlock detection procedure is simple and straightforward, and involves the detection and resolution of CSCC in the ER graph using the **search** function. In addition to invoking the deadlock detection routine at each unsuccessful **seize**, we would also need to invoke detection if the filling rule fails to reassign the available resource units in the **release** statement. Detection is required here since failure to reallocate available units of the resource may indicate the creation of a CSCC in the ER graph, possibly an evolution of an unresolved pending deadlock to a total deadlock situation.

Also, note that the depth-first search is rooted at the resource r that is unable to reassign the available resource units in the **release** statement. By construction, and assuming continuous detection and resolution of total deadlocks, the CSCC that we may discover is necessarily rooted at resource r. By our definition this component is necessarily the PSCC. In the following subsection we will present the algorithms necessary to detect deadlock in simulation systems.

5.3 Deadlock Detection

The deadlock detection scheme works in two phases. Initially we invoke the depth-first based **search** function to detect the existence of an SCC (or CSCC). During the second phase, we execute the reduction scheme based on the information generated from the function **search**. Depending on whether or not we choose to ignore pending deadlocks, we need to execute the **reduction** function to characterize the SCC as a pending deadlock or transient deadlock component. If we work under the premise that we detect and resolve pending and total deadlocks continuously, then it is sufficient to invoke deadlock detection after each unsuccessful resource request. The function **deadlock** is invoked by entity *e* in the edge (*e*,*r*,*k*) of the ER graph, and is shown in the definition of the *enhanced* **seize** statement in Line 8 of Figure 8. The function **deadlock** is used to detect and categorize the different types of deadlocks in the simulation. It is an integer-valued function that returns 0 for deadlock-free states, 1 for transient deadlock states, and 2 for pending deadlock states, and 3 for deadlock free states.

If we choose to not resolve pending deadlocks in the simulation, we would also need to invoke **deadlock** after failing to reassign any returned resource units in the **release** statement. This would be in addition to invoking deadlock detection through the **seize** statement as mentioned above. The function **deadlock** invocation is shown in Figure 9 (line 17) of the definition of the enhanced **release** statement. Here, PENDING_IGNORED is a global boolean variable that is set to false if we choose to ignore pending deadlocks in the ER graph, and vice versa. The function **deadlock** in turn invokes the **search** function to detect the presence of an SCC (or CSCC) component in the ER graph.

The **deadlock** function characterizes the result of **search**. If a CSCC is found then we report the existence of total deadlock, and appropriate action is taken. If we chose not to resolve pending deadlocks, then we do not need to invoke the **reduction** function. However, we invoke the reduction process if we find an SCC in the ER graph such that $r \in$ SCC, and we chose to detect and resolve pending deadlocks. Of course, if no component is found we return indicating that the simulation is deadlock-free.

We use the boolean function **additional_status** to characterize the reduction status of the ER graph. If the function **additional_status** reports that the ER graph is reducible, we report that the ER graph is in transient deadlock. The list of entities in the PSCC (*L*) to be reduced and the effective available capacity of the resources (EFF_CAP) are initialized by **search**. The function **additonal_status** uses **reduction** to distinguish between transient deadlock and pending deadlock. The details of **additonal_status** are shown in Figure 10.

The specific details of **reduction** are shown in Figure 11. The function **reduction** chooses an entity to reduce from the list *L*, whose resource request can be satisfied from the effective available capacity of the resources (lines 1-2). Failing this, **reduction** attempts to choose the next entity in L whose resource request can be satisfied, until all the entities in L are tested. Upon finding an entity that satisfies the resource constraints, all the resource units held by the entity are returned (line 3), and the entity is deleted from L (line 4), and the process starts again (line 7).

More precisely, let $G_{scc} \subseteq G_t$ be an arbitrary SCC in the ER graph. Let L be a list of entities where $e \in L$ such that either $[e \in G_{scc}]$ or $[e \notin G_{scc}, r \in G_{scc}, (r\ e, k) \in E_t]$. If G_{scc} is in transient deadlock then $L_f = \emptyset$, where L_f is the final list of entities obtained by executing **reduction** on L. Likewise, G_{scc} is in pending deadlock if and only if $L_f \neq \emptyset$ at the conclusion of executing **reduction** on L.

Notice that in line 2 of function **reduction** we use the entity attribute VREQ[e,r], the number of virtual requested units of resource r by entity e to check for the resource constraint. Recall that in the case of a no-pass resource queue, the virtual request of an entity is the maximum request size of all entities ahead of it in the queue. This guarantees that whenever an entity is processed during reduction, all other entities ahead of the entity can also be processed, thereby preserving the no-pass filling rule of the resource queue. The function **reduction** essentially operates on the list of entities in L as if the entities are resident in a resource queue with a pass filling rule. Thus, we utilize VREQ[e,r] to transform a resource queue with a no-pass filling rule to one with a pass filling rule. We will present the formal definition of VREQ[e,r] in the next section; for now the reader may ignore the difference between the two filling policies (i.e., assume REQ[e,r] is equal to VREQ[e,r]).

The **reduction** function determines if there exists a sequence of entity reductions that can satisfy the pending resource requests in the PSCC, given the current effective available capacity of the resources. If L is empty after executing **reduction**, the ER graph is in transient deadlock. If L is not empty, the ER graph is in pending deadlock.

6. Modeling Filling Rules

In the previous section the function **reduction** was used to detect pending deadlock in the PSCC of the ER graph. The function chooses an entity from the list of entities in the PSCC (L) whose resource request can be satisfied from the effective available capacity of the resources. Failing to reduce the entity, the function **reduction** chooses the next entity in L and attempts to reduce the newly chosen entity, until all the entities in L are tested. If some entity in L (with $L \neq \emptyset$) can be reduced then the reduction procedure starts again with a new list formed by deleting the reduced entity from list L. Eventually, no more entities in L can be reduced even though L is not empty, and the PSCC is in pending deadlock. Otherwise, L is empty and the PSCC is in transient deadlock. Careful thought will reveal that the **reduction** function reduces the PSCC as if the waiting entities are resident on resource queues that allow passing of entities. Consequently, the **reduction** function will violate the no-pass filling rule of a resource queue if there exists more than one entity from the resource queue in the list L.

As was demonstrated in Example 3 of Section 3.1, the classification of the deadlock situation in the simulation depends on the filling rule with multiple-unit resource requests. To accommodate both filling policies by the **reduction** function we now present a scheme that transforms a resource queue with a no-pass filling rule to one with a pass-filling rule. It should be noted that we assume the following scheme for a resource queue in the simulation: entities are inserted into the queue at an appropriate position, based on a queue discipline specified by the user at the beginning of the simulation. Further, entities are always picked from the front (or head) of the queue to satisfy resource demands.

To preserve the no-pass filling rule of any of the resource queues, we use a transformation that is a function of the request size (REQ[e,r]), the resource filling rule (PASSATTR[r]), and the position of the entity in the resource queue. This information is retained in an entity attribute called *virtual request*. The virtual request, VREQ[e_i,r], of entity e_i waiting for resource r is defined as follows:

$$\mathrm{VREQ}_t[e_i,r] = \begin{cases} \max_{j=1,\cdots,i} \mathrm{REQ}_t[e_j,r], & \text{if filling rule for } r \text{ is no-pass,} \\ \mathrm{REQ}_t[e_i,r], & \text{if filling rule for } r \text{ is pass,} \end{cases}$$

where e_i, $e_j \in \mathrm{QUEUE}_t[r]$ and i, j are the ordinal numbers of e_i, e_j in $\mathrm{QUEUE}_t[r]$.

Using the size of the virtual request in the function **reduction**, entity e_j cannot be reduced unless entity e_i can be reduced, if the entities are resident in a no-pass resource queue. Here, entities e_i, e_j are the i^{th} and the j^{th} entities in the resource queue, with $i<j$. Thus, virtual request essentially orders the entities in a no-pass resource queue in a non-decreasing fashion, and thereby preserves the no-pass filling rule of the resource queue during the reduction process. The VREQ[e,r] value

is updated every time an entity is inserted in or removed from the resource queue. Additional details are provided in Venkatesh *et al.* [22].

7. Deadlock Resolution

We next describe an approach that enables a simulation system to automatically recover from deadlock situations. The approach requires no additional language constructs and places no additional burden on the modeler. Deuermeyer *et al*. [5] distinguish between two types of deadlock situations that can occur in simulation systems with single-unit requests. The two configurations are termed *pull-processing* and *group-processing* deadlock.

Pull-processing occurs in the control of production systems using pull-control and flexible manufacturing systems. Deuermeyer *et al*. [5] suggest that the entities in deadlock need to *exchange* deadlocked resources to continue processing. For example, suppose entity 1 is finished with machine 1 and needs machine 2, while entity 2 is finished with machine 2 and needs machine 1; one way to model this situation is shown in Figure 12. Here, the deadlock should be resolved by the two entities exchanging the control of their resources. If this is accomplished, both the entities would proceed with the processing, and neither would be unnecessarily delayed. This type of pull-processing deadlock occurs typically in the modeling of a *machine-type* resource.

In the context of modeling manufacturing systems, we take the view that pull-processing deadlocks are unlikely to occur under the multiple-unit request conditions. For example, consider the deadlock in Figure 4b, except now, without entities e_3, e_4, e_5, and e_6. Suppose the deadlock represented a pull-processing situation; then an exchange of the resources held by entities e_1 and e_2 (probably representing parts in the manufacturing system) would be an appropriate resolution scheme. However, by doing this we have the unlikely prospect of entities e_1 and e_2 changing size during the exchange procedure. This would occur because the number of resource units requested by entity e_1 and the number of resource units currently assigned to entity e_2 are not the same. Such an occurrence is always imminent so long as any of the requested or assigned number of resource units (i.e., the value k in the edge triple (u,v,k)) are different in a particular cycle. In order to avoid this situation, we consider only group-processing deadlock resolution for the multi-unit seize case. It should be noted that if the requested and the assigned numbers of resource units match throughout a cycle, then the pull-processing resolution procedure developed in Deuermeyer *et al*. [5] is sufficient to resolve the deadlock.

In group-processing, more than one type of recourse is required simultaneously to perform a particular task. For example, such a case occurs when an operator is needed to load and subsequently unload a machine (refer to Example 1, Section 3.1). In this case, both the operator and the machine are required during the loading and unloading steps, but only the machine is required during the processing step. In the case of group-processing, deadlock occurs when none of the relevant entities (i.e., those in the PSCC) can gain control of the required set of resources and are, therefore, blocked from further processing. Deuermeyer *et al*. [5] suggest that the only way to resolve such a deadlock is for one entity to gain control of the resource units it cannot obtain by *displacing* some entities holding units of these resources. When an entity is so displaced, it is prevented from becoming active until the displaced entity is returned the resource. In the case of multiple-unit resource requests we use the displacement mechanism developed in Deuermeyer *et al*. [5] to resolve group-processing deadlocks.

When resolving deadlock using displacement, each displaced entity incurs a time penalty due to the resource preemption. This time penalty is the time period for which the entity is displaced. Because of this time penalty, it does not make sense to resolve transient deadlocks with this mechanism, at least not immediately, because there is the potential for the transient deadlock to be naturally resolved over time. Furthermore, since the duration of the transient deadlock is not known at the time of detection, our resolution strategy cannot be based upon time trade-offs. Consequently, in the group-processing case, we ignore transient deadlocks and resolve total deadlocks by displacement.

In preparation for the development of the resolution procedure we present the assumptions concerning the operation of the simulation system and some additional notation. Suppose entity e displaces entity e'' from resource r. Here, we assume that resource r maintains a list of the entities that are displaced. Also, we assume that entities are displaced from *all* the units of *all* the resources they are holding. This is a reasonable assumption in the case of group processing, since an entity cannot begin processing until it obtains control of all the requested resources.

After displacing an entity, the fill-rule of the **release** statement removes the first entity (say *e'*) of QUEUE[*r*] and creates the edge (*r*, *e'*) in *E*. It is necessary to prevent the fill-rule from selecting the displaced entities from QUEUE[*r*], since the displaced entity cannot begin to process until it obtains control of all the requested resources. To facilitate this, we assume as in Deuermeyer *et al*. [5] that each entity carries a *visibility index* that is initialized to zero. This index is incremented each time the entity is displaced from a resource and decremented each time the entity is restored on a resource queue. When the visibility index is zero, the entity is visible in the queue, otherwise the fill-rule will skip this entity. Thus, as entity *e″* is displaced from resource *r*, it is inserted into the displacement list maintained by resource *r*. As resource units become available, the displaced entity (*e″*) is reassigned the returned units, barring capacity constraints. The additional required extensions to **release** the statement are provided in Figure13.

The enhancements to the **release** statement are shown in lines 6-14 in Figure13. As entity *e* releases *k* units of resource *r*, in line 6 we check if there is an entity *e″* in the displaced list of resource *r* that should be given control. If so, lines 7-14 restore the displaced entity *e″*, decrementing the visibility index of *e″*. When the index reaches zero, entity *e″* will be selected by the fill-rule (lines 15-23) in the resource for which the entity is waiting. In addition, line 12 checks if a deadlock has been created due to the reassignment of entity *e″* to resource *r*. If no displaced entity is detected then lines 15-23 fill the resource as usual, from among the waiting entities in the resource queue.

We are now ready to present the deadlock resolution procedure. We work under the premise that function **resolution** is invoked as we detect pending or total deadlock in the simulation. The function **resolution** (Figure 14) is adapted from Coffman *et al*. [3] and Leung and Lai [14] for our purposes, and works in principle similar to function **reduction**. The function **resolution** begins with a list of entities (*L*) that need to be resolved (line 1). This list *L* is initialized with two classes of entities: entities that are in the PSCC; and entities that are not in the PSCC but that hold resources that are in the PSCC (i.e., $e \in L$ such that either [$e \in$ PSCC] or [$e \notin$ PSCC, $r \in$ PSCC, $(r, e, k) \in E_t$]). The effective capacity vector, EFF_CAP, is assigned the actual unassigned capacity of the resources in G_t at instant *t* (line 3). The set of entities to be displaced during the resolution is maintained in list *D*. This is identified by the function **dissolution** (Figure 15).

In the function **dissolution**, we choose an entity from list L and **displace** (Figure 16) *all* the resources it is holding. The displaced entity is removed from the list L and inserted in list D, and the resource units held by the displaced entity are added to the EFF_CAP (lines 2-4, function **dissolution**). The addition of the resources may help resolve the deadlock. To check if the addition of these resource units to EFF_CAP is sufficient to resolve the deadlock, we execute **reduction** on the remaining set of entities in the list L (line 5). If **reduction** does not resolve the deadlock (i.e., empties L) we displace *an* additional entity from the list L, and execute **reduction** again. This process is repeated until list L is empty, which indicates that the deadlock is resolved. The set of entities to be displaced is retained in D as the procedure is executed (line 4). Note that it is sufficient to displace the resources units held by the displaced entity within the PSCC. However, we choose the definition of the function **displace** identical to that of function **reduce** for the sake of simplicity.

We now present the rationale for the correctness of the resolution procedure. List L is initialized with the entities in the PSCC and entities holding resource units in the PSCC. Since G_{scc} is in permanent deadlock, we know that at the conclusion of executing **reduction** on G_{scc} , $L_f \neq \emptyset$. Thus to begin resolving the deadlock, we **displace** *an* entity in L (lines 2-3, function **dissolution** (Figure 15)). The effect of **displace** is similar to **reduce**, i.e., it strictly increases the vector EFF_CAP. This increase may help resolve the deadlock. It is possible in each iteration of the while loop (line 1, function **dissolution**) that we may not find *any* entity such that REQ[e_i,r] $\leq$ EFF_CAP[r] (i.e., the effective capacity will not be sufficient to satisfy the requested resource units for *any* entity in the list). In the worst case we would **displace** all but one entity in the list L, for if not, then REQ[e, r] > CAP[r] where entity e is the last entity in list L, thus leading to a contradiction. Note that by definition we have that VREQ[e,r] $\leq$ CAP[r] for all $e,r \in G_t$, in the case of the no-pass resource filling-rule. Therefore, we are always assured of a resolution of deadlock, since in the worst case we could always displace all the entities in the list L except the last entity. This displacement of all but the last entity would satisfy the resource request of the last entity. Thus **resolution** always resolves the deadlock in G_{scc}.

The function **resolution** works with the entities in the list L in *any* order; however, the order of entities in L can have an impact on the entities that are chosen for displacement (Leung and Lai [14]). For example, if the set of entities at the front of list L are such that $e \notin G_{scc}$, $r \in G_{scc}$, and (r e,

$k) \in E_t$ (as in a pending deadlock situation), then the displaced entities will be those entities that are currently processing. Thus, the sequence of entities in list L is critical, and requires further study and should be the subject of future research. The list of entities L can be initialized in a particular sequence providing the user some control over the outcome. For example, the list of entities in L can be ordered based on the number of times an entity has been displaced, a user-declared attribute (the attribute can be used to distinguish between different types of job priority, job processing times, etc.), or the number of resources and resource units an entity is holding.

It should be noted that it is possible that an entity may be displaced again and again during the resolution process (Deuermeyer *et al*. [5]). To prevent this from occurring, we use a priority level, initialized to 1 at the creation of the entity (or entry into the ER graph), and this priority level is used to order the entity in the list L. Each time an entity is displaced to resolve a deadlock, the priority level is multiplied by 0.5. This guarantees that the priority level of an entity (low numbers indicate high priority) would eventually reach a high enough value for the entity to proceed.

8. Conclusions

A graph theoretic model is used to automatically handle deadlocks in a general discrete simulation system. The graph model is based on the dynamic evolution of the entity-resource relationship that is developed by enhancing the **seize** and **release** statements of the simulation language. A key feature of the automatic scheme is that deadlock handling is accomplished by providing appropriate data structures to a simulation system, thereby using the information that already exists in the simulation system. Deuermeyer *et al*. [5] presented an automatic deadlock detection and resolution scheme with entities restricted to a single-unit of a resource. This paper is a generalization of the above approach extended for multiple-unit resource requests.

Sufficient conditions to identify three categories of deadlocks in a general simulation system based on the conditions of the ER graph have been presented. Initially, a depth-first search based algorithm identifies the existence of an SCC (or a CSCC) in the ER graph. A SCC that is closed in the ER graph represents a permanent deadlock in the simulation system requiring immediate intervention. However, the existence of an SCC that is not closed can represent either a transient or a pending deadlock. Once an SCC that is not closed is discovered, additional processing is required to distinguish between the transient and pending deadlock categories. A procedure based on the concept of reduction is adapted from Coffman *et al*. [3] to distinguish between the latter two deadlock categories. The reduction algorithm utilizes the information from the initial depth-first search procedure, thereby placing no additional computational burden for the information required by the reduction procedure. Also, this allows the user to turn off the reduction algorithm if the user knows that there can be no multiple-unit deadlock occurrences in the simulation.

It was demonstrated that the outcome of the deadlock detection scheme in multiple-unit requests can be affected by the resource filling rule. Appropriate algorithms are developed to model the different resource filling policies. These simple transformations allow for the development of a single deadlock handling procedure by ignoring the subtitles of correctly ordering entities in the resource queue, but without overlooking their effects.

Deuermeyer *et al*. [5] distinguish between group and pull processing deadlocks under single-unit resource requests. However, in the present paper, the view that pull-processing deadlocks are unlikely to occur in the context of modeling manufacturing systems is taken. In the case of group-processing deadlocks, appropriate algorithms to resolve the deadlocks based on the reduction procedure adapted from Coffman *et al*. [3] are developed. This is accomplished by automatically displacing some of the entities in deadlock. The procedure is applicable to both the categories of permanent deadlocks. As in Deuermeyer *et al*. [5] it is assumed that it does not make sense to resolve transient deadlocks in the case of group-processing. This is because there is a time penalty involved in displacing entities and the duration of the transient deadlocks at the time of detection is not known.

9. Acknowledgments

This research was funded in part by a SEMATECH/SRC grant on manufacturing systems research through grant number 88-MC-506.

10. References

[1] Aho, A., J.C. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA (1974).

[2] Cho, H., T.K. Kumaran, and R.A. Wysk, "Graph-Theoretic Deadlock Detection and Resolution for Flexible Manufacturing Systems," *IEEE Transactions on Robotics and Automation*, **11**, 3, 413-421 (1995).

[3] Coffman, E.G., M.J. Elphick, and A. Shoshani, "System Deadlocks," *ACM Computing Surveys*, **3**, 2, 67-78 (1971).

[4] Curry, G.L., B.L. Deuermeyer, and R.M. Feldman, *Discrete Simulation: Fundamentals and Microcomputer Support*, Holden-Day, Inc., Oakland, California (1989).

[5] Deuermeyer, B. L., G. L. Curry, A.D. Duchowski, and S. Venkatesh, "An Automatic Approach to Deadlock Detection and Resolution in Discrete Simulation Systems," *ORSA Journal of Computing* (*to appear*, 1995).

[6] Gold, E. M., "Deadlock Predition: Easy and Difficult Cases," *SIAM Journal of Computing*, Vol. 7, pp. 320-336, 1978.

[7] Gray, J.N., "Notes in Database Operating Systems," in *Operating Systems: An Advanced Course*, (*Lecture Notes in Computer Science 60*), R. Bayer, R.M. Graham, and G. Seegmuller, Eds., Springer-Verlag, NY (1979).

[8] Habermann, A.N., "Prevention of System Deadlocks," *Communications of the ACM*, **12**, 7, 373-377 (1969).

[9] Holt, R.C., "Comments on Prevention of System Deadlock," *Communications of the ACM*, **14**, 1, 36-38 (1971).

[10] Holt, R.C., "Some Deadlock Properties of Computer Systems," *ACM Computing Surveys*, **4**, 3, 179-196 (1972).

[11] Jun, K.P. and H.G. Perros, "Approximate Analysis of Arbitrary Configurations of Queueing Networks with Blocking and Deadlock," *Queueing Networks with Blocking: Proceedings of the First International Conference Workshop*, 259-279, Raleigh, NC (May 1988).

[12] Knapp, E., "Deadlock Detection in Distributed Databases," *ACM Computing Surveys*, **19**, 4, 303-328 (1987).

[13] Krishnaswamy, S. and N. Ghiya, "Design Optimization of Spin Apply Clusters using Simulation at the Packaging Plant in IBM, East Fishkill, NY," *Sematech 7th Operational Modeling Workshop*, Austin, TX (May 1993).

[14] Leung, J.Y.-T. and E.K. Lai, "On Minimum Cost Recovery from System Deadlocks," *IEEE Transactions on Computers*, **28**, 9, 671-677 (1979).

[15] Natarajan, N., "A Distributed Deadlock Scheme for Detecting Communication Deadlock," *IEEE Transactions on Software Engineering*, **12**, 4, 531-537 (1986).

[16] Pegden, C.D., R.P. Sadowski, and R.E. Shannon, *Introduction to Simulation Using SIMAN*, Systems Modeling Corporation, Sewickley, PA (1990).

[17] Pierce, N.G. and M.J. Drevna, "Development of Generic Simulation Models to Evaluate Wafer Fabrication Cluster Tools," *Proceedings of the 1992 Winter Simulation Conference*, 874-878, Arlington, VA (December 1992).

[18] Schriber, T.J., *Simulation Using GPSS*, John Wiley, NY (1974).

[19] Schriber, T.J., *An Introduction to Simulation Using GPSS/H*, John Wiley, NY (1990).

[20] Singhal, M. "Deadlock Detection in Distributed Systems," *IEEE Computer*, **22**, 12, 37-48 (1989).

[21] Venkatesh, S., J. Smith, B.L. Deuermeyer, and G. L. Curry, "Deadlock Properties in Discrete Simulation Systems," *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, 2563-2569, San Diego, CA (May 1994).

[22] Venkatesh, S., J. Smith, B.L. Deuermeyer, and G. L. Curry, "Deadlock Detection and Resolution for Discrete Event Simulation: Details and Algorithms," Technical Paper, Industrial Engineering Department, Texas A&M University (1994).

[23] Wysk, R.A., N.S. Yang, and S. Joshi, "Detection of Deadlocks in Flexible Manufacturing Systems," *IEEE Transactions on Robotics and Automation*, **7**, 6, 853-859 (1991).

11. Figures

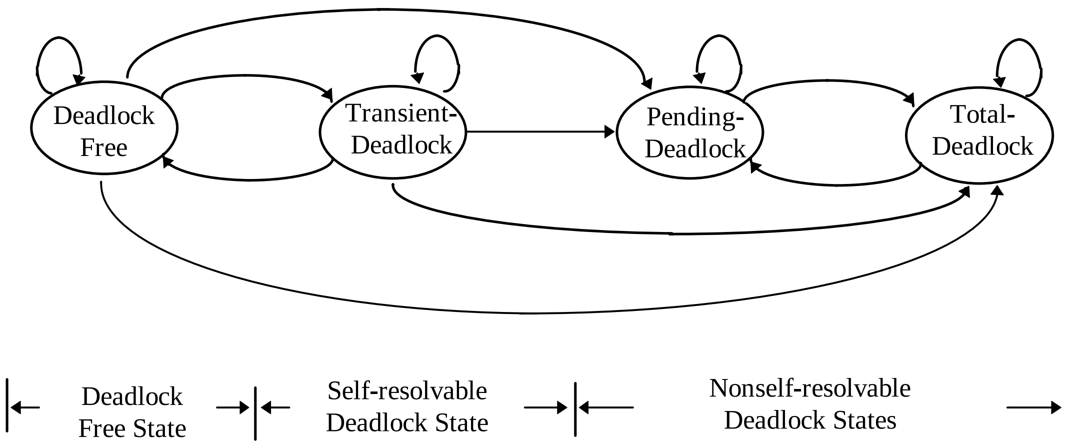

Figure 1. Transition diagram for simulation deadlock states.

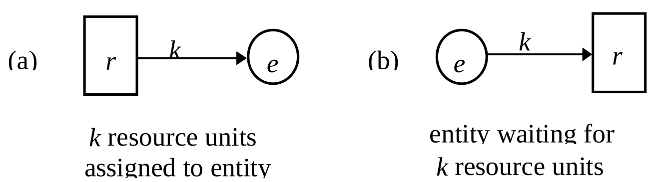

Figure 2. Entity-resource relationship scheme - (a) assignment edge, (b) request or queue edge.

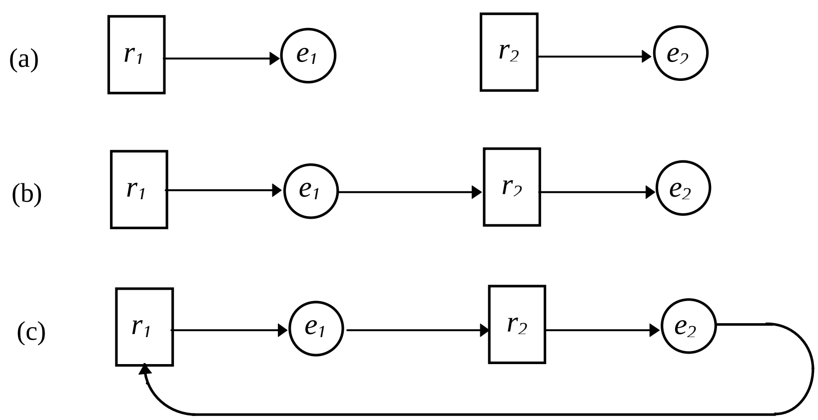

Figure 3. Sequence of events in Example 1 that leads to deadlock.

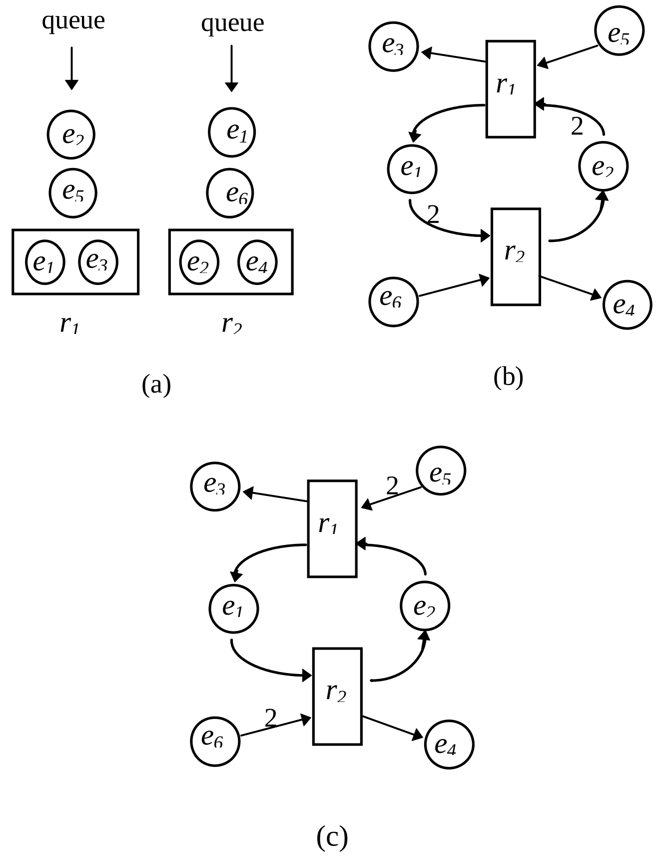

Figure 4. (a) System simulated in Examples 2 and 3; (b) Entity-resource relationship for Example 2; (c) Entity-resource relationship for Example 3.

seize(e,r,k) - entity e requests k units of resource r

```
if e ∉ V then
    insert(V,e);
if REQ[e,r] ≤ REM[r] then
    insert(E,(r,e,k));
    REM[r] = REM[r] - REQ[e,r];
else
    insert(E,(e,r,k));
endif
```

Figure 5. Definition of the **seize** statement.

```
release(e,r,k) - entity e releases k units of resource r
delete(E,(r,e,k));
delete(USES[r], e);
REM[r] = REM[r] + k;
if OWNS[e] = Ø then
    delete(V,e);
if QUEUE[r] ≠ Ø then        " resource fill-rule "
    e' = firstfit(r);
    dowhile(e' ≠ Ø)
        delete(E,(e',r,REQ[e',r]));
        insert(E,(r,e',REQ[e',r]));
        REM[r] = REM[r] - REQ[e',r];
        e' = firstfit(r);
    endwhile
endif
```

Figure 6. Definition of the **release** statement.

```
reduce(e)
for each r such that (r,e,HOLDS[e,r]) ∈ E do
    EFF_CAP[r] = EFF_CAP[r] + HOLDS[e,r];
```

Figure 7. Definition of function **reduce**.

```
seize(e,r,k) - entity e requests k units of resource r
if e ∉ V then
    insert(V,e);
if REQ[e,r] ≤ REM[r] then
    insert(E,(r,e,k));
    REM[r] = REM[r] - REQ[e,r];
else
    insert(E,(e,r,k));
    if deadlock(r) ≠ 0 then
        "Deadlock Detected"
endif
```

Figure 8. Definition of the enhanced **seize** statement.

```
release(e,r,k) - entity e releases k units of resource r

delete(E,(r,e,k));
delete(USES[r], e);
REM[r] = REM[r] + k;
if OWNS[e] = Ø then
    delete(V,e);
if QUEUE[r] ≠ Ø then
    e' = firstfit(r);
    if e' ≠ Ø then
        dowhile(e' ≠ Ø)
            delete(E,(e',r,REQ[e',r]));
            insert(E,(r,e',REQ[e',r]));
            REM[r] = REM[r] - REQ[e',r];
            e' = firstfit(r);
        endwhile
    else
        if PENDING_IGNORED then
            if deadlock(r) = 3 then        " total deadlock "
                "Deadlcok Detected"
    endif
endif
```

Figure 9. Definition of the enhanced **release** statement.

```
boolean additional_status(L, EFF_CAP)
reduction(L, EFF_CAP)
if |L| ≠ 0  then
    for each r such that r ∈ SCC
        if EFF_CAP[r] ≠ CAP[r] then
            RESSTATE[r] = true;
    endfor
    return(true);      " pending deadlock "
else
    return(false);     " transient deadlock "
endif
```

Figure 10. Details of function **additional_status**

```
boolean reduction(L, EFF_CAP)
for ( i = 1; i ≤ |L|, |L| ≠ 0; i++)
    if VREQ[e_i,r] ≤ EFF_CAP[r] then
        reduce(e_i);
        delete(L,e_i);
        i = 1;
    endif
endfor
```

Figure 11. Details of the **reduction** function.

Entity 1 Logic

```
seize mach_1
    wait
    seize mach_2
release mach_1
```

Entity 2 Logic

```
seize mach_2
    wait
    seize mach_1
release mach_2
```

Figure 12. Example entity logic for pull-processing resolution.

release (*e*, *r*, *k*) - entity *e* releases *k* units of resource *r*

```
delete(E,(r,e,k));
delete(USES[r], e);
REM[r] = REM[r] + k;
if OWNS[e] = Ø then
    delete(V,e);
if there is an (r, e″) contained in r→displace then
    if REQ[e″ , r] ≤  REM[r] then
        REM[r] = REM[r] + k;
        delete (r→displace, (r, e″));
        decrement (e″→visible);
        insert (E, (r, e″));
        if deadlock(e″, r) then
            “Resolve Deadlock”;
    endif
else if  QUEUE[r] ≠ Ø then        “ resource fill-rule ”
    e’ = firstfit(r);
    dowhile(e’ ≠ Ø)
        delete(E,(e’,r,REQ[e’,r]));
        insert(E,(r,e’,REQ[e’,r]));
        REM[r]  = REM[r] - REQ[e’,r];
        e’ = firstfit(r);
    endwhile
endif
```

Figure 13. Details of the **release** statement with extensions to prevent filling of a displaced entity.

resolution()

```
initialize L;
D = ∅ ;
EFF_CAP = REM;
dissolution(L, D, EFF_CAP);
```

Figure 14. Details of the **resolution** function.

dissolution (*L*, *D*, EFF_CAP)

```
while L ≠ ∅ do
    e = get(L, 1);
    displace(e);
    insert(D, e);
    reduction(L,EFF_CAP);
endwhile
return(D);
```

Figure 15. Details of the **dissolution** function.

displace (e)

1. **for** each r such that $(r,e,\text{HOLDS}[e,r]) \in E$ **do**
2. EFF_CAP[r] = EFF_CAP[r] + HOLDS[e,r];

Figure 16. Details of the **displace** function.