An Extended Banker's Algorithm for Deadlock Avoidance

Sheau-Dong Lang

Abstract—We describe a natural extension of the banker's algorithm for deadlock avoidance in operating systems. Representing the control flow of each process as a rooted tree of nodes corresponding to resource requests and releases, we propose a quadratic-time algorithm which decomposes each flow graph into a nested family of regions, such that all allocated resources are released before the control leaves a region. Also, information on the maximum resource claims for each of the regions can be extracted prior to process execution. By inserting operating system calls when entering a new region for each process at runtime, and applying the original banker's algorithm for deadlock avoidance, this method has the potential to achieve better resource utilization because information on the "localized approximate maximum claims" is used for testing system safety.

Index Terms—Banker's algorithm, deadlock, deadlock avoidance, graph reduction, operating system, worst-case complexity.

1 Introduction

A well-known deadlock avoidance algorithm used in operating systems is the banker's algorithm which was proposed by Dijkstra to handle a single resource type [2], and later extended by Habermann to handle multiple resource types [4], where the resources are assumed to be serially reusable [6]. In this algorithm, processes are required to declare their maximum claims of resources in advance. When a new request of resources is made, the banker's algorithm would grant the request if the resulting system remains in a "safe" state, in the sense that even in the "worst case" that all processes request their maximum claims, there is still a schedule of process execution so that all the requests will be granted eventually. When there are n processes and m resource types, Habermann's algorithm requires $O(mn^2)$ time for testing system safety. The efficiency of the algorithm was later improved to $O(mn)$ by Holt [6]. When restricted to a single resource type, Habermann [5] proposed an efficient algorithm that uses $O(n + r)$ space and $O(r)$ time to handle a resource request or release, where r is the number of resource units. More recently, Finkel and Madduri [3] proposed a new algorithm that maintains the resource allocation history in a binary tree and performs a safety test in $O(\log n)$ time.

Although the efficiency of the banker's algorithm is a major concern in practice, another concern is the algorithm's effectiveness in resource utilization. Since only a limited amount of resource usage information is known in advance (i.e., the maximum claims), it is possible that a request will be denied because it is putting the system into an "unsafe" state, even though the system is deadlock-free. Let us consider an example.

Example 1. Suppose there are two processes P_1 and P_2 sharing two types of resources R_1 and R_2, each having two units. Processes P_1 and P_2 request and release resources in a sequence of steps as depicted in Fig. 1. Using the banker's algorithm, the maximum claim matrix is:

$$\text{CLAIM} = \begin{bmatrix} 1 & 2 \\ 2 & 2 \end{bmatrix},$$

where the first row (1, 2) indicates the maximum need of 1 unit of R_1 and 2 units of R_2 from process P_1, and the second row (2, 2) indicates the maximum need of 2 units of R_1 and 2 units of R_2 from $P2$. Suppose the current allocation of resources is denoted by

$$\text{ALLOC} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix},$$

where the first row (1, 0) indicates 1 unit of $R1$ and 0 unit of $R2$ being allocated to P_1, and the second row (0, 0) indicates no allocations to P_2. Similarly, suppose the current request of resources is denoted by

$$\text{REQ} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

in which P_2 requests 1 unit of R_2. According to the banker's algorithm, the request of P_2 will be denied because by granting the request, and assuming a worst-case scenario that both P_1 and P_2 request their maximum claims, the system could be put into a deadlock. (See Fig. 2 for the general resource graph [6] used in the banker's algorithm which cannot be completely reduced.) Notice that the request of 1 unit of R_2 from P_2 could have been safely granted, if we knew that P_1 never needs both R_1 and R_2 simultaneously. However, the banker's algorithm only knows the maximum claims from each process, thus making the algorithm ineffective in resource utilization.

Attempts have been made to improve the effectiveness of the banker's algorithm [7]. It is shown in [7, pp. 1,033-1,038] that by knowing the current allocation resources and exactly what other resources are needed before the current allocated resources will be released, a modified banker's algorithm may be applied to determine whether granting a resource request is safe. Thus, the modified algorithm improves upon the original algorithm by knowing the "localized approximate maximum claims" as opposed to the global maximum claims of each process [7]. However, this algorithm requires that the resource units seized by one ALLOCATE-RESOURCE instruction be released by one DEALLOCATE-RESOURCE instruction, and that the resource requests be made in a "nested" form, in the sense that within each process and for each resource type, units seized last are released first. Also, this algorithm incurs more overhead by maintaining a set of dynamic Request-Assign graphs which keep track of the resource requests and allocations of each process, and by combining these graphs at runtime in order to determine whether granting a resource request could lead to a system deadlock.

In this paper, we consider a simple and natural extension of the banker's algorithm that offers the potential of improved resource utilization while incurring low overhead. By representing the control flow of each process as a rooted tree of resource requests and releases, we propose a quadratic-time preprocessing algorithm which decomposes these flow graphs into regions and determines the maximum resource claims for each of these regions. The detail of this algorithm and a formal definition of the region are given in Section 2. We also provide proofs that demonstrate the optimality of this decomposition, and explain how the prior knowledge of process regions and the associated maximum claims can be incorporated into the banker's algorithm for testing system safety. Section 3 concludes the paper and points out some future work.

• *S.D. Lang is with the School of Computer Science, University of Central Florida, Orlando, FL 32816. E-mail: lang@cs.ucf.edu.*

Manuscript received 13 Apr. 1989; revised 31 Oct. 1996.
Recommended for acceptance by R.R. Razouk.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 104057.

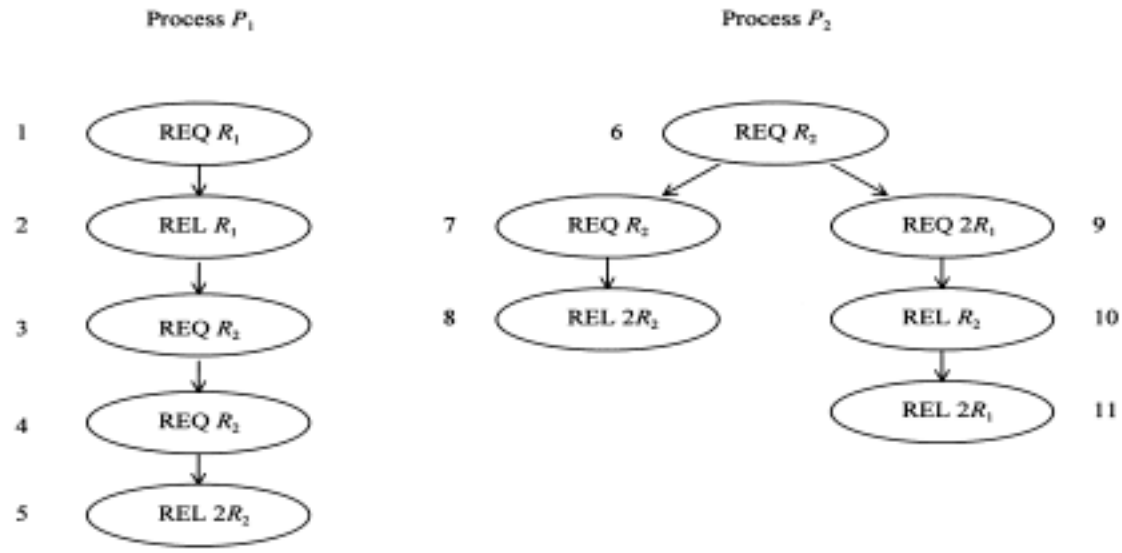

Fig. 1. Two resource-request graphs.

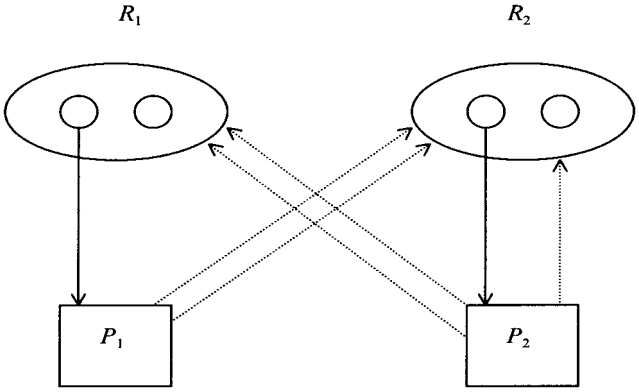

Fig. 2. The worst-case scenario by granting 1 unit of R_2 to P_2, dashed lines show maximum requests.

2 The Extended Banker's Algorithm

2.1 Decomposition into Regions

Suppose there are n processes P_1, ..., P_n sharing m types of multi-unit resources R_1, ..., R_m. We first examine the control flow of each process (such as a flow chart) and extract the calls of resource requests or releases, which are of the form REQ iR_j or REL iR_j representing, respectively, a request or release of i units of resource R_j. We assume the control flow of such resource-related calls is represented as a rooted tree of nodes, in which all allocated resources are released at the end of each leaf node, and no processes would release resources that are not allocated earlier. (Generalizations to other types of control flow graphs allowing loops and cross arcs are possible, but beyond the scope of this paper.) In the rooted tree, each node has one or more child nodes; this latter case means alternative execution paths. (See Fig. 1 for two process trees.) This process structure that models resource requests and releases is proposed in [7, p. 1,025] and called a *Resource-Request Graph*. The basic idea of the extended banker's algorithm is to preprocess these graphs into *regions* and determine the associated *maximum resource claims*, then to use this information at runtime to provide a more effective system safety test in the banker's algorithm. We now give some formal definitions.

Definition 1. *Suppose a process is represented as a resource-request graph. A prime region is a directed path (i.e., a sequence of connected nodes) such that:*

1) *no resources are allocated before the control enters the first node of the path;*
2) *all allocated resources are released when the control leaves the last node; and*
3) *no proper subpaths also satisfy the first two properties.*

A subprime region is a subpath of a prime region having the same end node.

Definition 2. *Corresponding to each prime and to each subprime region, we define the associated maximum resource claims (or needs) as the union of the maximum needs of each type of the resources in the region, which may include resources allocated prior to entering the region (in case of a subprime region).*

Using Fig. 1 for illustration, the path (6, 7, 8), consisting of the nodes REQ R_2, REQ R_2, and REL $2R_2$, is a prime region. The subpaths (7, 8) and (8) are both subprime regions. The path (1, 2, 3, 4, 5) is not a prime region, because it contains a subpath (1, 2) which is a prime region. In fact, the graphs in Fig. 1 have exactly four prime regions: (1, 2), (3, 4, 5), (6, 7, 8), and (6, 9, 10, 11). The maximum resource claims for the prime region (1, 2) in Fig. 1 can be written as a vector (1, 0) corresponding to the resources (R_1, R_2). Similarly, the maximum resource claims for the subprime region (7, 8) is (0, 2). Notice that in our convention, we assume the claims reflect the status after the start node of the region. Thus, if a region starts with a resource request node, that resource need is counted in the maximum resource claims for the region; however, if a region starts with a resource release node, that resource is not counted. For example, node 7 in region (7, 8) is REQ R_2, so the associated claim vector is (0, 2), but for the subprime region (10, 11) whose start node is REL R_2, the associated claim vector is (2, 0).

In order to test the system safety more effectively in the banker's algorithm, we will associate maximum resource claims with *each node* of the resource-request graph. One reason is that the maximum resource claims could be made more precise when a node gets closer to the end of a prime region; for example, in the prime region (6, 9, 10, 11), the maximum claims drop from (2, 0) as in node 10 to (0, 0) in node 11. Another reason for associating maximum resource claims with each node is that when a node is contained in more than one prime or subprime region (e.g., node 6 in Fig. 1), the corresponding maximum resource claims must accommodate the resource needs of all alternative execution paths. Thus, we give the following more general definitions.

DEFINITION 3. *A region can be a prime region, a subprime region, or the union of regions sharing a common beginning subpath.*

DEFINITION 4. *The region associated with a node u, denoted region(u), is defined as the union of those prime or subprime regions having u as their start node. The maximum resource claims associated with region(u), denoted claim(u), consists of the union of the claims corresponding to the prime or subprime regions in region(u).*

We define the MAX function and a binary relation "≤" on vectors as follows:

DEFINITION 5. *Let* $\mathbf{B}_1, ..., \mathbf{B}_q$ *be a set of vectors of the same arity.* MAX $\{\mathbf{B}_1, ..., \mathbf{B}_q\}$ *defines a vector whose j*th *component equals the largest j*th *component among all vectors* $\mathbf{B}_1, ..., \mathbf{B}_q$.

DEFINITION 6. *Suppose* **A** *and* **B** *are vectors of the same arity.* $\mathbf{A} \leq \mathbf{B}$ *means each entry of* **A** *is less than or equal to the corresponding entry of* **B**.

Using Fig. 1 as an example, the region associated with node 6 is *region*(6) = {6, 7, 8, 9, 10, 11}, which is a union of the prime regions {6, 7, 8}, and {6, 9, 10, 11}, covering all alternative execution paths starting at node 6. Since the latter two regions have the maximum claims (0, 2) and (2, 1), respectively, for resources (R_1, R_2), the union of these maximum claims gives *claim*(6) = MAX {(0, 2), (2, 1)} = (2, 2). Notice that we use the set notation { } for regions; but when a region is a prime or subprime region, we could also use the path notation ( ) which is common in Graph Theory. Table 1 gives the regions and the corresponding maximum resource claims for each of the nodes of process P_2 in Fig. 1.

We now present an algorithm which determines for each node of a resource-request graph the corresponding region and its maximum resource claims.

Region Decomposition Algorithm

Input. A resource-request graph G (i.e., a rooted tree) of root node *t* represented in an adjacency list structure, where *k* = the number of nodes, and *m* = the number of resource types.

Output. Two arrays *Regions*[1..*k*, 1..*k*] and *Claims*[1..*k*, 1..*m*] such that for each node *i*, $1 \leq i \leq k$, the row vectors *Regions*[*i*, 1..*k*] and *Claims*[*i*, 1..*m*] identify, respectively, the region and the associated maximum resource claims corresponding to node *i*. That is, *Regions*[*i*, *j*] = 1 if node $j \in region(i)$; 0 otherwise; and the row vector *Claims*[*i*, 1..*m*] = *claim*(*i*).

Method

call Decomp(*t*, **0**)

/* start from root *t*, where **0** is an *m*-vector indicating zero allocated resources in the beginning */

Procedure Decomp(*u*, *res*)

/* a recursive procedure which computes for node *u* (and its descendants) its region *Regions*[*u*, 1..*k*] and the associated maximum claim vector *Claims*[*u*, 1..*m*] using a depth-first traversal; the parameter *res* is an *m*-vector giving the allocated resources immediately before the control enters node *u*; the statement numbers are included in the code for reference */

TABLE 1
PROCESS P_2'S REGIONS AND MAXIMUM CLAIM VECTORS

Nodes	Regions	Maximum ClaimVectors (R_1, R_2)
6	(6, 7, 8, 9, 10, 11)	(2, 2)
7	(7, 8)	(0, 2)
8	(8)	(0, 0)
9	(9, 10, 11)	(2, 1)
10	(10, 11)	(2, 0)
11	(11)	(0, 0)

1) **if** node u = REQ iR_j **then**
2) $res = res + i\ \mathbf{e}_j$, where $\mathbf{e}_j$ is a unit vector with the value 1 in the *j*th position
3) **else if** node u = REL iR_j **then**
4) $res = res - i\ \mathbf{e}_j$
5) *Claims*[*u*, 1..*m*] = *res*
/* initialize the claims at *u* to be the allocated resources up to this point */
6) *Regions*[*u*, 1..*k*] = $\mathbf{e}_u$ /* initialize node *u*'s region to *u* itself */
7) **for each** child *v* of *u* **do /*** recursive calls ***/**
8) **call** Decomp(*v*, *res*)
9) **if** *res* ≠ **0 then**
/* node *u* is not at the end of a prime region */
10) $Claims[u, 1..m] = Claims[u, 1..m] + \underset{v \text{ is a child node of } u}{\text{MAX}} \{Claims[v, 1..m]\}$
11) $Regions[u, 1..k] = Regions[u, 1..k] + \underset{v \text{ is a child node of } u}{\text{MAX}} \{Regions[v, 1..k]\}$

end Decomp

Theorem 1 summarizes the major facts about region decomposition.

THEOREM 1. *The region decomposition algorithm applied to a resource-request graph of k nodes and m resource types satisfies the following properties*:

1) *The worst-case time and space complexity both are* $O(km + k^2)$;
2) *The row vector Regions*[*u*, 1..*k*] *correctly computes region*(*u*), *the region associated with node u*;
3) *The row vector Claims*[*u*, 1..*m*] *correctly computes claim*(*u*), *the maximum resource claims corresponding to region*(*u*);
4) *If u and v are two adjacent nodes of a resource-request graph, and v is a child node of u, then either claim*(*u*) = **0**, *i.e., node u is the end node of a prime region, or otherwise claim*(*v*) ≤ *claim*(*u*);
5) *The region region*(*u*) *is the smallest region which covers node u and its alternative execution paths until all the allocated resources are released; that is, for any region R that also has this property, region*(*u*) ⊆ *R; and*
6) *The regions corresponding to the nodes of the resource-request graph form a nested family, that is, any pair of these regions are either mutually exclusive or one is a subset of the other.*

PROOF. (***Part 1***). It is obvious that the algorithm performs a depth-first traversal of the tree which consists of *k* nodes and *k* – 1 edges; the traversal visits O(*k*) nodes and edges assuming an adjacency list representation of the tree [1]. During the traversal, the time of applying the MAX function in Step 10 of the algorithm is (number of child nodes of *u*) · O(*m*), for each node *u*. Thus, the total time contributed by Step 10 in all recursive calls is O(*k*)O(*m*) = O(*km*), because there are O(*k*) edges and nodes. Similarly, the time of executing Step 11 in all recursive calls is $O(k^2)$. Since the time of executing statements other than steps 10 and 11 is O(*k*), the total time complexity of the algorithm is $O(km + k^2)$. Also, the algorithm uses $O(km + k^2)$ space for the two arrays Claims[1..*k*, 1..*m*] and *Regions*[1..*k*, 1..*k*] (the adjacency list representation of the graph uses only O(*k*) space).

(***Parts 2 and 3***). Since the algorithm performs a depth-first traversal of the tree, it is obvious that the parameter *res* correctly updates the allocated resources at each node *u* using Steps 1, 2, 3, and 4, and this information is passed on to the child nodes of *u* in Step 8. Now we prove the correctness of computing the region and claims information for each node *u*. We use induction on the distance from the node to the root.

- ***Basis***. If node *u* is a leaf node, its allocated resources *res* must be equal to **0**, based on our assumption that all allocated resources will be released at a leaf node in a

resource-request graph. In this case, Steps 5 and 6 correctly computes the region and claims for node u, because none of the following Steps 7, 8, 9, 10, and 11 will be executed.

- ***Induction***. Assume the region and claims are correct for each child node of node u. First, node u's claims are initialized in Step 5 to be the allocated resources after entering the node. Then, the recursive calls in Steps 7 and 8 correctly compute the claims for the children of node u, by induction hypothesis. Finally, Step 10 takes *Claims*[v, 1..m], the claims of node u's children reflecting the maximum needs after entering each child node v, and forms the union of these claims with the allocated resources initialized in Step 5. This union computed by the MAX function gives the correct claim vector corresponding to node u, according to Definition 4. Thus by induction, *Claims*[u, 1..m] correctly computes *claim*(u) for all node u. A similar argument applies to the proof for the region computation.

(***Part 4***). If node u is not an end node of a prime region, then Step 10 computes *claim*(u) by applying MAX to the claim vectors of all its children, including *claim*(v); thus, *claim*(v) ≤ *claim*(u) in that case.

(***Parts 5 and 6***). These two parts can also be proved by induction, similar to that of Parts (2) and (3). The proof is omitted here for brevity. □

Notice that Part 5 of Theorem 1 implies that the regions computed by the decomposition algorithm are optimal in the sense that each region covers a minimum portion of the graph including alternative execution paths. As a result, the maximum claims at each node provide, prior to runtime, the best estimate of the maximum resource needs. This latter point will be formally proved in the next section. Since the original banker's algorithm uses a single region and a single maximum claim vector for each process for testing system safety, our method of separating process states into multiple regions has the potential to improve resource utilization, when the corresponding maximum claim vectors are dissimilar.

2.2 Testing System Safety

Based on the maximum claim vectors computed by the decomposition algorithm for each process, a simple way to avoid deadlocks is to make system calls at each node declaring the current maximum claims at runtime. For a resource request node, the maximum claims are declared before requesting the resource, but for a resource release node, the maximum claims are declared after releasing the resource. This is consistent with the way in which the array *Claims*[1..k, 1..m] is computed in the algorithm. Since these maximum claims cover the resource needs of all alternative execution paths until the resources are released, and there is a single claim vector for each process at any given time during process execution, the original banker's algorithm for testing system safety is still applicable [6]. We now summarize our extended banker's algorithm as follows:

Extended Banker's Algorithm

Preprocessing. Represent each process as a rooted tree of resource-request graph, and apply the decomposition algorithm to compute the regions and the associated maximum resource claims for each process.

Runtime. Insert operating system calls to declare the maximum resource claims at each resource request node and resource release node, so that the claims are declared before a resource request but after a resource release. The system would grant a resource request if the resulting state is safe, by applying the original banker's algorithm for safety testing [6]. Further, blocked requests will be checked for potential unblocking whenever there is a resource release call or a resource claims declaration with reduced resource needs.

As a matter of implementation detail, the system calls are inserted at a node only if its claims are different from that of the parent node's. Also, the region information (i.e., the array *Regions*[1..k, 1..k]) is not needed for testing the system safety, but was included in the decomposition algorithm for clarification purposes. Removing *Regions*[1..k, 1..k] and the related code reduces both the time and space complexity of the decomposition algorithm to O(km). If the n processes have respectively k_i nodes, $1 \le i \le n$, the total time and space needed in computing the claims information for all n processes is

$$O\left(\left(\sum_{i=1}^{n} k_i\right)m\right).$$

In the following, we will provide a formal proof that shows the correctness of the system safety test using this method; that is, deadlocks are avoided for every combination of the execution paths of the processes at runtime. Furthermore, we will prove that the converse of this statement is also true; that is, without knowing which combination of the alternative execution paths to use prior to runtime, we must include all potential resource needs in declaring the maximum resource claims. Thus, our region decomposition algorithm produces an optimal set of maximum resource claims to avoid deadlocks. We need some definitions in order to define the system safety more formally.

DEFINITION 7. *Suppose process P_i's current state of execution is represented by a node u_i in its resource-request graph, for $1 \le i \le n$. Let region(u_i) = $\bigcup_{j=1}^{ti} Pij$, a union of prime or subprime regions according to Definition 4. The maximum claim vectors C_{ij} corresponding to the regions P_{ij} are called the admissible claim vectors of process P_i for its current state of execution.*

When the runtime state of the system is represented by a set of nodes, one from each process's resource-request graph, and each node has a set of admissible claim vectors corresponding to the alternative execution paths, the system safety test should avoid deadlocks in all possible combinations of these claim vectors. Thus, we define the system safety as follows.

DEFINITION 8. *Let S_i denote the set of admissible claim vectors of process P_i for its current state, and let $S = \bigtimes_{i=1}^{n} S_i$ denote their Cartesian product. For each n-tuple $Z = (C_1, ..., C_n) \in S$, which is a possible combination of admissible claim vectors of the processes P_1 through P_n, let G_Z denote the general resource graph depicting the current resource allocation, the available resources, and the worst-case requests made by each process P_i based on its admissible claim vector C_i. The current system state is safe if the graph G_Z is completely reducible for every n-tuple $Z \in S$. A system state is unsafe if it is not safe.*

According to this definition, if $|S_i| = t_i$ for $1 \le i \le n$, a straightforward approach to testing system safety requires testing $\prod_{i=1}^{n} t_i$ possible combinations of claim vectors. In each scenario, since Holt's algorithm requires O(mn) time [6], the total time complexity

of the system safety test would require $O(mn\prod_{i=1}^{n} t_i)$ time. One of the main objectives of this paper is to prove that the extended banker's algorithm described previously, which uses a single maximum claim vector $claim(u_i) = MAX\ \{C \mid C \in S_i\}$ for each process P_i, provides an equivalent but more efficient procedure for testing the system safety.

THEOREM 2. *Suppose the runtime state of the system is represented by a set of nodes u_i, $1 \le i \le n$, each taken from the corresponding process P_i's resource-request graph. Let S_i denote the set of admissible claim vectors of process P_i for its current state, and let G denote the general resource graph depicting the current resource allocations, the available resources, and the worst-case requests using a single claim vector $claim(u_i) = MAX\ \{C \mid C \in S_i\}$ for each process P_i. Then the current system state is safe (according to Definition 8) if and only if the graph G is completely reducible.*

PROOF. The "if" part is obvious because for each n-tuple $Z = (C_1, ..., C_n) \in \bigtimes_{i=1}^{n} S_i$, since each claim vector $C_i \le claim(u_i)$, the general resource graph G_Z is completely reducible if the graph G is completely reducible. To prove the "only if" part, suppose G is not completely reducible. Then there are k processes, call them $P_1, ..., P_k$, for some $k \le n$, which are "blocked" after all possible reductions in G are complete [6]. That is, for each process P_i, $1 \le i \le k$, there exists a request to some resource which has no available units. Since the requests of P_i in graph G are based on the claims in the claim vector $claim(u_i) = MAX\ \{C \mid C \in S_i\}$, there exists an admissible claim vector C_i which has some request to an unavailable resource. Therefore, taking these claim vectors C_i, $1 \le i \le k$, and arbitrary admissible claim vectors C_{k+1} through C_n of processes P_{k+1} through P_n, respectively, the general resource graph G_Z, with $Z = (C_1, ..., C_n)$, can not be completely reduced. Thus the system state is unsafe according to Definition 8. This proves the "only if" part. □

The following example illustrates the "only if" part of Theorem 2.

EXAMPLE 2. Consider the two processes depicted in Fig. 1 sharing two types of resources R_1 and R_2, each resource having two units available. Suppose process P_1 has successfully entered node 3 (i.e., the request REQ R_2 is granted), and process P_2 is at node 6 making the request REQ R_2. At this point, process P_1 is in region {3, 4, 5}, and process P_2 is in region {6, 7, 8, 9, 10, 11} = {6, 7, 8} ∪ {6, 9, 10, 11}. The corresponding maximum claim vectors computed by the region decomposition algorithm are: $claim(3) = (0, 2)$ for process P_1, and $claim(6) = MAX\ \{(0, 2), (2, 1)\} = (2, 2)$ for process P_2. By granting P_2's request, the extended banker's algorithm would use these claim vectors in making the worst-case requests, along with the resource allocation information, to form a general resource graph as in Fig. 3. Since this graph is not completely reducible, granting P_2's request would result in an unsafe state according to the theorem; thus this request will be denied. The reason we should deny this request is because continuing from node 6, process P_2 may proceed into the region {7, 8}, which would make a request REQ R_2 at node 7, resulting in a deadlock.

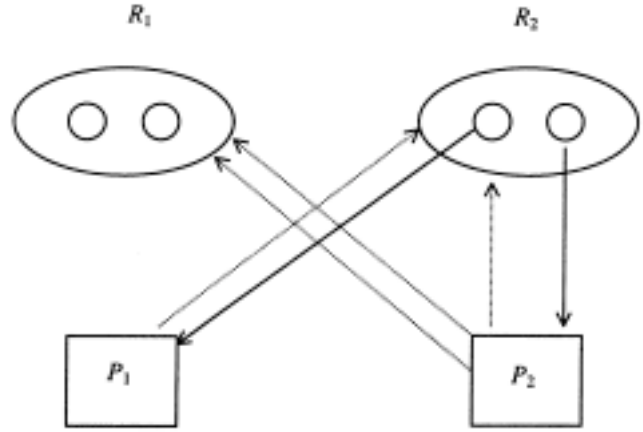

Fig. 3. The worst-case scenario by granting 1 unit of R_2 to P_2, dashed lines show maximum requests based on $claim(1) = (0, 2)$ and $claim(6) = (2, 2)$.

3 CONCLUSION

In this paper, we proposed an extension of the banker's algorithm for deadlock avoidance. Assuming that the control flow of the resource-related calls of each process forms a rooted tree, we proposed a quadratic-time algorithm which decomposes these trees into regions and computes the associated maximum resource claims, prior to process execution. This information is then used at runtime to test the system safety using the original banker's algorithm. We proved that the region decomposition algorithm provides an optimal set of maximum claim estimates, and that the criteria used in the system safety test are both necessary and sufficient to avoid deadlocks. Thus, the extended banker's algorithm has the potential to improve the resource utilization while incurring low runtime overhead. For future work, we plan to investigate the patterns of resource-related calls in real-time system software, and to investigate the practicality of the proposed deadlock avoidance algorithm. Also, we plan to generalize the region decomposition algorithm to consider more general resource-request structures.

ACKNOWLEDGMENTS

The author would like to thank the editors and three anonymous referees for their constructive comments on the earlier versions of the paper which streamlined the main ideas and led to an improved presentation. In particular, the proof of Theorem 2 was greatly simplified by one of the referees. This research was partially supported by the National Science Foundation under grant MIP-9496245.

REFERENCES

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, Mass.: Addison-Wesley, 1974.
[2] E.W. Dijkstra, "Cooperating Sequential Processes," *Programming Languages*, F. Genuys, ed., pp. 103-110, New York: Academic Press, 1968.
[3] R. Finkel and H.H. Madduri," An Efficient Deadlock Avoidance Algorithm," *Information Processing Lett*ers, vol. 24, no. 1, pp. 25-30, Jan. 1987.
[4] A.N. Habermann, "Prevention of System Deadlocks," *Comm. ACM*, vol. 12, no. 7, pp. 373-377, 385, July 1969.
[5] A.N. Habermann, *Introduction to Operating System Design*. Chicago: Science Research Assoc., Inc., 1976.
[6] R.C. Holt, "Some Deadlock Properties of Computer Systems," *ACM Computing Surveys*, vol. 4, no. 3, pp. 179-196, Sept. 1972.
[7] T. Minoura, "Deadlock Avoidance Revisited," *J. ACM*, vol. 29, no. 4, pp. 1,023-1,048, Oct. 1982.