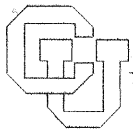
Some Applications of Finite State Automata Theory to the Deadlock Problem *

Gary J. Nutt

CU-CS-017-73



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* This work was supported by the National Science Foundation under Grant #GJ-660.

SOME APPLICATIONS OF FINITE STATE AUTOMATA THEORY TO THE DEADLOCK PROBLEM *

by Gary J. Nutt
Department of Computer Science
University of Colorado
Boulder, Colorado 80302

Report #CU-CS-017-73 April, 1973

* This work was supported by the National Science Foundation under Grant #GJ-660.

ABSTRACT

A new characterization of the system deadlock problem is presented in which sequences of resource activity are treated as potential members of the set of strings accepted by a finite state automaton. Membership in the set indicates a deadlock situation. A detection algorithm and an avoidance algorithm are provided to illustrate the approach. Some new areas of research based on the ideas are briefly mentioned.

INTRODUCTION

The existence of concurrent processes in a multiprogrammed or multiprocessor computer system may cause that system to become incapable of further activity, (without operator intervention) due to the deadlock problem. Deadlock is characterized by the situation in which one or more of the process is prevented from further activity because it is waiting for some condition to hold which will never be satisfied.

A simple example of the deadlock problem can be given in which there are two processes, P_1 and P_2, each of which may use resources R_1 and R_2 where there is one unit of each resource. Suppose that P_1 requests and receives R_1, then begins activity. Meanwhile, P_2 has requested and received resource R_2 and began concurrent activity. At some later time, P_1 requests R_2 while retaining R_1. Since P_2 has control of R_2, P_1 must wait until P_2 is ready to relinquish control of R_2. Unfortunately, P_2 may request R_1 while retaining R_2, causing P_2 to wait. When this situation arises, P_1 holds R_1 and requests R_2 while P_2 holds R_2 and requests R_1; P_1 and P_2 are incapable of further activity unless the operator intervenes to preempt a resource from one process or the other.

Deadlock is possible only when three conditions are satisfied in an operating system: [1,3]

1. A process can claim exclusive control over the resource it holds.

2. A resource cannot be preempted from a process.

3. A process can hold a unit of resource while requesting another

unit of a (possibly different) resource, i.e., a circular wait is possible.

Since these conditions are necessary for deadlock to exist, the problem can be prevented if an operating system violates any or all of the conditions. Havender suggests such an approach to operating system design.[5]

If an operating system satisfies all three conditions necessary for deadlock, different strategies can be applied to cope with the problem. One may design an avoidance algorithm which is able to specify the action of the system in such a way that deadlock will not occur. Habermann[4], Hebalkar[6], and Holt[7,8] have all given avoidance algorithms that rely on some information about the amount of resource allocated at any given time. A second strategy that can be employed in operating system design is detection. This approach would allow the detection algorithm to be executed periodically to ascertain the state of the system with respect to deadlock. Detection algorithms are given in references 1 and 8. The final strategy that might be applied in operating system design is the "no strategy" approach. The philosophy here is that deadlock occurs so infrequently that the system overhead required for prevention, avoidance, or detection are not cost-effective over a long period of time. When deadlock does occur in such a system, operator intervention is required both for detection and correction.

System State Diagrams

Holt[7,8] has described a graph model composed of system states corresponding to nodes and a set of processes to label directed edges. The sets of states and processes are allowed to be infinite. The interpretation of

the model is that a system can move from state S_i to state S_j under the influence of process P_k if and only if there exists a directed edge from node S_i to node S_j and the edge is labeled P_k. The system state diagram is nondeterministic in the sense that there may be more than one directed edge emanating from any given state with the same edge label. Using the graph model, Holt has derived definitions of blocked processes, deadlock states, effective deadlock, and total deadlock. We paraphrase the definitions of a blocked process and a deadlocked process:

> If there exists no state, S_j, such that a transformation from state S_i to S_j exists under the action of process P_k, then process P_k is <u>blocked</u> in state S_i.

That is, if no directed edge leaves state S_i, where the label of the edge corresponds to process P_k, then P_k is blocked in state S_i.

> If for all states S_j, such that a path of transformations leads from S_i to S_j and process P_k is blocked in S_i, then P_k is <u>deadlocked</u> in state S_i.

Less formally, if the set of all transformation paths leading from state S_i result only in process P_k still being blocked, then P_k is deadlocked in S_i. If all processes are deadlocked in state S_i, (i.e., there is no directed edge from state S_i), S_i is a <u>total deadlock state</u>.

Figure 1 is a system state diagram composed of four system states that change due to the influence of two processes. S_0 is the initial state and process P_1 (P_2) can change the state of the system from S_0 to S_1 (S_2). If

the current state of the system is S_1 (S_2), then process P_1 (P_2) is blocked but not deadlocked. State S_3 is a total deadlock state, since all processes are deadlocked in S_3.

The system state diagrams can be interpreted as finite state automata if the number of processes and the number of system states are both finite. Further constraining the model, a workable automata theoretic approach to the deadlock problem can be formulated. The remainder of this paper will discuss that approach.

FINITE STATE AUTOMATA

Basic Definitions

The definitions given in this section conform to those given in Hopcroft and Ullman[9].

A finite state automaton, M, is defined by the 5-tuple

$$M = (K, \Sigma, \delta, S_0, F)$$

where

K is a nonempty, finite set of states,

Σ is a nonempty, finite alphabet,

δ is a mapping of $K \times \Sigma$ into K,

$F \subseteq K$ is a set of final states.

Conceptually, M is a machine with a finite control which reads an input tape, containing symbols from the alphabet. Initially, the machine is in state S_0 and is scanning the left-most symbol on the tape. δ defines the next state by recognizing the input symbol while in the current state. A state change causes the machine to scan the next symbol to the right of the currrent symbol.

δ is also defined for the domain K x Σ^*, (where Σ^* is the set of all possible combinations of members of Σ), by defining:

$$\text{If } x\epsilon\Sigma^*,\ a\epsilon\Sigma, \text{ and } S\epsilon K, \text{ then } \delta(S,xa) = \delta(\delta(S,x),a).$$

A sentence, $x\epsilon\Sigma^*$, is accepted by M if the resulting action leaves the machine in a final state after the last character in x has been scanned. The machine does not accept any string where δ is not defined on K x Σ. The set of all sentences, T(M), is thus defined by

$$T(M) = \{x | x\epsilon\Sigma^* \text{ and } \delta(S_0,x)\epsilon\ F\}$$

The system state diagram shown in Figure 1 can be interpreted as a finite state automaton, M, such that

$$M = (K,\ \Sigma,\ \delta,\ S_0,\ F)$$

where

$K = \{S_0,S_1,S_2,S_3\}$ $\quad$ $\delta(S_0,P_1) = S_1$ $\quad$ $\delta(S_1,P_2) = S_2$

$\Sigma = \{P_1,P_2\}$ $\quad$ $\delta(S_0,P_2) = S_2$ $\quad$ $\delta(S_2,P_1) = S_1$

$F = \{S_3\}$ $\quad$ $\delta(S_1,P_2) = S_3$ $\quad$ $\delta(S_2,P_1) = S_3$

$\delta(S_1,P_1) = S_0$ $\quad$ $\delta(S_2,P_2) = S_0$

M has been defined such that S_3, the total deadlock state, is the only final state.

Application to Deadlock Detection

In the previous example, the set of all strings accepted by M corresponds to the sequence of process actions that bring the model into a state of total deadlock. Similarly, one could define the set of final states such

that strings representing process actions are accepted if the actions lead to a deadlock state for any process.

Since the automaton is nondeterministic, the process action string will always lead to a final or deadlock state whenever possible. Consider the sequence of actions: $P_1P_2P_1P_2$

The first appearance of P_1 changes the state of M from S_0 to S_1. The first appearance of P_2 may either change the system state to S_2 or S_3. Let us suppose that the next state "chosen" by the automaton was S_2. The second P_1 returns the system state to S_1 where the automaton must again choose between S_2 and S_3. The formal definition of a nondeterministic finite state automaton guarantees that the first choice will result in the next state being S_2 and the second choice being S_3, hence $P_1P_2P_1P_2$ is accepted by M and does cause the system to reach a state of total deadlock.

For the system state diagram shown in Figure 1, one could conceive of a string of process actions, $P_1P_2P_1P_2$, that left the system in state S_2. Why is it that the formal finite state automaton leads to deadlock? This phenomenon is best explained by considering the classes of deterministic and nondeterministic finite state automata. A fundamental result from automata theory states that for each nondeterministic finite state automaton, there exists a deterministic finite state automaton that accepts the same set of strings. By applying this result to the automaton discussed earlier, one can obtain the (deterministic) automaton shown in Figure 2. Note that in this automaton, any string of alternating P_i's of length greater than or equal two is accepted by the automaton (corresponding to the original nondeterministic finite state automaton).

Each P_i label in the original system state diagram represented some particular action by process P_i. However, in modeling this situation, the model has failed to distinguish between the various actions taken by a process; instead it assumes a "worst case" in which the set of actions leads to deadlock if at all possible. If the processes are deterministic, then a change of the system state from S_i to S_j or S_k is well-defined. Hence, the original model could be made deterministic without changing the set of states or the transitions by redefining the alphabet of the automaton to reflect actions as well as processes. For example, if the system shown in Figure 1 is in state S_1, then a particular action, (e.g., a resource deallocation), by process P_2 will change the state to S_2; alternatively, P_2 may perform a distinct action, (e.g., request another resource), causing the state of the system to change from S_1 to S_3 (and deadlock). For the automaton shown in Figure 1, let the alphabet be redefined to be

$$\Sigma = \{a_1, a_2, b_1, b_2\}$$

where the symbol describes an action and the subscript identifies the process. Figure 3 shows the new (deterministic) automaton that corresponds to the original automaton.

Assume that a deterministic finite state automaton has been defined to represent the operating system. Each appearance of a symbol in the alphabet corresponds to a unique action by a unique process in the operating system and this action-process can be identified when it occurs.

1. Initially, the system is in state S_0, $i \leftarrow 0$.

2. Action of a process causes the corresponding symbol, a_j, to be generated; (action type a by process P_j).

3. Apply $\delta(S_i,a_j) = S_k$. (If $\delta(S_i,a_j)$ is not defined, the model is improperly defined, stop).

4. If $S_k \varepsilon F$ then DEADLOCK else $i \leftarrow k$.

5. Goto Step 2.

Application to Deadlock Avoidance

In the previous section, a model is given which formally describes sequences of action (by a series of processes) which lead to deadlock. A deadlock detection algorithm was presented, based on a deterministic finite state automaton, M, which accepted exactly those strings leading to a state of deadly embrace. An avoidance algorithm can be more useful if the frequency of deadlocks is significant, or if the cost of preemption is high. In this section, the finite state automaton model is applied to derive an avoidance algorithm.

Given a (deterministic) system state model, it is desirable to be able to classify, (i.e., build an automaton to accept) strings which avoid deadlock states. By avoiding a deadlock state, we mean:

i. the system does not halt in this state,

ii. the system does not "pass through" this state,

iii. the system does not enter a state which leads only to a deadlock state.

The third characterization of deadlock avoidance states that the model must be able to detect system states which will lead only to final states in the deterministic finite state sutomaton. Figure 4 illustrates a possible state transition diagram segment in which this case arises. From any of the S_j states, the only transitions possible lead to a final state, $(S_{k_1}, S_{k_2},\ldots,S_{k_r})$. For purposes that the automaton is to be used, the sets of S_j can be treated as final states. An automaton that is derived to satisfy this treatment will be called a reduced finite state automaton. The procedure for deriving a reduced finite state automaton from a given deterministic automaton, M, will now be discussed.

Let $M = (K, \Sigma, \delta, S_0, F)$ be a deterministic finite state automaton;

Define $M' = (K, \Sigma, \delta, S_0, F')$ such that the set of final states is redefined by the following procedure:

Suppose $F = \{S_{k_1}, S_{k_2},\ldots,S_{k_r}\}$

1. $a \leftarrow 1$; Set $F' = F$.

2. Let $\overline{S}_{k_a} = \{S_{j_b} \mid \delta(S_{j_b},x) = S_{k_a}$, for some $x\epsilon\Sigma\}$

   Denote $\overline{S}_{k_a}$ by $\{R_1,R_2,\ldots,R_r\}$

   (Note that $0 < |\overline{S}_{k_a}| \leq r = |F'| \leq |K| < \infty$).

3. $LC \leftarrow 1$.

4. If (there exists $x\epsilon\Sigma$ and $S\epsilon K-F'$ such that $\delta(R_{LC},x) = S$) then goto Step 5. Else $F' \leftarrow F' \cup \{R_{LC}\}$ and $r \leftarrow r+1$.

5. LC ← LC + 1; if LC > $|\overline{S}_{k_a}|$ then goto Step 6 else goto Step 4.

6. a ← a + 1; if a > r then stop else goto Step 2.

The formal description of a finite state automaton (given previously) allows the next-move map, δ, to be partial on $K \times \Sigma$. The rule is imposed that if the next-move map is not defined, the string is rejected by the automaton. An equivalent formulation is to include another state, S', in K such that for each $(S,x) \in K \times \Sigma$, where $\delta(S,x)$ is not defined in M, define $\delta(S,x) = S'$, (it follows that $\delta(S',x) = S'$). This completely specified map replaces the rule mentioned above, since δ is now total on $K \times \Sigma$.

Recalling that the original incompletely specified automaton has a partial next-move map whenever some process action is inappropriate for the given state, a reasonable convention for the following discussion is to place the "trap state", S', in the set F as well as K. The set of final states is becoming a set of states which accepts undesirable (in some sense) strings.

Provided that a deterministic (possibly incompletely specified) automaton has been reduced and the trap state has been made final, consider how the resulting automaton, M, could be further modified to accept the set of strings that define all legal moves in the given system. The set that is of interest is given by

$$\Sigma^* - (T(M) \cup \{\alpha\beta \mid \alpha \in T(M),\ \beta \in \Sigma^*\}$$

This set is accepted by a completely specified, deterministic finite state automaton that roughly corresponds to the complement machine of M. (Note that the class of finite state automata is closed under complement.) This avoidance automaton for M is defined by

Let $M = (K, \Sigma, \delta, S_0, F)$ be a completely specified, reduced deterministic finite state automaton.

Define $M' = (K', \Sigma, \delta', S_0, F')$

where

$\delta'(S_i,p) = S_j$ if $\delta(S_i,p) = S_j$ and $S_i \varepsilon K-F$

$\delta'(S_i,p) = S_i$ if $\delta(S_i,p) = S_j$ and $S_i \varepsilon F$

$K' = K - \{S_j \mid$ There exists no $S_i \varepsilon K$ and $p \varepsilon \Sigma$ such that $\delta'(S_i,p) = S_j\}$

$F' = K' - F$

Verbally, the new automaton is derived from M by complementing M and by replacing all transitions <u>from</u> a member of F with new transitions that never allow a state change from a member of F. The new set of states, K', is reduced by eliminating all states that are no longer reachable under the next-move map, δ'.

The automaton shown in Figure 3 is incompletely specified; an equivalent completely specified counterpart is shown in Figure 5, along with its formal definitions. S_4 is the added "trap state." The automaton that accepts deadlock free strings is shown in Figure 6, with its formal specification.

An avoidance automaton can be generated from a deterministic finite state automaton by the following steps:

1. Obtain the reduced finite state automaton.

2. Generate the completely specified representation.

3. Construct the avoidance automaton, (a finite state automaton), from the completely specified representation.

Assume that an avoidance automaton has been defined to represent the operating system. The following procedure is an avoidance algorithm.

1. Initially, the system is in state S_0; $i \leftarrow 0$.

2. A process, P_j, requests permission to perform action a, generating symbol a_j.

3. Apply $\delta(S_i,a_j) = S_k$.

4. If $S_k \varepsilon F$ then goto Step 5 else "suspend P_j" and goto Step 2.

5. $i \leftarrow k$; perform a_j; goto Step 2.

IMPLEMENTATION CONSIDERATIONS

The primary consideration in using automata theoretic models for operating systems is the number of system states involved. As the level of detail of the model increases, the number of system states increases to the point that analysis is not possible under conventional methods. The level of detail implied by the deadlock problem is determined by the resource types, and the number of processes involved in the system. Although the number of states in such a model may be very large, the model can still be handled by orthodox methods. In the previous algorithms, the critical portions of the automaton that must be available for implementation are

a. determination of membership in the set of final states, and

b. the next-move map.

Both of these portions of the automaton can be encoded, once the model has been initially defined, such that the next move map is written as a procedure (or function program) which returns the next state value. A better encoding might also allow the procedure definition to determine (analytically) the membership of the functional evaluation in the set of final states. The cost of implementing such a system is then reduced to a function evaluation whenever a resource is involved in a system state change. The following simple example, (motivated by Holt[7,8]) illustrates the point.

Suppose that a system services two processes that share two identical units of a single resource type. Each process will never request more than two units of the resource and can only request one unit of the resource at a time. A process can be in one of the following states:

0 - the process holds no units of the resource.

1 - the process has requsted a unit.

2 - the process holds one unit of the resource.

3 - the process holds one unit and requests the other.

4 - the process holds both units of the resource.

Each state of the system can be represented by S_{jk} where j indicates the state of process P_1, and k indicates the state of process P_2. The actions that may be taken by the processes are a request by P_i, denoted r_i; allocation of a unit of resource to P_i, denoted a_i; and deallocation of a unit of resource from P_i, denoted d_i. Figure 7 is the automaton of the system. The final (total deadlock) state is S_{33}. Note that Figure 7 has 20 states to describe

a system with two process and two units of one type of resource. Using this scheme for representing n units of one type of resource for two processes results in an automaton with

$$N = 4n^2 + 2n + 2 - 2 \sum_{i=1}^{n-1} (2i - 1)$$

states. This number of states is becoming unreasonable for values of n greater than 5 or 6, since more than N specifications of the next-move map are required if the tabular form is used. (It should also be noted that under the scheme, there are n - 1 total deadlock states, i.e., the set of final states remains relatively small.) The conventional tabular form of the next-move map can be replaced by a procedure which accepts the subscripts of the current state and the current symbol as arguments and determines if the next state is defined; if it is defined, the procedure can also determine the indices of the next state. Since the set of final states is small, the result of a procedure call can easily be checked for membership. For the particular automaton described in this example, (because of its orderly growth with n), the final states are easily determined within the next state procedure.

CONCLUSIONS

The interpretation of system state diagrams as finite state automata transition diagrams can be a powerful theoretic tool for the analysis of operating systems in terms of the deadlock problem. It is possible to insure deadlock avoidance by inspecting the sequence of resource requests, allocations, and deallocations using the simplest of the abstract machines from formal language studies. The generation of legal (deadlock-free) strings is analogous to parsing expressions, thus the possibility that syntactic analysis techniques can be applied to "resource sequences." Another theoretic tool that could be employed for operating systems analysis is the regular expression to characterize the set of legal (or illegal) strings for the system.

Even if the automata-theoretic techniques are not employed as algorithms within an operating system, the models can be used to analyze the system to determine its relative freedom from deadlock. For example, models could be built of various operating system submodules. As the submodules are combined to create larger components in the operating system, the deadlock free models can be concatenated to obtain a more comprehensive model to be used for analysis. Casting the resource allocation process as an abstract machine provides enough foundation for quantitative studies of systems in terms of the (non-zero) possibility of deadlock. The study of the problem as a probabilistic automaton is another avenue of approach that deserved attention.

Perhaps the most significant aspect of this paper is the application of another established discipline to operating system design and analysis.

While the various theorems have not been applied directly, the technique for manipulating the abstract machines has been employed in deriving various algorithms and in motivating the general ideas presented here.

GJN:cah

REFERENCES

1. Coffman, E. G., Jr., Elphick, M. J., and Shoshani, A., "System Deadlocks," Computing Surveys, Vol. 3, No. 2, (June, 1971), 67-78.

2. Dijkstra, E. W., "Cooperating Sequential Processes," in Programming Languages, ed. by F. Genuys, Academic Press, London (1968).

3. Denning, P. J., "Third Generation Computer Systems," Computing Surveys, Vol. 3, No. 4, (December, 1971), 175-216.

4. Habermann, A. N., "Prevention of System Deadlocks," Communications of the ACM, Vol. 12, No. 7, (July, 1969), 373-377.

5. Havender, J. W., "Avoiding Deadlock in Multitasking Systems," IBM Systems Journal, Vol. 7, No. 2, (1968) 74-84.

6. Hebalkar, P. G., "Deadlock-Free Resource Sharing in Asynchronous Systems," Ph.D. Dissertation, Electrical Engineering Department, MIT, Cambridge, Mass., (September, 1970).

7. Holt, R. C., "On Deadlock in Computer Systems," Technical Report CSRG-6, University of Toronto, (July, 1972), (also available as Ph.D. Dissertation, Department of Computer Science, Cornell University, Ithaca, New York, (January, 1971).

8. Holt, R. C., "Some Deadlock Properties of Computer Systems," Computing Surveys, Vol. 4, No. 3, (September, 1972), 179-196.

9. Hopcroft, J. E., and Ullman, J. D., Formal Languages and their Relation to Automata, Addision-Wesley, (1969).

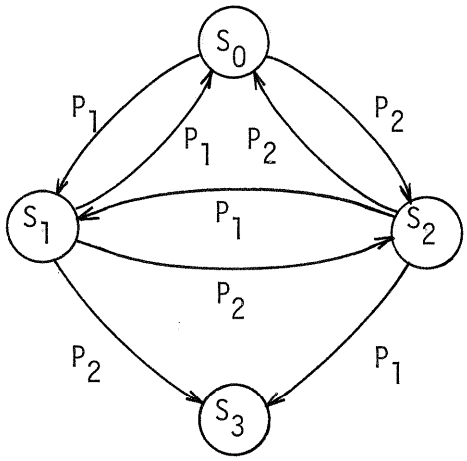
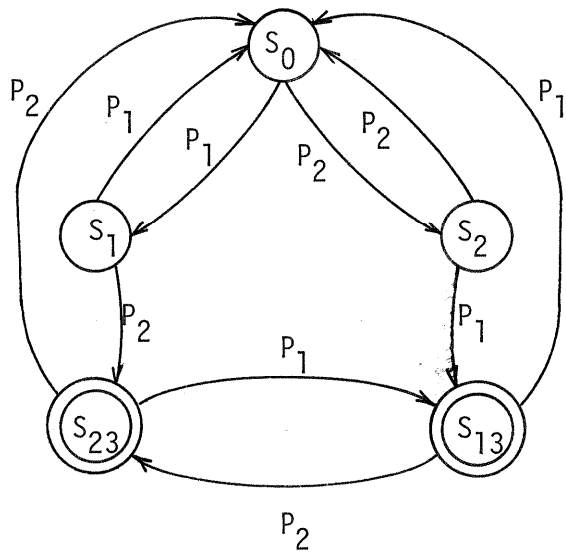


FIGURE 1



$M' = (K', \Sigma, \delta', S_0, F)$

$K' = \{S_0, S_1, S_2, S_{13}, S_{23}\}$

$F' = \{S_{13}, S_{23}\}$

$\delta'(S_0, P_1) = S_1$

$\delta'(S_0, P_2) = S_2$

$\delta'(S_1, P_1) = S_0$

$\delta'(S_1, P_2) = S_{23}$

$\delta'(S_2, P_1) = S_{13}$

$\delta'(S_2, P_2) = S_0$

$\delta'(S_{13}, P_1) = S_0$

$\delta'(S_{13}, P_2) = S_{23}$

$\delta'(S_{23}, P_1) = S_{13}$

$\delta'(S_{23}, P_2) = S_0$

FIGURE 2

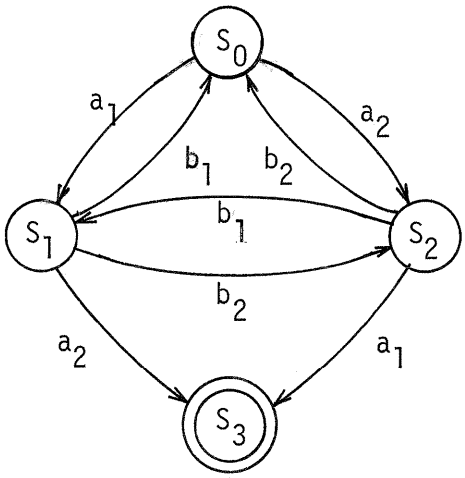


FIGURE 3

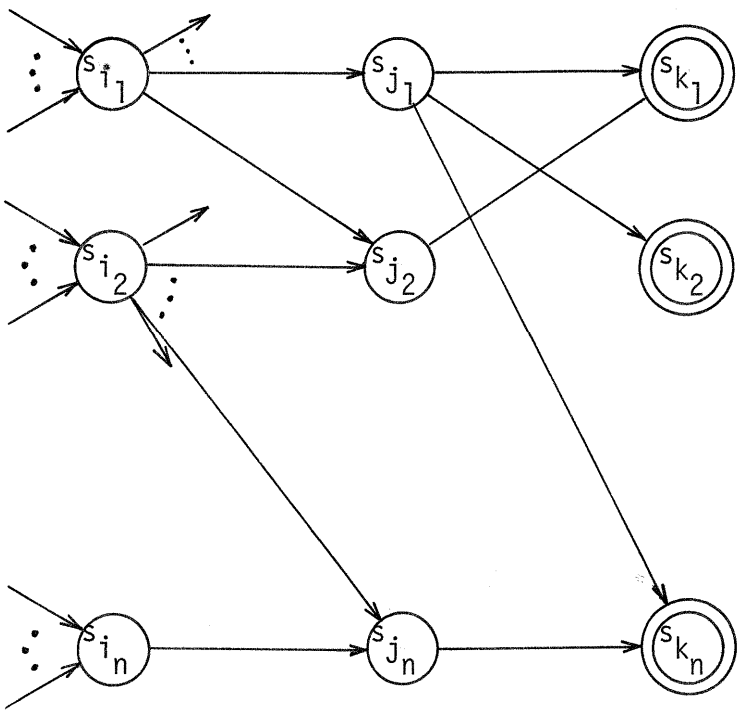
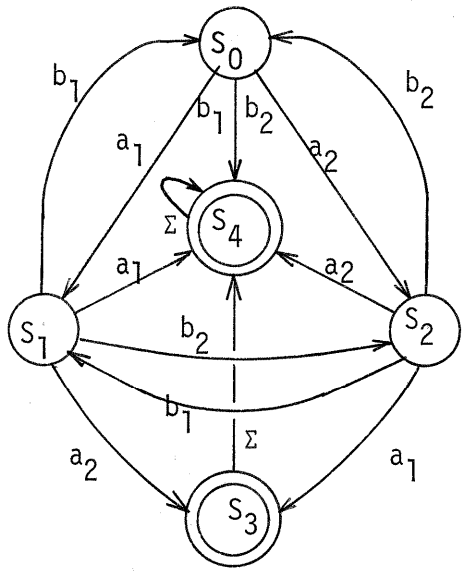


FIGURE 4

$M = (K, \Sigma, \delta, S_0, F)$

$K = \{S_0, S_1, S_2, S_3, S_4\}$

$F = \{S_3, S_4\}$



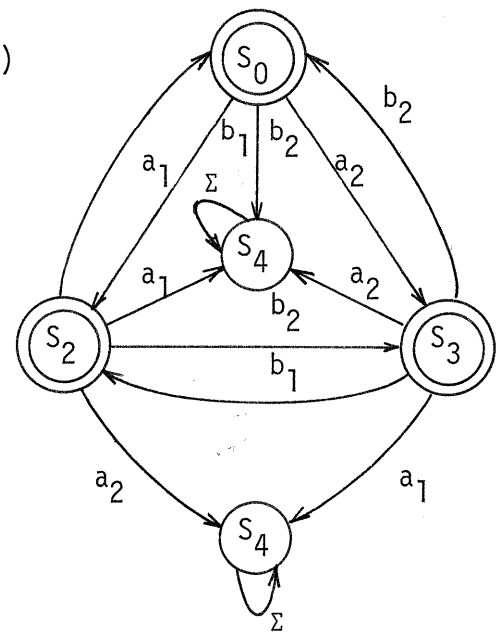
$\delta(S_0, a_1) = S_1$
$\delta(S_0, a_2) = S_2$
$\delta(S_0, b_1) = S_4$
$\delta(S_0, b_2) = S_4$
$\delta(S_1, a_1) = S_4$
$\delta(S_1, a_2) = S_3$
$\vdots$
$\delta(S_3, b_2) = S_4$
$\delta(S_4, a_1) = S_4$
$\delta(S_4, a_2) = S_4$
$\delta(S_4, b_1) = S_4$
$\delta(S_4, b_2) = S_4$

FIGURE 5

$M' = (K, \ , \ ', S_0, F')$

$F' = K' - F$

$= \{S_0, S_1, S_2\}$



$\delta'(S_0, a_1) = S_1$
$\delta'(S_0, a_2) = S_2$
$\delta'(S_0, b_1) = S_4$
$\delta'(S_0, b_2) = S_4$
$\delta'(S_1, a_1) = S_4$
$\delta'(S_1, a_3) = S_3$
$\vdots$
$\delta'(S_3, b_2) = S_3$
$\delta'(S_4, a_1) = S_4$
$\delta'(S_4, a_2) = S_4$
$\delta'(S_4, b_1) = S_4$
$\delta'(S_4, b_2) = S_4$

FIGURE 6

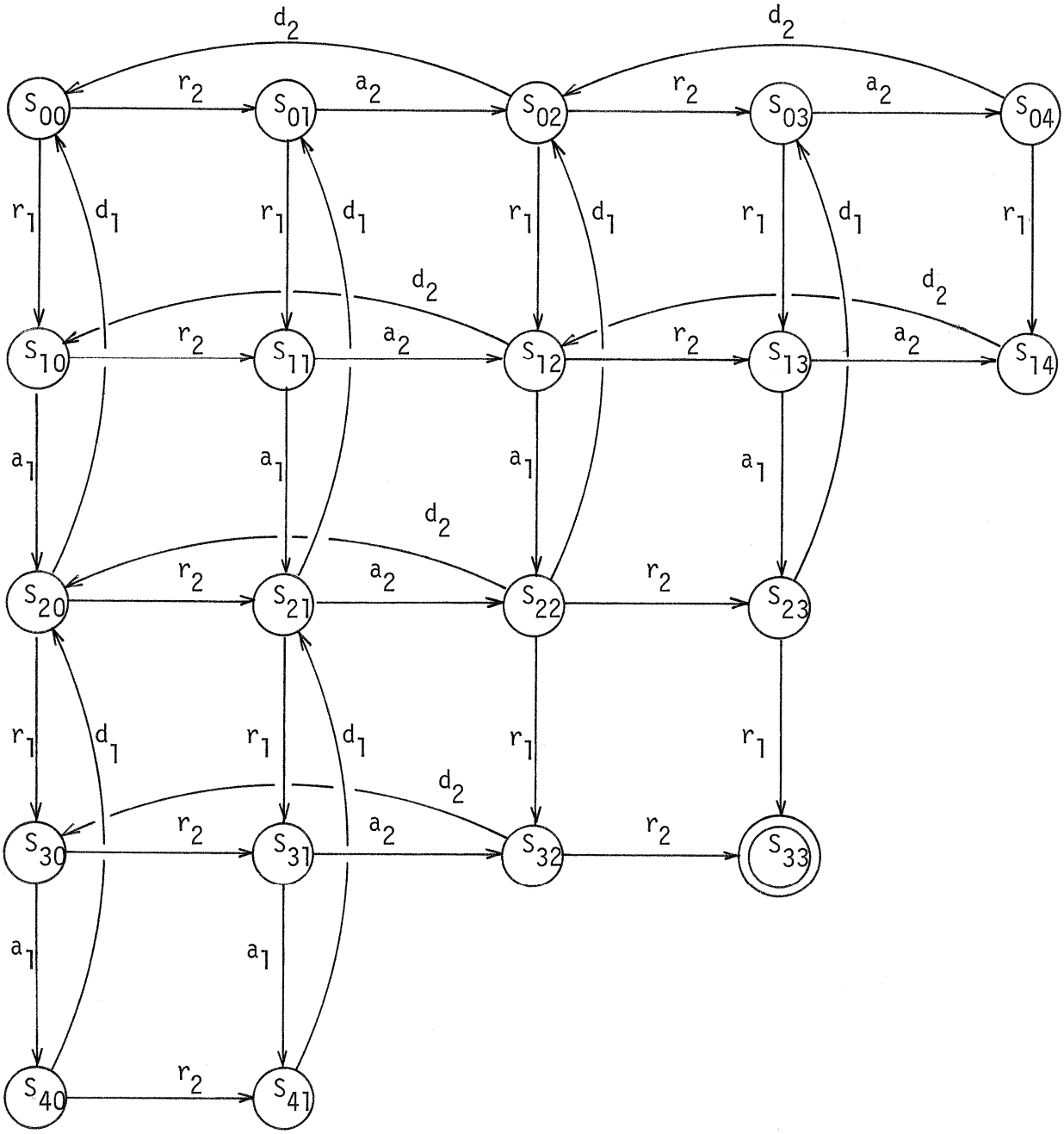

S_{00} S_{01} S_{02} S_{03} S_{04}
S_{10} S_{11} S_{12} S_{13} S_{14}
S_{20} S_{21} S_{22} S_{23}
S_{30} S_{31} S_{32} S_{33}
S_{40} S_{41}
r_1 r_2 a_1 a_2 d_1 d_2

FIGURE 7

ERRATA for TR #CU-CS-017-73, April 1973--SOME APPLICATIONS OF FINITE STATE AUTOMATA THEORY TO THE DEADLOCK PROBLEM by Gary J. Nutt

page 5, line 1: the state of the system is $S_1(S_2)$, then process $P_2(P_1)$ changes the state to S_3.

page 10, lines 14-21)
page 11, lines 1-2) replaced by:

Suppose $F=\{S_{k_1}, S_{k_2}, \ldots, S_{k_r}\}$

1. a←1; Let $F'=F$.
2. ADD←false
3. Let $\hat{S}_{k_a} = \{S_{j_b} \mid \delta(S_{j_b}, x) = S_{k_a}$, for some $x \in \Sigma\}$

   Denote $\hat{S}_{k_a}$ by $\{R_1, R_2, \ldots, R_c\}$

   (Note that $0 < |\hat{S}_{k_a}| \leq |K| < \infty$)
4. LC←1
5. If [$\exists x \in \Sigma$ and $S \in K-F'$ ∋ $\delta(R_{LC}, X)=S$]

   then goto Step 6.

   else begin ADD←true;

   r←r+1;

   F←F'∪$\{S_{k_r}\}$

   (where S_{k_r} is R_{LC})

   end
6. LC←LC+1; if LC>c then goto Step 7 else goto Step 5.
7. a←a+1; if a≤r then goto Step 3.
8. If ADD then goto Step 2 else stop.

page 15, line 4:

$$N=4n^2+2n+2-2\sum_{i=1}^{n-1}(2i-1)=2n^2+6n$$

Figure 6

$M'=(K, \Sigma, \delta', S_0, F')$

$F'=K-F$

the state designated S_2 is S_1

the state designated S_3 is S_2

the state designated S_4 is S_3

ERRATA for TR #CU-CS-017-73, April, 1973 -- SOME APPLICATIONS OF FINITE STATE AUTOMATA THEORY TO THE DEADLOCK PROBLEM, by Gary J. Nutt.

page 5, line 1: the state of the system is $S_1(S_2)$, then process $P_2(P_1)$ changes the state to S_3.

page 15, line 4: $N = 4n^2 + 2n + 2 - 2 \sum_{i=1}^{n-1} (2i - 1) = 2n^2 + 6n$

Figure 6: $M' = (K, \Sigma, \delta', S_0, F')$