LIVENESS OF MARKED GRAPHS AND COMMUNICATION AND VLSI SYSTEMS REPRESENTED BY THEM

Mohamed G. Gouda

Department of Computer Sciences
The University of Texas at Austin

TR-163 December 1980

ABSTRACT

Marked graphs are a graph model to represent and study systems of parallel computations and communicating processes. Live marked graphs correspond to systems without deadlocks. An $O(n^3)$-time algorithm to examine the liveness of a marked graph with n nodes is presented. The algorithm is based on a set of reduction rules which can be applied repeatedly to a live marked graph to reduce it to an empty graph.

A subclass of marked graphs called connection graphs is identified to represent the relationships between communicating processes. A reduction algorithm to examine liveness of connection graphs is presented; its time is $O(n)$ where n is the number of nodes in the given connection graph.

A subclass of connection graphs, called array graphs, is identified to represent the relationships between communicating cells in a VLSI array. The array cells in an array graph are identical; and two arrays graphs with identical array cells are said to belong to the same array graph family. We prove that any array graph with arbitrary shape and size is live iff a minimal array graph of the same array graph family is live. This yields an $O(n)$-time algorithm to examine the liveness of all members of an array graph family, where n is the number of nodes in a single array cell.

Keywords: Communicating processes, deadlock-free communication, marked graphs, liveness, parallel computation models, polynomial-time algorithms, VLSI arrays.

I. INTRODUCTION

Petri nets and related graph models have been proposed to represent [1], [2], analyze [3], [4], and synthesize [5], [6] systems of parallel computations and communicating processes. For a comprehensive presentation of Petri nets and their use in system modelling, analysis, and synthesis, the reader is referred to [7]. A subclass of Petri nets called marked graphs has been shown sufficient [8] to represent a variety of such systems; e.g., binary communication protocols, pipelined operations, and parallel CPU and disk activities. Later in this paper we discuss two other examples where sets of communicating processes and arrays of VLSI are represented as marked graphs.

Live marked graphs represent systems which are free of deadlocks. Therefore, to examine whether or not a given system is deadlock-free is equivalent to examining whether or not its marked graph representation is live. If the marked graph is large; i.e., has a large number of nodes, it is important to use an efficient algorithm to examine its liveness. Such an algorithm is especially important during the design stages of the given system to explore the effects of many design decisions on system liveness.

In this paper, we present three polynomial-time algorithms to examine the liveness of marked graphs and two interesting subclasses of marked graphs called connection and array graphs. A connection graph models a set of communicating processes, while an array graph models a set of identical communicating cells in a VLSI array. The algorithm for marked graphs, discussed in sections II and III, requires a time of $O(n^3)$ where n is the number of nodes in the graph. The algorithm for connection graphs, discussed in sections IV and V, requires a time of $O(n)$ where n is the number

of nodes in the graph. The algorithm for array graphs, discussed in sections VI and VII, requires a time of O(n) where n is the number of nodes in a single cell in the graph regardless of the graph's size or shape.

II. LIVENESS OF MARKED GRAPHS

A marked graph is a directed graph with a nonnegative number of tokens assigned to each directed edge. A node in a marked graph is said to be enabled if each of its input edges has at least one token. A node without input edges is always enabled. An enabled node may fire; the firing of an enabled node consists of removing one token from each of its input edges and adding one token to each of its output edges.

Let M be a marked graph. A state s (or an initial state s_0) of M is defined by the number (or the initial number, respectively) of tokens on each directed edge in M. A state s is reachable if there exists a node firing sequence which transforms the initial state s_0 to s. A state s of M is live if every transition in M is enabled at s or can be enabled through some sequence of firings starting from s. M is live if each of its reachable states is live. The following theorem is proved in [9].

Theorem 1:

A marked graph M is live iff every directed cycle in M has at least one token.

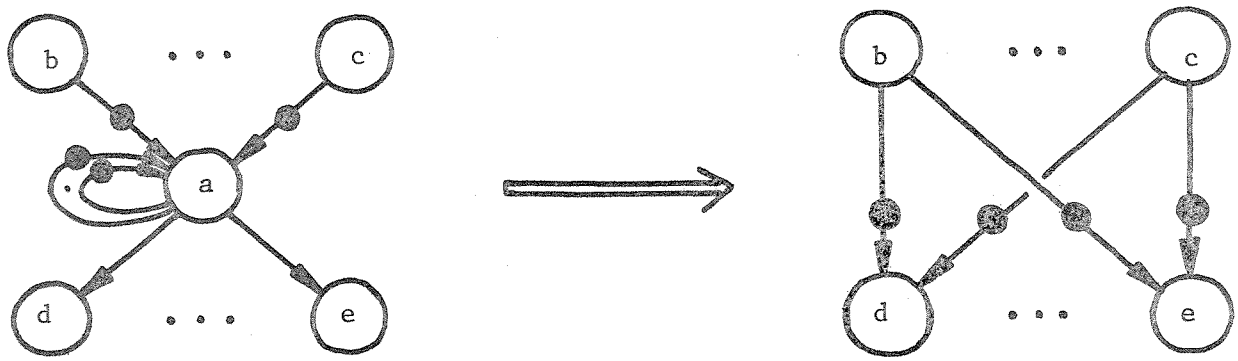
Based on this theorem, the liveness of a marked graph M can be examined by generating all the directed cycles in M and counting the number of tokens on each of them. Since the number of directed cycles in M is exponential in n [10], where n is the number of nodes in M, the execution time of this algorithm is exponential in n. In the next section, we present a polynomial-time algorithm to examine liveness of marked graphs. It is based on a set of four reduction rules which can reduce a live marked graph to an empty graph; i.e., one with no nodes and no edges.

III. A POLYNOMIAL-TIME ALGORITHM FOR LIVENESS OF MARKED GRAPHS

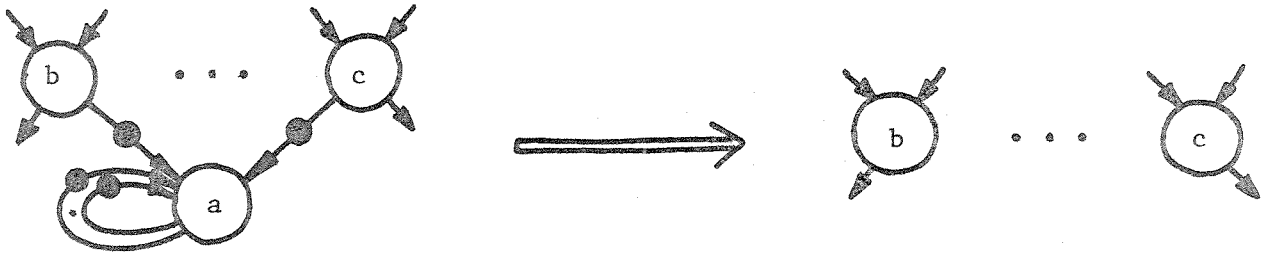
Consider the four reduction rules in Figure 1; each of them can be applied to remove one node from a marked graph. Notice that any rule is applicable to remove node "a" only if each input edge of "a" has at least one token. Whether or not the output of node "a" have tokens is irrelevant. Rule R1 (Figure 1a) is applicable when node "a" has inputs and outputs other than its self-loops. Rule R2 (Figure 1b) is applicable when node "a" has no outputs other than its self-loops. Rule R3 (Figure 1c) is applicable when node "a" has no inputs other than its self-loops. Rule R4 (Figure 1d) is applicable when node "a" has neither inputs or outputs other than its self-loops. In each case, the rule is applied by removing node "a" along with its inputs and outputs; but in rule R1 edges are added as follows. An edge with a token is added from any node immediately preceeding node "a" to any node immediately succeeding node "a". Based on these four reduction rules, the following reduction algorithm can be applied to a marked graph M to examine its liveness.

Algorithm 1

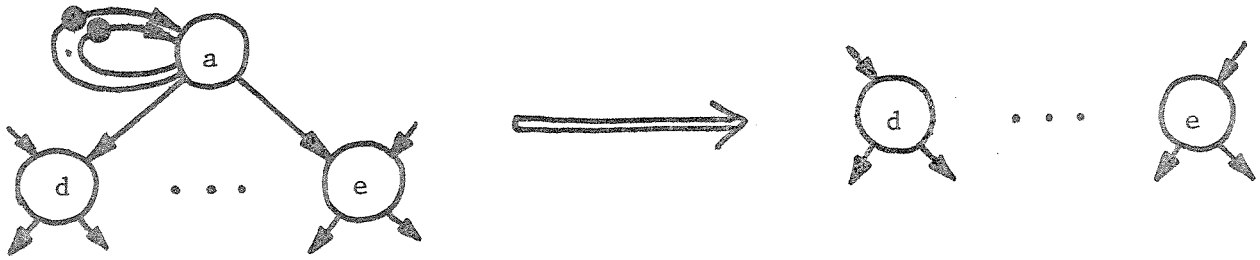
```
temp := M;
do [ R1 is applicable to "temp" → temp := R1 (temp)
   □ R2 is applicable to "temp" → temp := R2 (temp)
   □ R3 is applicable to "temp" → temp := R3 (temp)
   □ R4 is applicable to "temp" → temp := R4 (temp)
   ] od;
result M is live iff temp = empty
end
```



(a) R1



(b) R2



(c) R3

(d) R4

Figure 1. Reduction rules for marked graphs

Notes:

(i) Variable "temp" is of type marked graph; it stores the intermediate marked graph during the reduction. At the beginning, "temp" stores the original marked graph M. At the end, "temp" is the empty graph; i.e., one with no nodes and no edges, iff M is live, as shown in Theorem 2 (below).

(ii) The notation: do [...□ $guard_i \rightarrow statement_i$ □...] od defines a repetitive guarded command as introduced by Dijkstra [11]. When the boolean expression $guard_i$ is true, $statement_i$ may be executed. When all guards are false, the execution control moves to the next statement.

(iii) The statement: temp := Ri (temp) means apply the reduction rule Ri to the marked graph "temp" and store the resulting marked graph back in "temp".

Now, we prove that Algorithm 1 is both correct (Theorem 2) and efficient (Theorem 3).

Theorem 2:

If Algorithm 1 is applied to a marked graph M, then

(i) it terminates, and

(ii) it yields an empty marked graph iff M is live.

Proof:

(i) Termination:

Each application of a reduction rule reduces the number of nodes in M by one. Since M has a finite number of nodes, Algorithm 1 must terminate.

(ii) Final marked graph is empty iff M is live:

By inspecting the reduction rules in Figure 1, it is straightforward to show the following three properties.

P1: No reduction rule adds new cycles to the marked graph; i.e., any cycle in the graph after some sequence of reductions corresponds to a cycle in M.

P2: No reduction rule adds tokens to cycles without tokens before the reduction; and no reduction rule removes all the tokens from a cycle with tokens before the reduction. Hence, any cycle with tokens in the graph after some sequence of reductions corresponds to a cycle with tokens in M. Moreover, any cycle without tokens in the graph after some sequence of reductions corresponds to a cycle without tokens in M.

P3: Any reduction rule removes only self-loops with tokens. From P_2, these self-loops correspond to cycles with tokens in M_1.

If Part:

Assume that M is not live. From Theorem 1, M must have at least one cycle C without tokens. From P2, no reduction rule can add tokens to C. From P3, no reduction rule can remove C. Therefore, the final marked graph is not empty since it contains C at least.

Only If Part:

Assume that the final marked graph M' is not empty. M' does not contain a node with tokens on all its inputs, otherwise one of the rules R1, R2, R3, or R4 can still be applied to M' and reduce it even further. Hence, no node can fire in M', and M' is not live. From Theorem 1, M' must have at least one cycle C without tokens. From P1 and P2, M must also have cycle C without tokens. From Theorem 1, M is not live. □

Theorem 3:

The execution time of Algorithm 1 when applied to a marked graph M is $O(n^3)$, where n is the number of nodes in M.

Proof:

Since each reduction removes exactly one node from M, Algorithm 1 performs at most n reductions when applied to M. Each reduction consists of the following three steps. First, traverse and examine the edges in the marked graph to check whether or not any reduction is applicable. Second, remove one node and its inputs and outputs. Third, add new edges if needed; i.e., when reduction rule R1 is applied. The execution time of all three steps is $O(n^2)$; and the execution time of Algorithm 1 is $O(n^3)$.

The above result is based on the assumption that the total number of edges in the marked graph is $O(n^2)$ after any sequence of reductions. This is valid if there are no duplicate edges between any two nodes in the graph. To guarantee this condition, duplicate edges should be detected and removed from the original graph M and after each application of reduction rule R1 which adds new edges to the marked graph. Duplicate edges from node n_1 to node n_2 in a marked graph are removed as follows. If all edges from n_1 to n_2 have tokens, then remove all of them except one, any one. Otherwise, keep exactly one edge without tokens from n_1 to n_2. It is straightforward to show that this removal of duplicate edges does not affect the liveness of a marked graph. □

Algorithm 1 can be made more efficient as follows. First, detect the strongly connected components* in the given marked graph M. Then, show that each of them is live by reducing it to an empty graph using only

---

*A strongly connected component m of a marked graph M satisfies the following three conditions: (i) m is a marked subgraph of M, where the tokens on each edge e in m are the same as those on e in M. (ii) m is strongly connected; i.e., there is a directed path from any node to any other node in m. (iii) If m_1 is any other strongly connected component of M, then m_1 does not have any common nodes with m.

reduction rules R1 and R4. Finally, it can be shown that M is live iff all its strongly connected components are live. This modification makes the algorithm more effecient since in many cases strongly connected components represent a small part of M; however, in the worst case the execution time for the algorithm is still $O(n^3)$. We leave the details to the reader.

For any given subclass of marked graphs, it may be possible to find another set of reduction rules to speed-up the reduction algorithm for that particular subclass. In the next section, we identify such a subclass; members of that subclass are called connection graphs. A connection graph is a marked graph which represents the relationships between a number of communicating processes, called cyclic processes. The communications between cyclic processes in a connection graph are free of deadlocks iff the connection graph is live.

IV. CONNECTION GRAPHS

A cyclic process is a directed graph with two types of nodes, called sending and receiving nodes, and two types of directed edges, called links and buffers such that the following four conditions hold:

(i) Each sending node is represented as a rectangular node, and has one input link, one output link and, one output buffer (Figure 2a).

(ii) Each receiving node is represented as a circular node, and has one input link, one output link, and one input buffer (Figure 2b).

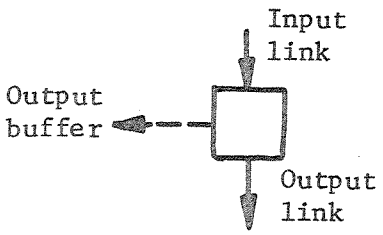
(iii) One of the nodes in the process, identified by a small arrow input to it, is called the initial node.

(iv) The links, and the sending and receiving nodes form a directed cycle, whereas the buffers represent the inputs and outputs of this cycle.

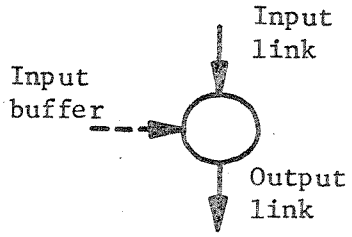
Figure 2c shows a cyclic process P. It has five nodes, three of them are sending nodes, and the other two are receiving nodes. Receiving node 1 is the initial node of P.

A sending (or receiving) operation is associated with each sending (or receiving) node. The associated operation can be executed only when its node is "enabled". At the beginning, the initial node is enabled. Then, after its associated operation is executed, the following node in the process is enabled, and so on. The "enabling" sequence continues such that at any instant only one node in the process is enabled. To identify the enabled node at any instant, a token, called the control token is placed on the node input link. In Figure 2c, the control token is placed on the input link of the initial node.

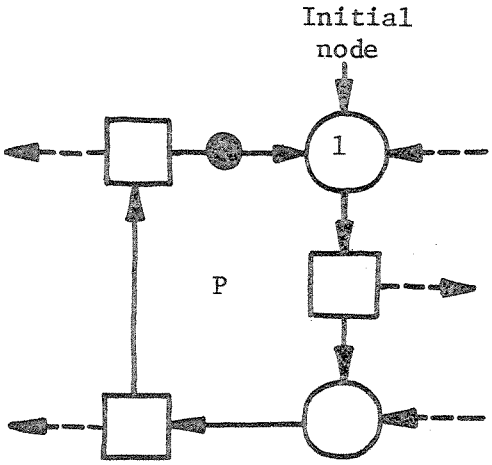
A sending node in a cyclic process is enabled if the control token is on



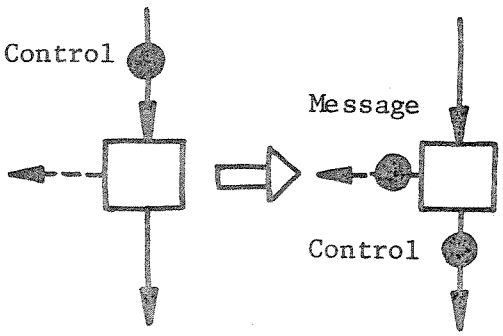
(a) Sending node



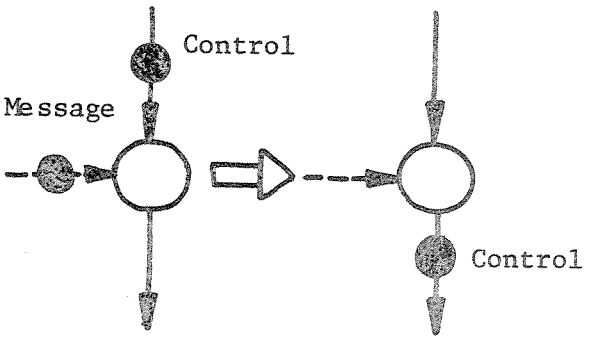
(b) Receiving node



(c) An example of a cyclic process



(d) Sending a message



(e) Receiving a message

Figure 2. The model of cyclic processes

its input link. In this case, the sending node can be executed. The execution consists of (i) moving the control token to the node's output link, and (ii) putting a message token on the node's output buffer (Figure 2d).

A receiving node in a cyclic process is enabled if the control token is on its input link. In this case, it can be executed only if there is at least one message token on its input buffer. The execution consists of (i) moving the control token to the node's output link, and (ii) removing one message token from its input buffer (Figure 2e).

Interactions between cyclic processes are achieved by exchanging message tokens via their buffers. This requires that processes share (or be connected by) common buffers.

A set of cyclic processes is called connected if any input (or output) buffer of one process is an output (or input, respectively) buffer of another process in the set. The directed graph showing the processes and their connecting buffers is called the connection graph of the set.

The state of a connection graph is defined by the number of tokens on each buffer, and the enabled node in each process. The initial state is a state in which all buffers are empty, and the enabled node in each process is its initial node. A state s is called reachable iff there is an execution sequence that leads from the initial state to s. As an example, Figure 3 shows a connection graph of three cyclic processes P_1, P_2, and P_3; the connection graph is in its first reachable state after its initial state.

A connection graph is in a deadlock state if in this state some of its processes satisfy the following two conditions:

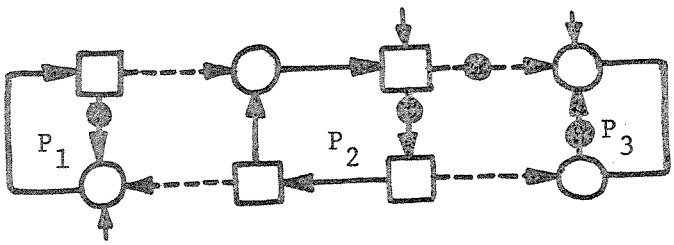


Figure 3. A connection graph of three cyclic processes P_1, P_2, and P_3.

(i) In this state, the enabled nodes in these processes are receiving nodes with empty input buffers.

(ii) In all the states which can be reached from this state, these buffers remain empty.

The communication in a connection graph G is deadlock-free iff G can never reach a deadlock state.

Any connection graph is a marked graph where sending and receiving nodes are simply nodes, buffers and links are just directed edges, and message and control tokens are simply tokens. It is straightforward to show that the communication in G is free of deadlocks iff G is live. Thus, one way to examine that the communication in G is deadlock-free is to use Algorithm 1 to examine the liveness of G. Another alternative is to develop a more efficient algorithm to examine liveness of connection graphs specifically. Such an algorithm is discussed in the next section. It is based on six reduction rules; each of them, when applied, removes two nodes from the connection graph. Thus, the algorithm performs at most $\frac{n}{2}$ reductions, where n is the number of nodes in the connection graph.

V. A POLYNOMIAL-TIME ALGORITHM TO EXAMINE LIVENESS OF CONNECTION GRAPHS

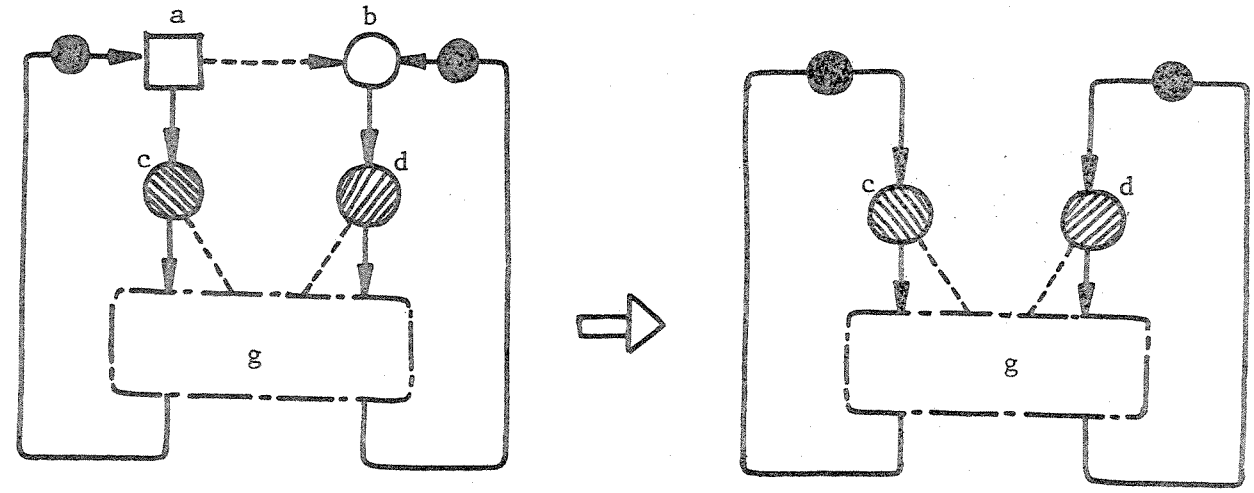
Consider the six reduction rules S1 to S6 in Figure 4, where g is that part of the connection graph not involved in the reduction. Each of these rules can be applied to remove a sending node "a" and the receiving node "b" connected to it via a buffer, provided that there is a control token on the input link of node "a". Whether or not the receiving node "b" has a control token on its input link is irrelevant.

Each reduction rule involves two cyclic processes. Rules S1 and S2 are applicable only if each of the involved processes has at least two nodes. Rules S3, S4, and S5 are applicable only if one process has exactly one node, and the other has at least two nodes. Rule S6 is spplicable only if each of the two involved processes has exactly one node. Based on these six reduction rules, the following reduction algorithm can be applied to any connection graph G to examine its liveness.

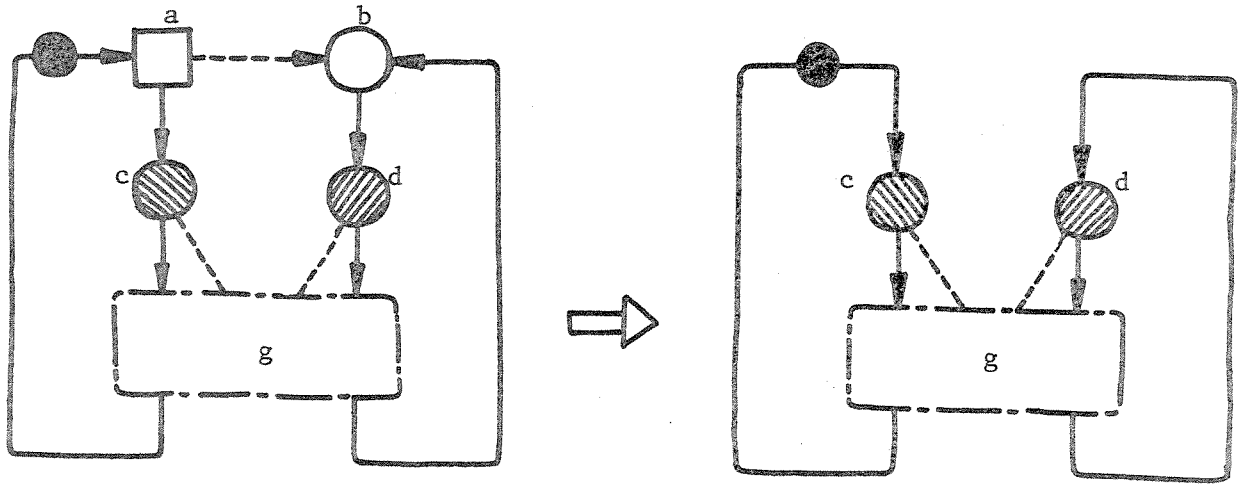
```
Algorithm 2:
     step 1:  temp := G;
     step 2:  find every sending node whose input link has a token
                  in temp:
              let N be the set of all those nodes;

     step 3:  while N is not empty do
                  remove any element n from N;
                  let  n' be the receiving node connected to n by a
                       buffer in temp;
                  apply the appropriate reduction rule to remove
                       buffer (n, n') from temp;
                  let  temp be the graph after reduction;
                  examine whether the reduction has produced* new
                          sending nodes whose input links have tokens
                          in temp;
                  if so, add these new nodes to N;
              endwhile

     step 4:  result communication in G is deadlock-free iff temp=empty
end
```

---

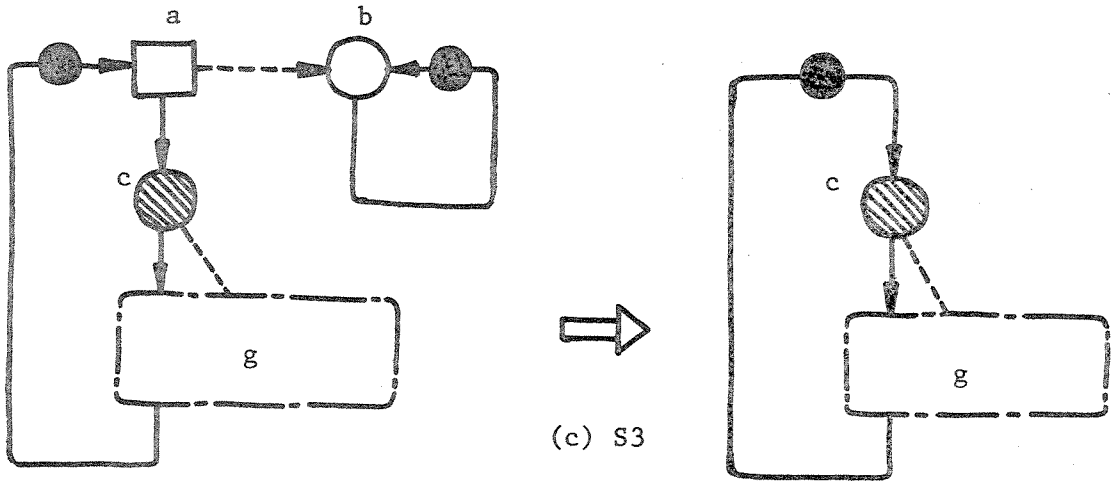
* Referring to Figure 4, reduction rule S1 can produce up to two such nodes; rules S2, S3, and S4 can produce at most one; but S5 and S6 cannot produce any such nodes.



(a) S1

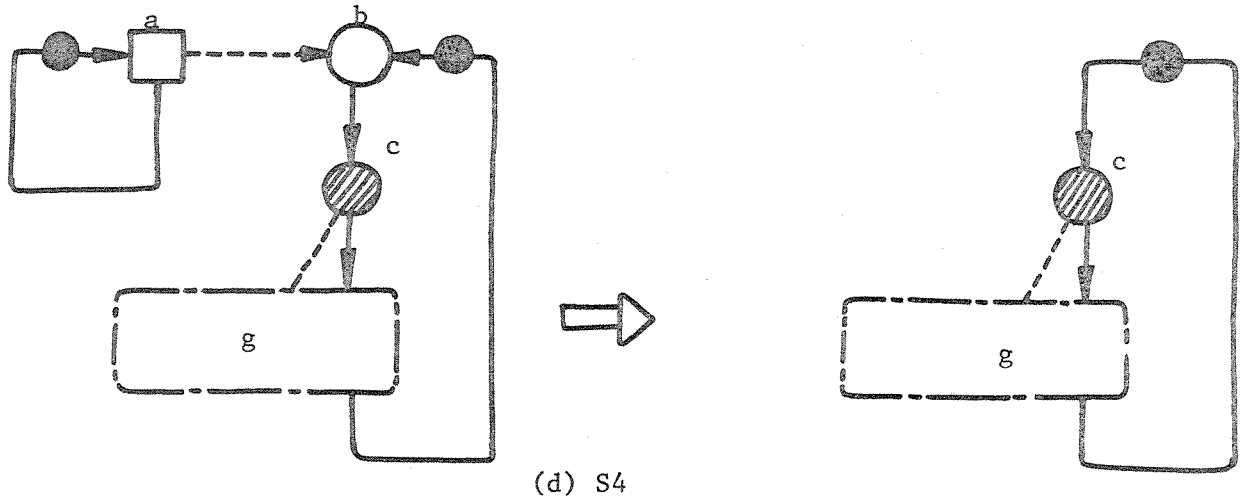


(b) S2

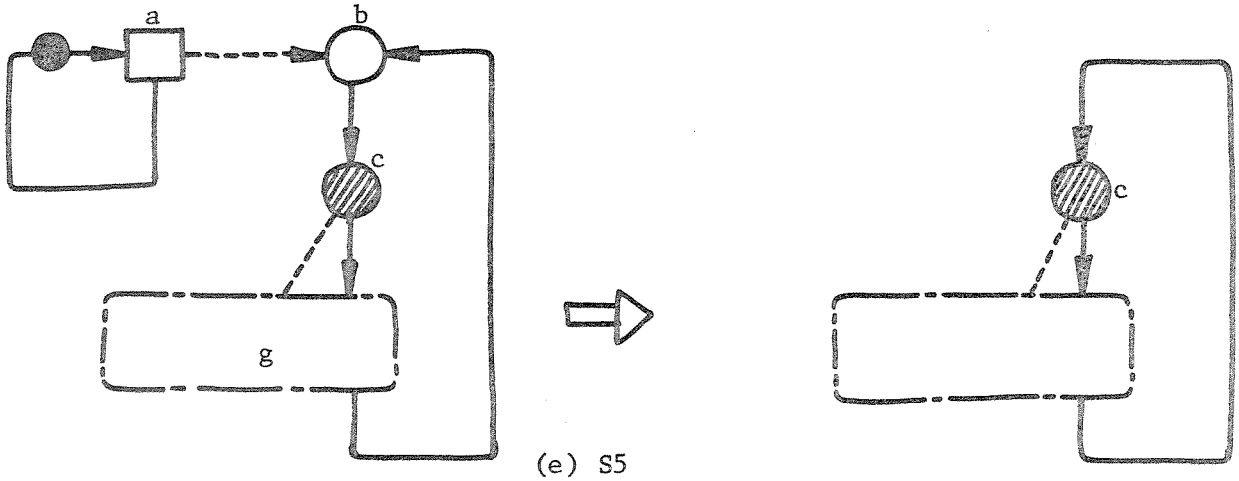


(c) S3

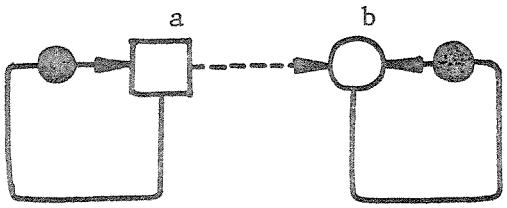
Figure 4. Reduction rules for connection graphs



(d) S4



(e) S5



empty

(f) S6

Figure 4. Continued

Now we prove that Algorithm 2 is correct (Theorem 4), and more efficient than Algorithm 1 (Theorem 5).

Theorem 4:

If Algorithm 2 is applied to a connection graph G, then

(i) it terminates, and

(ii) it yields an empty connection graph iff the communication in G is deadlock-free.

Proof:

(i) Termination:

Each application of a reduction rule reduces the number of nodes in the connection graph by two. Thus, the number of iterations in step 3 is at most $\frac{n}{2}$ where n is the number of nodes in G. Therefore, step 3 must terminate, and so does Algorithm 2.

(ii) Final graph is empty iff communication in G is deadlock-free:

Since each reduction rule is applicable only when a sending node has a token on its input link, step 2 finds all such nodes in the original graph G. For each such node, step 3 performs the appropriate reduction and finds any new sending node whose input link has tokens due to the reduction. This continues until no more reductions are possible. In effect, step 2 and step 3 perform all the possible reductions on G using rules S1 to S6.

If Part:

Any of the six reduction rules may remove cycles from the connection graph. We show that each of them removes only those cycles which contain tokens. Then, on reaching an empty graph, we conclude that each cycle in the original connection graph has at least one token; hence the original graph is free of deadlocks by Theorem 1. The reduction rules can be grouped into

three sets:

a. Rules S1 and S2:

Either of these two rules removes one buffer with its tail and head nodes. However, as shown in Figures 4a and 4b, each cycle which contains the buffer also contains at least one token.

b. Rules S3, S4, and S5:

Each of these rules removes one cycle which has one control token, as shown in Figures 4c, 4d, and 4e.

c. Rule S6:

This rule removes two cycles; each of them has one control token, as shown in Figure 4f.

Only If Part:

Assume that the communication in a connection graph is free of deadlocks. We want to show that the reduction rules can reduce the connection graph to an empty graph.

Assume that during the application of these rules to a connection graph G, we have reached a nonempty graph G' to which no rule can be applied. In G', no token is on the input link of a sending node, otherwise one of the transformations can be applied. All control tokens should be on the input links of receiving nodes with empty buffers. This is a deadlock state for G'; i.e., G' is not live. From Theorem 1, G' must have a cycle which does not have tokens. This cycle should also exist in the original connection graph G. Hence, the communication in the connection graph is deadlocked contradicting the assumption at the beginning. □

Theorem 5:

The execution time of Algorithm 2 when applied to a connection graph G is

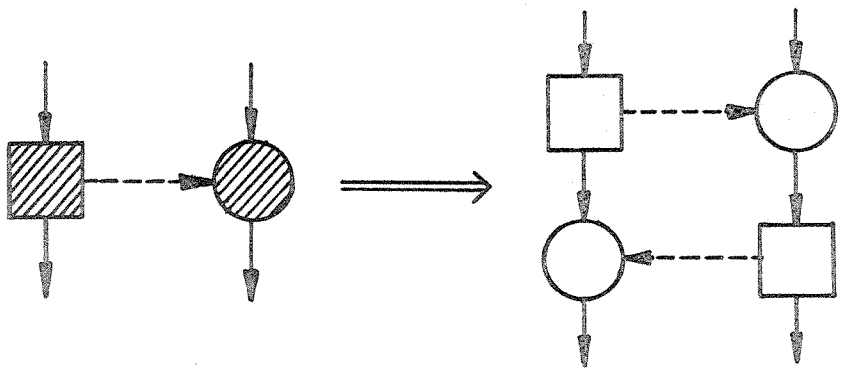
$O(n)$, where n is the number of nodes in G.

Proof:

The execution time for Algorithm 2 is dominated by that of step 3. In the worst case, step 3 executes $\frac{n}{2}$ iterations; each of them requires an execution time bounded by some constant regardless of the number of nodes or edges in G. Thus, the execution time of Algorithm 2 is $O(n)$. □

Processes in a connection graph communicate using a non-blocking send/blocking receive communication protocol; i.e., a process resumes execution immediately after sending a message and does not wait for the other process to receive it. In [12], Hoare employs a blocking send/blocking receive communication protocol where both the sending and the receiving processes wait until the message transmission occurs; then they both resume execution. If this protocol is used for the communication between cyclic processes, then the resulting class of systems can still be represented as connection graphs. Figure 5 shows how to represent a blocking send/blocking receive communication as two successive non-blocking send/blocking receive communications. The result is a connection graph with twice the number of nodes as in the original graph; thus, Algorithm 2 can still be applied to examine the liveness of the original graph in a linear time with respect to the number of nodes in the original graph.

In the next sections, we discuss how to use Algorithm 2 more efficiently to examine the liveness of connection graphs where most processes are identical. Such connection graphs are called array graphs; they are useful in modelling the communications within VLSI arrays [13].



(a) Blocking send/
blocking receive

(b) Non-blocking send/
blocking receive

Figure 5 Mapping from blocking send/blocking receive to non-blocking send/blocking receive.

VI. ARRAY GRAPHS

An array graph consists of a number of identical array cells and a number of end cells. Communications between the different cells in an array graph are assumed to be asynchronous. This assumption does not limit our study to asynchronous self-timed VLSI arrays [14]; the results are also applicable to synchronous arrays since if an asynchronous communication is deadlock-free then any synchronous version of that communication is also deadlock-free. In this section, we define array graphs with rectangular cells, then extend the results to arrays with hexagonal cells in the next section. These two types of cells are of special importance since they fully utilize the chip area. In [15], it is shown that many matrix operations can be realized using rectangular and hexagonal cells.

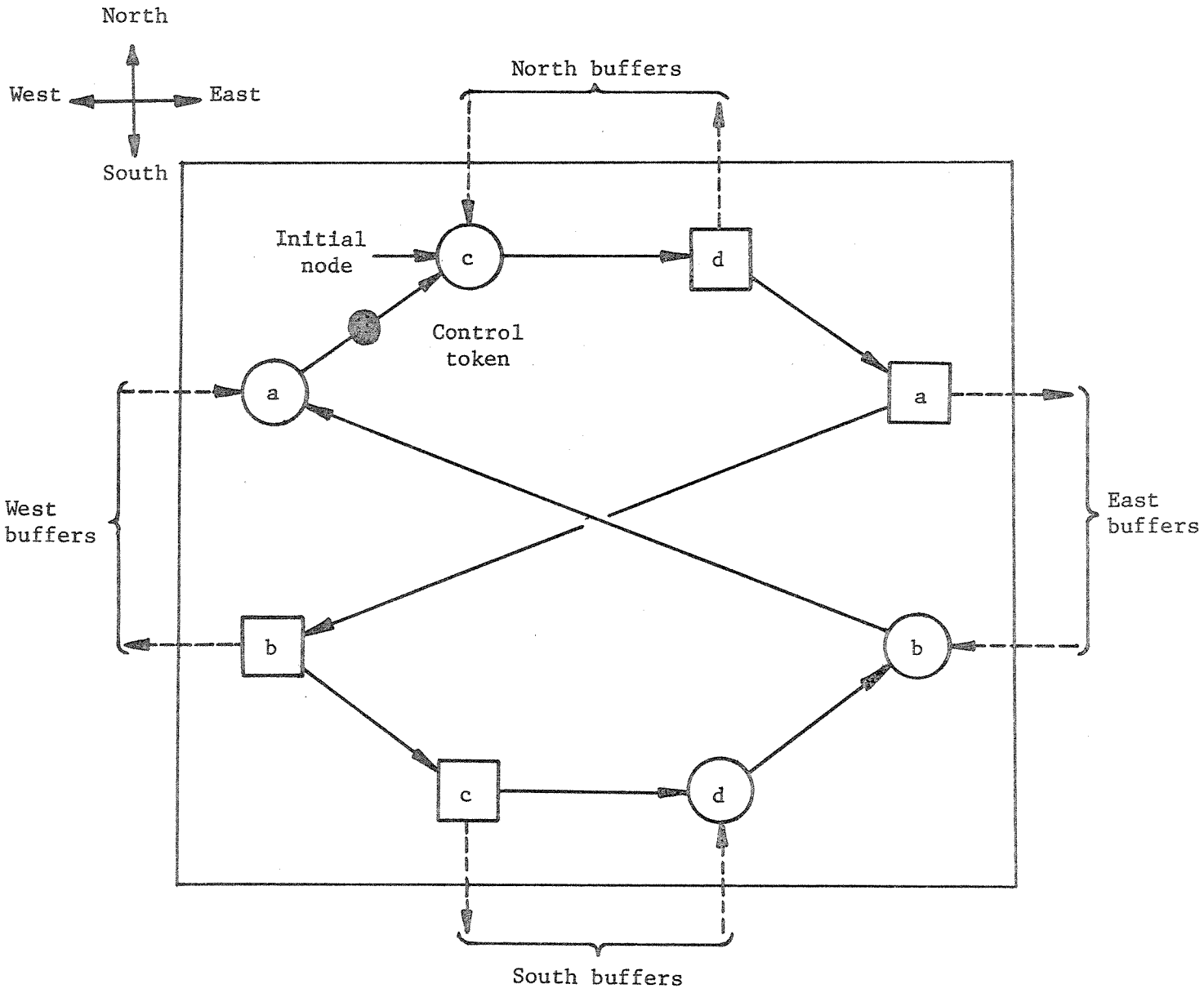
An array cell A is a cyclic process, as defined in section IV, with the following three additional conditions:

(i) All sending nodes in A are adjacent and followed by all receiving nodes.

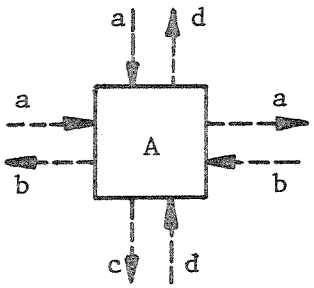
(ii) Each input or output buffer has a name and a direction out of the four directions: west, east, north and south.

(iii) For each input (or output) buffer, there is exactly one output (or input respectively) buffer with the same name and the opposite direction.

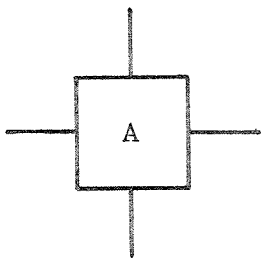
As an example Figure 6a shows an array cell A. It has four adjacent sending nodes followed by four adjacent receiving nodes. For convenience, the name of each input (or output) buffer is written inside the associated receiving (or sending) node in Figure 6a. Sending node "d" is the initial node; so the control token of A is on its input link initially as shown in Fig. 6a. Notice that cell A satisfies the above condition (iii); for



(a) Detailed



(b) Block



(c) Abstract

Figure 6 Different graphical representations for an array cell A

instance, A has an input west buffer named "a", and an output east buffer also named "a". Figures 6b and 6c show less-detailed graphical representations for array cell A.

Because of condition (iii), there is a one-to-one correspondence between west and east buffers, and between north and south buffers in an array cell. Therefore, identical cells can be connected such that the west (or north) buffers of one cell are connected to the east (or south respectively) buffers of another cell. Ultimately, this yields a mesh structure as shown in Figure 7a.

Let A be an array cell. A cyclic process W is called a west cell for A iff for each input (or output) buffer in W, there is exactly one output (or input respectively) west buffer in A, and vice versa. Similarly, define E, N, and S to be east, north, and south cells for A. The four cells W, E, N, and S are also called compatible end cells for A.

A west cell W can be connected to an array cell A by connecting each buffer in W to its corresponding west buffer in A, as shown in Figure 7b. Similarly, cells E, N, and S can be connected to array cells as shown in Figures 7c, 7d, and 7e.

Let A be an array cell; and let W, E, N and S be a compatible set of end cells. A connected directed graph G is called an array graph based on A, W, E, N, and S iff G is constructed by connecting cells identical to A, W, E, N, and S such that the following three conditions are satisfied:

(i) Each end cell is connected to one array cell.

(ii) Each array cell is connected to four cells, at most three of them are end cells.

(iii) Restriction: The west buffers of at least one array cell must be connected to the east buffers of another array cell; and the north buffers of at least one array cell must be connected to the south buffers of other array cells.

Restriction (iii) is only intended to simplify the presentation in the next section. But this restriction means that all the array cells in an array graph should not form a single one-dimensional array; because this restricted case is an important one, we will discuss it separately later in the next section.

Because of restriction (iii), a minimal array graph must have at least three array cells forming an L-shape as shown in Figure 8. There are four such minimal graphs.

For any compatible set of cells A, W, E, N, and S, a family of array graphs can be based. Members of such a family differ in their sizes and shapes; but each of them is based on the same set of cells. In the next section we show that every member in such a family is live iff a specific member, namely an L-shaped array graph (Figure 8), is live. Based on this result, we give an $O(n)$-time algorithm to examine the liveness of every member in an array graph family, where n is the number of nodes in a single array cell. Also in the next section we discuss similar results for one-dimensional array graphs and for hexagonal array graphs.

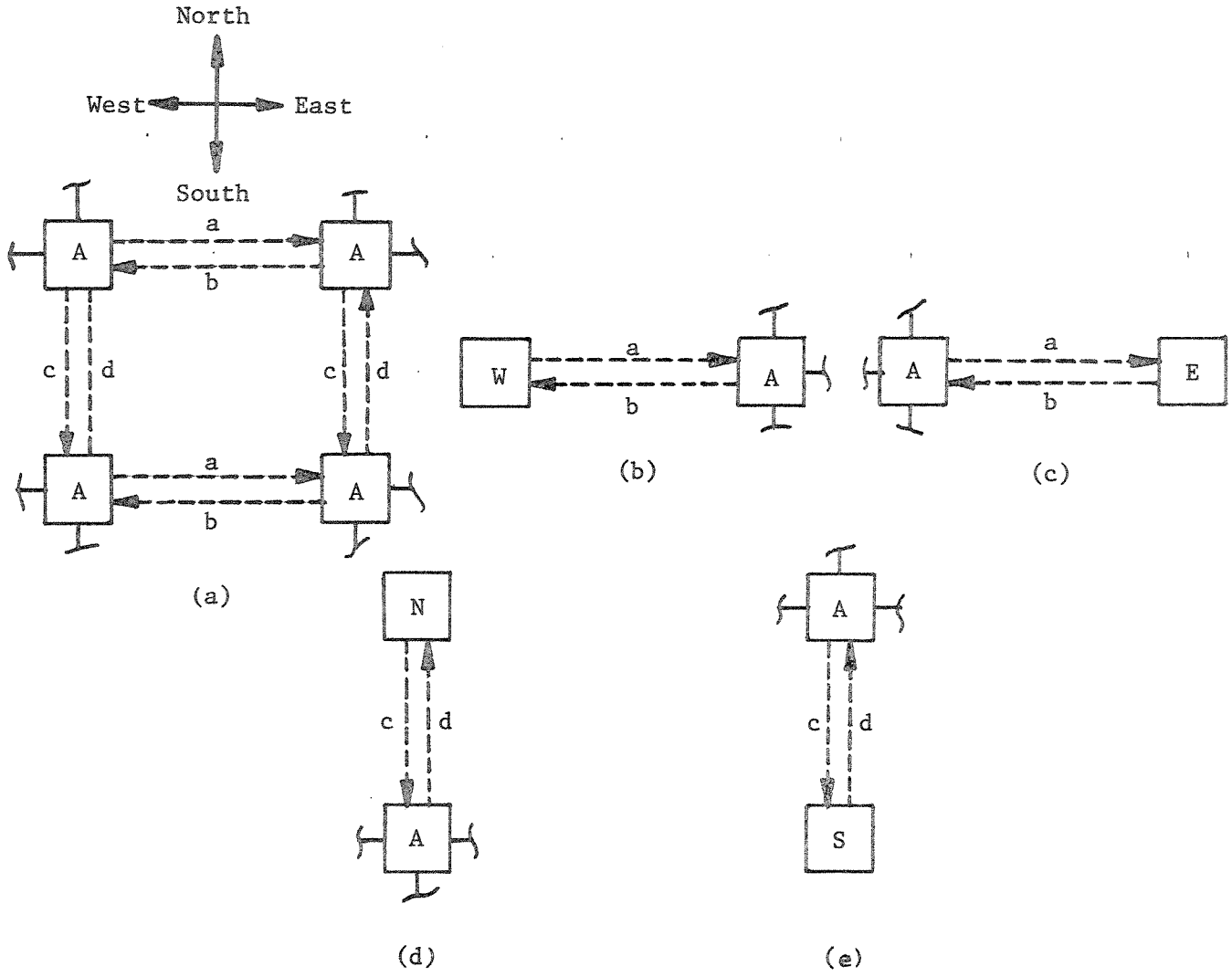


Figure 7 Five types of connections in an array graph.

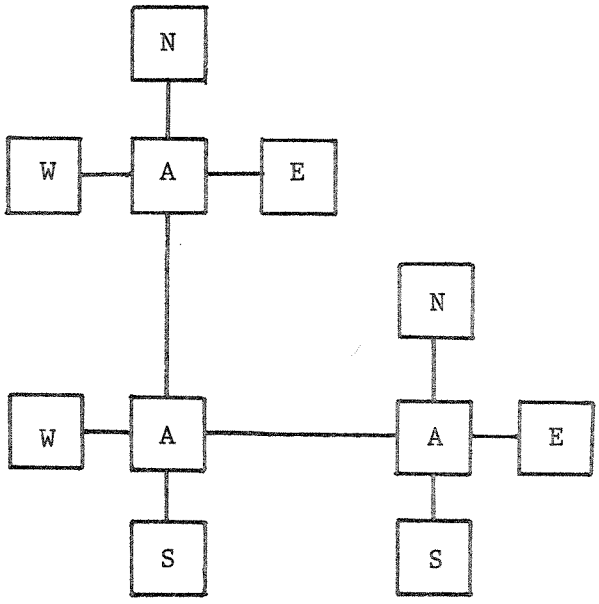


Figure 8 An L-shaped Array Graph. (A minimal array graph)

VII. LIVENESS OF ARRAY GRAPHS

Since array graphs are marked graphs, Theorem 1 holds for them as well. From Theorem 1, an array graph G is live iff every directed cycle in G has at least one token. There are two types of directed cycles in G, namely cycles which consist solely of links and cycles which have one or more buffers. Each cycle of the first type is contained completely in one cell and is guaranteed to have one token, namely the control token of that cell. Therefore, we need only consider cycles which have buffers.

Figure 9 shows a cycle with four buffers; it spans three cells E_1, E_2, and E_3. This cycle consists of four buffers "a," "b," "c," and "d" and four parts, called internal paths, inside the three cells. Each internal path starts with a receiving node and ends with a sending node. For example, the internal path in cell E_1 begins with the receiving node for input buffer "a" and ends with the sending node for output buffer "b"; this internal path is denoted [a,b>. Cell E_2 has two internal paths [b,c> and [d,a>; and cell E_3 has one internal path [c,d>. This cycle has a token iff at least one of the four internal paths [a,b>, [b,c>, [c,d>, and [d,a> has initially the control token of its cell. These concepts are useful to prove the following two lemmas which state, in principle, that if an array graph has a "large" cycle without tokens then it must also have a "small" cycle without tokens.

Lemma 1:

An array graph G has a cycle C_1, without tokens, which spans at least one array cell A and one end cell E iff G has a cycle C_2, without tokens, which spans exactly cells A and E.

Proof: is in Appendix I. □

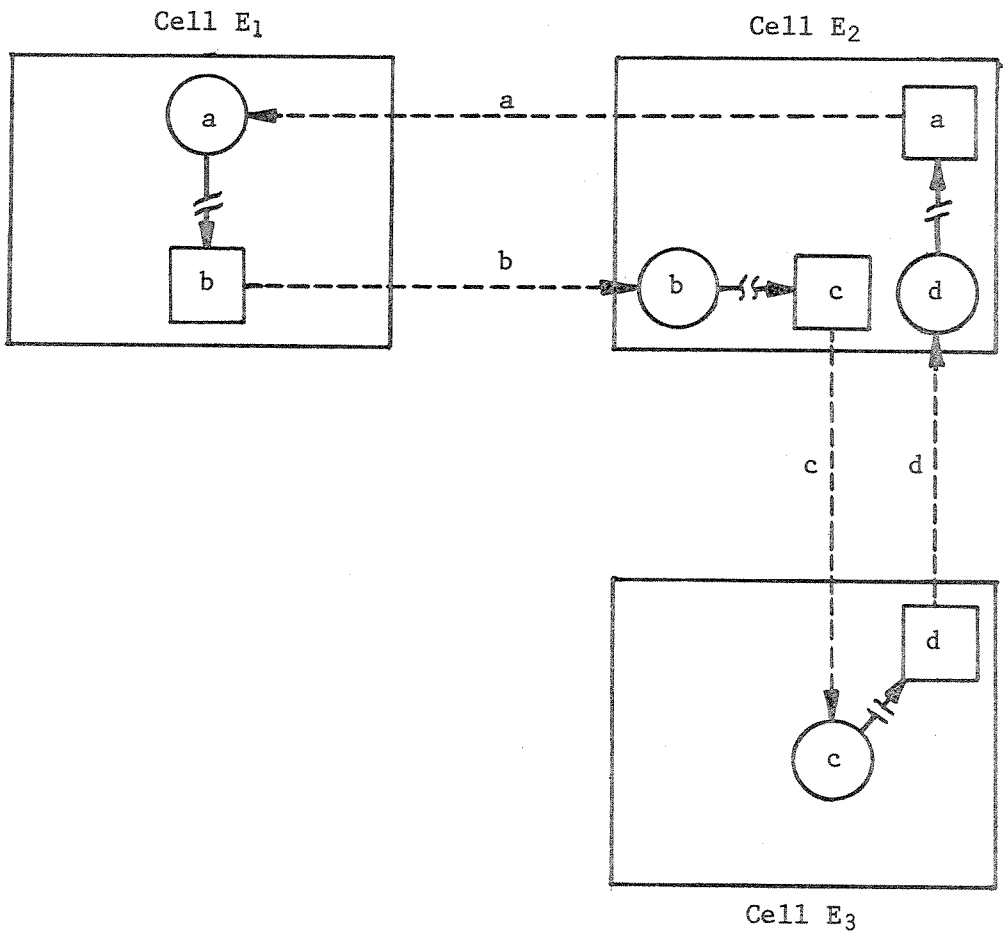


Figure 9 A cycle with four buffers "a", "b", "c", and "d", which spans three cells E_1, E_2, and E_3 in a rectangular array graph.

Lemma 2:

An array graph G has a cycle C_1, without tokens, which spans array cells only iff G has a cycle C_2, without tokens, which spans exactly two array cells.

Proof: is in Appendix I. □

From these two lemmas we can prove the following theorem.

Theorem 6:

Let G be an array graph; and let G_L be an L-shaped array graph (as in Figure 8) which is based on the same cells as G. Then,

G is live iff G_L is live.

Proof:

G is not live

iff G has a cycle C_1 without tokens (Theorem 1)

iff either C_1 spans at least one end cell

iff G has a cycle C_2, without tokens, which spans exactly one array cell and one end cell (Lemma 1)

iff G_L has a cycle, similar to C_2, without tokens

or C_1 spans array cells only

iff G has a cycle C_2, without tokens, which spans exactly two array cells (Lemma 2)

iff G_L has a cycle, similar to C_2, without tokens

iff G_L is not live (Theorem 1). □

Corollary 1:

An array graph G is live iff each member of the array graph family of G is live. □

Theorem 6 suggests the following algorithm to examine the liveness of an

array graph G based on cells A, W, E, N, and S.

Algorithm 3:

step 1: Construct an L-shaped array graph G_L based on the cells A, W, E, N, and S

step 2: Use Algorithm 2 to examine the liveness of G_L

step 3: result G is live iff G_L is live

end

From Corollary 1, Algorithm 3 examines, as a side effect, the liveness of every member of the array graph family of G. The following theorem establishes the time complexity of Algorithm 3.

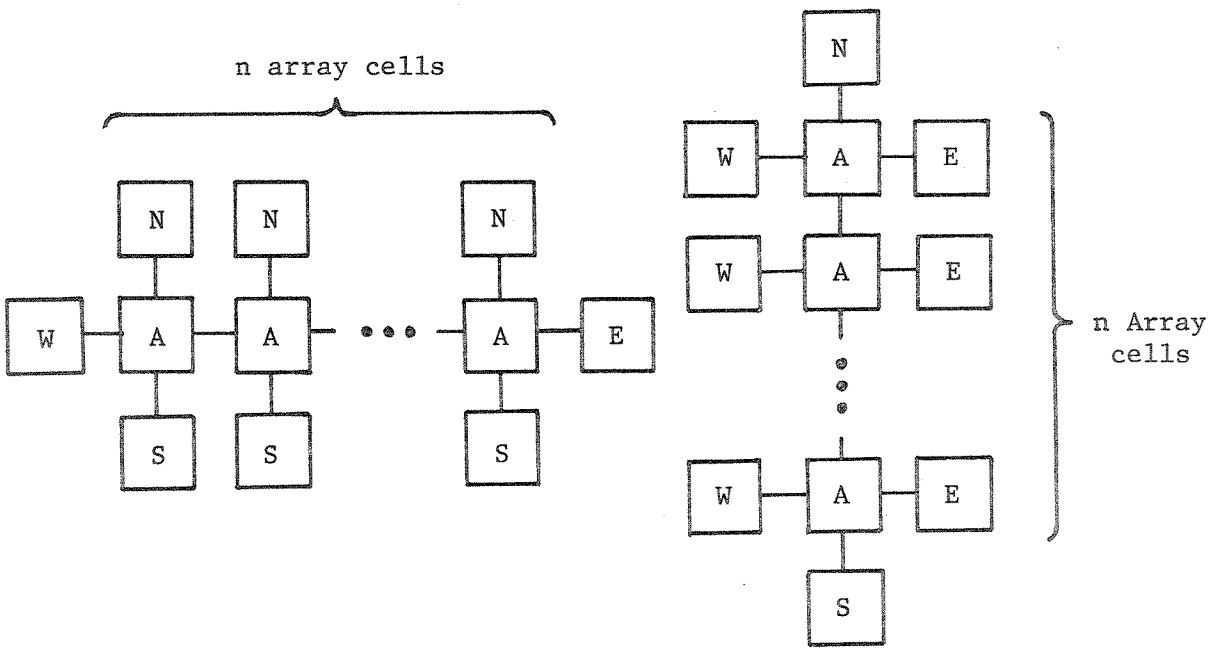
Theorem 7:

The execution time of Algorithm 3 when applied to an array graph G is O(n), where n is the number of nodes in a single array cell in G.

Proof:

The execution time of Algorithm 3 is dominated by that of its second step. From Theorem 5, the execution time is O(m), where m is the number of nodes in G_L. As shown in Figure 8, G_L consists of three array cells (of n nodes each), and eight end cells. Since every four end cells have n nodes, we get m = 5n. Therefore, the execution time of Algorithm 3 is O(n). □

Algorithm 3 does not apply to one-dimensional array graphs (namely horizontal and vertical array graphs shown in Figure 10) because of our restricted definition of array graphs in the previous section. Nevertheless, Lemmas 1 and 2 are applicable to any one-dimensional array graph $G_N(k)$ of order k where k is the number of array cells in the graph (Figure 10). These two lemmas can be used to prove the following theorem.



(a) Horizontal array graph $G_H(n)$ of order n ($n \geq 2$)

(b) Vertical array graph $G_V(n)$ of order n ($n \geq 2$)

Figure 10 One-dimensional array graphs of order n ($n \geq 2$)

Theorem 8:

Let $G_N(k)$ be a one-dimensional array graph of order $k(k\geq 2)$. $G_N(k)$ is live iff $G_N(2)$ is live.

Proof: is identical to that of theorem 6 except replace G by $G_N(k)$ and G_L by $G_N(2)$. □

Now we can modify Algorithm 3 by replacing G by $G_N(k)$ and G_L by $G_N(2)$ so that it becomes applicable to one-dimensional array graphs. Its time complexity remains linear in the number of nodes in one array cell.

The above discussion for rectangular array cells can be extended to hexagonal cells. A hexagonal array cell is a cyclic process with adjacent sending nodes followed by receiving nodes; every sending (or receiving) node is associated with an output (or input) buffer. Each buffer has a name and a "direction" from the following six directions: north, northwest, north-east, south, south-west, and south-east. For each input (or output) buffer with one direction, there exists an output (or input respectively) buffer with the same name but with the "opposite" direction, as illustrated in Figure 11. Two identical hexagonal array cells can be connected by connecting each north (or northwest or northeast) buffer in one cell to the corresponding south (or southwest or southeast respectively) buffer in the other cell, as illustrated in Figure 12a.

For any hexagonal array cell A, define a north end cell N to be a cyclic process such that for each north input (or output) buffer in A there is one output (or input) buffer in N. Similarly define NW, NE, S, SW, and SE to be a northwest, northeast, south, southwest, and southeast end cells for A. The connections between A and these end cells are illustrated in Figure 12.

Let A be a hexagonal array cell: and let N, NW, NE, S, SW, and SE be a

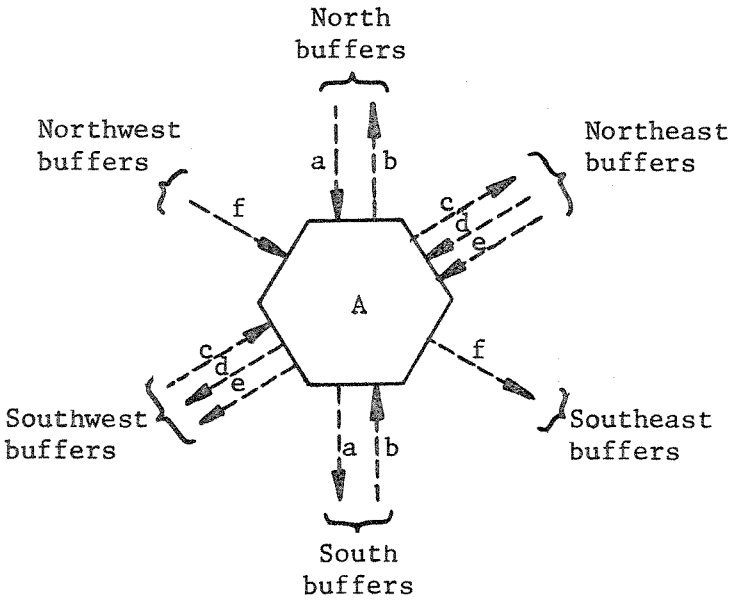


Figure 11 A block representation of a hexagonal array cell.

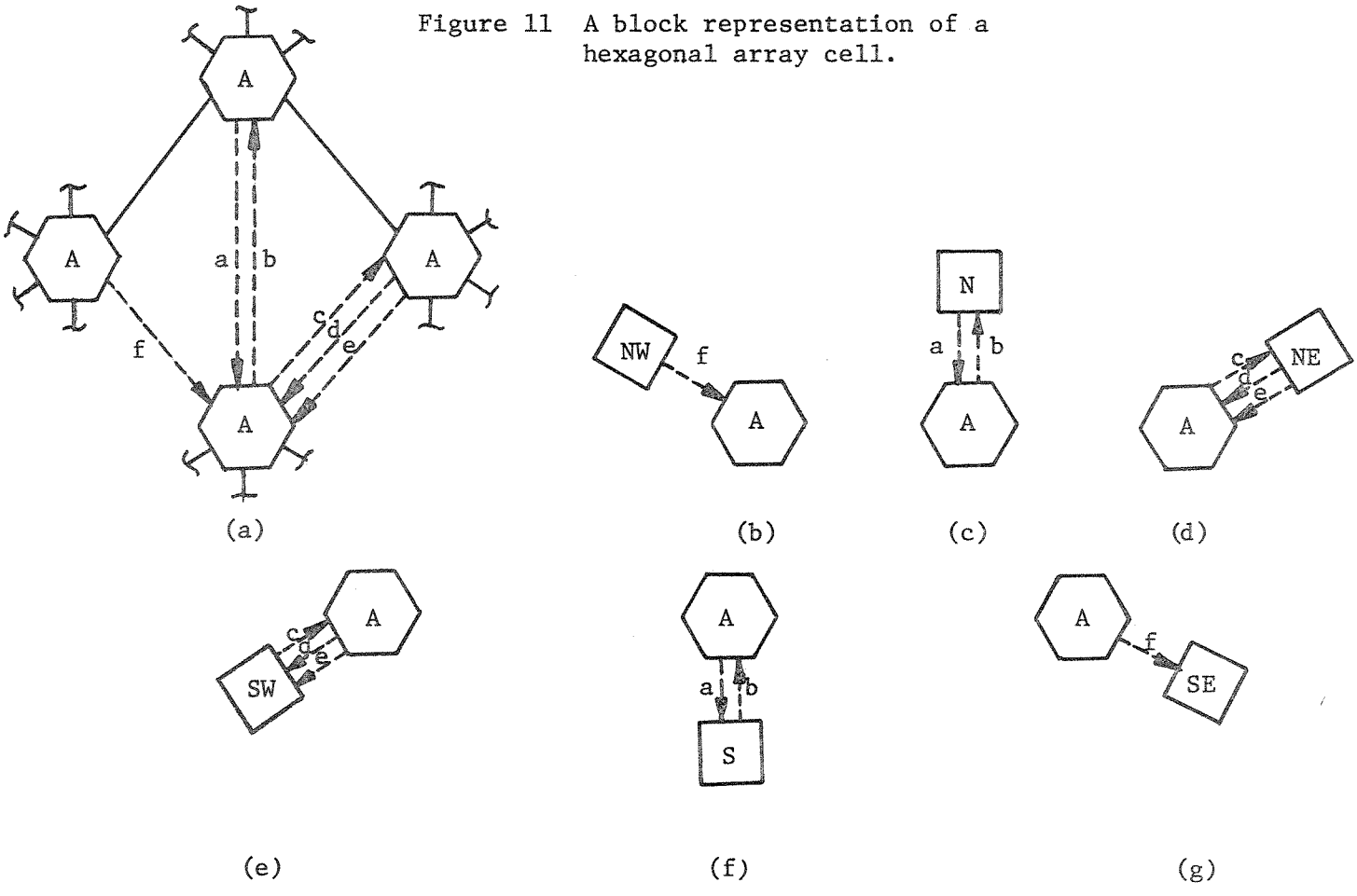


Figure 12 Different connections in a hexagonal array graph.

compatible set of end cells. A hexagonal array graph based on A, N, NW, NE, S, SW, and SE is a connected directed graph constructed by connecting cells identical to A, N, NW, NE, S, SW, and SE such that the following three conditions are satisfied: (i) Each end cell is connected to one array cell; (ii) each array cell is connected to six cells; and (iii) G must have at least four array cells connected as shown in Figure 12a. Because of condition (iii), the minimal hexagonal array graph based on A, N, NW, NE, S, SW, and SE is shown in Figure 13.

Lemma 1 for rectangular array graphs is also applicable to hexagonal array graphs with an identical proof. The following Lemma 3 and Theorem 9 for hexagonal graphs are analogous to Lemma 2 and Theorem 8 for rectangular graphs.

Lemma 3:

A hexagonal array graph G has a cycle C_1, without tokens, which spans hexagonal array cells only iff it has a cycle C_2, without tokens, which spans at most three hexagonal array cells.

Proof: In Appendix I. □

Theorem 9:

Let G be a hexagonal array graph; and let G_M be the minimal array graph based on the same cells as G. Then G is live iff G_M is live.

Proof: is identical to that of Theorem 6 except for replacing G_L by G_M, and replacing the reference to Lemma 2 by that for Lemma 3. □

From Theorem 9, an algorithm to examine the liveness of a hexagonal array graph G is identical to Algorithm 3 except for replacing G_L by G_M. The execution time for this algorithm is O(n) where n is the number of nodes in one array cell; the proof is similar to that of Theorem 8.

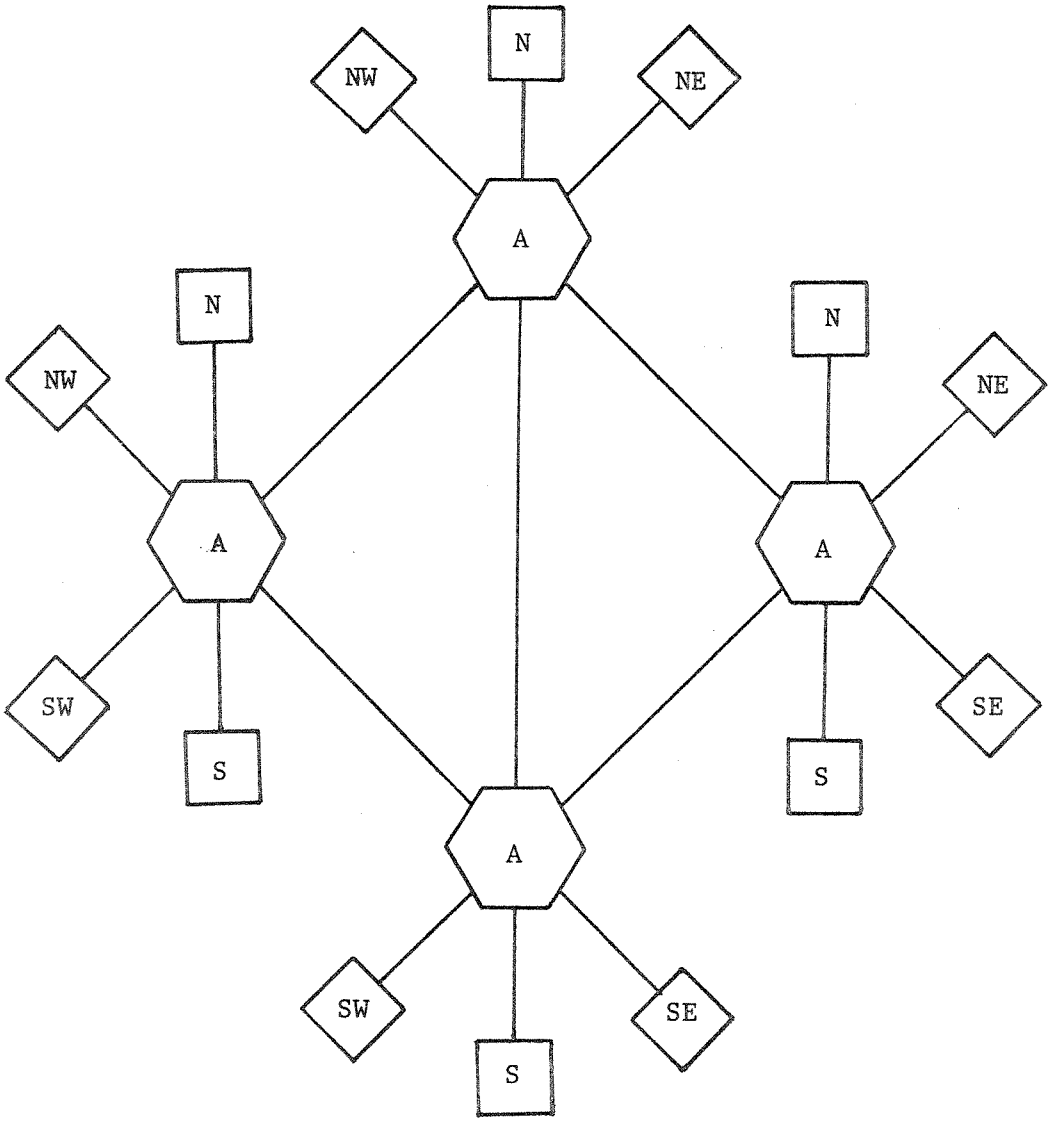


Figure 13 The minimal hexagonal array graph based on Cells A, N, NW, NE, S, SW, SE.

VIII. CONCLUSIONS

We have presented three algorithms to examine the liveness of marked graphs, connection graphs, and array graphs. The time complexity for the first algorithm is $O(n^3)$ where n is the number of nodes in the marked graph. The time complexity for the second algorithm is O(n) where n is the number of nodes in the connection graph. The time complexity for the third algorithm is O(n) where n is the number of nodes in one array cell.

The second and third algorithms are most useful during the design stages of communicating processes and VLSI arrays to ensure that their communications are deadlock-free.

One research problem is to extend the analysis in this paper to communicating processes whose structures are more complex than cycles of sending and receiving nodes.

ACKNOWLEDGEMENTS

The author is thankful to J. L. Peterson for his helpful comments and to Sheila Foster for her careful typing.

REFERENCES

[1] T. Agerwala, "Putting Petri Nets ot Work," Computer, Vol. 12, pp.85-94, Dec. 1979.

[2] P. M. Merlin, "A Methodology for the Design and Implementation of Communication Protocols," IEEE Tran. Commun., Vol. COM-24, pp. 614-621, June 1976.

[3] J. L. Peterson, "Petri Nets," Computing Surveys, Vol. 9, pp.223-252, Sept. 1977.

[4] C. V. Ramamoorthy and G. S. Ho, "Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets," IEEE Trans. Software Eng., Vol. SE-6, pp. 440-449, Sept. 1980.

[5] T. Agerwala and Y. C. Choed-Amphai, " A Synthesis Rule for Concurrent Systems," in Proc. 15th Design Auto Conf., Las Vegas, NV, June 1978, pp. 305-311.

[6] T. Murata, "Circuit Theoretic Analysis and Synthesis of Marked Graphs," IEEE Trans. Circuits Syst., Vol. CAS-24, pp. 400-405, July 1977.

[7] J. L. Peterson, Petri Net Theory and the Modelling of Systems, Prentice-Hall, Inc., Englewood Cliffs, 1981.

[8] T. Murata, "Synthesis of Decision-Free Concurrent Systems for Prescribed Resources and Performance," IEEE Trans. Software Eng., Vol. SE-6, pp. 525-530, Nov. 1980.

[9] F. G. Commoner et al., "Marked Directed Graphs," J. Comp. Syst. Sci., Vol. 5, pp. 511-523, Oct. 1971.

[10] E. M. Reingold et al., Combinatorial Algorithms: Theory and Practice, Prentice-Hall, Inc., Englewood Cliffs, 1977, pp. 348-349.

[11] E. W. Dijkstra, "Guarded Commands, Nondeterminacy, and Formal Deprivation of Programs," Comm. of the ACM, Vol. 18, No. 8, Aug. 1975, pp. 453-457.

[12] C. A. R. Hoare, "Communicating Sequential Processes," Communications of the ACM, Vol. 21, No. 8, pp. 666-667, Aug. 1978.

[13] C. Mead and L. Conway, Introduction to VLSI Systems, Addison-Wesley Publishing Company, Inc., 1980.

[14] C. L. Seitz, System Timing, Chapter 7 in reference [13], pp. 218-262.

[15] H. T. Kung and C. E. Leiserson, Algorithms for VLSI Processor Arrays, Section 8.3 in reference [13], pp. 271-292.

APPENDIX I

PROOFS OF LEMMAS

Proof of Lemma 1:

Figure 14a shows an outline of a cycle C_1, without tokens, which spans at least one array cell A and one end cell E. In this cycle, [a,b> and [c,d> are two internal paths in A and [b,c> is an internal path in E. The directed path from output buffer "d" to input buffer "a" may involve cell edges other than in A or E. Since C_1 has no tokens, then the control token of A is not initially on the internal paths [a,b> and [c,d>. Also, the control token of E is not initially on the internal path [b,c>. There are only four arrangements to order the two internal paths [a,b> and [c,d> in A; they are shown in Figure 14b. (This is because all sending nodes should be adjacent and followed by all receiving nodes in A.) In each of these four arrangements, since the control token of A is not initially on the internal paths [a,b> and [c,d>, then it is not initially on the internal path [c,b>. Therefore, a cycle C_2 without tokens can be constructed, as shown in Figure 14c, from the two buffers b and c, and the two internal paths [c,b> in A and [b,c> in E. Notice that this cycle spans cells A and E only. □

Proof of Lemma 2:

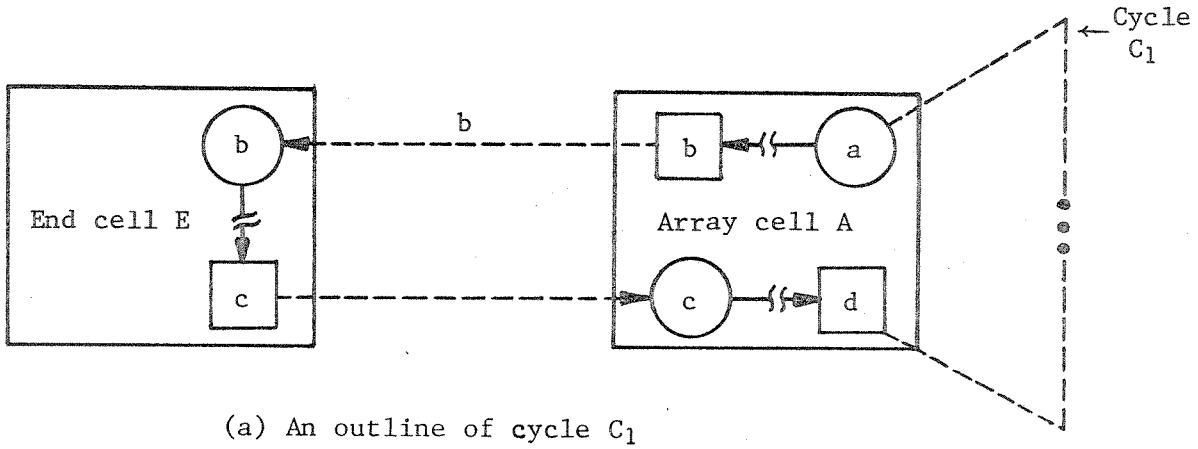
To prove Lemma 2, we need first to state and prove the following two lemmas.

Lemma 4:

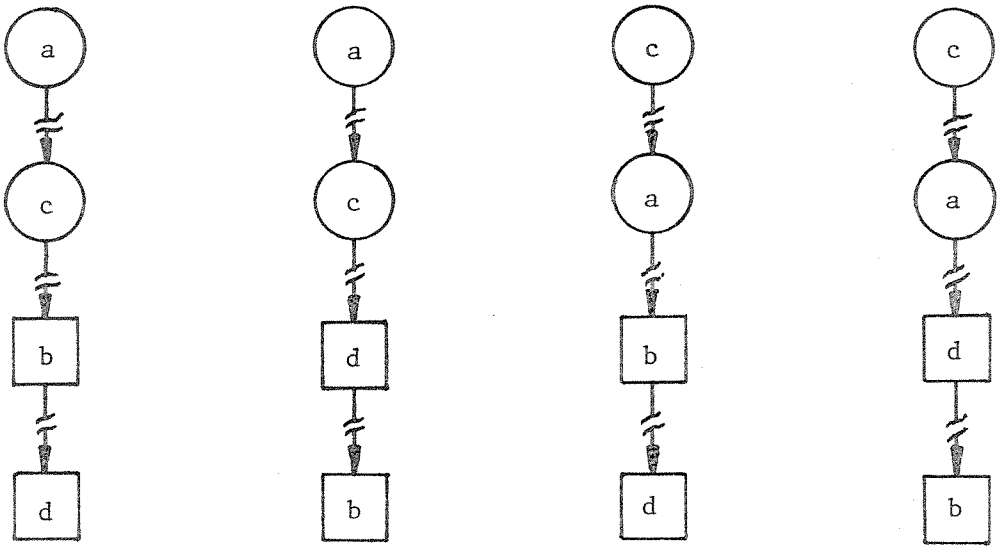
If two internal paths [a,b> and [b,c> in an array cell have no tokens, then internal path [a,c> has no tokens.

Proof:

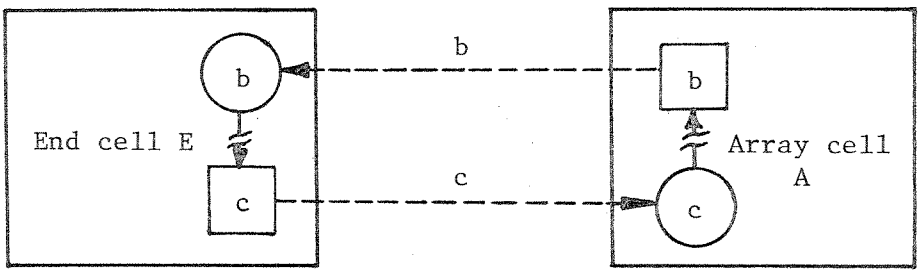
Figure 15 shows all the four arrangements in which the four operations receive from "a", send to "b", receive from "b", and send to "c" can be



(a) An outline of cycle C_1



(b) Four possibilities to order the two internal paths [a,b and [c,d in array cell A.



(c) An outline of cycle C_2

Figure 14 Proof of Lemma 1

arranged in an array cell. (This is because all sending nodes should be adjacent.) In each arrangement if $[a,b\rangle$ and $[b,c\rangle$ have no tokens, then $[a,c$ has no tokens. □

Lemma 5:

For any positive integer k ($k \geq 2$), if k internal paths $[a_0,a_1\rangle$, $[a_1,a_2\rangle$, ..., $[a_{k-1},a_k\rangle$ in any array cell have no tokens, then the internal path $[a_0,a_k$ has no tokens.

Proof: (by induction on k)

For k = 2, if $[a_0,a_1\rangle$ and $[a_1,a_2\rangle$ have no tokens, then $[a_0,a_2\rangle$ has no tokens by Lemma 4. Consider the case when the n + 1 internal paths $[a_0,a_1\rangle$, ..., $[a_n,a_{n+1}\rangle$ have no tokens. Then $[a_0,a_n\rangle$ has no tokens by induction hypothesis. Since $[a_0,a_n\rangle$ and $[a_n,a_{n+1}\rangle$ have no tokens, then $[a_0,a_{n+1}\rangle$ has no tokens by Lemma 4. □

Lemma 2 can now be proved as follows.

If Part: Assume that G has a cycle C_1, without tokens which spans array cells only. Cycle C_1 must have two intermediate buffers b and b' with opposite directions w.r.t. their tail array cells, as shown in Figure 16a. Let the buffers from b to b' on cycle C_1 be b_1, b_2, ..., and b_k. Then the internal paths $[b,b_1\rangle$, $[b_1,b_2\rangle$,..., and $[b_k,b'\rangle$ have no tokens since C_1 has no tokens. From Lemma 5, the internal path $[b,b'\rangle$ has no tokens. Similarly, we can show that the internal path $[b',b$ has no tokens. Therefore, there exists a cycle C_2 in G constructed from the two buffers b and b', and the two internal paths $[b,b'\rangle$ and $[b',b\rangle$, as shown in Figure 16b. Cycle C_2 has no tokens; and it spans two array cells only.

Only If Part: Immediate. □

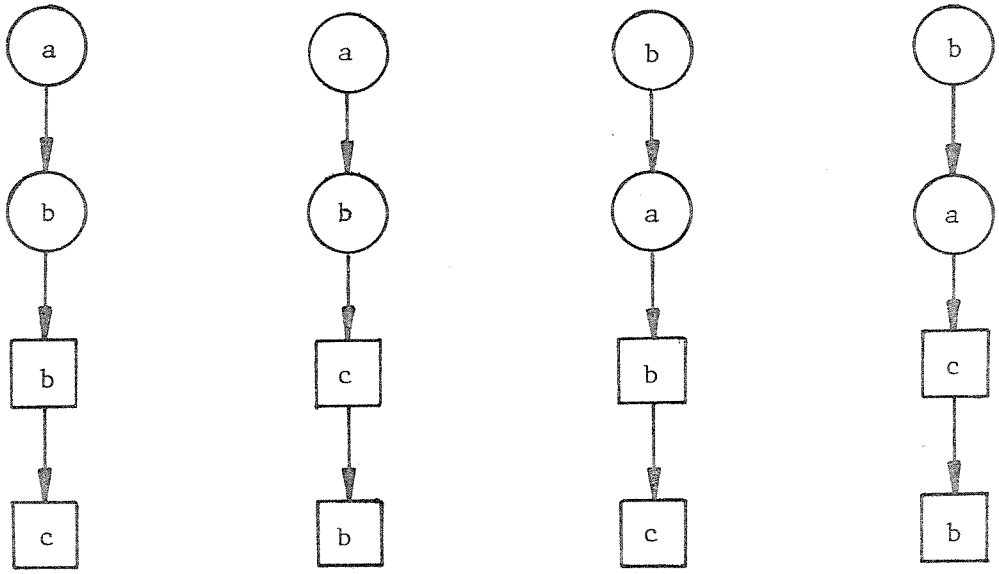


Figure 15 Proof of Lemma 4

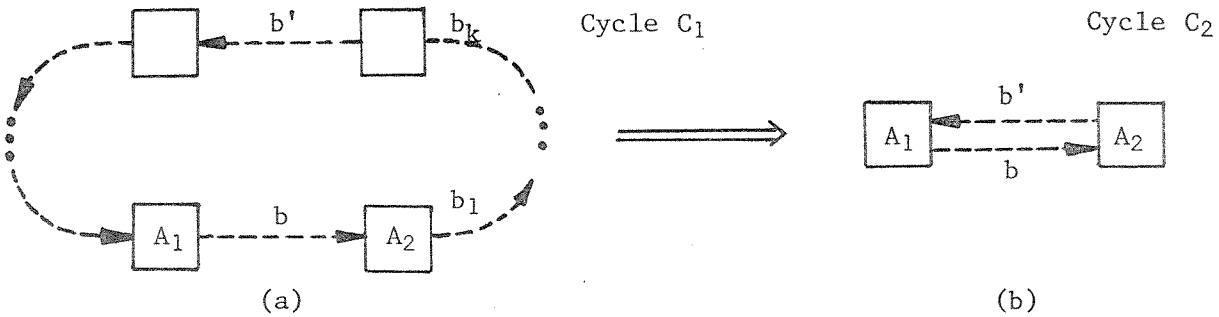


Figure 16 Proof of Lemma 2

Proof of Lemma 3:

Lemmas 4 and 5 in the appendix are also valid for hexagonal array cells.

If Part: Assume that G has a cycle C_1, without tokens, which spans hexagonal array cells only. Cycle C_1 must satisfy at least one of the following two properties:

(i) C_1 contains two buffers b and b' of opposite directions w.r.t. their tail array cells.

(ii) C_1 contains three buffers b, b', and b'' of directions

either north, south-east, and south-west

or south, north-west, and north-east

w.r.t. their tail array cells.

From (i) and Lemma 5, we have that the two internal paths [b,b'> and [b',b> have no tokens. Therefore, there exists a cycle C_2 in G which consists of the two buffers b and b', and the two internal paths [b,b'> and [b',b> have no tokens. Cycle C_2 spans two array cells only.

From (ii) and Lemma 5, we have that three internal paths [b,b'>, [b',b''>, and [b'',b> have no tokens. Therefore, there exists a cycle C_2 in G which consists of the three buffers b, b', and b'', and the three internal paths [b,b'>, [b',b''>, and [b'',b> and has no tokens. Cycle C_2 spans three array cells only.

Only If Part: Immediate. □