HARDWARE/SOFTWARE DEADLOCK DETECTION ALGORITHM AND IMPLEMENTATION

GIT-CC-02-46

SEPTEMBER 4, 2002

PUN HANG SHIU

Abstract. This report introduces a new theorem and its proof about the problem of deadlock detection. First, we examine how to represent the problem of deadlock with a directed graph. Then, translation from a directed graph into a matrix is elaborated. The theorem and its proof are based on this matrix representation. By applying this theorem, we present a novel parallel deadlock detection algorithm, which we hypothesize has a run-time complexity of $O_{hw}(\min(m,n))$ in a parallel hardware implementation, where m, n are the number of processors and resources involved in deadlock detection respectively.

1. The Deadlock Problem

Deadlock[1] is a system state when processors are waiting for resources held by other processors which, in turn, are also waiting for some resources held by the previous processors.

Example 1. We have two processors p_1 and p_2. In addition, we have two resources q_1 and q_2. Processor p_1 is holding a resource q_1 and makes a new request of resource q_2. At the same time, another processor p_2 is holding q_2 and makes a new request for resource q_1. Processor p_1 needs both resources q_1 and q_2 to complete its task, thus processor p_1 will not release resource q_1 unless processor p_1 obtains the resource q_2. The converse is true for processor p_2: p_2 needs both resources q_1 and q_2 to proceed in its program before releasing any resource. Therefore, both p_1 and p_2 are waiting for a never-released resource from each other. At the end, neither processor p_1 nor p_2 are performing any useful work at all.

The situation of a system being in deadlock is also called a **deadlock state**[1]. In general, a "processor" can be any entity capable of requesting a resource; in this technical report, however, we will use "processor" to refer to either a standard Von Neumann style processor or to custom VLSI hardware able to request other hardware (or, in some cases, software) resources. A system in a deadlock state

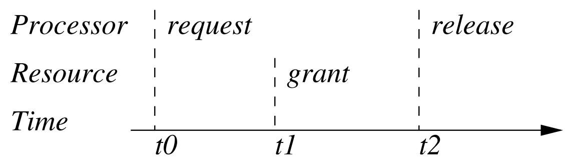


FIGURE 1. Some Terms

does not perform useful work because processors are blocked from either competing for other resources or communicating with other processors.

Resources can be classified into two groups. A **consumable resource** is characterized by (1) no fixed total number of units (units can be created or destroyed), (2) when the processor finishes using the acquired resource, the resource ceases to exist, and (3) an unblocked processor of the resource may release any number of units. These units then become immediately available to requesting processors. An example of a consumable resource is a message. A **reusable resource** is characterized by (1) fixed total inventory (units are neither created nor destroyed) and (2) units are requested and acquired by processors from a pool of available units. When the processor finishes using the acquired reusable resource, the resource is returned to the resource pool so that other processors can have a chance to use the resource. Note that in this technical report, we only consider reusable resources. Therefore, all further references to "resource" should be read as "reusable resource." The number of reusable resources can be classified into two classes. The first class is a **multiple-resource system** and has resources that have multiple units per resource type. The second class is a **single-resource system** and has resources that have one unit per resource type. This report will focus on a reusable single-resource systems which, we predict, will be commonly found in future System-on-a-Chip (SoC) designs.

In general, a system assumes the following guidelines when sharing resources among processors: (1) a processor must request a resource before using it; (2) a processor cannot proceed to use the resource until the processor's request is granted; (3) a processor must release the resource when the processor finishes using the resource; and (4) a processor may request as many resources as it likes as long as the requests do not exceed the total number of available resources.

From the hardware point of view, all the resources and processors are known to the system. Before using any resources exclusively, a processor must first ask for permission to obtain exclusive access to a particular resource. A processor is allowed to access a resource only after permission is given. A processor can give up the right of exclusive access to a particular resource by releasing the resource. In

Figure 1, there are three time stamps: $t0$, $t1$, and $t2$. The initial request for a resource begins at time $t0$. The resource is first granted at time $t1$. The moment of giving up exclusive access to the resource is time $t2$. The period from $t0$ to $t1$ is called **request**. The period from $t1$ to $t2$ is called grant of usage or **grant**.

From an Operating System (OS) point of view, **request** is an OS kernel routine, which enables processors to obtain shared resources. Also, the OS can keep track of availabilities of resources. Following the previously stated guidelines, a processor requests a resource and the OS schedules the resource to be given to the processor. When a processor requests a busy resource, the processor is constantly waiting for the busy resource to be assigned to the processor. During this time, the processor is unable to execute important task(s) which require the resource in order to be executed. During such a waiting period, we say that the processor is **blocked**. On the other hand, if the requested resource is available, a **grant** routine is called to update the resource allocation data structure internal to the operating system. Once the grant routine is completed, the processor can execute tasks requiring the obtained resource. Another OS kernel routine is **release**; release explicitly terminates the exclusive access of a shared resource.

1.1. Motivation.

In a System-on-a-Chip (SoC), there may be several processors or processing elements on a single chip. Besides processing elements, there are also a lot of hardware units for various functions, such as telecommunication functions, image processing, and special hardware accelerators. deadlock detection in hardware will also enhance hardware/software debugging. Each processing element can have a different policy of using resources to meet a specific requirement.

Development of a real-time System-on-a-Chip (SoC) demands a deterministic and fast Real-Time Operating System (RTOS), which provides services and manages resources between software and hardware. However, the algorithms implementing RTOS services may be non-deterministic or may have long execution times. Since the RTOS also competes for the shared CPU on which the RTOS executes, RTOS services may be even less deterministic. For real-time systems, optimization beyond assembly code is desired, such as a custom hardware unit similar to FASTCHART[8]. Therefore, implementing deadlock detection in hardware can provide a better alternative which not only reduces the load of a shared CPU but also improves determinism of the overall SoC system.

Furthermore, moving deadlock detection out of the RTOS and into custom hardware gives more bandwidth to the rest of the RTOS services, allowing the RTOS to handle more services with faster run time, more concurrency, and better utilization of the underlying SoC.

Note that in the previous section we could have discussed a specific task on each processor which requests the resource(s); however, given our target architecture, we focus on a coarse level of granularity where we represent the request as coming from the processor, even though a specific task or set of tasks on the processor requires the resource(s).

1.2. **Organization.**

This report is organized as follows. A graph model applicable to deadlock detection is introduced in Section 2. Deadlock definition and properties are discussed in Section 3. A theorem and novel algorithm for parallel deadlock detection is presented in Section 4. Finally, this report is closed with an conclusion in Section 5.

2. Graph Models

Before proceeding further to apply graph theory to a deadlock problem, basic terms are introduced next and then some properties are pointed out.

2.1. **Definitions of the Resource Allocation Graph.**

Definition 1. *Let $P = \{p_1, p_2, \ldots, p_m\}$ be a set of m requesters or processors which may request and/or hold a number of resources at any time.*

If a processor needs to use a resource, the processor has to make a request first. Once a request is acknowledged by a grant, the processor can then safely use the resource exclusively.

Definition 2. *Let $Q = \{q_1, q_2, \ldots, q_n\}$ be a set of n resources which provide a specific functions usable by the processors. Each resource q_i can serve only one processor at any given time.*

Processors can obtain resources dynamically. Such an interaction between a processor and a resource is carried out by requests (Definition 3), grants (Definition 4), and release routines, which are denoted by various types of edges (except the releases, which only remove grant edges.).

Definition 3. *Let R be the set of request edges. Let an ordered pair (p_i, q_j) be a request edge, where the first node is a processor $p_i \in P$ and the second node is a resource $q_j \in Q$. Thus a set of request edges R can be written as follows:*

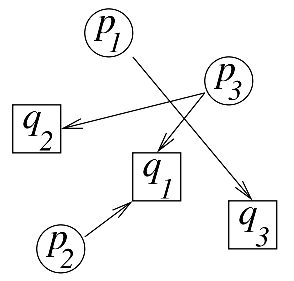
$R = \{(p_i, q_j) \| i \in \{1, 2, 3, \ldots, m\}, j \in \{1, 2, 3, \ldots, n\}$, *and processor p_i is requesting resource q_j*$\}$.

An ordered pair (p_i, q_j) can also be represented by $p_i \to q_j$, where the arrow represents a request edge. Another notation for a request edge is r_{ij}, where the first index represents the processor p_i and the second index represents the resource q_j.

These three notations are used to facilitate interpretation. Notation (p_i, q_j) is used in a graph or set expression. Notation $p_i \to q_j$ is used in diagrams illustrations. and square represents p_i and q_j respectively. Notation r_{ij} is used in a table or matrix, where the subindex ij is implicitly understood as determined by the location of r in row i and column j.

Example 2. Consider Figure 2. Processor p_2 makes one request of resource q_1. This request is represented by an edge (p_2, q_1) in set R, arrow $p_1 \to q_2$ shown in the graph of Figure 2, and r_{21} in the matrix on the right hand side of Figure 2. Note that r_{21} is the r located in row 2, column 1. Processor p_1 makes one request of resource q_3. Such request is represented by an edge (p_1, q_3) in set R and by r in row 1, column 3 of the matrix. Processor p_3 makes two requests of resource q_1 and q_2. Such requests are represented by two edges (p_3, q_1) and (p_3, q_2) in the request set R, arrows in the graph of Figure 2, and r entries in the matrix of Figure 2. The final resulting request set R is also shown in Figure 2.

$$R=\{(p_1, q_3), (p_3, q_2), (p_3, q_1), (p_2, q_1)\}$$



	q_1	q_2	q_3
p_1			r
p_2	r		
p_3	r	r	

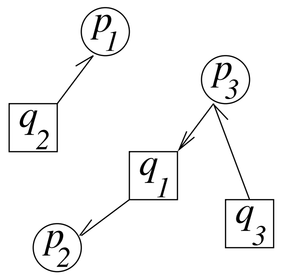
FIGURE 2. Example of Request

Definition 4. *Let $\boldsymbol{G}$ be a set of grant edges. Let an ordered pair (q_j, p_i) be a grant edge, where the first node is a resource and the second node is a processor. Thus a set of grant edges $\boldsymbol{G}$ can be written as $\boldsymbol{G} = \{(q_j, p_i)$, such that $i \in \{1, 2, 3, \ldots, m\}$ and $j \in \{1, 2, 3, \ldots, n\}\}$. An ordered pair (q_j, p_i) can* *also be represented by a $p_i \leftharpoonup q_j$, where the harpoon "$\leftharpoonup$" represents a grant edge. Another notation is g_{ij}, where the first index represents the processor p_i and the second index represents the resource q_j.* $G = \{(q_j, p_i) \mid q \in Q \wedge p \in P \wedge (p, q) \notin R\}$

Several notations are used to facilitate interpretation. Notation (q_j, p_i) is used in graph or set expression. Notation $p_i \leftharpoonup q_j$ is used in diagrams or figures illustration. Notation g_{ij} is used in a table or a matrix.

$$R=\{(p_3, q_1)\}$$

$$G=\{(q_2, p_1), (q_1, p_2), (q_3, p_3)\}$$



	q_1	q_2	q_3
p_1		g	
p_2	g		
p_3	r		g

FIGURE 3. Example of Grant and Request edges

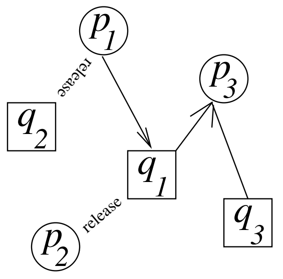
Example 3. Consider Figure 3. Resource q_1 is granted to processor p_2. This case is represented by (q_1, p_2) in set G or g_{21} in the matrix (i.e., entry g in row 2, column 1 of the matrix in Figure 3). Resource q_2 is granted to processor p_1; this grant is represented by (q_2, p_1) in set G, a harpoon arrow in the graph of Figure 3, and g in row 1, column 2 of the matrix in Figure 3. Resource q_3 is granted to processor p_3. Resource q_1 is currently granted to processor p_2, thus processor p_3 has to wait for resource q_1 to be free. Such grant and request edges are represented by two edges (q_3, p_3) and (p_3, q_1) in the grant set G and request set R respectively. Therefore, the request set R contains $\{(p_3, q_1)\}$ and the grant set G contains $\{(q_2, p_1), (q_1, p_2), (q_3, p_3)\}$. The union of the request edge set and the grant edge set is $\{(p_3, q_1), (q_2, p_1), (q_1, p_2), (q_1, p_3)\}$.

Definition 5. *If a grant edge ceases to exist in a graph, that we say that the grant edge is* ***released****.*

Example 4. In this example, we start with the system state shown in Figure 3. Processor p_2 releases resource q_1 which is immediately granted to processor p_3, as shown in Figure 4. The release of resource q_1 is represented by the removal ("release") of grant edge (q_1, p_2) and the removal of g_{21} from the matrix of Figure 4. Furthermore, processor p_1 releases resource q_2, which is now available to serve other processors. Thus grant edge (q_2, p_1) is removed ("released") from the grant set G. Processor p_1 at the same time makes a new request of resource q_1, which is being used by processor p_3 exclusively, and thus processor p_1 has to wait for resource q_1 to be available. Overall, this case is represented by replacing a request edge (p_3, q_1) by a grant edge (q_1, p_3), by adding a request edge (p_1, q_1), and by eliminating ("releasing") two edges. Note that "release" in the middle diagram of Figure 4 represents a release of a resource by a processor.

$$R=\{(p_1, q_1)\}$$

$$G=\{(q_1, p_3), (q_3, p_3)\}$$



	q_1	q_2	q_3
p_1	r		
p_2			
p_3	g		g

FIGURE 4. Example of Release, Grant, and Request Edges

Definition 6. *A given system with processors and resources can be abstracted by a Resource Allocation Graph (**RAG**). A RAG is a directed graph $\gamma = \{V, E\}$, such that V is a non-empty set of nodes and E is a set of ordered pairs or edges*[10]. *Note that the edge set E maybe empty. Using Definitions 1-5, a RAG can be described as a bipartite graph $\gamma = \{V, E\}$, where $V = \{P \cup Q\}$ and $E = \{R \cup G\}$. The set V, the set of nodes in the RAG, can be divided into two disjoint subsets P and Q such that $P \cap Q = \emptyset$, where the **processor subset** is represented by $P = \{p_1, p_2, p_3, \cdots, p_m\}$ (Definition 3) and the **resource subset** is represented by $Q = \{q_1, q_2, q_3, \cdots, q_n\}$ (Definition 4). Therefore, graph γ is bipartite. The set E, the set of directed edges in the RAG, can be divided into two disjoint subsets R and G such that $R \cap G = \emptyset$, where the request subset is represented by $R = \{(p_i, q_j) \mid 1 \leq i \leq m, 1 \leq j \leq n\}$ and the grant subset is represented by $G = \{(q_j, p_i) \mid 1 \leq i \leq m, 1 \leq j \leq n\}$. The total number of nodes V in a system γ_i is $V = P + Q = m + n$, where the subindex i represents a particular set V of a RAG γ.*

Definition 7. *The edge set E is equal to $R \cup G$. An edge is represented by (v_i, v_j) such that $v_i, v_j \in V$ and either $v_i \in P$ and $v_j \in Q$, or $v_i \in Q$ and $v_j \in P$. An edge (v_i, v_j) denotes a request edge r_{ij} if the first node v_i is a processor node; e.g., $(p_i, q_j) = r_{ij}$. On the other hand, an edge (v_i, v_j) denotes a grant edge g_{ij} if the first node is a resource node; e.g., $(q_i, p_j) = g_{ij}$.*

In the figures in this report, a circle represents a processor, while a square represents a resource. Hence, a group of circles forms a set P of processors and a group of boxes forms a set Q of resources.

Furthermore, as stated earlier, in the figures of this report, an arrow "$\rightarrow$" represents a request edge while a harpoon "$\leftharpoonup$" (or "$\rightharpoonup$") represents a grant edge.

Definition 8. *Given RAG γ, let function $E(\gamma)$ be defined as the set of edges E of RAG γ. Note that, from Definition 6, we know that $E = \{R \cup G\}$. The function $R(\gamma)$ is defined as the set of request edges R of RAG γ. Similarly, the function $G(\gamma)$ is defined as the set of grant edges G of a RAG γ. Let function $V(\gamma)$ be defined as the set of nodes $V = \{P \cup Q\}$ of RAG γ. The function $P(\gamma)$ is defined as a set of processors P of RAG γ. The function $Q(\gamma)$ is defined as a set of resources Q of RAG γ.*

Example 5. Let the RAG in Figure 4 having three processors and three resources be system γ_i. The function $E(\gamma_i)$ gives a set of edges $\{(p_1, q_1), (q_1, p_3), (q_3, p_3)\}$. The function $R(\gamma_i)$ gives a set of request edges $\{(p_1, q_1)\}$. The function $G(\gamma_i)$ gives a set of grant edges $\{(p_1, q_1), (q_3, p_3)\}$. The function $V(\gamma_{ij})$ gives a set of nodes $\{p_1, p_2, p_3, q_1, q_2, q_3\}$. The function $P(\gamma_{ij})$ gives a set of processor nodes $\{p_1, p_2, p_3\}$. The function $Q(\gamma_{ij})$ gives a set of resource nodes $\{q_1, q_2, q_3\}$.

Definition 9. *First of all, note that for a given (fabricated) SoC or Printed Circuit Board (PCB), the processors and resources are already decided upon and do not change. Therefore, a particular system $\gamma_i = \{V, E\}$ representing this SoC or PCB will never change its set V of vertices (processors and resources). We define $\gamma_{ij_1}, \gamma_{ij_2}, \gamma_{ij_3}, \ldots$ to be different instances or* ***states*** *of the same SoC or PCB (same set V). Note that the edge set $E(\gamma_{ij})$ is different for each $j \in \{1, 2, 3, \ldots\}$. Since the node set $V = \{P \cup Q\}$ is constant for a given system γ_i, the edge set E has enough information to represent a current state γ_{ij}, defined by the function $E(\gamma_{ij})$ of a state γ_{ij} of a given system γ_i, where the second subindex j represents a particular set of E of a system γ_i. Thus a particular state γ_{ij} is uniquely defined relative to γ_i by $E(\gamma_{ij}) = \{R \cup G\}$. A system γ_i changes from one state γ_{ij} to another state γ_{ik} when handling requests, grants, and releases of resources*[9].

Example 6. In Figure 5, a given system γ_i in a particular system state γ_{ij} is shown. $V = \{P \cup Q\}$ is the set of nodes and $E = \{R \cup G\}$ is the set of edges in γ_{ij}. A circle in Figure 5 represents a processor, while a box represents a resource. The set P of processors are shown by three circle nodes, which are $\{p_1, p_2, p_3\}$. The set Q of resources are shown by three box nodes, which are $\{q_1, q_2, q_3\}$. The set $E = \{R, G\}$ of edges has two disjunct sets. The set $R = \{(p_1, q_2), (p_2, q_3)\}$ of edges are requests shown

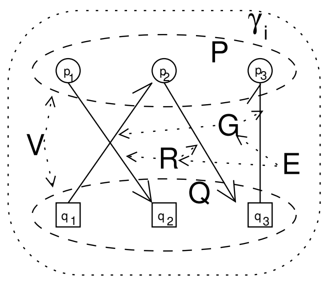


FIGURE 5. An Example of RAG in Bipartite Graph.

by arrows from set P to set Q. The set $G = \{(q_1, p_2), (q_3, p_3)\}$ of edges are grants shown by harpoons pointing from set Q to set P.

A SoC or PCB system is represented by a bipartite graph γ. A system γ_i defines a fixed set P of processors and a fixed set Q of resources, while a system state γ_{ij} represents the current actions (a set of requests R and a set of grants G).

An adjacency matrix M_{ij} is another representation of system state γ_{ij}. The dimension of matrix M_i is $m \times n$, m rows and n columns respectively. The set P of processors is mapped to the rows of matrix M. And the set Q of resources is mapped to the column of matrix M. In other words, the s^{th} row of matrix M represents all the edges (requests from or grants to) belonged to processor p_s. Similarly, The t^{th} column of matrix M represents all the edges (requests to or grants from) belonged to resource q_t. Each entry m_{st} (at s^{th} row and t^{th} column) in the matrix can be either request r_{st}, grant g_{st}, and available (as empty or release). For clarity in the matrix M_{ij}, r_{st} is written as r in the row s and column t. Similarly, it is the same for g_{st}. If there is a grant edge $(q_t, p_s) \in G$, there is a g_{st}(or g at row s and column t in matrix M_{ij}). If there is a request edge $(p_s, q_t) \in R$, there is a r_{st} (or r at row s and column t in the matrix M_{ij}).

Definition 10. *This definition aims to define matrices which correspond to graph γ, system γ_i, and state γ_{ij}. A **RAG matrix M** represents an arbitrary system with processors and resources. A **system matrix M_i** is defined as a matrix representation of system γ_i where the rows (fixed in size) of matrix M_i represent the fixed set P of processor nodes of γ_i, and the columns (fixed in size) of matrix M_i represent the fixed set Q of resource nodes of γ_i. A **state matrix M_{ij}** represents to a system state γ_{ij}. Edges in system state γ_{ij} are mapped into the array elements using the following rule:*

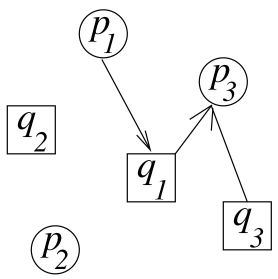
Given $E = \{R \cup G\}$ from γ_{ij}, *for all rows* $0 \leq s \leq m$, *and for all columns* $0 \leq t \leq n$*:*

$m_{st} = r_{st}$ *(r for clarity), if there exists a request edge* $(p_s, q_t) \in R$

$m_{st} = g_{st}$ *(g for clarity), if there exists a grant edge* $(q_t, p_s) \in G$

$m_{st} = 0_{st}$ *(blank for clarity), otherwise*

Example 7. Example 4 shows an equivalent state matrix (on the right hand side of Figure 4) of the system state γ_{ij} described in Example 4. The system state γ_{ij} and state matrix M_{ij} corresponding to Example 4 are shown below in Figure 6. The request (p_1, q_1) from γ_{ij} is represented by $m_{11} = r_{11} = r$ in the top left entry of the matrix as shown in Figure 6. Similarly, the grant (q_3, p_3) from γ_{ij} is represented by $m_{33} = g_{33} = g$ in M_{ij} as shown in Figure 6.



(a) system state graph γ_{ij}

M_{ij}	q_1	q_2	q_3
p_1	$m_{11}=r$	$m_{12}=0$	$m_{13}=0$
p_2	$m_{21}=0$	$m_{22}=0$	$m_{22}=0$
p_3	$m_{31}=g$	$m_{32}=0$	$m_{33}=g$

(b) system state matrix M_{ij}

FIGURE 6. Example of a system state graph γ_{ij} and the corresponding system state matrix M_{ij}.

Definition 11. *We overload the equality operator "=" in this technical report as follows: whenever an expression* $M_{ij} = \gamma_{ij}$ *is seen, this means that matrix* M_{ij} *is created from* γ_{ij} *using Definition 10.*

In summary, in this technical report, we use notation as follows. A RAG γ, as defined in Definition 6, represents a RAG, a directed bipartite graph with a set of nodes V and a set of edges E. A **system** γ_i represents a particular system, where the node set $V = \{P, Q\}$ represents the system and does not change. A **state** γ_{ij} represents a particular instance of interactions (requests and grants) between processors and resources of a given system γ_i. Such interaction is captured in the edge set $E = \{R \cup G\}$ which is used to represent formally the state γ_{ij} of a given system γ_i.

2.2. Definitions of Types of Deadlock or Near-Deadlock States.

Note that for some state γ_{ij}, it may be possible to fulfill all requests in any arbitrary order without ever entering into a deadlock, and all requests are fulfilled in a timely fashion. Such states $\{\gamma_{ij}\}$ are called **secure**.

Definition 12. *For a particular system γ_i, $\Gamma^k = \{\gamma_{ij_1}, \gamma_{ij_2}, \gamma_{ij_3}, \ldots\}$ is a set of states, possibly empty, such that all states in Γ^k are secure.*

Now, consider the case where there exists at least one request which is never fulfilled or is fulfilled so seldomly that the processor requesting the resource(s) is unable to perform its tasks in a timely fashion. Such a case does not lead to deadlock but is not a secure case either. This case is called **starvation**.

Definition 13. *For a given system γ_i, we define Γ^h as to be a set of states $\{\gamma_{ij_1}, \gamma_{ij_2}, \gamma_{ij_3}, \ldots\}$, possibly empty, such that all states in Γ^h are starvation states.*

In this report, we refer to a sequence of resource allocations via requests and grants as a **resource scheduling**. Now, consider a set of states Γ^j where there exists at least one resource scheduling that keeps a system out of deadlock. This case is called **safe**.

Definition 14. *For a given system γ_i, we define Γ^j to be a set of states $\{\gamma_{ij_1}, \gamma_{ij_2}, \gamma_{ij_3}, \ldots\}$, non-empty, such that all states in Γ^j are safe states.*

Note that all secure and starvation states are, by definition, safe states. More formally, $\Gamma^k \subset \Gamma^j$ and $\Gamma^h \subset \Gamma^j$. Some authors also refer to safe states as **reducible**[1].

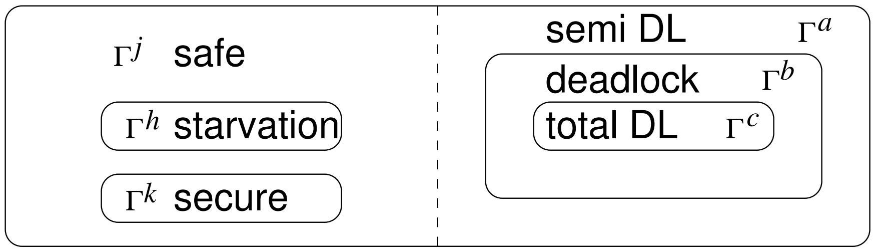


FIGURE 7. Sets of System States $\Gamma^h, \Gamma^j, \Gamma^k, \Gamma^a, \Gamma^b, \Gamma^c$.

Now, consider the case where a system will enter a deadlocked state regardless of in which order requests are granted due to new requests in the near future. Note that the system may or may not currently be deadlocked. This case is called **semi-deadlock**. Some authors also refer to semi-deadlock states as "non-safe"[1]. Clearly, absent knowledge of the future requests, it is not possible to detect if a system is in a semi-deadlock state.

Definition 15. *For a given system γ_i, we define Γ^a to be a set of states $\{\gamma_{ij_1}, \gamma_{ij_2}, \gamma_{ij_3}, \ldots\}$, possibly empty, such that all states in Γ^a are semi-deadlock states.*

Now, consider the case where there are some processors and resources deadlocked. There may also be some other processors or resources not part of the deadlock. This case is called **deadlock**.

Definition 16. *For a given system γ_i, we define Γ^b to be a set of states $\{\gamma_{ij_1}, \gamma_{ij_2}, \gamma_{ij_3}, \ldots\}$, possibly empty, such that all states in Γ^a are deadlock states.*

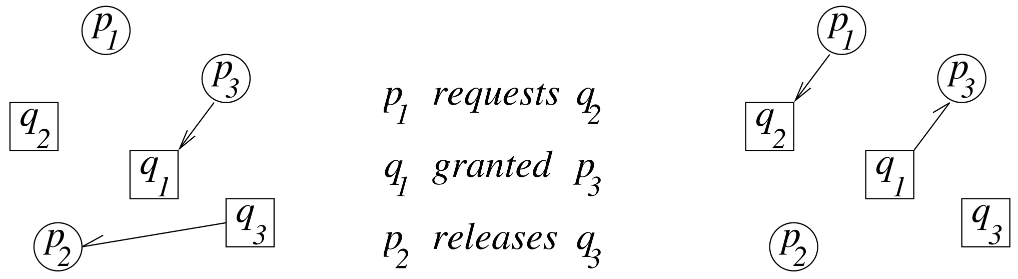
Now, consider the case where all processors and resources are deadlocked. This case is called **total-deadlock**.

Definition 17. *For a given system γ_i, we define Γ^c to be a set of states $\{\gamma_{ij_1}, \gamma_{ij_2}, \gamma_{ij_3}, \ldots\}$, possibly empty, such that all states in Γ^c are total deadlock states.*

Using Definitions 15, 16, and 17, note that $\Gamma^c \subset \Gamma^b \subset \Gamma^a$. We say that a state γ_{ij} is unsafe if state γ_{ij} is in one of the semi-deadlock states Γ^a – note that, by definition, state γ_{ij} could possibly also be in Γ^b or Γ^c as well. is a critical difference between deadlock and starvation. In a deadlock state, a processor waits for resources held by other processor(s) that will never be released, while in a starvation state, at least one processor never obtains enough to execute even though the resources periodically become available (only to be snatched up by other processors first).

The set Γ contains the union of the set of safe states and the set of semi-deadlock states. More formally, $\Gamma = \Gamma^j \cup \Gamma^a$.

Example 8. Consider system γ_j as shown in Figure 8. In state γ_{ji_1}, processor p_3 requests resource q_1, while at the same time p_2 is using resource q_3 which has been granted to p_2. State γ_{ji_1} is defined uniquely by $P = \{p_1, p_2, p_3\}$, $Q = \{q_1, q_2, q_3\}$, and the edge set $E = \{(p_3, q_1), (q_3, p_2)\}$. Now consider what happens if processor p_1 requests resource q_2, resource q_1 is granted to processor p_3, and p_2 releases resource q_3. The resulting state γ_{ji_2} is represented by the edge set $E = \{(p_1, q_2), (q_1, p_3)\}$ and is shown in Figure 8(c).



(a) State γ_{ji_1} (b) Actions (c) State γ_{ji_2}

FIGURE 8. Relationship between a State γ_{ji} (RAG) and edges E.

Now, let us put the previous definitions together and see the complexity of the deadlock detection problem as shown in Table 1 and Figure 9 for a system γ_k with two processors and two resources.

Definition 18. *Note that at any point in time a processor may request or release a resource. Furthermore, at any point in time, an outstanding request for a resource may be granted. We refer to any such request, release, or grant as an* ***action****. Each action is associated with a* ***time-stamp****, which captures the relative timing among actions. Thus, the timing of a sequence of actions can be represented concisely by the sequence of time-stamps.*

Example 9. Consider a system γ_k with two processors $\{p_1, p_2\}$ and two resources $\{q_1, q_2\}$. At some point in time, each processor requires both resources at the same time to perform certain tasks. In this system, each processor performs the actions shown in Table 1 at the time-stamps shown in Table 1. The processor p_1 will go from time-stamp s_0 (p_1 has no action) to s_1(an action that p_1 requests q_1) and then from time-stamp s_1 to time-stamp s_2(an action that q_1 is granted to p_1). Finally, in our example, processor p_1 goes from time-stamp s_4(an action that q_2 is granted to p_1) to time-stamp s_5(no action, but p_1 is holding q_1). When the processor p_1 reaches time-stamp s_5, the processor p_1 will go back to time-stamp s_0 and will repeat the previous sequence of time-stamps. The processor p_2 will go from time-stamp t_0(p_2 has no action) to time-stamp t_1(an action that p_2 requests q_2) and then from time-stamp t_1 to time-stamp t_2(an action that q_2 is granted to p_2). Finally, in our example, processor p_2 goes from time-stamp t_4(an action that q_1 is granted to p_2) to time-stamp t_5(no action but p_2 is holding q_2). When the processor p_2 reaches time-stamp t_5, the processor p_2 will go back to time-stamp t_0 and will repeat the previous sequence of time-stamps.

multicolumn2cActions of p_1		Actions of p_2	
Time-Stamp	Action	Time-Stamp	Action
s_0	no action	t_0	no action
s_1	request q_1	t_1	request q_2
s_2	grant q_1	t_2	grant q_2
s_3	request q_2	t_3	request q_1
s_4	grant q_2	t_4	grant q_1
	release q_2		release q_1
s_5	no action	t_5	no action
	release q_1		release q_2

TABLE 1. Action Sequence for Processors p_1 and p_2

In Figure 9, both left and right horizontal arrows are action transitions due to actions by p_1. Furthermore, in Figure 9, both up and down vertical arrows are action transitions due to actions by p_2. In state γ_{k20} in Figure 9, processor p_1 is at time-stamp s_3: p_1 is holding q_1 while requesting q_2. At the same time, in state γ_{k20} processor p_2 is at time-stamp t_3: p_2 is holding q_2 while requesting q_1. Clearly, processors p_1 and p_2 are deadlocked in state γ_{k20}, which is also a total deadlock state in this system γ_k because all the processors and resources are involved. States γ_{k14}, γ_{k19} and γ_{k15} are semi-deadlock states because p_2 and p_1 are going to deadlock in the future (given the known and unchanging execution patterns or action sequences we have described for p_1 and p_2, we can prove that the system will definitely enter total deadlock state γ_{k20}). Although there are no secure states in Figure 9, there are safe states: namely, the twenty-three states which are not total deadlock states (γ_{k20}) nor semi-deadlock states (γ_{k14}, γ_{k15} and γ_{k19}).

In this report, we assume no knowledge about future execution patterns resulting in future requests and grants. Therefore, we are unable to detect semi-deadlock states. However, we can detect any any deadlock or total deadlock state. Similarly, we do not detect starvation states (note that starvation is typically due to a poor resource scheduling policy which can possibly be changed to avoid starvation). The scope of this report is limited to the fast detection of deadlock and total deadlock states.

2.3. **Definitions and Properties of Various Edges.**

This section further refines various relationships among edges. Such relationships of edges give the properties and definition of a deadlock. In other words, a particular set of edges corresponds to a particular system state, in which we would like to detect if there is a deadlocked scenario or not.

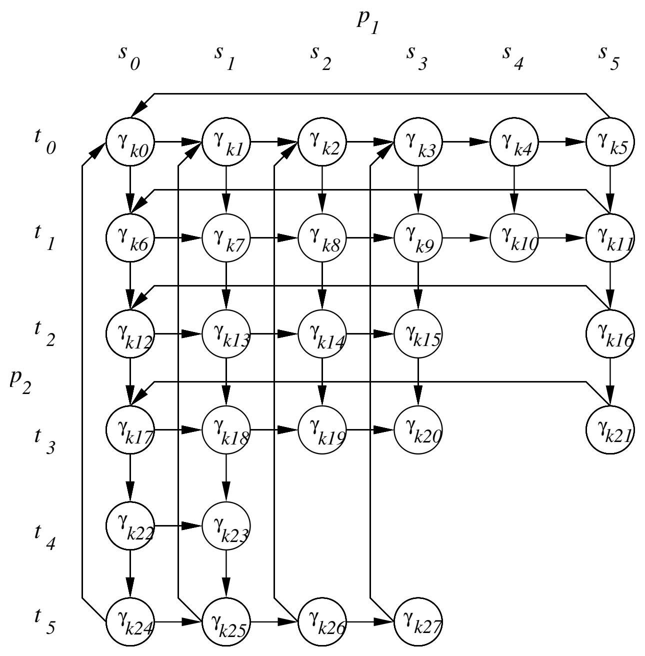

p_1
s_0 s_1 s_2 s_3 s_4 s_5
p_2
t_0 t_1 t_2 t_3 t_4 t_5
γ_{k0} γ_{k1} γ_{k2} γ_{k3} γ_{k4} γ_{k5}
γ_{k6} γ_{k7} γ_{k8} γ_{k9} γ_{k10} γ_{k11}
γ_{k12} γ_{k13} γ_{k14} γ_{k15} γ_{k16}
γ_{k17} γ_{k18} γ_{k19} γ_{k20} γ_{k21}
γ_{k22} γ_{k23}
γ_{k24} γ_{k25} γ_{k26} γ_{k27}

FIGURE 9. System States of Table 1

Definition 19. *The **out-degree** of a node v is the number of directed edges going to other nodes from node v.*

Definition 20. *The **in-degree** of a node v is the number of directed edges coming in to node v from other nodes.*

Definition 21. *The **degree** of node v is the total number of directed edges connected to node v. Note that the sum of the out-degree of node v and the in-degree of node v equals the **degree** of node v. More formally, degree(v)=out-degree(v) + in-degree(v).*

Definition 22. *A **isolated** node v_ϵ is a node that does not have any in-coming edges nor any out-going edges; more formally, node v_ϵ is isolated if both $E \cap \{(v, v_\epsilon) \mid v \in V\} = \emptyset$ and $E \cap (v_\epsilon, v) \mid v \in V = \emptyset$.*

In other words, the number of edges of a isolated node is zero because the in-degree is zero and out-degree is zero.

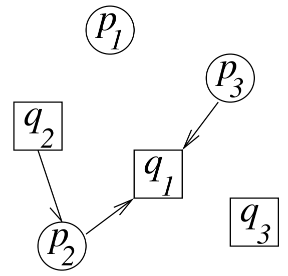
Definition 23. *Given a RAG in state γ_{ij} (see Definition 9), let $\epsilon(\gamma_{ij})$ be a function which returns a set $\{v_{\epsilon_1}, v_{\epsilon_2}, \ldots, v_{\epsilon_p}\}$ of isolated vertices. Furthermore, let $\Sigma_\epsilon = \{v_{\epsilon_1}, v_{\epsilon_2}, \ldots, v_{\epsilon_p}\}$. Note that it is possible for Σ_ϵ to be empty, i.e., it may be the case that $\Sigma_\epsilon = \emptyset$.*

Recall that a node in a RAG can be either a processor or a resource (Definition 6). If a processor does not request nor hold any resource, that processor is said to be an isolated processor. Obviously, an isolated processor cannot participate in any deadlock states. A resource is said to be isolated when that resource is not requested nor held by any processor. Such isolated nodes need not be considered by a deadlock detection algorithm. The advantage of identifying any isolated node(s) is that the problem space can be shrunk by ignoring the isolated node(s).

Definition 24. *If a node is not isolated, it is called* ***non-isolated****. The set of non-isolated node is* $V - \Sigma_\epsilon$.

When a resource is being requested by a processor, that resource and the processor are said to be non-isolated, because there is an action (interaction) between the processor and the resource. Specifically, the processor has an outgoing edge, while the resource has an incoming edge.

Example 10. Consider the left hand side of Figure 10. Processor p_1 is not requesting nor holding any resource, thus processor p_1 is an isolated processor node. Similarly, resource q_3 is not being requested nor held by any processor, thus resource q_3 is an isolated resource node. It is easy to identify any isolated node by checking either column j (for resource q_j) or row i (for processor p_i). On the right hand side of Figure 10, row 1 (for processor p_1) is empty; thus, processor p_1 is an isolated processor node. Again, consider the matrix on the right hand side of Figure 10: column 3 (for resource q_3) is empty and thus resource q_3 is an isolated node. Nodes p_2, p_3, q_1, and q_2 are non-isolated nodes, because there are edges connected to or from each of these nodes. A non-isolated node is also easy to identify using the matrix: when a row or column is not empty, the corresponding processor or resource is non-isolated.



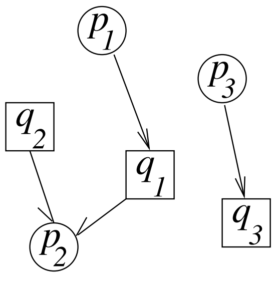
	q_1	q_2	q_3
p_1			
p_2	r	g	
p_3	r		

FIGURE 10. Isolated and non-isolated nodes

Definition 25. *A node v_α is a **sink** node if node v_α both does not have any out going edges (out-degree(v_α) $= 0$) and does have at least one incoming edge (in-degree(v_α) ≥ 1). More formally, v_α is a sink node if both $E \cap \{(v_\alpha, v) \mid v \in V\} = \emptyset$ and $E \cap \{(v, v_\alpha) \mid v \in V\} \mid\geq 1$.*

Definition 26. *Given a RAG in state γ_{ij} (see Definition 9), let $\alpha(\gamma_{ij})$ be a function which returns a set $\{v_{\alpha_1}, v_{\alpha_2}, \ldots, v_{\alpha_p}\}$ of sink vertices. Furthermore, let $\Sigma_\alpha = \{v_{\alpha_1}, v_{\alpha_2}, \ldots, v_{\alpha_p}\}$. Note that it is possible for Σ_α to be empty, i.e., it may be the case that $\Sigma_\alpha = \emptyset$. $\Sigma_{\alpha_{ij}}$ is used to denote the set of sink nodes in state γ_{ij}.*

Definition 27. *Given a RAG in state γ_{ij} and a set $\alpha(\gamma_{ij})$ of sink vertices, let $E_{\alpha_{ij}}$ be the set of edges connected to sink vertices $v_\alpha \in \alpha(\gamma_{ij})$. In other words, $E_{\alpha_{ij}} = \{(v_x, v_y)$ such that $v_x \in \alpha(\gamma_{ij})\}$ – the tail of the edge is the sink vertex. (Recall that edge (v_x, v_y) was defined by Definition 7.)*



	q_1	q_2	q_3
p_1	r		
p_2	g	g	
p_3			r

FIGURE 11. Sink nodes and edges

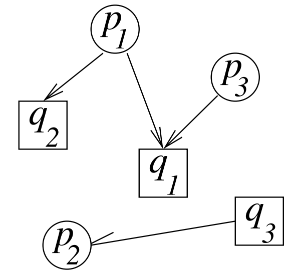
Example 11. Let the left hand side of Figure 11 define state γ_{ij}. Processor p_2 is a sink processor node ($p_2 \in \alpha(\gamma_{ij}) = \Sigma_{\alpha_{ij}}$) with two grant edges: (q_2, p_2) and (q_1, p_2) both elements of $E_{\alpha_{ij}}$. Grant edges $\{(q_2, p_2), (q_1, p_2)\}$ can be called sink edges with respect to the processor p_2. Resource q_3 is a sink resource node ($q_3 \in \alpha(\gamma_{ij})$) with one request edge (p_3, q_3) which is in the set $E_{\alpha_{ij}}$. Request edge (p_3, q_3) can be called a sink edge with respect to resource q_3. One can easily identify a sink node using the matrix on the right hand side of Figure 11. If a row i for processor p_i contains only grant edges, then the processor p_i is a sink node and all the corresponding edges in that row i are called sink edges with respect to processor p_i. If a column j for resource q_j contains only one grant edge, then the resource q_j is a source node and the corresponding edge is called a source edge with respect to resource q_j.

Definition 28. *A node v_β is a* **source** *node if v_β both has at least one outgoing edge (out-degree(v_β) ≥ 1) and does not have any incoming edges (in-degree(v_β) $= 0$). More formally, v_β is a source node if both $E \cap \{(v, v_\beta) \mid v \in V\} = \emptyset$ and $E \cap \{(v_\beta, v) \mid v \in V\} \mid \geq 1$.*

Definition 29. *Given a RAG in state γ_{ij} (see Definition 9), let $\beta(\gamma_{ij})$ be a function which returns a set $\{v_{\beta_1}, v_{\beta_2}, \ldots, v_{\beta_p}\}$ of source vertices. Furthermore, let $\Sigma_\beta = \{v_{\beta_1}, v_{\beta_2}, \ldots, v_{\beta_p}\}$. Note that it is possible for Σ_β to be empty, i.e., it may be the case that $\Sigma_\beta = \emptyset$. $\Sigma_{\beta_{ij}}$ is used to denote the set of source nodes in state γ_{ij}.*

Definition 30. *Given a RAG in state γ_{ij} and a set $\beta(\gamma_{ij})$ of source vertices, let $E_{\beta_{ij}}$ be the set of edges connected to source vertices $v_\beta \in \beta(\gamma_{ij})$. In other words, $E_{\beta_{ij}} = \{(v_x, v_y)$ such that $v_x \in \beta(\gamma_{ij})\}$ – the head of the edge is the source vertex.*

Let us apply the sink and source definitions to an SoC scenario. A processor is a sink processor when the processor only has grant edges. Alternatively, a resource is a sink resource when the resource only has request edges. On the other hand, a processor is a source processor when the processor only has request edges. A resource is a source resource when the resource only has one edge, a grant edge. In such situations, sink or source processors or resources do not satisfy the four necessary conditions for deadlock to occur[1].



	q_1	q_2	q_3
p_1	r	r	
p_2			g
p_3	r		

FIGURE 12. Source Nodes and Edges

Example 12. Let the RAG on the left of Figure 12 define γ_{ij}. Processor p_1 is a source processor node ($p_1 \in \beta(\gamma_{ij}) = \Sigma_{\beta_{ij}}$) because p_1 has not been granted any resources and is making two requests: one request is for resource q_1 and the other request is for resource q_2. The request edges are $(p_1, q_2) = r_{12}$ and $(p_1, q_1) = r_{11}$. We find that $r_{12} \in E_{\beta_{ij}}$ and $r_{11} \in E_{\beta_{ij}}$. Resource q_3 is a source resource node ($q_3 \in \beta(\gamma_{ij})$) because q_3 has only one edge: a grant edge (q_3, p_2) pointing to processor p_2. Grant edge

$(q_3, p_2) = g_{32}$ is an element of $E_{\beta_{ij}}$. Using Definition 25, resources q_2 and q_1 are both sink resource nodes because they only have incoming edges.

Definition 31. *A node v_τ is a **terminal** node, if node v_τ is either a sink or a source node.*

Definition 32. *Given a RAG in state γ_{ij} (see Definition 9), let $\tau(\gamma_{ij})$ be a function which returns a set $\{v_{\tau_1}, v_{\tau_2}, \ldots, v_{\tau_p}\}$ of terminal vertices. Furthermore, let $\Sigma_\tau = \{v_{\tau_1}, v_{\tau_2}, \ldots, v_{\tau_p}\}$. Note that it is possible for Σ_τ to be empty, i.e., it may be the case that $\Sigma_\tau = \emptyset$. Also note that $\Sigma_\tau = \Sigma_\alpha \cup \Sigma_\beta$. Finally, $\Sigma_{\tau_{ij}}$ is used to denote the set of terminal nodes in state γ_{ij}.*

Definition 33. *Given a RAG in state γ_{ij} and a set $\tau(\gamma_{ij})$ of terminal vertices, let $E_{\tau_{ij}}$ be the set of edges connected to terminal vertices $v_\tau \in \tau(\gamma_{ij})$. In other words, $E_{\tau_{ij}} = \{(v_x, v_y)\}$ such that either $v_x \in \tau(\gamma_{ij})$ or $v_y \in \tau(\gamma_{ij})$ (or both).*

Example 13. In Figure 12, let the RAG shown define γ_{ij}. The node p_1 in Figure 12 is a terminal node because p_1 is a source node ($p_1 \in \beta(\gamma_{ij})$ and $p_1 \in \tau(\gamma_{ij})$). The node q_1 is a terminal node because q_1 is a sink node ($q_1 \in \alpha(\gamma_{ij})$ and $q_1 \in \tau(\gamma_{ij})$). The set of terminal nodes $\Sigma_{\tau_{ij}}$ is $\{p_1, p_2, p_3, q_1, q_2, q_3\}$.

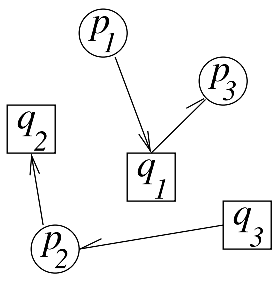
Now let the RAG in Figure 10 define γ_{ij}. In this case, the node p_2 is not a terminal node because it is neither a sink nor a source node. Processor p_1 and resource q_3 are not terminal nodes because they are isolated nodes. The set of terminal nodes in Figure 10 is $\Sigma_{\tau_{ij}}$ is $\{p_3, q_1, q_2\}$.

Definition 34. *A **link** node v_λ is a node that has exactly one in-coming edge and one out-going edge, such that $\|E \cap \{(v, v_\lambda) \mid v \in V\}\| = 1$ and $\|E \cap \{(v_\lambda, v) \mid v \in V\}\| = 1$. Clearly, the number of edges of a link node is two (degree$(v_\lambda) = 2$).*

Definition 35. *Given a RAG in state γ_{ij} (see Definition 9), let $\lambda(\gamma_{ij})$ be a function which returns a set $\{v_{\lambda_1}, v_{\lambda_2}, \ldots, v_{\lambda_p}\}$ of link vertices. Furthermore, let $\Sigma_\lambda = \{v_{\lambda_1}, v_{\lambda_2}, \ldots, v_{\lambda_p}\}$. Note that it is possible for Σ_λ to be empty, i.e., it may be the case that $\Sigma_\lambda = \emptyset$.*

Definition 36. *Given a RAG in state γ_{ij} and a set $\lambda(\gamma_{ij})$ of link vertices, let $E_{\lambda_{ij}}$ be the set of edges connected to link vertices $v_\lambda \in \lambda(\gamma_{ij})$. In other words, $E_{\lambda_{ij}} = \{(v_x, v_y)\}$ such that either $v_x \in \lambda(\gamma_{ij})$ or $v_y \in \lambda(\gamma_{ij})$ (or both).*

A resource is said to be a link resource when that resource is being used by one processor and at the same time is being requested by another processor. The concept of a link resource can also be applied to a processor. When a processor is holding a resource and at the same time requesting an additional resource, that processor is said to be link processor.



	q_1	q_2	q_3
p_1	r		
p_2		r	g
p_3	g		

FIGURE 13. Link Nodes and Edges

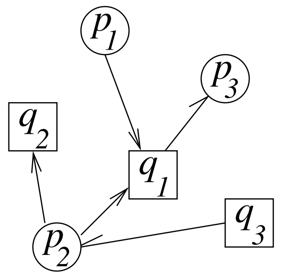
Example 14. Let the system state shown in Figure 13 be γ_{ij}. Processor p_2 is a link processor node ($p_2 \in \lambda(\gamma_{ij})$) because p_2 is involved in both a request (p_2, q_2) and a grant (q_3, p_2). With respect to processor p_2, edges (p_2, q_2) and (q_3, p_2) are link edges. Resource q_1 is also another link node (link resource node) because there are a request edge from processor p_1 and a grant edge to p_3. Processor p_1 is not a link processor node: instead, p_1 is a source processor node. Similarly, processor p_3 is not a link processor node but instead is a sink processor node. Edges (p_1, q_1) and (q_1, p_3) are called link edges with respect to resource q_1. The result is as follows: $\lambda(\gamma_{ij}) = \{p_2, q_1\}$ and $E_{\lambda_{ij}} = \{(p_2, q_2), (q_3, p_2), (p_1, q_1), (q_1, p_3)\} = \{r_{22}, g_{32}, r_{11}, g_{13}\}$
It is easy to identify link nodes using the matrix in Figure 13. For processor p_2, row 2 contains requests r_{22} and grants g_{23}, therefore, p_2 is a link node. For resource q_1, column 4 contains request r_{11} and grant g_{31}; therefore, q_1 is a link node. Resource q_2 is a sink node. Similarly, resource q_3 a sink node.

Definition 37. *A **branch** node v_ω has one or more incoming edges and one or more outgoing edges, such that the total number of edges is greater than or equal to three. More formally, three conditions must hold for v_ω to be a branch node: (1) the in-degree of v_ω must be one or more (in-degree(v_ω) ≥ 1); (2) the out-degree of v_ω must be one or more (out-degree(v_ω) ≥ 1); and (3) the degree of v_ω must be three or more (degree(v_ω) ≥ 3).*

Definition 38. *Given a RAG in state γ_{ij} (see Definition 9), let $\omega(\gamma_{ij})$ be a function which returns a set $\{v_{\omega_1}, v_{\omega_2}, \ldots, v_{\omega_p}\}$ of branch vertices. Furthermore, let $\Sigma_\omega = \{v_{\omega_1}, v_{\omega_2}, \ldots, v_{\omega_p}\}$. Note that it is possible for Σ_ω to be empty, i.e., it may be the case that $\Sigma_\omega = \emptyset$.*

Definition 39. *Given a RAG in state γ_{ij} and a set $\omega(\gamma_{ij})$ of branch vertices, let $E_{\omega_{ij}}$ be the set of edges connected to branch vertices $v_\omega \in \omega(\gamma_{ij})$. In other words, $E_{\omega_{ij}} = \{(v_x, v_y)\}$ such that either $v_x \in \omega(\gamma_{ij})$ or $v_y \in \omega(\gamma_{ij})$ (or both).*

The difference between a branch node and a link node is that while a branch node must have three or more edges, a link node must have only two edges. A resource is said to be a branch resource if the resource is held by one processor and is being requested by two or more processors. A processor is said to be a branch processor if the processor either holds one or more resources while requesting two or more additional resources, or if the processor holds two or more resources while requesting one or more resources.



	q_1	q_2	q_3
p_1	r		
p_2	r	r	g
p_3	q		

FIGURE 14. Branch Nodes and Edges

Example 15. Let the system state shown in Figure 14 be γ_{ij}. Processor p_2 is a branch node because the number of edges to and from p_2 is greater than two and there is at least one request edge and at least one grant edge. Similarly, resource q_1 is also a branch node. The edges connected to a branch node are called branch edges with respect to the branch node. Thus, $\{(p_2, q_2), (q_3, p_2), (p_2, q_1)\}$ are branch edges with respect to branch processor node p_2, and $\{(p_2, q_1), (q_1, p_3), (p_1, q_1)\}$ are branch edges with respect to branch resource node q_1. We end up with the following for this example: $\omega(\gamma_{ij}) = \{p_2, q_1\}$ and $E_{\omega_{ij}} = E(\gamma_{ij})$ (since all edges connect to the two branch nodes).

It is also easy to identify a branch node by examining the matrix on the right hand side of Figure 14. A processor p_i is a branch node if row i has three or more entries with at least one r and one g entry. Similarly, a resource q_j is a branch node if the column j has three or more entries with at least one r entry and one g entry.

Definition 40. *A node v_ϕ is a **connect** node if node v_ϕ is either a link node or a branch node.*

Definition 41. *Given a RAG in state γ_{ij} (see Definition 9), let $\phi(\gamma_{ij})$ be a function which returns a set $\{v_{\phi_1}, v_{\phi_2}, \ldots, v_{\phi_p}\}$ of connect vertices. Furthermore, let $\Sigma_\phi = \{v_{\phi_1}, v_{\phi_2}, \ldots, v_{\phi_p}\}$. Note that it is possible for Σ_ϕ to be empty, i.e., it may be the case that $\Sigma_\phi = \emptyset$.*

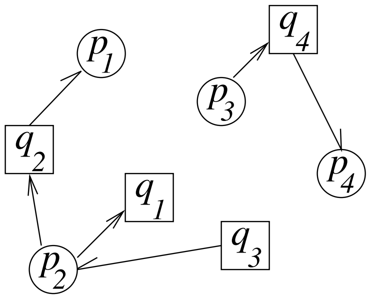
Definition 42. *A **path** $(v_1, v_2, v_3, \ldots, v_{k-1}, v_k)$, $k \geq 2$, is a consecutive ordered sequence of alternating request and grant edges (v_1, v_2), (v_2, v_3), $\ldots$, (v_{k-1}, v_k) where every node in the path is distinct and where every other node belongs to the same set. In other words, every odd node along a path belongs to one node set of V (either P or Q) and every even node along the same path belongs to the other node set of V.*

The "," between two nodes can represent either a request edge or a grant edge. The "," represents a request edge if the previous node is a processor node, while the "," represents a grant edge if the previous node is a resource node. To explicitly illustrate the action (interaction) between nodes, the arrow (request) and harpoon (grant) symbols can be used instead of the comma ",". Thus, a path $v_1 \rightarrow v_2 \rightharpoonup v_3 \rightarrow \cdots \rightharpoonup v_{k-1} \rightarrow v_k$ can be another representation of a path $(v_1, v_2, v_3, \ldots, v_{k-1}, v_k)$ where node v_1 is a processor node in set P. Similarly, path $v_1 \rightharpoonup v_2 \rightarrow v_3 \rightharpoonup \cdots \rightarrow v_{k-1} \rightharpoonup v_k$ can be used to represent a path $(v_1, v_2, v_3, \ldots, v_{k-1}, v_k)$ where node v_1 is a resource node in set Q.

Example 16. In Figure 5, resource q_1 is granted to processor p_2. At the same time, processor p_2 requests resource q_3, which is granted to processor p_3. Thus, Figure 5 has the path (q_1, p_2, q_3, p_3). Resource q_1 is a source node and processor p_3 is a sink node. The processor p_2 and resource q_3 are link nodes in this path. In short, the paths in Figure 5 are (q_1, p_2, q_3, p_3), (q_1, p_2, q_3), (p_2, q_3, p_3), (q_1, p_2), (p_2, q_3), (q_3, p_3) and (p_1, q_2).

Definition 43. *A **simple path** is a path $(v_i, v_{i+1}, \ldots, v_k)$ such that both v_i and v_k are terminal nodes and all other nodes are link nodes. Formally, a path $(v_i, v_{i+1}, \ldots, v_k)$ is simple if both $v_i, v_k \in \Sigma_\tau$ and $v_{i+1}, \ldots, v_{k-1} \in \Sigma_\lambda$.*

Definition 44. *A **dangling path** is a path $(v_i, v_{i+1}, \ldots, v_j)$ such that either v_i is a terminal node and v_j is a branch node, or v_i is a branch node and v_j is a terminal node. Formally, a path $(v_i, v_{i+1}, \ldots, v_j)$ is dangling if either $(v_i \in \Sigma_\tau) \wedge (v_j \in \Sigma_\phi)$ or $(v_i \in \Sigma_\phi) \wedge (v_j \in \Sigma_\tau)$. In short, a dangling path either begins with a terminal node and ends with a branch node, or vice versa.*



	q_1	q_2	q_3	q_4
p_1		g		
p_2	r	r	g	
p_3				r
p_4				g

FIGURE 15. Nodes and Edges of Dangling and Simple Paths.

Example 17. Consider Figure 15. Processor p_2 is a branch node, while resources q_2 and q_4 are link nodes. Resources q_1 and q_3 and processors p_1, p_3, and p_4 are terminal nodes. There are three dangling paths: (q_3, p_2), (p_2, q_1), and (p_2, q_2, p_1). There is one simple path (p_3, q_4, p_4)

The reason for defining a dangling path will become clear later on when we use it in a proof.

Definition 45. *The* ***reachable*** *set Σ_{v_π} of a node v_π is the set of nodes $\{v_i \mid \exists(v_\pi, \ldots, v_i)\}$, which means a set of nodes such that there exists a path from v_π to node v_i.*

Example 18. In Figure 16 the reachable set Σ_{p_6} of node p_6 is $\{q_5, p_1, q_4, p_3, q_1, p_5, q_2\}$. Note that although q_3 is connected to the node p_5, nevertheless q_3 is not reachable from p_6.

Definition 46. *A* ***cycle*** *C is an ordered sequence of vertices $(v_1, v_2, \ldots, v_k)$, $k \geq 5$, such that v_1 and v_k are the same and no other in C are the same. Σ_c is the set of nodes involved in a path C.*

Example 19. In Figure 16, nodes q_4, q_1, p_3, and p_5 form a cycle C of $(q_4, p_3, q_1, p_5, q_4)$, starting and ending with node q_4. The set Σ_c is $\{q_4, p_3, q_1, p_5\}$.

Definition 47. *A state γ_{ij} of a system γ_i is said to be an* ***expedient*** *state if γ_{ij} does not contain any sink resources. In other words, there are no resources which are unallocated* ***and*** *have pending requests: thus, all resources are either isolated (no requests) or are granted to some processors. In an expedient system, all satisfiable requests are granted without delay. As soon as there is one request of a resource which is an isolated resource, that resource will be granted to the requesting processor and will become a source resource. If a resource has two requests, the resource will be granted to one processor and become a link resource; the processor which was not granted the resource has to wait. If a resource has more than two requests, the resource will be granted to one processor only and become a branch resource.*

Example 20. Figure 16 shows a given system γ_i in state γ_{ij_1}.

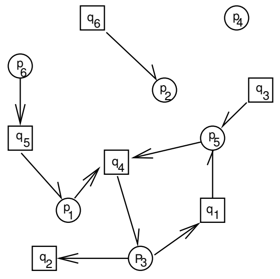
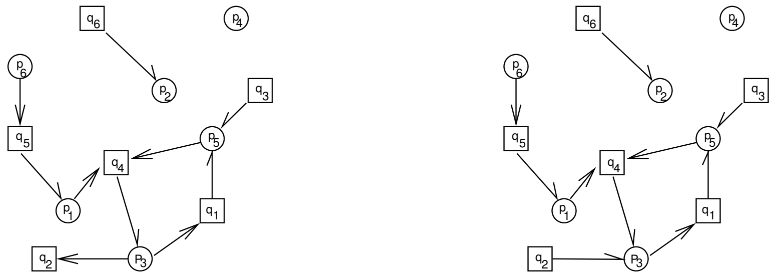


FIGURE 16. An Example of RAG with Request(s), Grant(s), and Cycle(s).

In Figure 16, system γ_i has processors p_1, p_2, p_3, p_4, p_5, and p_6. γ_i also has resources q_1, q_2, q_3, q_4, q_5, and q_6. State γ_{ij_1} is not an expedient state, because the satisfiable request (p_3, q_2) has not been granted yet. However γ_{ij_1} can be transformed into expedient state γ_{ij_2} when the request edge (p_3, q_2) is turned into a grant edge (q_2, p_3), which is possible since q_2 is not allocated to any processor. In state γ_{ij_1}, p_4 is an isolated processor node, while the rest of the processor and resource nodes are either link, sink or source nodes.

Example 21. In Figure 17, the graph γ_{ij_1} is transformed into an expedient graph γ_{ij_2} by changing $p_3 \rightarrow q_2$ into $q_2 \rightharpoonup p_3$. In Figure 17 (b), since p_2 and q_6 do not connect to the rest of the graph, the existence of path $q_6 \rightharpoonup p_2$ will not affect the other subgraphs. Thus $q_6 \rightharpoonup p_2$ can be safely ignored when searching for the deadlock condition.



(a) p_3 requests q_2 in γ_{ij_1}

(b) p_3 acquires q_2 in γ_{ij_2}

FIGURE 17. RAG Reduction of Figure 16.

Lemma 1. *The number of edges $\|E\|$ in a system γ_i is less than or equal $\|E\| \leq m \times n$,*

Proof: Since the graph γ_i is bipartite, each edge is permitted only to go from one set P of processor nodes to the other set Q of resource nodes or vice versa. A processor node p_i can have one edge to each resource, thus a processor node p_i can have at most $n = \|Q\|$ request edges $\{(p_i, q_j) \mid q_j \in Q, j = 1, 2, 3, \ldots, n\}$. Since there are $m = \|P\|$ processor nodes, the maximum number of request edges is $m \times n$. Since each grant edge replaces the corresponding request edge, the total number of request edges will be decreased by one whenever the total number of grant edges is increased by one. Thus, there is no change in the maximum total number of possible edges if the system is maintained in an expedient state. Overall, the total number of edges $\|E\|$ is less than $((m \times n - k) + k) = m \times n \mid k = 0, 1, 2, \ldots, n$, where k is the number of request edges that have been transformed into grant edges. Note that the maximum total number of grants at any given time is $k \leq n$, since at most all n resources can be granted.

The above property give us the approximate computational complexity of an algorithm based on either edges or nodes. An edge based algorithm has $O(e)$ run time complexity, where e is the number of edges. Also, the fact that there do not exist edges (p_i, p_j) from one processor to another and that there do not exist edges (q_i, q_j) from one resource to another is an important property for the hardware architecture. In short, a two dimensional adjacency matrix of edges is sufficient to represent all the possible different types of edges for the systems we consider. Such a two dimensional adjacency matrix provides an efficient hardware architecture implementation.

Therefore, OS routines and a RAG together can be used to model both system states and state transitions of an SoC or PCB with processors and resources. When a processor p_i makes a request for resource q_j, the OS inserts a request edge $r_{ij} = (p_i, q_j)$ into the edge set R. When a processor p_i obtains a resource q_j, the OS removes the request edge $r_{ij} = (p_i, q_j)$ from R and inserts a grant edge $g_{ji} = (q_j, p_i)$ in the set G. When a processor p_i releases a resource q_j, the OS removes the grant edge $g_{ji} = (q_j, p_i)$ from the set G.

3. Deadlock Properties

This section describes various properties and theorems relating to deadlock. For a system γ_{ij}, these properties and theorems can be applied to the matrix representation M_{ij} of γ_{ij}. In the systems we consider – reusable single-resource systems (defined in Section 1) – a cycle is a sufficient condition for deadlock[1]. Generally speaking, it is desirable to identify if a system state is deadlocked or not as soon as possible. Finding and constructing a cycle is not computationally efficient because the run time complexity of a cycle-search algorithm is similar to that of depth-first-search or breath-first-search. A technique based on a reduction sequence applied to the RAG of the system has been shown previously[1].

In general, any algorithm based on a RAG has a computational complexity in software of $O_{sw}(m \times n)$, where the "sw" in "O_{sw}" refers to the fact that the algorithm is run in software on a processor. We introduce a new technique – based on the notion of a *matrix reduction sequence* – which provides a better solution and can identify a deadlock state in linear time complexity in a hardware implementation.

Theorem 1. *A cycle is a necessary and sufficient condition for deadlock in a reusable single-resource expedient system with $m \geq 2$ requesters and $n \geq 2$ resources..*

Proof: The proof is available in Chapter 4 of Operating Systems - Advanced Concepts[1].

Example 22. A system with three processors and three resources is shown in Figure 18. The existence of a cycle C of path $(p_5, q_4, p_3, q_1, p_5)$ is necessary and sufficient to indicate a deadlocked system. The set of nodes in cycle C is $\Sigma_c = \{p_5, q_4, p_3, q_1\}$. Processor p_5 and resource q_4 are branch nodes. The grant edge (q_3, p_5) and request edge (p_2, q_4) are dangling paths. The reachable set of p_5 is $\Sigma_{p_5} = \{p_5, p_3, q_1, q_4\} = \Sigma_c$. The reachable sets Σ_{p_3}, Σ_{q_1}, and Σ_{q_4} are equal to Σ_c. The reachable set Σ_{p_2} is $\{\Sigma_c \cup p_2\}$ but processor p_2 is not part of cycle C. Processor p_2 is connected to the cycle C. The node q_3 has a reachable set of $\{\Sigma_c \cup q_3\}$ and resource q_3 is not part of cycle C either. Since both processor p_2 and resource q_3 are not part of the Σ_c, the existence of the cycle C is not affected by the absence of edges $p_2 \rightarrow q_4$ and $q_3 \rightharpoonup p_5$. In such a single-resource expedient system, a cycle is necessary and sufficient condition to identify a deadlocked state.

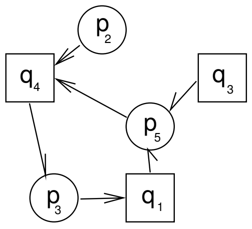


FIGURE 18. A Deadlock Cycle in RAG

Before formally defining the notion of a reduction step, we first give an informal description. One can consider a reduction step as emulating processor p_i either (i) releasing a held resource q_j, or (ii) releasing the resource q_j.

A processor p_i can complete its computation and then release all of the resources p_i holds only if the processor p_i has been granted access to all the resources p_i has requested. When a processor p_i does release all of the resources which p_i holds, the processor p_i is said to be **reduced**. Clearly, a reduced processor which makes no more requests cannot participate in a deadlock. When a granted resource q_j is released by the last requester p_i, that resource q_j is said to be reduced. Clearly, a reduced resource which receives no more requests also cannot participate in deadlock. This is why we are interested in formally defining the notion of a reduction step.

Definition 48. *A* ***sink reduction step*** $\boldsymbol{\delta_{sink}}$ *is a unary operator* $\delta_{sink} : \gamma_{ij} \mapsto \gamma_{i,j+1}$*, where* δ_{sink} *calculates the sink set* $\alpha(\gamma_{ij})$ *of* γ_{ij} *and returns* $\gamma_{i,j+1}$ *such that all sink edges* $E_{\alpha_{ij}}$ *found are removed and do not appear in* $\gamma_{i,j+1}$*. To determine the sink set,* δ_{sink} *uses Definition 25:* $\Sigma_{\alpha_{ij}} = \alpha(\gamma_{ij})$ *which returns a set sink nodes. Next, the sink reduction step* δ_{sink} *deletes all sink edges* $E_{\alpha_{ij}}$ *found connected to vertices in the sink set* $\alpha(\gamma_{ij})$*. Such deletions reduce the number of edges. The formula for* $\delta_{sink}(\gamma_{ij})$ *is shown in the second line of Equation 1:*

$$\begin{aligned} \gamma_{i,j+1} &= \delta_{sink}(\gamma_{ij}) \\ &= (V, (E(\gamma_{ij}) - E_{\alpha_{ij}})) \end{aligned} \tag{1}$$

The function $\delta_{sink}(\gamma_{ij})$ is called a sink reduction step because Equation 1 simply subtracts out (reduces) the set of sink edges. Note that the node set V is not changed. If the outgoing edge of a link node connects to a sink node in γ_{ij}, then that link node will become a sink node in $\gamma_{i,j+1} = \delta_{sink}(\gamma_{ij})$. If the outgoing edge of a branch node connects to a sink node, that branch node can become a link node when that edge is removed (if the degree of the branch node is three). If the outgoing edges of a branch node all connect to sink nodes, that branch node can become a sink node when those edges are removed. Thus, the removal of sink edges may make the set Σ_ϕ of connect nodes smaller and may create new sink nodes. When δ_{sink} is applied again to the next system state $\gamma_{i,j+1}$, any sink edges in $\gamma_{i,j+1}$ must have been derived from the set of connect edges in γ_{ij}.

Definition 49. *If a system state* γ_{ij} *can be transformed by a reduction step to another state* $\gamma_{i,j+1}$ *resulting in* $\gamma_{i,j+1} \neq \gamma_{ij}$*, then the system state* γ_{ij} *is said to be* ***reducible****. If a system state* γ_{ij} *cannot be reduced to another different state* $\gamma_{i,j+1}$ *(because the resulting* $\gamma_{i,j+1}$ *is equal to* γ_{ij}*), then system state* γ_{ij} *is said to be* ***irreducible****.*

Example 23. Assume we are given a system state $\gamma_{ij} = (V, E)$, as shown in Figure 19(a), where $V = \{P \cup Q\}$ and $E = \{R \cup G\}$:

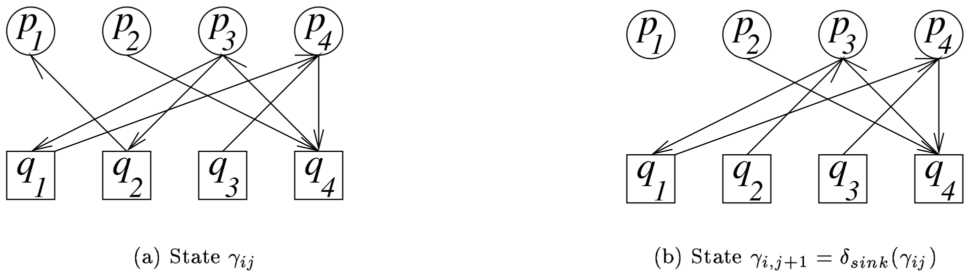


FIGURE 19. Apply a Sink Reduction Step δ_{sink} to State γ_{ij}.

$P = \{p_1, p_2, p_3, p_4\}$ and $Q = \{q_1, q_2, q_3, q_4\}$.

$R = \{(p_2, q_4), (p_4, q_4), (p_3, q_1), (p_3, q_2)\}$ and $G = \{(q_2, p_1), (q_1, p_4), (q_4, p_3), (q_3, p_4)\}$.

$\Sigma_{\alpha_{ij}} = \alpha(\gamma_{ij}) = \{p_1\}$ since p_1 is a sink processor node.

$E_{\alpha_{ij}} = \{(q_2, p_1)\}$ because there is only one edge into $p_1 = \alpha(\gamma_{ij})$.

Resource q_2 is a link node in γ_{ij} while q_2 is a sink node in $\gamma_{i,j+1}$:

$$\begin{aligned} \gamma_{i,j+1} &= \delta_{sink}(\gamma_{ij}) \\ &= (V, E - E_{\alpha_{ij}}) \\ &= (V, \{R \cup G\} - \{(q_2, p_1)\}) \\ &= (V, \{(p_2, q_4), (p_4, q_4), (p_3, q_1), (p_3, q_2)\} \cup \{(q_1, p_4), (q_4, p_3), (q_3, p_4)\}) \end{aligned} \tag{2}$$

The result, $\gamma_{i,j+1}$, has a smaller set E because the sink reduction step δ_{sink} removed an edge, as can be seen in Figure 19(b). Note that in state γ_{ij} there are no isolated nodes, while in state $\gamma_{i,j+1} = \delta_{\text{sink}}(\gamma_{ij})$ the set of isolated nodes is $\{p_1\}$.

Definition 50. *A **sink reduction sequence** Δ_{sink} is defined as a finite sequence of sink reduction steps δ_{sink}, such that (i) $\gamma_{ij} \mapsto \gamma_{i,j+1} \mapsto \cdots \mapsto \gamma_{i,j+k}$; (ii) $\gamma_{i,j+k}$ is irreducible; and (iii) $\{\gamma_{i,j+h}, 0 \leq h < k\}$ are all reducible. An equation expression of sink reduction sequence Δ_{sink} is $\gamma_{i,j+k} = \Delta_{sink}(\gamma_{ij}) = \delta_{sink}(\cdots \delta_{sink}(\delta_{sink}(\gamma_{ij}))\cdots)$, where the sink reduction step δ_{sink} is applied recursively $k \geq 0$ times until $\gamma_{i,j+k}$ is irreducible. The formula $\Delta_{sink}(\gamma_{ij})$ is shown in Algorithm 1.*

Algorithm 1. *Sink Reduction Sequence Algorithm*

```
1   $\Delta_{sink}$ $(\gamma_{ij})$ {
2       $k = 0$;
3       $\gamma_{iterate} = \gamma_{ij}$;
4       while $(\alpha(\gamma_{iterate}) \neq \emptyset)$ {
5           $k = k + 1$;
6           $\gamma_{temp} = \delta_{sink}(\gamma_{iterate})$
                $= (V(\gamma_{iterate}), E(\gamma_{iterate}) - E_{\alpha_{iterate}})$;
7           $\gamma_{iterate} = \gamma_{temp}$;
8       }
9       $\gamma_{i,j+k} = \gamma_{iterate}$;
10      return $\gamma_{i,j+k}$;
11  }
```

The right hand side of line 6 in Algorithm 1 is the core of the algorithm: $\delta_{sink}(\gamma_{\text{iterate}})$ is calculated recursively on itself until there are no more sink edges left in γ_{iterate}. Equation 3 below shows another way of expressing the execution of Algorithm 1 to perform a sink reduction sequence, where each γ_{iterate} is replaced with $\gamma_{\text{i,j+h}}$ where h has the appropriate value corresponding to the algorithm iteration step:

$$\begin{array}{llcl} \text{if } \alpha(\gamma_{ij}) \neq \{\emptyset\}, & \gamma_{i,j+1} & = & \delta_{sink}(\gamma_{ij}) = (V, E(\gamma_{ij}) - E_{\alpha_{ij}}) \\ \text{if } \alpha(\gamma_{i,j+1}) \neq \{\emptyset\}, & \gamma_{i,j+2} & = & \delta_{sink}(\gamma_{i,j+1}) = (V, E(\gamma_{i,j+1}) - E_{\alpha_{i,j+1}}) \\ \vdots & \vdots & & \\ \text{if } \alpha(\gamma_{i,j+k-1}) \neq \{\emptyset\}, & \gamma_{i,j+k} & = & \delta_{sink}(\gamma_{i,j+k-1}) = (V, E(\gamma_{i,j+k-1}) - E_{\alpha_{i,j+k-1}}) \\ \alpha(\gamma_{i,j+k}) = \{\emptyset\}. & & & \end{array} \tag{3}$$

Definition 51. *A system state $\gamma_{i,j+k}$ is said to be* ***completely reduced*** *if $E(\gamma_{i,j+k}) = \emptyset$. Otherwise, a system state $\gamma_{i,j+k}$ is said to be* ***incompletely reduced*** *if $E(\gamma_{i,j+k}) \neq \emptyset$.*

Example 24. Let us apply the sink reduction sequence Δ_{sink} to two examples. We will consider a deadlocked example first and then a non-deadlocked case.

For the first example, γ_{ij} is the same as in Example 23: $\gamma_{ij} = (V, E) = (\{P \cup Q\}, \{R \cup G\})$ as shown in Figure 19.

$P = \{p_1, p_2, p_3, p_4\}$ and $Q = \{q_1, q_2, q_3, q_4\}$.

For the request set, we have $R = \{(p_2, q_4), (p_4, q_4), (p_3, q_1), (p_3, q_2)\}$,

For the grant set, we have $G = \{(q_2, p_1), (q_1, p_4), (q_4, p_3), (q_3, p_4), (q_3, p_4)\}$.

The result is shown as follows:

$$\begin{aligned}
\alpha(\gamma_{ij}) &= \{p_1\} \\
E_{\alpha_{ij}} &= \{(q_2, p_1)\} \\
\gamma_{i,j+1} &= (V, \{(p_2, q_4), (p_4, q_4), (p_3, q_1), (p_3, q_2)\} \cup \{(q_2, p_1), (q_1, p_4), (q_4, p_3)\} - \{(q_2, p_1)\}) \\
&= (V, \{(p_2, q_4), (p_4, q_4), (p_3, q_1), (p_3, q_2)\} \cup \{(q_1, p_4), (q_4, p_3)\})
\end{aligned} \tag{4}$$

$$\alpha(\gamma_{i,j+1}) = \{\emptyset\}$$

Therefore, we stop and find that k = 1. Note that the result $\gamma_{i,j+1}$ is incompletely reduced since $E(\gamma_{i,j+1}) \neq \emptyset$.

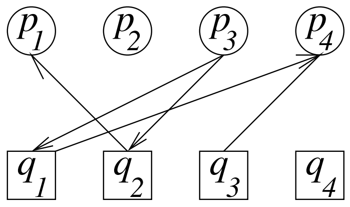


FIGURE 20. Apply Sink Reduction Sequence Δ_{sink} to State γ_{ij}.

For the second example shown in Figure 20, the $\gamma_{ij} = (V, E) = (\{P \cup Q\}, \{R \cup G\})$ is given as follows:

$P = \{p_1, p_2, p_3, p_4\}$ and $Q = \{q_1, q_2, q_3, q_4\}$.

$R = \{(p_3, q_1), (p_3, q_2)\}$ and $G = \{(q_2, p_1), (q_1, p_4), (q_3, p_4)\}$.

The result of the first sink reduction step is shown as follows:

$$\begin{aligned}
\alpha(\gamma_{ij}) &= \{p_1, p_4\} \\
E_{\alpha_{ij}} &= \{(q_2, p_1), (q_1, p_4), (q_3, p_4)\} \\
\gamma_{i,j+1} &= (V, \{(p_3, q_1), (p_3, q_2), (q_2, p_1), (q_1, p_4), (q_3, p_4)\} - \{q_2, p_1), (q_1, p_4), (q_3, p_4)) \\
\gamma_{i,j+1} &= (V, \{(p_3, q_1), (p_3, q_2)\})
\end{aligned} \tag{5}$$

Figure 21 shows the result after one sink reduction step. Resources q_1 and q_2 have now become sink nodes. Therefore, they are removed by the next sink reduction step:

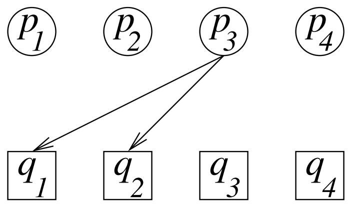


FIGURE 21. After one Sink Reduction Step δ_{sink}

$$\begin{aligned}
\alpha(\gamma_{i,j+1}) &= \{q_1, q_2\} \\
E_{\alpha_{i,j+1}} &= \{(p_3, q_1), (p_3, q_2)\} \\
\gamma_{i,j+2} &= (V, \{(p_3, q_1), (p_3, q_2)\} - \{(p_3, q_1), (p_3, q_2)\}) \\
\gamma_{i,j+2} &= (V, \emptyset)
\end{aligned} \tag{6}$$

$$\alpha(\gamma_{i,j+2}) = \emptyset$$

We stop and find that k = 2. Note that the result $\gamma_{i,j+2}$ is completely reduced since $E(\gamma_{i,j+2}) = \emptyset$. Also note that the second sink reduction step assumes that resources q_1 and q_2 were both granted to and then released by processor p_3.

A sink reduction step δ_{sink} of a state (thus removing edges to sink nodes) might unblock a waiting processor, e.g., as happened to processor p_3 in the example above. Depending on which order requests are turned into grants, in fact deadlock could arise. Such an occurrence is equivalent to a safe state becoming a deadlock state (see Section 2.2). The sink reduction step as we have defined it takes an optimistic view about the future behavior of a system in state γ_{ij}; in other words, if a safe sequence of requests and grants that keeps the system γ_i out of deadlock exists, then the sink reduction step assumes that this safe sequence is in fact the sequence that will be chosen.

Theorem 2. *A processor p_i is not part of a deadlock cycle in state γ_{ij} iff there exists a sequence of sink reduction steps in γ_{ij} which allows processor p_i to be able to acquire all requested resources.*

Proof: The proof is also available in Chapter 4 of Operating Systems - Advanced Concepts[1].

Lemma 2. *A cycle C in system state γ_{ij} must contain alternating resource nodes and processor nodes.*

Proof: By Definition 6, γ_{ij} cannot have any edge from processor p_i to processor p_j for any two processors p_i, $p_j \in V(\gamma_{ij})$. Similarly, by Definition 6, γ_{ij} cannot have any edge from resource q_i to resource q_j for any two resources q_i, $q_j \in V(\gamma_{ij})$. Therefore, since cycle C is composed of edges, any cycle C must contain alternating resource nodes and processor nodes connected by edged.

Lemma 3. *In a system state γ_{ij}, the number of edges in a cycle must be a multiple of 2.*

Proof: This lemma follows trivially from Lemma 2 and the definition of a cycle, Definition 46.

Theorem 3. *Let γ_{ij} be an expedient state of a reusable resource system γ_i. Sink reduction sequence $\Delta_{sink}(\gamma_{ij})$ reduces the state γ_{ij} to a state $\gamma_{i,j+k}$ which is irreducible. State γ_{ij} is not a deadlock state if and only if $\gamma_{i,j+k}$ is completely reduced ($E(\gamma_{i,j+k}) = \emptyset$).*

Proof: The proof is available as Theorem 4.3 in Chapter 4 of Operating Systems – Advanced Concepts[1].

By Theorem 3, if the state γ_{ij} is expedient, then detection of a cycle in γ_{ij} will be a sufficient condition for determining that γ_{ij} is a deadlock state.

Corollary 1. *If state $\gamma_{i,j+k} = \Delta_{sink}(\gamma_{ij})$ is completely reduced ($E(\gamma_{i,j+k}) = \emptyset$), then γ_{ij} is not a deadlock state.*

Example 25. In Figure 22 (a), the system state is expedient. The set of sink nodes is $\Sigma_\alpha = \{p_2\}$. The set of link nodes is $\Sigma_\lambda = \{q_5, p_1, q_4, p_5, q_1, p_3\}$, while the set of source nodes is $\Sigma_\beta = \{q_3, q_6, q_2\}$. Therefore, a sink reduction step δ_{sink} removes an edge (q_6, p_2) incident to the sink node p_2. The reduced state is shown in Figure 22 (b). At that time, the sink set $\Sigma_\alpha = \{\emptyset\}$, and hence the state is irreducible. Since the state in Figure 22 (b) contains non-empty set of connect nodes, $\Sigma_\tau \neq \emptyset$, the original state shown in Figure 22 (a) is deadlocked. Note that Figure 22 (b) contains two dangling paths which do not participate in the deadlock cycle C.

Algorithm 2. *Deadlock Detection Algorithm Reducing Sink Nodes*
```
Deadlock_Detect_Sink ($\gamma_{ij}$) {
    $\gamma_{i,j+k} = \Delta_{sink}(\gamma_{ij})$;
    if ($E(\gamma_{i,j+k}) = \emptyset$) {
        return 0; /* no deadlock */
    } else {
        return 1; /* deadlock detected */
    }
}
```

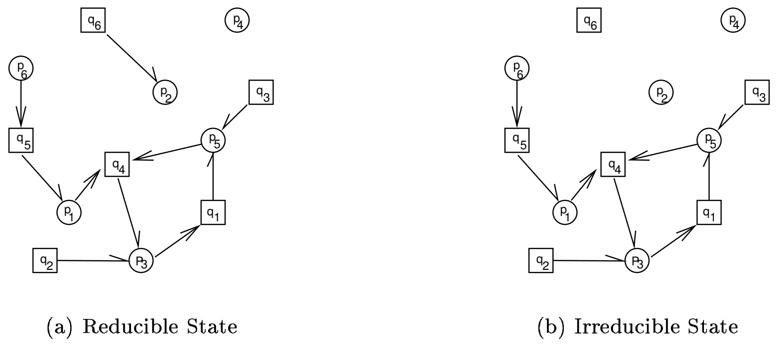


FIGURE 22. A System state that is Irreducible.

Algorithms 1 and 2 represent the classical solution for deadlock detection [2, 3, 1]. This solution and all other solutions known to the author take time $O_{sw}(m \times n)$, where the "sw" in "O_{sw}" refers to the fact that the algorithm is run in software on a processor. Although other authors give their algorithm complexity in terms of $e = \|E\|$, $e \leq m \times n$ so that $O_{sw}(e) = O_{sw}(m \times n)$ [4, 6].

4. Equivalent Deadlock Detection Theorem

In this section we will define a new algorithm for deadlock detection with $O_{sw}(m \times n)$. The advantage of this new algorithm will not be seen until the next section when we show how to implement the algorithm in a matrix form. This matrix form can be implemented in hardware yielding complexity $O_{hw}(min(m,n))$, where the "hw" in "O_{hw}" refers to the fact that the algorithm is run in a special hardware configuration (to be explained in detail in Section 6).

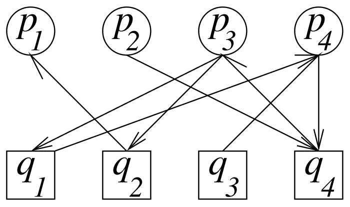
For the proposed algorithm, Definitions 48 and 50 yield insight into how to check in parallel if a state γ_{ij} is a deadlock state or not. The insight is that by recursively removing sink nodes (Algorithm 1), we can determine if a cycle exists. From this insight, we will expand the notion of a reduction step to include the removal of source nodes as well as sink nodes. Then we will prove that this does not alter the resulting deadlock detection properties.

Definition 52. *A* ***reduction step δ*** *is a unary operator $\delta : \gamma_{ij} \mapsto \gamma_{i,j+1}$, where δ calculates the terminal edge set $\tau(\gamma_{ij})$ of γ_{ij} and returns $\gamma_{i,j+1}$ such that all terminal edges $E_{\tau_{ij}}$ found are removed and do not appear in $\gamma_{i,j+1}$. To determine the terminal set, δ uses Definition 31: $\Sigma_{\tau_{ij}} = \tau(\gamma_{ij})$ which returns a set of edges connected to terminal (sink or source) nodes. Next, the reduction step δ deletes all terminal* *edges found in the terminal set $\alpha(\gamma_{ij})$. Such deletions reduce the number of edges. The formula for $\delta(\gamma_{ij})$ is shown in the second line of Equation 7:*

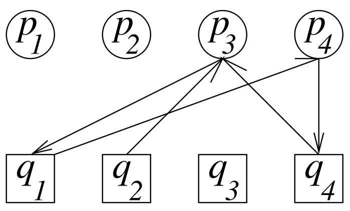
$$\begin{aligned} \gamma_{i,j+1} &= \delta(\gamma_{ij}) \\ &= (V, E(\gamma_{ij}) - E_{\tau_{ij}}) \end{aligned} \tag{7}$$

The function $\delta(\gamma_{ij})$ simply subtracts out (reduces) the set of terminal edges. The node set V is not changed. If only one edge of a link node connects to a terminal node in γ_{ij}, then that link node will become a terminal node in $\gamma_{i,j+1} = \delta(\gamma_{ij})$. If both edges of a link node connect to terminal nodes in γ_{ij}, then the link node will become an isolated node in $\gamma_{i,j+1}$. If edges of a branch node connect to a terminal node γ_{ij}, then, in $\gamma_{i,j+1}$, the branch node will either (i) remain a branch node, (ii) become a link node, (iii) become a terminal node, or (iv) become an isolated node. Thus, all connect nodes (link and branch nodes) either stay the same, convert to a different connect node, or become reduced to a terminal node or an isolated node. Thus, the removal of edges to terminal nodes can only leave the set Σ_ϕ of connect nodes the same size or smaller. When δ is applied again to the next system state $\gamma_{i,j+1}$, any edges to terminal nodes $((v_x, v_y) \in E_{\tau_{i,j+1}})$ in $\gamma_{i,j+1}$ must have been derived from the set of edges to connect nodes $((v_x, v_y) \in E_{\phi_{ij}})$ in γ_{ij}.

Example 26. This example starts with the same system state $\gamma_{ij} = (V, E)$ as in Example 23; the system is redrawn in Figure 23(a), where $V = \{P \cup Q\}$ and $E = \{R \cup G\}$:



(a) State γ_{ij}



(b) State $\gamma_{i,j+1} = \delta(\gamma_{ij})$

FIGURE 23. Apply a Reduction Step δ to State γ_{ij}.

$R = \{(p_2, q_4), (p_4, q_4), (p_3, q_1), (p_3, q_2)\}$ and $G = \{(q_2, p_1), (q_1, p_4), (q_4, p_3), (q_3, p_4)\}$.

$\Sigma_{\tau_{ij}} = \tau(\gamma_{ij}) = \{p_1, p_2, q_3\}$.

$E_{\tau_{ij}} = \{(q_2, p_1), (p_2, q_4), (q_3, p_4)\}$.

$$\begin{aligned} \gamma_{i,j+1} &= \delta(\gamma_{ij}) \\ &= (V, E - E_{\tau_{ij}}) \\ &= (V, \{R \cup G\} - \{(q_2, p_1), (p_2, q_4), (q_3, p_4)\}) \\ &= (V, \{(p_4, q_4), (p_3, q_1), (p_3, q_2)\} \cup \{(q_1, p_4), (q_4, p_3)\}) \end{aligned} \tag{8}$$

The result, $\gamma_{i,j+1}$, has a smaller set E because the reduction step δ removed three edges, as can be seen in Figure 23(b). Note that in state γ_{ij} there are no isolated nodes, while in state $\gamma_{i,j+1} = \delta_{sink}(\gamma_{ij})$ the set of isolated nodes is $\{p_1, p_2, q_3\}$.

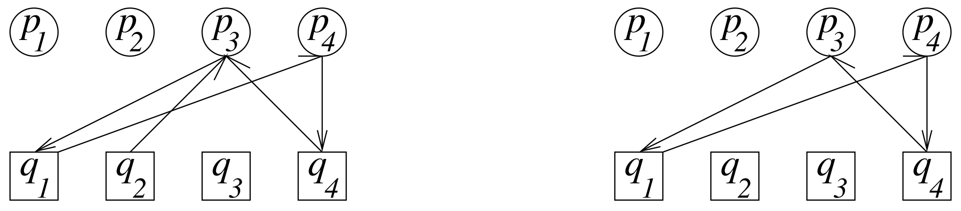
Definition 53. *A* ***reduction sequence*** Δ *is defined as a finite sequence of reduction steps δ, such that (i) $\gamma_{ij} \mapsto \gamma_{i,j+1} \mapsto \cdots \mapsto \gamma_{i,j+k}$; (ii) $\gamma_{i,j+k}$ is irreducible; and (iii) $\{\gamma_{i,j+h}, 0 \leq h < k\}$ are all reducible. An equation expression of a reduction sequence Δ is $\gamma_{i,j+k} = \Delta(\gamma_{ij}) = \delta(\cdots\delta(\delta(\gamma_{ij}))\cdots)$, where the reduction step δ is applied recursively $k \geq 0$ times until $\gamma_{i,j+k}$ is irreducible. The formula $\Delta(\gamma_{ij})$ is shown in Algorithm 3.*

Algorithm 3. *Reduction Sequence Algorithm*

```
1   $\Delta(\gamma_{ij})$ {
2       $k = 0$;
3       $\gamma_{iterate} = \gamma_{ij}$;
4       while ($\tau(\gamma_{iterate}) \neq \emptyset$) {
5           $k = k + 1$;
6           $\gamma_{temp} = \delta(\gamma_{iterate})$
                  $= (V(\gamma_{iterate}), E(\gamma_{iterate}) - E_{\tau_{iterate}})$;
7           $\gamma_{iterate} = \gamma_{temp}$;
8       }
9       $\gamma_{i,j+k} = \gamma_{iterate}$;
10      return $\gamma_{i,j+k}$;
11  }
```

The right hand side of line 6 in Algorithm 3 is the core of the algorithm: $\delta(\gamma_{\text{iterate}})$ is calculated recursively on itself until there are no more terminal edges left in γ_{iterate}.

Example 27. Let us apply the reduction sequence Δ to the previous example (Example 26). The first application of δ is exactly as shown in Example 26 and results in $\gamma_{i,j+1}$ as shown in Figure 24. Therefore, we show the second application of δ.



(a) State $\gamma_{i,j+1}$ (b) State $\gamma_{i,j+2} = \delta(\gamma_{i,j+1})$

FIGURE 24. Apply Reduction Operator Δ

The result is as follows (and is shown in the graph of $\gamma_{i,j+2}$ on the right hand side of Figure 24):

$$\begin{aligned}
\Sigma_{\tau_{i,j+1}} &= \tau(\gamma_{i,j+1}) = \{q_2\} \\
E_{\tau_{i,j+1}} &= \{(q_2, p_3)\} \\
\gamma_{i,j+2} &= \delta(\gamma_{i,j+1}) \\
&= (V, E(\gamma_{i,j+1}) - E_{\tau_{i,j+1}}) \\
&= (V, \{(p_4, q_4), (p_3, q_1), (q_1, p_4), (q_4, p_3)\})
\end{aligned} \tag{9}$$

$$\Sigma_{\tau_{i,j+2}} = \tau(\gamma_{i,j+2}) = \{\emptyset\}$$

Therefore, we stop and find that k = 2. Note that the result $\gamma_{i,j+2}$ is incompletely reduced since $E(\gamma_{i,j+2}) \neq \emptyset$.

Algorithm 4. *Deadlock Detection Algorithm Reducing Sink and Source Nodes*

```
Deadlock_Detect ($\gamma_{ij}$) {
    $\gamma_{i,j+k} = \Delta(\gamma_{ij})$; /* call Algorithm 3 */
    if ($E(\gamma_{i,j+k}) = \emptyset$) {
        return 0; /* no deadlock */
    } else {
        return 1; /* deadlock detected */
    }
}
```

Algorithms 3 and 4 represent a slightly modified version of the classical solution for deadlock detection explain in Section 3. We turn now to formally proving that the algorithm finds all deadlock conditions accurately.

Lemma 4. *Given system γ_i in state γ_{ij} without any cycles, all nodes $v_\pi \in V$ must have the property that the reachable set Σ_{v_π} of node v_π cannot include v_π.*

Proof: If v_π were reachable from itself, it would form a cycle C.

Lemma 5. *Given system γ_i in state γ_{ij} with $E \geq 1$ and without any cycles, there must exist at least one path $(v_1, v_2, v_3, \ldots, v_{k-1}, v_k)$ with both v_1 and v_k terminal nodes.*

Proof: We will prove this lemma by contradiction. Suppose that state γ_{ij} with $E \geq 1$ and without any cycles does not have any path $(v_1, v_2, v_3, \ldots, v_{k-1}, v_k)$ with both v_1 and v_k terminal nodes. Choose any path $(v_1, v_2, v_3, \ldots, v_{k-1}, v_k)$ (since $E \geq 1$ we know that there exists at least one path). Then either v_1 or v_k is a connect node. Suppose, without loss of generality, that v_k is a connect node. Then $(v_1, v_2, v_3, \ldots, v_{k-1}, v_k, v_{k+1})$ is a path. Again, either v_1 or v_{k+1} is a connect node. Continuing in this way, we will either find a path with both its first and last nodes as terminal nodes, or else we will come to a path $(v_1, v_2, v_3, \ldots, v_{V-1}, v_V)$. If we find a path with both its first and last nodes being terminal nodes, then we have a contradiction and are finished with the proof. However, on the other hand, if we do not find a path with both its first and last nodes being terminal nodes, then we will eventually come to path $(v_1, v_2, v_3, \ldots, v_{V-1}, v_V)$ without both v_1 and v_V terminal nodes. Without loss of generality, assume that v_V is a connect node. Since all nodes have been used already in path $(v_1, v_2, v_3, \ldots, v_{V-1}, v_V)$, and since all connect nodes have at least two edges, v_V has an edge to v_h where $v_h \neq v_{V-1}$. In this case, then, there exists a cycle beginning and ending with v_h, which contradicts our assumption that γ_{ij} does not have any cycles, and we are done. QED.

Lemma 6. *Given system γ_i in state γ_{ij} with a cycle C (i.e., γ_{ij} is a deadlock state), removing edges connected to terminal nodes will not alter the cycle C.*

Proof: Every node in cycle C is a connect node. Furthermore, every node in cycle C must have an edge **to** another node in the cycle and **from** another node in the cycle. Therefore, if a node in cycle C has an edge to a terminal node, the terminal node cannot be in the cycle. Thus, removal of the edge to the terminal node leaves the cycle edges intact, since none of the edges to other nodes in cycle C are edges to terminal nodes.

Theorem 4. *Algorithms 3 and 4 detect deadlock in state γ_{ij} iff there exists a cycle in γ_{ij}.*

Proof: First let us prove the "if" part of Theorem 4: for the "if" part of this proof, we are given that there exists a cycle C in state γ_{ij}. By Lemma 6, each application of δ does not remove cycle C. Since there are a finite number of nodes in γ_{ij}, at some point in the algorithm no more terminal nodes will be found. By Lemma 6, the cycle C is left intact, and so $E(\gamma_{i,j+k}) \neq \emptyset$ since there are edges in cycle C. The algorithm reports a deadlock state, which, by Theorem 1 is correct because γ_{ij} has a cycle.

Now for the "only if" part of the proof: for the "only if" part of the proof, we are given that Algorithms 3 and 4 detected deadlock given state γ_{ij} as input. In order for deadlock to have been detected, it must be the case that $\gamma_{i,j+k} = \Delta(\gamma_{ij})$ has a least one edge so that we have $E(\gamma_{i,j+k}) \neq \emptyset$. Note that Algorithm 4 always executes $\gamma_{i,j+k} = \Delta(\gamma_{ij})$. Now, note that Algorithm 3 (to calculate $\Delta(\gamma_{ij})$), cannot exit until line 4 results in $\tau(\gamma_{iterate}) = \emptyset$ (where $\gamma_{iterate} = \gamma_{i,j+k}$ for some value of k). The two possible cases are (i) there is no cycle C or (ii) there is a cycle C. Suppose we have case (i): in this case, by Lemma 5, as long as $E \geq 1$ we will continue to find a path with terminal nodes. Eventually, then, since the number of edges is finite, we will find that $E < 1$ which implies that we much have $E = 0$. In this case $E(\gamma_{i,j+k}) = \emptyset$ and the algorithm would indicate that there is **not** a deadlock. By Theorem 1 this result is correct since γ_{ij} does not have a cycle. Now suppose we have case (ii): there is a cycle C. In this case (ii), by Lemma 6, the cycle C is never affected. Therefore, since the number of edges is finite, eventually no more terminal nodes are left and line 4 of Algorithm 3 results in $\tau(\gamma_{iterate}) = \emptyset$ causing an exit with $\gamma_{i,j+k} = \Delta(\gamma_{ij})$ as a return value. In this case, since there are still edges in $\gamma_{i,j+k}$ due to the connect nodes of cycle C, $\|E(\gamma_{i,j+k})\| \neq 0$. Thus $E(\gamma_{i,j+k}) \neq \emptyset$ causing the algorithm to indicated that a deadlock has been detected. This proves the "only if" part of our proof since deadlock is detected by the algorithm only when a cycle C exists. QED.

Theorem 5. *Algorithms 3 and 4 detect deadlock in state γ_{ij} iff there exists a deadlock in γ_{ij}.*

Proof: By Theorem 4, algorithms 3 and 4 detect deadlock in state γ_{ij} iff there exists a cycle in γ_{ij}. By Theorem 1, this is equivalent to saying that algorithms 3 and 4 detect deadlock in state γ_{ij} iff there exists a deadlock in γ_{ij}. QED.

Lemma 7. *Given system γ_i in state γ_{ij} with $P = m$ and $Q = n$, the maximum number of nodes in any path is $2 \times min(m, n) + 1$.*

Proof: We will prove this by construction. By Definition 42, a path $(v_1, v_2, v_3, \ldots, v_{k-1}, v_k)$, $k \geq 2$, is a consecutive ordered sequence of alternating request and grant edges $\{ (v_1, v_2), (v_2, v_3), \ldots, (v_{k-2}, v_{k-1}), (v_{k-1}, v_k) \}$ where every vertex with an odd index $(v_1, v_3, \ldots)$ is a member of the same set (P or Q) and every vertex with an even index $(v_2, v_4, \ldots)$ is a member of the other set (P or Q). Clearly, the smaller set (P or Q) should have all of its nodes involved in the maximum sized path in γ_{ij}. Due to the required sequence of alternating requests and grants, the larger set can at most have one more node in the path than the smaller set. Thus, the maximum number of vertices in a path is twice $min(m, n)$ plus one: $2 \times min(m, n) + 1$. QED

5. Parallel Deadlock Detection Theorem

So far we have covered deadlock detection using a graph model and have proposed a parallel algorithm that reduces the run-time complexity in a parallel implementation. In this section we show how to implement the proposed parallel algorithm in custom hardware. Similar to edge labeling[4], a RAG is mapped into modified adjacency matrix edge by edge. Since a parallel algorithm can perform multiple operations at the same time, the parallel run-time complexity is thus reduced. In the proposed algorithm, the reduction step δ borrows some ideas from Quine-McCluskey's prime implicant chart.

The four necessary deadlock conditions are also mapped into the matrix representation. Only the processor p_i which has been granted a resource q_j can release q_j – in other words, holding of resources is non-preemptive. If a processor p_i requests resource q_j, then a request r_{ij} is recorded in matrix M. If a processor p_i is granted a resource q_j, then the corresponding r_{ij} is transformed into a grant g_{ij}. A release of resource q_j by processor p_i transforms the grant g_{ij} into an empty edge. Resourceq_j can only be granted to one processor p_i at any time. Thus, since each column in M corresponds to a unique resource, there is at most one g in any column in M at any given time. Another processor p_k which requests q_j has to wait for the resource q_j to be released by the owner p_i. Furthermore, since each row in M corresponds to a unique processor, a row containing both r and g forms a hold-and-wait condition in an expedient system state (which means that all satisfiable requests have been granted). We say that such a row has a **horizontal link edge**. This is because the processor corresponding to the row is holding the resource corresponding to the g column while waiting for the resource corresponding to the r column (which, since the system state is expedient, is granted to another processor). Similarly, if a column has both an r and a g, then another hold-and-wait condition is also formed. We say that such a column has a **vertical link edge**. Thus, in a given M_{ij}, if we can find a circular path (circular hold-and-wait condition) formed by alternative horizontal and vertical link edges, then a cycle is formed and a given system state γ_{ij} is in deadlock.

This section proposes an implementation of the Algorithms 3 and 4 where state γ_{ij} is represented by the matrix M_{ij} corresponding to state γ_{ij}. The matrix representation will enable a custom hardware implementation which perform each iteration of Algorithm 3 very fast.

Definition 54. *A* ***matrix reduction step*** $\boldsymbol{\delta_M}$ *is a unary operator $\delta_M : M_{ij} \mapsto M_{i,j+1}$, where δ_M calculates the terminal edge set $\tau(M_{ij}) = \tau(\gamma_{ij})$ and returns $M_{i,j+1}$ such that all terminal edges $\tau(M_{ij})$ found are removed and do not appear in $M_{i,j+1}$.*

To determine the set of connect nodes, δ_M uses Definition 31: $\Sigma_{\tau_{ij}} = \tau(\gamma_{ij}) = \tau(M_{ij})$ which returns a set of edges connected to terminal (sink or source) nodes. Next, the matrix reduction step δ_M deletes all terminal edges found in the terminal set $\tau(M_{ij})$ by removing corresponding entries in the matrix M_{ij}. Such deletions reduce the number of entries in M_{ij}. The formula for $\delta_M(M_{ij})$ is shown in the third line of Equation 10 (see Definition 11 for what we mean by $M_{i,j+1} = \gamma_{i,j+1}$):

$$\begin{aligned} M_{i,j+1} &= \delta_M(M_{ij}) \\ &= \delta_M(\gamma_{ij}) \\ &= (V, E(\gamma_{ij}) - \tau(\gamma_{ij})) \\ &= \gamma_{i,j+1} \end{aligned} \tag{10}$$

Definition 54 is completely analogous to Definition 52. The only difference is that in Definition 54 the operations are on matrix M_{ij} instead of directly on γ_{ij}.

Definition 55. *A matrix M_{ij} is* **reducible** *if γ_{ij} corresponding to M_{ij} is reducible (Definition 49). Similarly, a matrix M_{ij} is* **irreducible** *if γ_{ij} corresponding to M_{ij} is irreducible.*

Definition 56. *A* ***matrix reduction sequence*** $\boldsymbol{\Delta_M}$ *is defined analogously to* ***reduction sequence*** $\boldsymbol{\Delta}$ *(Definition 53): a* ***matrix reduction sequence*** $\boldsymbol{\Delta_M}$ *is a sequence of reduction steps δ_M such that (i) $M_{ij} \mapsto M_{i,j+1} \mapsto \cdots \mapsto M_{i,j+k}$; (ii) $M_{i,j+k}$ is irreducible; and (iii) $\{M_{i,j+h}, 0 \le h < k\}$ are all reducible. Another representation of a matrix reduction sequence is $M_{i,j+k} = \delta_M(\cdots \delta_M(M_{i,j+1}\delta_M(M_{ij}))\cdots))$. A matrix reduction sequence is called a* ***complete matrix reduction*** *when the sequence of matrix reduction* *steps corresponding to Δ_M results in $M_{i,j+k}$ such that the irreducible state matrix $M_{i,j+k}$ contains all zero entries (note that this means that $\gamma_{i,j+k}$ corresponding to $M_{i,j+k}$ has no edges: $E(\gamma_{i,j+k}) = \{\emptyset\}$). A matrix reduction sequence is called an* ***incomplete matrix reduction*** *when Δ_M returns $M_{i,j+k}$ with at least one non-zero entry (note that this means that $\gamma_{i,j+k}$ corresponding to $M_{i,j+k}$ has at least one edge: $E(\gamma_{i,j+k}) \neq \{\emptyset\}$). The formula for $\Delta_M(M_{ij})$ is shown in the second line of Equation 11:*

$$\begin{aligned} M_{i,j+k} &= \Delta_M(M_{ij}) \\ &= \delta_M(\ldots\delta_M(\delta_M(M_{ij}))\ldots) \end{aligned} \tag{11}$$

Example 28. In Figure 25, we start with γ_{ij} shown in Figure 25 (a). γ_{ij} is reduced by the parallel deadlock detection algorithm defined by Definition 56. In step 1.1, Figure 25 (a) has $q_3 \rightharpoonup p_5$ removed, and the system state γ_{ij} is transformed to Figure 25 (b). In step 1.2, $q_2 \rightharpoonup p_5$ is removed. In step 1.3, $p_6 \rightarrow q_5$ is removed, resulting system state $\gamma_{i,j+1}$ as can be seen in Figure 25 (d). In step 2, $q_5 \rightharpoonup p_1$is removed. System state is $\gamma_{i,j+5}$ as can be seen in Figure 25 (e). Finally, in step 3, $p_1 \rightarrow q_4$ is removed. Thus, all the dangling paths have been removed, and so we have that $\tau(\gamma_{i,j+3}) = \emptyset$, as can be seen by inspection of Figure 25 (f) where clearly no terminal (sink or source) nodes are present. The system state $M_{i,j+3} = \gamma_{i,j+3}$ contains only link edges that form a cycle c_1. The cycle c_1, present in M_{ij}, is preserved intact in $M_{i,j+3}$. Thus, we will detect deadlock in this case.

Example 29. An equivalent example of Figure 25 is shown in Table 2. The matrix M_{ij} represents the system state γ_{ij}. For this case, let $M_{i,j+1}$ be the first state derived by applying a reduction step to M_{ij} – note that we show three intermediate steps to derive $M_{i,j+1}$. $M_{i,j+3}$ is the non-empty irreducible matrix found at the end of the algorithm. $M_{i,j+3}$ preserves the cycle from M_{ij}. Since $M_{i,j+3} \neq \emptyset$, a deadlock exists among processor nodes p_5 and p_3 and resource nodes q_1 and q_4. Note that the matrix reduction sequence is done sequentially to illustrate the idea. The proposed algorithm makes use of parallelism to improve the run time performance, since multiple dangling paths are removed simultaneously and independently of each other.

The algorithm find out if a given M_{ij} contains cycles. If M_{ij} has cycles, then a set P_d of deadlocked processors and a set Q_d of deadlocked resources can be identified.

We use the matrix representation to implement Algorithms 3 and 4. By Theorem 5, we always correctly find out if deadlock exists or not.

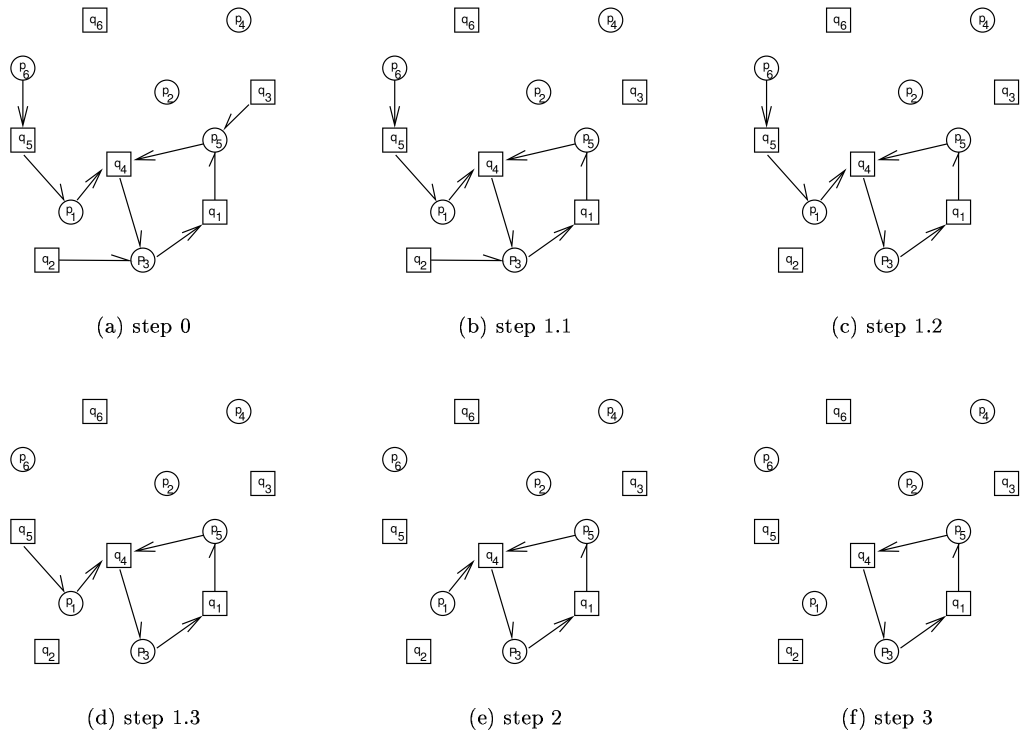


FIGURE 25. Graph Reduction Sequence of both Sinks and Sources

The following terms are defined for the matrix reduction sequence.

Definition 57. *A **terminal column** τ_{ct} is a column t (recall that column t corresponds to resource q_t) of a matrix M with either (i) all non zero entries $\{m_{st} \neq 0, 1 \leq s \leq m\}$ are request entries r_{st} with at least one non-zero (request) entry, or (ii) one entry $m_{st}, 1 \leq s \leq m$ is a grant $g_{s_h,t}$ with the rest of the entries $\{m_{st}, 1 \leq s \leq m, s \neq s_h\}$ equal to zero.*

Definition 58. *A **terminal row** $M_{s,\tau}$ is a row s (recall that row s corresponds to processor p_s) of matrix M with either (i) all non-zero entries $\{m_{st} \neq 0, 1 \leq t \leq n\}$ are request entries r_{st} with at least one non-zero (request) entry, or (ii) all non-zero entries $\{m_{st} \neq 0, 1 \leq t \leq n\}$ are grant entries g_{st} with at least one non-zero (grant) entry.*

Definition 59. *A **connect column** ϕ_{ct} is a column t with at least one request r and at least one grant g in column t.*

M_{ij}	q_1	q_2	q_3	q_4	q_5	q_6
p_1				r	g	
p_2						
p_3	r	g		g		
p_4						
p_5	g		g	r		
p_6					r	

Step 1.1	q_1	q_2	q_3	q_4	q_5	q_6
p_1				r	g	
p_2						
p_3	r	g		g		
p_4						
p_5	g			r		
p_6					r	

Step 1.2	q_1	q_2	q_3	q_4	q_5	q_6
p_1				r	g	
p_2						
p_3	r			g		
p_4						
p_5	g			r		
p_6					r	

$M_{i,j+1}$	q_1	q_2	q_3	q_4	q_5	q_6
p_1				r	g	
p_2						
p_3	r			g		
p_4						
p_5	g			r		
p_6						

$M_{i,j+2}$	q_1	q_2	q_3	q_4	q_5	q_6
p_1				r		
p_2						
p_3	r			g		
p_4						
p_5	g			r		
p_6						

$M_{i,j+3}$	q_1	q_2	q_3	q_4	q_5	q_6
p_1						
p_2						
p_3	r			g		
p_4						
p_5	g			r		
p_6						

TABLE 2. Matrix Reduction Sequence of both Sinks and Sources

Definition 60. *A* ***connect row*** $\boldsymbol{\phi_{rs}}$ *is a row s with at least one request r and at least one grant g in row s.*

We first show how to use the matrix representation to implement Algorithm 4.

Algorithm 5. *Deadlock Detection Algorithm Reducing Sink and Source Nodes Using Matrix Representation*

```
1   Deadlock_Detect_Matrix ($\gamma_{ij}$) {
2       $M[s,t] = [m_{st}]$, where
3           $s = 1, \ldots, m$ and $t = 1, \ldots, n$
4           $m_{st} = r$, if $\exists(p_s, q_t) \in E(\gamma_{ij})$
5           $m_{st} = g$, if $\exists(q_s, p_t) \in E(\gamma_{ij})$
6           $m_{st} = 0$, otherwise.
7       $M_{i,j+k} = \Delta_M(M_{ij})$; /* call Algorithm 6 */
8       if ( $M_{i,j+k} = [0]$ ) {
9           return 0; /* no deadlock */
10      } else {
11          return 1; /* deadlock detected */
12      }
13  }
```

Next, in the following algorithm, we show a parallel implementation of Δ_M (Definition 56) implemented by a version of Algorithm 3 suitably modified to operate on matrices.

Algorithm 6. *Parallel Reduction Sequence Algorithm*

```
1  $\Delta_M(M_{ij})$ {
2      $k = 0$;
3      $M_{iterate} = M_{ij}$;
4      while (($\exists s$ such that $\exists \tau_{rs}$) or ($\exists t$ such that $\exists \tau_{ct}$)) {
5          $k = k + 1$;
           /* concurrently execute lines 6-8 and lines 9-11 at the same time */
6          for all $s$ such that $\exists \tau_{rs}$ { /* Definition 58 */
7              all entries in row $s$ of $M_{iterate}$ are set to zero entries;
8          }
9          for all $t$ such that $\exists \tau_{ct}$ { /* Definition 57 */
10             all entries in column $t$ of $M_{iterate}$ are set to zero entries;
11         }
12     }
13     $M_{i,j+k} = M_{iterate}$ ;
14     return $M_{i,j+k}$ ;
15 }
```

Example 30. In Table 3, the previous example of Table 2 or Figure 25 is used to illustrate the parallel algorithm. The system state γ_{ij} is captured in matrix M_{ij} in lines 2-6 of Algorithm 5. Next, line 7 of Algorithm 5 calls Algorithm 6.

In M_{ij}, there are several dangling paths. First, $q_2 \rightharpoonup p_3$ is a dangling path with the terminal node in the path being a source node (q_2 is a source). Second, $q_3 \rightharpoonup p_5$ is another dangling path, and it connects to a cycle involving processor node p_5. Third, there is a long dangling path (p_6, q_5, p_1, q_4), where q_4 is involved in a cycle.

Therefore, in line 4 of Algorithm 6, there do exist terminal columns (columns 2 and 3) and a terminal row (row 6). Thus, in the first iteration, there are three terminal nodes which can be removed at the same time, resulting in the removal of three edges: $g_{3,2}$, $g_{5,3}$ (sink edges) and $r_{6,5}$ (a source edge), where the subscript index pair denotes the processor node and the resource node in the matrix table coordinate. The result of lines 7 and 10 of Algorithm 6 can be seen in matrix $M_{i,j+1}$ of Table 2.

The second execution of line 4 of Algorithm 6 finds that there is one terminal column: column 5. Thus, column 5 is set to zero thereby removing $g_{1,5}$ corresponding to sink vertex q_5. The result of line 10 of Algorithm 6 can be seen in matrix $M_{i,j+2}$ of Table 2.

In the third iteration, row 1 is set to zero, deleting $r_{1,4}$. The result is $M_{i,j+3}$. However, no more terminal columns nor terminal rows exist, so Algorithm 6 returns non-empty and irreducible $M_{i,j+3}$ to Algorithm 6.

Thus, a cycle has been revealed. The cycle is $p_3 \rightarrow q_1 \rightharpoonup p_5 \rightarrow q_4 \rightharpoonup p_3$. Note: In Table 3, the arrows in the matrix $M_{i,j+3}$ are not request nor grant edges, they are used to illustrate the cycle path more explicitly. We have discovered deadlock!

$M_{i,j}$	q_1	q_2	q_3	q_4	q_5	q_6
p_1				r	g	
p_2						
p_3	r	g		g		
p_4						
p_5	g		g	r		
p_6					r	

$M_{i,j+1}$	q_1	q_2	q_3	q_4	q_5	q_6
p_1				r	g	
p_2						
p_3	r			g		
p_4						
p_5	g			r		
p_6						

$M_{i,j+2}$	q_1	q_2	q_3	q_4	q_5	q_6
p_1				r		
p_2						
p_3	r			g		
p_4						
p_5	g			r		
p_6						

$M_{i,j+3}$	q_1	q_2	q_3	q_4	q_5	q_6
p_1						
p_2						
p_3	r	$\cdots$	$\rightarrow$	g		
p_4	$\leftarrow$			$\rightarrow$		
p_5	g	$\leftarrow$	$\cdots$	r		
p_6						

TABLE 3. Matrix Reduction of both Sinks and Sources

Lemma 8. *In RAG γ with $P = m$ and $Q = n$, the maximum number of edges in any path is $k \leq 2 \times min(m, n)$.*

Proof: We start by repeating the definition of a path. A path $(v_1, v_2, v_3, \ldots, v_{k-1}, v_k)$, $k \geq 2$, is a consecutive ordered sequence of alternating request and grant edges (v_1, v_2), (v_2, v_3), …, (v_{k-1}, v_k) where every node in the path is distinct and where every other node belongs to the same set. In other words, every odd node along a path belongs to one node set of V (either P or Q) and every even node along the same path belongs to the other node set of V.

The number of nodes in a processor set is $m = P$ and the number of nodes of a resource set is $n = Q$. For an arbitrary graph of $(m + n)$ nodes, the upper bound of the number of edges of a path is $m + n - 1$ (note that by Definition 42, a path cannot have a cycle since all nodes are distinct), assuming the graph is not a bipartite graph. We use k to represent the length of a path, i.e., the number of edges along a path. Thus the upper bound of k is shown as follows:

$$k = (m + n) - 1 \tag{12}$$

However, a RAG is by Definition 6 a bipartite graph. For a bipartite graph, there exists three different special cases that limit the upper bound of the length k even further. An edge (see Definition 7) is either a request or a grant. A request edge begins with a processor node and ends with a resource node, while a grant edge begins with a resource node and ends with a processor node.

Case 1 – $P = Q$ **:** In this case, the number m of processor nodes and the number n of resource nodes are the same. A path has maximum number of edges when it contains all nodes $v \in V$. If a path begins with a processor node, then the path must end with a resource node. On the other hand, if a path begins with a resource node, then the path must end with a processor node. Therefore, all nodes are covered by the longest path and the number of edges k is as follows:

$$k = (m + n) - 1 = 2 \times m - 1 = 2 \times n - 1, \text{ if } m = n \tag{13}$$

Case 2 – $P > Q$ **:** In this case, the number m of processor nodes is greater than the number n of resource nodes. A path has maximum number of edges when it contains all resource nodes $q \in Q$. Since there are more processor nodes than resource nodes the longest path starts with a processor node and ends with another distinct processor node. Therefore, the number of nodes in the longest path is $n + 1 + n$ and the number of edges is as follows:

$$k = (n + 1) + n - 1 = 2 \times n, \text{ if } m > n \tag{14}$$

Case 3 – $P < Q$ **:** In this case, the number m of processor nodes is less than the number n of resource nodes. This is an exact mirror case of Case 2 and therefore the number of edges k is as follows:

$$k = m + (m + 1) - 1 = 2 \times m, \text{ if } n > m \tag{15}$$

Summarizing all cases (Equations 13, 14, and 15) the length of the longest path, i.e., the number of edges, is as follows:

$$k = \begin{cases} 2 \times m - 1 = 2 \times n - 1, & \text{if } m = n \\ 2 \times n, & \text{if } m > n \\ 2 \times m, & \text{if } m < n \end{cases} \tag{16}$$

It can be seen from Equation 16 that k is always limited by the smallest subset of V as follows:

$$k \leq 2 \times \min(m, n) \tag{17}$$

Lemma 9. *An iteration of Algorithm 6 reduces the length of any path with at least one terminal node by at least one.*

Proof: The while-loop of Algorithm 6, i.e., lines 4 to 12, is referred to as an iteration. The iteration is executed if and only if the condition in line 4 is true, i.e., if there exists at least one terminal node (see Definition 31) which is a terminal column (see Definition 57) or a terminal row (see Definition 58). The body of the iteration (lines 5 to 12) removes all edges corresponding to terminal nodes. A simple path (Definition 43) has two terminal nodes, therefore the length of this path is reduced by two in an iteration. At the same time, a dangling path (Definition 44) has only one terminal node, therefore its length is reduced by one in an iteration. There are no cases of paths with terminal node(s) other than simple paths and dangling paths. In the worst case, the graph has only one dangling path whose length is reduced by one in every iteration of Algorithm 6.

Theorem 6 below assumes that the time complexity measure is the number of iterations of lines 4-12 of Algorithm 6. The statements in lines 6-8 and 9-11 of Algorithm 6 are executed in parallel in constant time for the hardware implementation described in Section 6.

Theorem 6. *Algorithm 6 completes operation on state γ_{ij} in time $O_{hw}(min(m, n))$ for a parallel implementation.*

Proof:

Note that a path p without at least one terminal node is either (i) contained in another path with at least one terminal node or (ii) not contained in an other path with at least one terminal node. Case (ii) indicates the presence of at least one cycle some of whose nodes either are already part of the path p or end up being included in the "expanded" paths which contain path p. Clearly, we need only consider "expanded" paths defined by case (i) and may ignore paths defined by case (ii). The reason why we need only consider "expanded" paths with terminal nodes is that paths contained in such "expanded" paths clearly have less edges and thus will have edges removed in future iterations once the "expanded" path(s) containing it has (have) enough edges removed. Finally, given the set of all paths with at least one terminal node, according to Lemma 8, the maximum number of edges of any path in this set of paths with at least one terminal node is $k \leq 2 \times \min(m, n)$. According to Lemma 9, one iteration of Algorithm 6 reduces the length of all of these paths by at least one. Therefore, the remaining set of paths with terminal nodes has no path with length greater than $k - 1$. Continuing in this way, Algorithm 6 terminates when there are no more paths with terminal nodes left in the graph. Hence Algorithm 6 has time complexity of $O_{hw}(\min(m, n))$.

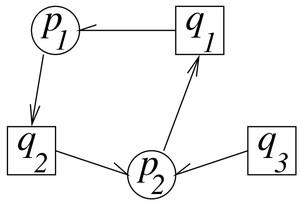
6. Parallel Hardware Deadlock Detection Architecture

The architecture described in this section will be able to perform all calculations of an iteration of Algorithm 6 (lines 4-11) in parallel. First, we illustrate the architecture derivation using two examples. Second, we generalize the architecture of the Deadlock Detection Unit (DDU) more formally. Third, we describe the components of the architecture in detail.

6.1. **Deriving Deadlock Detection Architecture.**

We will illustrate the derivation of the architecture based on two examples. We will show how to perform all calculations of an iteration of Algorithm 6 in parallel. The first example describes how a deadlock state is detected, and the second example describes how a safe state is detected.

Example 31. Consider the example shown in Table 4 that has a deadlock situation. In this example, we have the following initial M_{ij} (Line 1 of Algorithm 6):



P\Q	q_1(IcP)	q_2(PCI)	q_3(WI)
p_1(DSP)	g	r	0
p_2(VSP)	r	g	g

TABLE 4. Example-I with Two Processors and Three Resources.

$$M_{ij} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \end{bmatrix} = \begin{bmatrix} g & r & 0 \\ r & g & g \end{bmatrix} \tag{18}$$

Each m_{st} can have a value either 0, g or r. A natural choice for m_{st} encoding would be one-hot encoding [7] because it clearly distincts different types of edges. We define m_{st} as a pair of two bits $m_{st} = (m^r_{st}, m^g_{st})$. If an entry m_{st} is a request r_{st}, then bit m^r_{st} is set to one and bit m^g_{st} is set to zero. If an entry m_{st} is a grant g_{st}, then bit m^r_{st} is set to zero and bit m^g_{st} is set to one. If there is no edge in entry m_{st}, then both bits m^r_{st} and m^g_{st} are set to zero. Hence, an entry m_{st} can be either one of the following binary encodings 00 (no edge), 01 (grant edge), and 10 (request edge). We call the new binary encoded matrix $M_{ij;\, B}$:

$$M_{ij} \Rightarrow M_{\text{iter};\, B} = \begin{bmatrix} (m^r_{11}, m^g_{11}) & (m^r_{12}, m^g_{12}) & (m^r_{13}, m^g_{13}) \\ (m^r_{21}, m^g_{21}) & (m^r_{22}, m^g_{22}) & (m^r_{23}, m^g_{23}) \end{bmatrix} = \begin{bmatrix} 01 & 10 & 00 \\ 10 & 01 & 01 \end{bmatrix} \tag{19}$$

The matrices M_{iterate} and $M_{i,j+k}$ are encoded in the same way. The encoded matrices are labeled as $M_{\text{iter};\, B}$ and $M_{i,j+k;\, B}$, correspondingly.

In the beginning of the first iteration, the $M_{\text{iter};\, B}$ is equal to $M_{ij;\, B}$. In order to find out whether a system state matrix $M_{iterate}$ is reducible or not (line 4 of Algorithm 6), we need to determine if there exists any terminal column τ_{ct} or any terminal row τ_{rs} in M_{iterate}. If M_{iterate} contains at least one τ_{rs} or τ_{ct}, then M_{iterate} is reducible. Otherwise, M_{iterate} is irreducible. If M_{iterate} is irreducible, then the Algorithm 6 stops iterating. We use two vectors to indicate whether M_{iterate} is reducible or not. One of the vectors ($M_{\text{iter};\, CBO}$) indicates the existence of edges on resources (columns). The second vector

($M_{\text{iter};\,RBO}$) indicates the existence of edges on processors (rows). The vectors are calculated from the binary encoded matrix $M_{\text{iter};\,B}$.

Consider the column 1 in $M_{\text{iter};\,B}$. The existence of a request edge means that at least one m^r_{s1} is "1" in $M_{\text{iter};\,B}$. The existence of a grant edge means that one m^g_{s1} is "1" in $M_{\text{iter};\,B}$. Therefore, we perform a bitwise OR between all pairs (m^r_{s1}, m^g_{s1}) to find the first element of $M_{\text{iter};\,CBO}$ – $(m^r_{c;\,1}, m^g_{c;\,1})$. Similarly, we use bitwise OR for other columns to find the other elements of $M_{\text{iter};\,CBO}$.

Consider the row 1 in $M_{\text{iter};\,B}$. The existence of a request edge means that at least one m^r_{1t} is "1" in $M_{\text{iter};\,B}$. The existence of a grant edge means that at least one m^g_{1t} is "1" in $M_{\text{iter};\,B}$. Therefore, we perform a bitwise OR between all pairs (m^r_{1t}, m^g_{1t}) to find the first element of $M_{\text{iter};\,RBO}$ – $(m^r_{r;\,1}, m^g_{r;\,1})$. Similarly, we use bitwise OR for other rows to find the other elements of $M_{\text{iter};\,RBO}$.

The resulting vectors $M_{\text{iter};\,CBO}$ and $M_{\text{iter};\,CBO}$ are as follows:

$$\begin{aligned} M_{\text{iter};\,CBO} &= \begin{bmatrix} (m^r_{c;\,1}, m^g_{c;\,1}) & (m^r_{c;\,2}, m^g_{c;\,2}) & (m^r_{c;\,3}, m^g_{c;\,3}) \end{bmatrix} = \begin{bmatrix} 11 & 11 & 01 \end{bmatrix} \\ &\text{where } m^r_{c;\,t} = \bigvee_{s=1}^{2} m^r_{st}, \text{ and } m^g_{c;\,t} = \bigvee_{s=1}^{2} m^g_{st}, \text{ for } 1 \le t \le 3 \end{aligned} \tag{20}$$

$$\begin{aligned} M_{\text{iter};\,RBO} &= \begin{bmatrix} (m^r_{r;\,1}, m^g_{r;\,1}) \\ (m^r_{r;\,2}, m^g_{r;\,2}) \end{bmatrix} = \begin{bmatrix} 11 \\ 11 \end{bmatrix} \\ &\text{where } m^r_{r;\,s} = \bigvee_{t=1}^{3} m^r_{st}, \text{ and } m^g_{r;\,s} = \bigvee_{t=1}^{3} m^g_{st}, \text{ for } 1 \le s \le 2 \end{aligned} \tag{21}$$

If there is an entry "00" in column t of $M_{\text{iter};\,CBO}$, then there are no edges in column t (resource q_t is neither requested nor granted). If there is an entry "01" in column t of $M_{\text{iter};\,CBO}$, then there is one grant edge in column t (resource q_t is granted to only one processor and is not requested by any processor), and the corresponding resource is a terminal node. If there is an entry "10" in column t of $M_{\text{iter};\,CBO}$, then there is at least one request edge in column t (resource q_t is not granted to any processor and is requested by at least one processor), and the corresponding resource is a terminal node. If there is an entry "11" in column t of $M_{\text{iter};\,CBO}$, then there exists at least one request edge and exactly one grant edge in column t (resource q_t is granted to one processor and requested by at least one processor), and the corresponding resource is a connect node. Vector $M_{\text{iter};\,RBO}$ is similarly interpreted. If there is an entry "00" in row s of $M_{\text{iter};\,RBO}$, then there are no edges in row s (processor p_s is not requesting nor holding any resource). If there is an entry "01" in row s of $M_{\text{iter};\,RBO}$, then there is at least one grant edge in row s (processor p_s is holding at least one resource but is not requesting any resource), and the corresponding processor is a terminal node. If there is an entry "10" in row s of $M_{\text{iter};\,RBO}$, then there is at least one request edge in row s (processor p_s is not holding any resource but requesting at least one resource), and the corresponding processor is a terminal node. If there is an entry "11" in column t of $M_{\text{iter};\,CBO}$, then there exists at least one request edge and exactly one grant edge in column t (processor p_s is both holding and request at least one resource), and the corresponding processor is a connect node.

It can be seen from the cases described above that the existence of terminal nodes is always indicated by the fact that the bits differ in pairs $(m^r_{c;\,t}, m^g_{c;\,t})$ and $(m^r_{r;\,s}, m^g_{r;\,s})$. Therefore, we perform XOR operation between the bits of these pairs.

Consider the column 1 in $M_{\text{iter};\,CBO}$. We perform XOR between $(m^r_{c;\,1}$ and $m^g_{c;\,1})$ to find the first element τ_{c1}. Similarly, we perform XOR for other columns to find other elements – τ_{c2}, τ_{c3}. We collect all the resulting bits into vector $X_{\text{iter};\,CBO}$. Analogously, we peform XOR between $m^r_{r;\,s}$ and $m^g_{r;\,s}$ for all rows to find all elements τ_{rs} of $X_{\text{iter};\,RBO}$.

In this example, columns 1 and 2 of $M_{\text{iter};\,CBO}$ have value "11", which means that the resources q_1 and q_1 are both granted to one processor and are requested by at least one processor. Column 3 of $M_{\text{iter};\,CBO}$ has value "01", which means that the resource q_3 is granted to only one processor and is not requested by any processor. Thus, column 3 is a terminal column. As shown in $M_{\text{iter};\,RBO}$ all processors are requesting and holding at least one resource.

$$X_{\text{iter};\,CBO} = \begin{bmatrix} \tau_{c1} & \tau_{c2} & \tau_{c3} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \quad \text{where } \tau_{ct} = m^r_{c;\,t} \oplus m^g_{c;\,t} \tag{22}$$

$$X_{\text{iter};\,RBO} = \begin{bmatrix} \tau_{r1} \\ \tau_{r2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \text{where } \tau_{ct} = m^r_{c;\,t} \oplus m^g_{c;\,t} \tag{23}$$

The vector $X_{\text{iter};\,CBO}$ is used to identify which column(s) is (are) terminal column(s) τ_{ct} – the corresponding column entry is (entries are) "1". Similarly, the vector $X_{\text{iter};\,RBO}$ is used to identify which row(s) is (are) terminal row(s) τ_{rs} – the corresponding row entry is (entries are) "1". In this example, there is only one "1" entry in the column 3 of $X_{\text{iter};\, CBO}$. Thus, the column 3 of M_{iter} is a terminal column and according to line 10 of Algorithm 6, all entries in the column 3 will be set to zero. This means that m_{23} is changed from a grant edge to no edge (releasing resource q_3). In other words, resource q_3 is a terminal node while serving processor p_2, and is reduced to an isolated node after serving processor p_2. The "0" entries in $X_{\text{iter};\, RBO}$ indicate that there are no terminal rows and thus line 7 of Algorithm 6 is not executed in this iteration. A new state matrix $M_{i,j+1}$ is obtained at the end of the first iteration. The matrix reduction step $\delta_M(M_{ij})$ is represented by statements from Line 6 to Line 12 of Algorithm 6. The content of the new matrix $M_{i,j+1}$ is shown as follows:

$$M_{i,j+1;\, B} = M_{\text{iter};\, B} = \begin{bmatrix} 01 & 10 & 00 \\ 10 & 01 & 00 \end{bmatrix} \quad \left(M_{i,j+1} = M_{\text{iteration}} = \begin{bmatrix} g & r & 0 \\ r & g & 0 \end{bmatrix} \right) \tag{24}$$

At the same time when $X_{\text{iter};\, CBO}$ and $X_{\text{iter};\, RBO}$ are calculated, we can also determine whether M_{iter} contains any connect column ϕ_{ct} and connect row ϕ_{rs} by generating another two vectors: (1) $A_{\text{iter};\, CBO}$ from $M_{\text{iter};\, CBO}$, and (2) $A_{\text{iter};\, RBO}$ from $M_{\text{iter};\, RBO}$. The vectors $A_{\text{iter};\, CBO}$ and $A_{\text{iter};\, RBO}$ are used to determine whether a reduction sequence Δ_M is a complete reduction or an incomplete reduction (Line 8 to Line 12 of Algorithm 5). If a matrix M_{iter} is incompletely reduced, then $M_{i,j+k} = \Delta_M(M_{ij})$ must contain at least one cycle (See Theorem 4 and Theorem 5), whose nodes are all connect nodes. Since the nodes in a cycle of an irreduced state matrix $M_{i,j+k}$ are not reduced, and are in the set Σ_ϕ of connected nodes, the connect row ϕ_{rs} and the connect column ϕ_{ct} can be used to distinct between complete and incomplete reductions of a matrix reduction sequence $\Delta_M(M_{ij})$. If state matrix $M_{i,j+k}$ contains connect rows and connect columns at the end of a matrix reduction sequence, the matrix reduction sequence is an incomplete reduction. Otherwise, the matrix reduction sequence is a completely reduction.

The vectors $A_{\text{iter};\, CBO}$ and $A_{\text{iter};\, RBO}$ are calculated according to Definitions 40, 59, and 60. A node is a connect node if it has both request and grant edges, i.e., the value $(m^r_{c;\, t}, m^g_{c;\, t})$ (or $(m^r_{r;\, s}, m^g_{r;\, s})$) is "11". Thus, an element of the vector $A_{\text{iter};\, CBO}$ is AND of every two bits ($m^r_{c;\, t}$ and $m^g_{c;\, t}$) of $M_{\text{iter};\, CBO}$, and an element of the vector $A_{\text{iter};\, RBO}$ is AND of every two bits ($m^r_{r;\, s}$ and $m^g_{r;\, s}$) of $M_{\text{iter};\, RBO}$. The vector $A_{\text{iter};\, CBO}$ is used to identify which column t is a connect column – ϕ_{ct} is "1". Similarly, the vector $A_{\text{iter};\, RBO}$ is used to identify which row s is a connect row – ϕ_{rs} is "1". In this example, there are two "1"s in $A_{\text{iter};\, CBO}$ and two "1"s in $A_{\text{iter};\, RBO}$. In other words, processors p_1 and p_2 are connect nodes – they are either inside a path or a cycle. Similarly are connect nodes resources q_1 and q_2.

$$A_{\text{iter};\, CBO} = \begin{bmatrix} \phi_{c1} & \phi_{c2} & \phi_{c3} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix} \quad \text{where } \phi_{ct} = m^r_{c;\, t} \wedge m^g_{c;\, t}, \text{ for } 1 \le t \le 3 \tag{25}$$

$$A_{\text{iter};\, RBO} = \begin{bmatrix} \phi_{r1} \\ \phi_{r2} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad \text{where } \phi_{rs} = m^r_{r;\, s} \wedge m^g_{r;\, s}, \text{ for } 1 \le s \le 2 \tag{26}$$

In this iteration, $A_{\text{iter};\, CBO}$ and $A_{\text{iter};\, RBO}$ are ignored because the current values of $X_{\text{iter};\, CBO}$ and $X_{\text{iter};\, RBO}$ show that the current state matrix M_{iter} is reducible. It is still worth to calculate these vectors at every iteration to speedup the decisions in Algorithm 5. This is similar to speculative excution [11] found in modern computer architecturess.

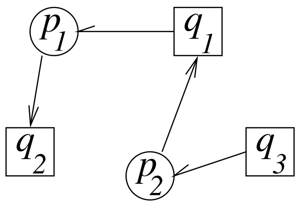
Now we will show the calculations that occur in the second iteration. For this iteration, the matrix $M_{\text{iter};\, B}$ is as follows:

$$M_{\text{iter};\, B} = \begin{bmatrix} 01 & 10 & 00 \\ 10 & 0 & 00 \end{bmatrix} \quad \left(M_{\text{iterate}} = \begin{bmatrix} g & r & 0 \\ r & g & 0 \end{bmatrix} \right) \tag{27}$$

Vectors $M_{\text{iter};\, CBO}$, $M_{\text{iter};\, RBO}$, $X_{\text{iter};\, CBO}$, $X_{\text{iter};\, RBO}$, $A_{\text{iter};\, CBO}$, and $A_{\text{iter};\, RBO}$ are calculated from $M_{\text{iter};\, B}$ like in the first iteration.

$$
\begin{aligned}
M_{\text{iter};\,CBO} &= \begin{bmatrix} 11 & 11 & 00 \end{bmatrix} & M_{\text{iter};\,RBO} &= \begin{bmatrix} 11 & 11 \end{bmatrix}^T \\
X_{\text{iter};\,CBO} &= [\ 0 \ \ 0 \ \ 0\ ] & X_{\text{iter};\,RBO} &= [\ 0 \ \ 0\ ]^T \\
A_{\text{iter};\,CBO} &= \begin{bmatrix} 1 & 1 & 0 \end{bmatrix} & A_{\text{iter};\,RBO} &= \begin{bmatrix} 1 & 1 \end{bmatrix}^T
\end{aligned} \tag{28}
$$

Since there are only "0"s in vectors $X_{\text{iter};\,CBO}$ and $X_{\text{iter};\,RBO}$, the state matrix $M_{\text{iter};\,B}$ does not have any terminal columns nor rows. In other words, the system state matrix $M_{\text{iter};\,B}$ is in an irreducible state. Also, vectors $A_{\text{iter};\,CBO}$ and $A_{\text{iter};\,RBO}$ indicate that there exist connect nodes. Since the state matrix $M_{\text{iter};\,B}$ is irreducible (line 4 of Algorithm 6), the Algorithm 6 returns $M_{\text{iter};\,B}$ as $M_{i,j+k;\,B}$. Algorithm 5 then checks the vectors $A_{\text{iter};\,CBO}$ and $A_{\text{iter};\,RBO}$ to determine if the matrix reduction sequence $\Delta_M(M_{ij})$ is completely reduced or not. Since vector $A_{\text{iter};\,CBO}$ contains "1"s in column 1 and 2, connect nodes exist (Line 11 of Algorithm 5). Therefore, a deadlock exists in state matrix M_{ij}. Note that the value k, which would help us to distinguish between state matrices, is not computed.



P\Q	q_1	q_2	q_3
p_1	g	r	0
p_2	r	0	g

TABLE 5. Example-II with 2 Processors and 3 Resources

Example 32. Consider another example shown in Table 5, which does not contain a cycle. There are two processors, p_1 and p_2, and three resources, q_1, q_2, and q_3. The state matrix M_{iterate} and its one-hot encoded representation $M_{\text{iter};\,B}$ are shown as follows:

$$
M_{\text{iterate}} = \begin{bmatrix} g & r & 0 \\ r & 0 & g \end{bmatrix} \qquad M_{\text{iter};\,B} = \begin{bmatrix} 01 & 10 & 00 \\ 10 & 00 & 01 \end{bmatrix} \tag{29}
$$

In the first iteration, $X_{\text{iter};\, CBO}$ indicates that all entries in the second and third columns should be set to zeros. However, there are no terminal rows because $X_{\text{iter};\, RBO}$ has only "0"s. Therefore, all entries in second and third columns are replaced by zeros as shown in the next iteration.

$$\begin{aligned} M_{\text{iter};\, CBO} &= \begin{bmatrix} 11 & 10 & 01 \end{bmatrix} & M_{\text{iter};\, RBO} &= \begin{bmatrix} 11 & 11 \end{bmatrix}^T \\ X_{\text{iter};\, CBO} &= \begin{bmatrix} 0 & 1 & 1 \end{bmatrix} & X_{\text{iter};\, RBO} &= \begin{bmatrix} 0 & 0 \end{bmatrix}^T \\ A_{\text{iter};\, CBO} &= \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} & A_{\text{iter};\, RBO} &= \begin{bmatrix} 1 & 1 \end{bmatrix}^T \end{aligned} \tag{30}$$

In the beginning of the second iteration M_{iterate} and $M_{\text{iter};\, B}$ are as follows:

$$M_{\text{iterate}} = \begin{bmatrix} g & 0 & 0 \\ r & 0 & 0 \end{bmatrix} \qquad M_{\text{iter};\, B} = \begin{bmatrix} 01 & 00 & 00 \\ 10 & 00 & 00 \end{bmatrix} \tag{31}$$

In the second iteration $X_{\text{iter};\, RBO}$ has two "1"s which indicate that both rows should have their entries set to zeros.

$$\begin{aligned} M_{\text{iter};\, CBO} &= \begin{bmatrix} 11 & 00 & 00 \end{bmatrix} & M_{\text{iter};\, RBO} &= \begin{bmatrix} 01 & 10 \end{bmatrix}^T \\ X_{\text{iter};\, CBO} &= \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} & X_{\text{iter};\, RBO} &= \begin{bmatrix} 1 & 1 \end{bmatrix}^T \\ A_{\text{iter};\, CBO} &= \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} & A_{\text{iter};\, RBO} &= \begin{bmatrix} 0 & 0 \end{bmatrix}^T \end{aligned} \tag{32}$$

The third iteration is not performed because M_{iterate} is irreducible as shown by vectors $X_{\text{iter};\, CBO}$ and $X_{\text{iter};\, RBO}$ – both contain only zeros. At the same time, the vectors $A_{\text{iter};\, CBO}$ and $A_{\text{iter};\, RBO}$ indicate that there is no deadlock – both vectors contain only zeros.

$$
\begin{aligned}
&M_{\text{iterate}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} && M_{\text{iter};\,B} = \begin{bmatrix} 00 & 00 & 00 \\ 00 & 00 & 00 \end{bmatrix} \\
&M_{\text{iter};\,CBO} = \begin{bmatrix} 00 & 00 & 00 \end{bmatrix} && M_{\text{iter};\,RBO} = \begin{bmatrix} 00 & 00 \end{bmatrix}^T \\
&X_{\text{iter};\,CBO} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} && X_{\text{iter};\,RBO} = \begin{bmatrix} 0 & 0 \end{bmatrix}^T \\
&A_{\text{iter};\,CBO} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} && A_{\text{iter};\,RBO} = \begin{bmatrix} 0 & 0 \end{bmatrix}^T
\end{aligned} \tag{33}
$$

6.2. **Generalization of Deadlock Detection Hardware Architecture.**

In this subsection, we will generalize the equations used in the previous subsection. A given system state γ_{ij} is equivalently represented by a system state matrix M_{ij}, which is used as the input to the DDU to perform deadlock detection. The system state matrix M_{ij} is explicitly represented in Equation 34.

$$
M_{ij} = \begin{bmatrix}
m_{11} & \cdots & m_{1t} & \cdots & m_{1n} \\
\vdots & \ddots & \vdots & & \vdots \\
m_{s1} & \cdots & m_{st} & \cdots & m_{sn} \\
\vdots & & \vdots & \ddots & \vdots \\
m_{m1} & \cdots & m_{mt} & \cdots & m_{mn}
\end{bmatrix} \tag{34}
$$

Based on the generalization of Equation 19, we derive the following binary encoded state matrix $M_{\text{iter};\,B}$, where m^r_{st} and m^g_{st} can have values “0” and “1”:

(35)
$$M_{\text{iter};\,B} = \begin{bmatrix} (m^r_{11}, m^g_{11}) & \cdots & (m^r_{1t}, m^g_{1t}) & \cdots & (m^r_{1n}, m^g_{1n}) \\ \vdots & \ddots & \vdots & & \vdots \\ (m^r_{s1}, m^g_{s1}) & \cdots & (m^r_{st}, m^g_{st}) & \cdots & (m^r_{sn}, m^g_{sn}) \\ \vdots & & \vdots & \ddots & \vdots \\ (m^r_{m1}, m^g_{m1}) & \cdots & (m^r_{mt}, m^g_{mt}) & \cdots & (m^r_{mn}, m^g_{mn}) \end{bmatrix}$$

Based on the generalization of Equation 20, we derive the following vector $M_{\text{iter};\,CBO}$:

(36)
$$\begin{aligned} M_{\text{iter};\,CBO} &= \begin{bmatrix} (m^r_{c;\,1}, m^g_{c;\,1}) & (m^r_{c;\,2}, m^g_{c;\,2}) & \cdots & (m^r_{c;\,t}, m^g_{c;\,t}) & \cdots & (m^r_{c;\,n}, m^g_{c;\,n}) \end{bmatrix} \\ &\text{where } \forall t, \quad m^r_{c;\,t} = \bigvee_{s=1}^{m} m^r_{st}, \text{ and } \quad m^g_{c;\,t} = \bigvee_{s=1}^{m} m^g_{st} \end{aligned}$$

Based on the generalization of Equation 21, we derive the following vector $M_{\text{iter};\,RBO}$:

(37)
$$\begin{aligned} M_{\text{iter};\,RBO} &= \begin{bmatrix} (m^r_{r;\,1}, m^g_{r;\,1}) & (m^r_{r;\,2}, m^g_{r;\,2}) & \cdots & (m^r_{r;\,s}, m^g_{r;\,s}) \cdots & (m^r_{r;\,n}, m^g_{r;\,n}) \end{bmatrix}^T \\ &\text{where } \forall s, \quad m^r_{r;\,s} = \bigvee_{t=1}^{n} m^r_{st}, \text{ and } \quad m^g_{r;\,s} = \bigvee_{t=1}^{n} m^g_{st} \end{aligned}$$

Based on the generalization of Equation 22, we derive the following binary vector $X_{\text{iter};\,CBO}$ that indicates the existence of terminal processor node(s):

(38)
$$\begin{aligned} X_{\text{iter};\,CBO} &= \begin{bmatrix} \tau_{c1} & \tau_{c2} & \cdots & \tau_{ct} & \cdots & \tau_{cn} \end{bmatrix} \\ &\text{where } \tau_{ct} = m^r_{c;\,t} \oplus m^g_{c;\,t} \end{aligned}$$

Based on the generalization of Equation 23, we derive the following binary vector $X_{\text{iter};\,RBO}$ that indicates the existence of terminal resource node(s):

(39)
$$\begin{aligned} X_{\text{iter};\,RBO} &= \begin{bmatrix} \tau_{r1} & \tau_{r2} & \cdots & \tau_{rs} & \cdots & \tau_{rm} \end{bmatrix}^T \\ &\text{where } \tau_{rs} = m^r_{r;\,s} \oplus m^g_{r;\,s} \end{aligned}$$

From the description in Examples 31 and 32, we have the following feedbacks from $X_{\text{iter:}CBO}$ and $X_{\text{iter:}RBO}$ of iteration k to update $M_{\text{iter; }B}$ for the next iteration ($k+1$):

$$(m^r_{st}, m^g_{st})^{k+1} = \begin{cases} (m^r_{st}, m^g_{st})^k, & \text{if } \tau_{ct} = 0 \text{ and } \tau_{rs} = 0 \\ (0,0), & \text{if } \tau_{ct} = 1 \text{ or } \tau_{rs} = 1 \end{cases} \tag{40}$$
$$\text{where k refers to } k^{\text{th}} \text{ iteration, and k + 1 refers to } k+1^{\text{th}} \text{ iteration}$$

Based on the generalization of Equation 25, we derive the following binary vector $A_{\text{iter; }CBO}$ that indicates the existence of connect node(s) among processors:

$$A_{\text{iter; }CBO} = \begin{bmatrix} \phi_{c1} & \phi_{c2} & \cdots & \phi_{ct} & \cdots & \phi_{cn} \end{bmatrix} \tag{41}$$
$$\text{where } \phi_{ct} = m^r_{c;\ t} \wedge m^g_{c;\ t}$$

Based on the generalization of Equation 26, we derive the following binary vector $A_{\text{iter; }RBO}$ that indicates the existence of connect node(s) among resources:

$$A_{\text{iter; }RBO} = \begin{bmatrix} \phi_{r1} & \phi_{r2} & \cdots & \phi_{rs} & \vdots & \phi_{rm} \end{bmatrix}^T \tag{42}$$
$$\text{where } \phi_{rs} = m^r_{r;\ s} \wedge m^g_{r;\ s}$$

From the description in Examples 31 and 32, we have the irreducability condition to terminate the iteration of Algorithm 6 that is calculated from $X_{\text{iter:}CBO}$ and $X_{\text{iter:}RBO}$:

$$d_r = \neg\tau_c \wedge \neg\tau_r, \quad \text{where } \tau_c = \bigvee_{t=1}^{n} \tau_{c;\ t}, \text{ and } \tau_r = \bigvee_{s=1}^{m} \tau_{r;\ s} \tag{43}$$

From the description in Examples 31 and 32, we also have the following deadlock detection condition (line 8 of Algorithm 5) that is calculated form $A_{\text{iter:}CBO}$ and $A_{\text{iter:}RBO}$:

$$d_x = \phi_c \vee \phi_r, \qquad \text{if } d_r = 1 \tag{44}$$
$$\text{where } \phi_c = \bigvee_{t=1}^{n} \phi_{c;\ t}, \text{ and } \phi_r = \bigvee_{s=1}^{m} \phi_{r;\ s}$$

6.3. Components of the Deadlock Detection Unit.

In this subsection we describe the **Deadlock Detection Unit** (DDU): a hardware unit that determines whether a given state M_{ij} (or it's equivalent γ_{ij}) is a deadlock state or not.

We define some more equations to facilitate the description of DDU architecture. We define d as a pair (d_r, d_x). From Equations 38 and 41 we define bottom weight vector as follows:

$$W_c = \begin{bmatrix} w_{c1} & w_{c2} & \cdots & w_{ct} & \cdots & w_{cn} \end{bmatrix} \tag{45}$$
$$\text{where } w_{ct} \text{ is a pair } (\tau_{c;\,t}, \phi_{c;\,t})$$

Each element w_{ct} in W_c is called a column weight cell. From Equations 41, and 42 we define the row weight vector as follows:

$$W_r = \begin{bmatrix} w_{r1} & w_{r2} & \cdots & w_{rs} & \cdots & w_{rm} \end{bmatrix}^T \tag{46}$$
$$\text{where } w_{rs} \text{ is a pair } (\tau_{r;\,s}, \phi_{r;\,s})$$

Each element w_{rs} in W_r is called a row weight cell. We refer to weight cell w as a column weight cell w_{ct} or row weight cell w_{rs}. Also, we refer to weight vector W as a column weight vector W_c or a row weight vector W_r.

Putting together all equations described previously, the architecture of the DDU consists of three parts. Part 1 is the system matrix M_i consisting of an array of matrix cells m_{st} that represent (m^r_{st}, m^g_{st}). Part 2 consists of two weight vectors: (i) one column weight vector W_c below the system matrix M_i, and (ii) one row weight vector W_r on the right hand side of the system matrix M_i. Part 3 consists of one decide cell d at the bottom right of the system matrix M_i. All cells are interconnected via buses.

In Figure 26 the decide cell d calculates Equations 44 and 43. Each matrix cell m_{st} calculates Equation 40. All column weight cells w_{ct} calculate Equations 36, 38, and 41, while all row weight cells w_{rs} calculate Equations 37, 39, and 42.

The most straightforward way to store the state of an iteration is to implement the matrix cells as FSMs that calculate Equation 40. That would require $4 \times m \times n$ flip-flops – half of them would be needed to store the initial M_{ij} and half of them would be needed to store the state M_{iterate}. From other side, the extra information M_{iterate} carries is the list of terminal nodes. This can be avoided by storing the

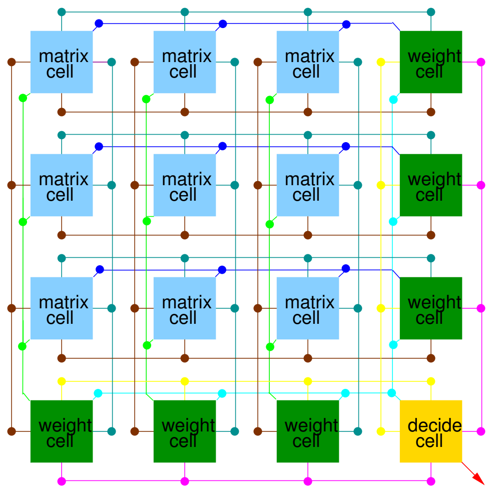


FIGURE 26. Deadlock Detection Architecture with three processors and three resources.

list of terminal nodes in weight cells because weight cells are actually determining the terminal nodes. The number of needed flip-flops would be then $2 \times (m+1) \times (n+1)$. The following description of cells follows the second possibility – the matrix cells are implemented as combinational functions and weight cells are implemented as FSMs.

6.3.1. *Matrix Cell.*

Each matrix cell m_{st} in Figure 27 has two inputs ${\tau_{ct}}^k$ and ${\tau_{rs}}^k$ as the current state of W_c and W_r, respectively. The inputs from the initial state matrix M_{ij} are implicitly understood by the indexes s and t (${m^r_{st}}^0$ and ${m^g_{st}}^0$). Each matrix cell m_{st} in Figure 27 has four outputs: ${t_{st}}^k$ on the top, ${b_{st}}^k$ on the bottom, ${l_{st}}^k$ on the left, and ${r_{st}}^k$ on the right of each side of the matrix cell m_{st}.

$$\begin{aligned} {t_{st}}^k &= {m^r_{st}}^0 \wedge {\tau_{rs}}^k & {l_{st}}^k &= {m^r_{st}}^0 \wedge {\tau_{ct}}^k \\ {b_{st}}^k &= {m^g_{st}}^0 \wedge {\tau_{rs}}^k & {r_{st}}^k &= {m^g_{st}}^0 \wedge {\tau_{ct}}^k \end{aligned} \tag{47}$$

Note that the superscript k denotes the current iteration.

6.3.2. *Weight Cell.*

Each column weight cell w_{ct} in Figure 28 has two inputs ${l_{st}}^k$ and ${r_{st}}^k$ as the current state of ${m_{st}}^k$. Also, each column weight cell w_{ct} in Figure 28 has two outputs: ${\tau_{ct}}^k$ on the top and ${\phi_{ct}}^k$ on the bottom.

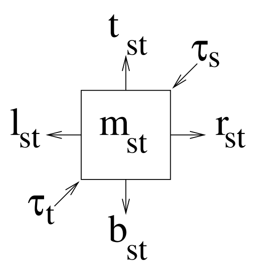
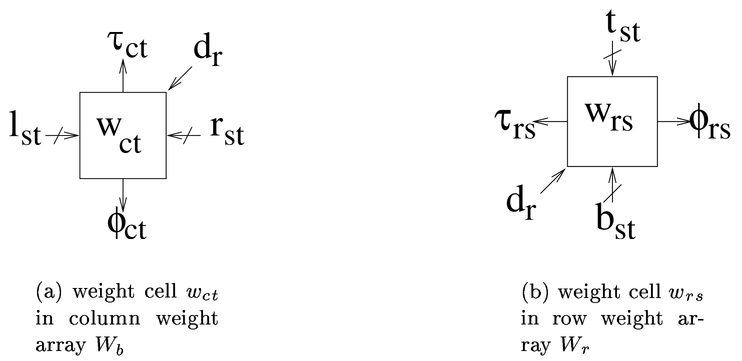


FIGURE 27. Matrix Cell m_{st} in the matrix array M_{ij}

Similarly, each row weight cell w_{rs} in Figure 28 has two inputs ${t_{st}}^k$ and ${b_{st}}^k$ as the current state of ${m_{st}}^k$. Also, each row weight cell w_{rs} in Figure 28 has two outputs: ${\tau_{rs}}^k$ on the left and ${\phi_{rs}}^k$ on the right.

$$\begin{aligned} {\tau_{ct}}^{k+1} &= \left(\bigvee_{s=1}^{m} l_{st}^k\right) \oplus \left(\bigvee_{t=1}^{n} r_{st}^k\right) \\ {\tau_{rs}}^{k+1} &= \left(\bigvee_{t=1}^{n} t_{st}^k\right) \oplus \left(\bigvee_{t=1}^{n} b_{st}^k\right) \end{aligned} \tag{48}$$

Note that the superscripts, k and $k+1$, indicate that Equation 48 is the next station function of the corresponding FSM.



(a) weight cell w_{ct} in column weight array W_b

(b) weight cell w_{rs} in row weight array W_r

FIGURE 28. Weight Cells in W_b and in $\cup W_r$

6.3.3. *Decide Cell.*

The decide cell d in Figure 29 has fours inputs τ_{ct} and ϕ_{rs} from W_c and τ_{rs} and ϕ_{rs} from W_c. The decide matrix cell d in Figure 29 has two outputs: d_r on the top left, d_x on the bottom right. d_r is calculated according to Equation 43 and d_x is calculated according to Equation 44.

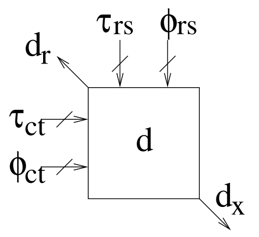
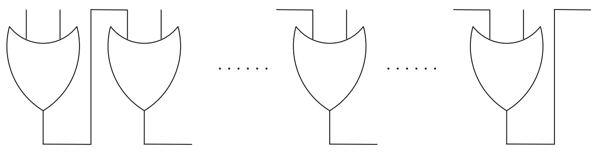
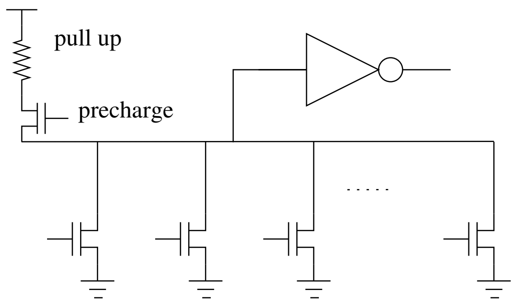


FIGURE 29. Decide Cell d



(a) A Chain of OR gates



(b) A Dynamic Wired OR gate

FIGURE 30. Different Implementations of Interconnections of DDU

6.4. **Interconnections of DDU.**

The number of inputs of each column weight cell (Figure 28) is proportional to the number of processors m. The number of inputs of each row weight cell (Figure 28) is proportional to the number of resources n. The number of inputs of the decide cell (Figure 28) is proportional to the number of processors m and the number of resources n. Hence, the hardware implementation does not scale very well, i.e., it has high fanin. One possible implementation is shown in Figure 30(a). A faster but significantly larger would be to implement this as an OR gate tree. However, we can optimize both area and speed futher by using either of the solutions shown in Figure 30(b).

6.5. **Synthesized Result of DDU.**

We used Synopsys Design Compiler (DC) to synthesize all modules of DDU using American Microsystems Inc. AMI 0.3μm standard cell library. The column "Gates" in Table 6 refers to the number of typical gates (NAND, NOR, INV, and XOR) and to the number of registers (D, JK, and T flip-flop). The column "Area" denotes that the area is relative to a 2 input NAND gate in AMI 0.3 library. The unit of the Delay column is ns. The longest delay is feedback path between state register of the weight cell. The area of DDU can be approximated as follows:

$$A = (m \times n) \times A_m + (m + n) \times A_w + A_d \tag{49}$$

$A_m = 5.1$ is the area of matrix cell m_{st}. $A_w = 13.9$ is the area of weight cell w_{bt} or w_{rs}. $A_d = 3.2$ is the area of the decide cell λ_2. Besides logic optimization, DC replaces all wired OR gates with ordinary OR gates, which increases the area and delay. Hence the difference between the Area from DC and Equation 49. Table 6 shows a good correlation of the expected and synthesized results about the scalability and speed of the DDU.

Module	Line	Area	Eq. 49	Gates	Delay
matrix	20	5.1	-	5+0	0.26
weight	60	13.9	-	7+1	0.64
decide	26	3.2	-	3+0	0.16
ie2x3	49	186.2.7	103	119+ 5	0.91
ie5x5	73	364.1	270	197+10	2.21
ie7x7	102	455.2	448	220+14	2.51
ie10x10	162	621.5	791	382+20	3.66
ie50x50	2682	14142.2	14143	13202+100	4.12

TABLE 6. Synthesis Results of DDU

6.6. **Timing Performance.**

Although the software cycle (mostly bus cycle) and the hardware cycle (delay between matrix cells and decide cell throught weight cells) are different, it can still be used to compare the design performance in a conservative way, since the hardware deadlock detection unit has less than 10 logic gate levels between weight cells. The VCS simulation and ISA emulation are carried out assuming an ideal situation (ideal CPI, no pipeline stalls, no cache misses, no bus arbitration or wait period, no memory read or write cycles, no interrupts). This assumption is applicable to ISA emulator that does not have a cache, memory, nor bus model. Also, such situation serves as the best case for comparison.

First, both hardware and software deadlock detections are modeled in Matlab to understand the properties of the algorithm and to give a rough initial performance an estimation. This rough estimate points out $\frac{O_{sw}(n \times m)}{O_{hw}(min(n,m))}$ performance improvement (see proof of Theorem 6), where m is the number of processors and n is the number of resources. Second, the software deadlock detection algorithm is implemented and tested in C, and hardware deadlock detection algorithm is implemented and tested in Verilog. Third, a codesign environment is developed using Tcl/Tk and "expect" to interact with and extract information from VCS Verilog simulator and ISA ARM7TDMI emulator. The experiments show that software algorithms has about 1000 to 800 software cycles and the hardware algorithm has about 10 cycles (which makes 99% run time improvement).

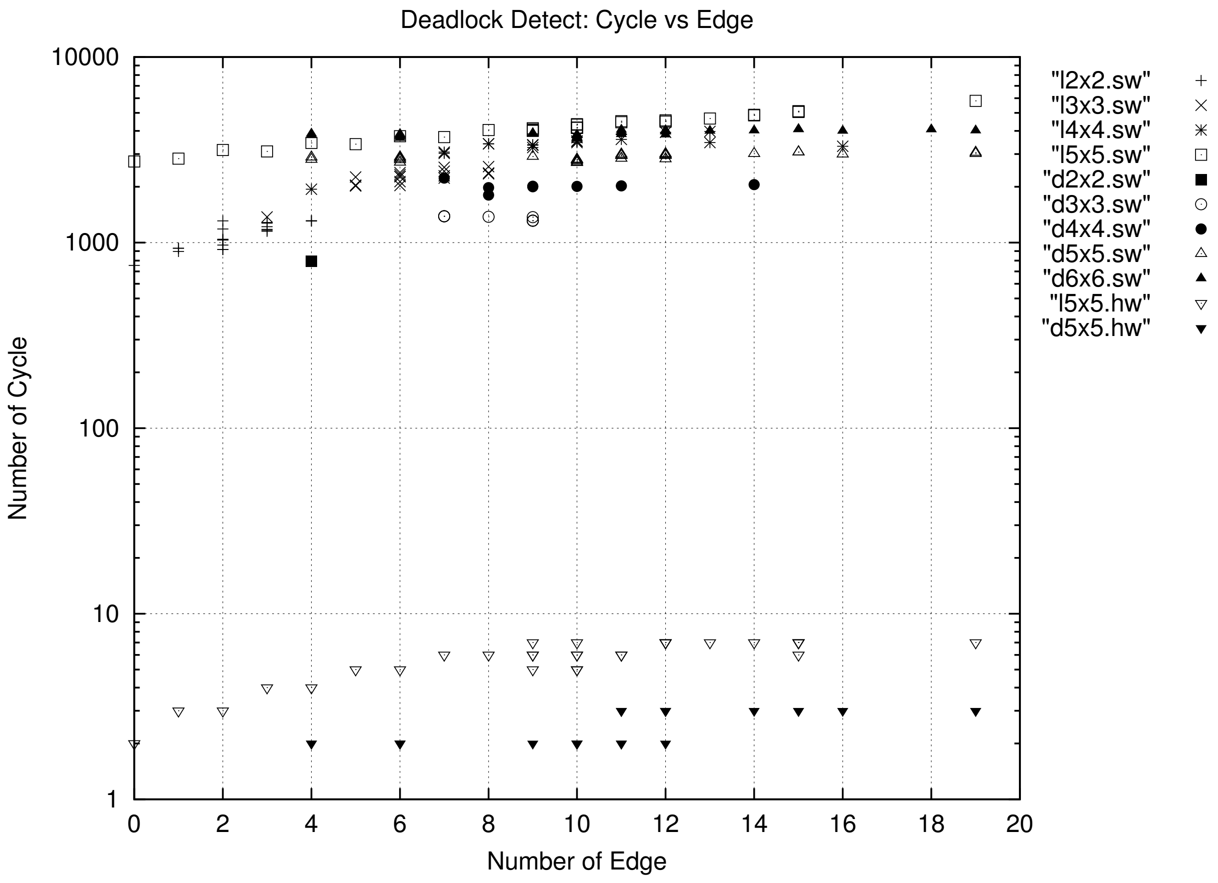


FIGURE 31. Hardware vs Software Run Time Complexity of the Deadlock Detection.

Note: In Figure 31 “Number of Cycles” means hardware cycles when used for hardware run time, and software cycle when used for software run time. “Number of Edges” means the total number of both request and grant edges in a given matrix representation. The plot in the top part, showing the run time complexity of the software deadlock detection algorithm, is about 1000 cycles and more. Also the range of run time complexity is about 1000 to 5000 cycles, which makes the timing characteristics of a system hard to predict. The plot in the bottom half, showing the run time complexity of the hardware deadlock detection, is about less then 10 cycles. The range of run time complexity is also less than 10 cycles.

7. Conclusion

In this report, a new parallel algorithm is proposed and proven. The main difference of the new algorithm is that the new parallel algorithm deals with the dangling path instead of explicitly finding out the exact cycles. Most of the previous algorithms [1]-[6] require back-tracking if a dead-end path is found. That increases computation time. However, the proposed algorithm implicitly finds out if a given system state contains a cycle without actually tracing a path. The proof of the new algorithm is based on several observed properties that are not addressed by the path-based tracing algorithms.

References

[1] M. Maekawa, A. E. Oldhoeft and R. R Oldehoeft, Operating Systems - Advanced Concepts, Benjamin-Cummings Pub., 1987.

[2] R. C. Holt, Some Deadlock Properties of Computer Systems, ACM Computing surveys, Vol.4, No. 3, september 1972.

[3] A. Shoshani. and E. G. Coffman Jr., Detection, Prevention and Recover From Deadlocks in Multiprocess, multiple resource systems, Proc. 4th Annual Princeton Conf. on Information Sciences and System, Mar. 1970.

[4] I. Cahit, Deadlock Detection Using (0,1)-Labeling of Resource Allocation Graphs, IEE Proceedings Compute. Digit. Tech. Vol. 145, No. 1, January 1998.

[5] F. Belik, An Efficient Deadlock Avoidance Technique, Trans., 1990, C-38 (7), pp. 882-888.

[6] J. G. Kim and K. Hoh, An O(1) Time Deadlock Detection Scheme in Single Unit and Single Request Multiprocessor System, IEEE TENCON'91, Vol 2, pp. 219-223, August , 1991.

[7] Giovanni De Micheli, Synthesis and Optimization of Digital Circuits, McGraw-Hill, 1994.

[8] Lennard Lindh, FASTCHART = Idea and Implementation, IEEE, International Conference on Computer Design (ICCD), Boston, USA, October 1991.

[9] http://www.cis.temple.edu/˜ingargio/cis307/index.html

[10] Giovanni De Micheli, Chapter 2 of Synthesis and Optimization of Digital Circuits, McGraw-Hill, 1994.

[11] David A. Patterson, John L. Hennessy, Computer Architecture: A Quantitative Approach, Second Edition Morgan Kaufmann Pub., 1996.

[12] Pun H. Shiu, and Vincent J. Mooney III, The Pricinple of Parallel Deadlock Detection, technical report GIT-CC-00-30, december 24, 2000, Georgia Institute of Technology.