A Deadlock Prevention Method for a Sequence Controller for Manufacturing Control

Toshimi Minoura
Chihan Ding
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-3902

89-60-13

A Deadlock Prevention Method for a Sequence Controller for Manufacturing Control

Toshimi Minoura and Chihan Ding
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

Abstract

In a manufacturing system, machines, robots and storage areas are used as serially-reusable resources. If the usage of these resources is not properly controlled, deadlocks may occur. In this paper, we present a simple deadlock prevention method for a linear manufacturing line. Our deadlock prevention method can be implemented simply by adding some dummy resources to a Petri net-based sequence controller. Heuristic strategies for the algorithms that assist us to design such sequence controllers are also discussed.

Key Words and Phrases: manufacturing line, sequence controller, Petri net, deadlock prevention.

1 Introduction

Computer systems are widely used in modern manufacturing systems for such functions as inventory control, job scheduling, and machine control. An automatic manufacturing system usually consists of manufacturing lines containing machine stations, robots, and transport devices controlled by computers. While a job, which is often embodied as a workpiece, goes through manufacturing stages, various resources such as machines, robots, and storage areas are allocated to it, and a manufacturing operation is performed at each manufacturing stage. Furthermore, a manufacturing system normally accommodates multiple jobs at various stages of manufacturing. A Petri net-based sequence controller that guarantees mutually exclusive accesses to serially reusable resources can be designed for scheduling job movements [KOMO-84, MAIM-83]. However, without a deadlock prevention or avoidance mechanism, this type of controller may encounter deadlocks.

This paper presents a simple deadlock prevention method for a Petri net-based sequence controller for a linear manufacturing line. A linear manufacturing line consists of a linear sequence of manufacturing stages, and it neither splits into multiple manufacturing lines nor merges with other manufacturing lines. Deadlocks are prevented by restricting accesses to dummy resources added to a Petri net-based sequence controller. These dummy resources serve as generalized locks that prevent the system from entering unsafe states.

Section 2 describes a model of a linear manufacturing line. A deadlock prevention method using dummy resources is presented in Section 3. Algorithms that assist us in implementing the deadlock prevention method are described in Section 4, and their effectiveness is compared in Section 5. Section 6 concludes this paper.

2 Model

A *linear manufacturing line* consists of a sequence of *stages*, and at each of these stages a specific operation such as moving a workpiece onto or off from a conveyor and machining a workpiece is performed. While each *job* proceeds through a manufacturing line, undergoing a manufacturing activity at each stage, various *serially-reusable* resources are allocated to the job. When a job proceeds to a new stage, all the resources required by the job at that stage must be allocated. If this requirement is not satisfied, progression of the job is blocked until those resources become available. A serially-reusable resource can be allocated to at most one job at any time.

A linear manufacturing line can be characterized by a quadruple (M, N, u, w_0), where

M indicates the number of stages in the manufacturing line, N indicates the number of *resource types* utilized by the manufacturing line, the *resource usage matrix* $u[1..M, 1..N]$ is a two-dimensional array of non-negative integers of size $M * N$, and the *resource array* $w_0[1..N]$ is an array of positive integers of size N. An matrix element $u[i, j]$ indicates the number of resource units of type j used by a job at stage i. An array element $w_0[j]$ indicates the number of resource units of type j initially available.

An example of a manufacturing line is shown in Fig. 1. This manufacturing line utilizes five resource types: one unit of conveyor space (C), one unit of robot 1 (R1), two units of storage (S), one unit of robot 2 (R2), and one unit of machine (M), Therefore, $w_0 = < 1, 1, 2, 1, 1 >$. This manufacturing line involves the following manufacturing stages as shown in Fig. 2: *in-conveyor*, *unloading1*, *in-storage*, *loading1*, *machining*, *unloading2*, *out-storage*, *loading2*, and *out-conveyor*. An incoming job J must first reach the conveyor. At this point J is in stage *in-conveyor*. It then is picked up by robot 1 and moved to a slot in the storage. While J is being moved from the conveyor to the storage, it is in stage *unloading1*. The stage *unloading1* requires one unit of each of C, R1, and S. While J is in the storage, it is in stage *in-storage*. The job then is picked up by robot 2 and loaded onto the machine (*loading1*). After machining (*machining*), robot 2 unloads the job from the machine and places it in a slot of the storage (*unloading2*). Finally, robot 1 moves J back onto the conveyor (*loading2*). The conveyor and the storage are shared by both incoming and outgoing jobs. Thus, the complete resource usage matrix is the following:

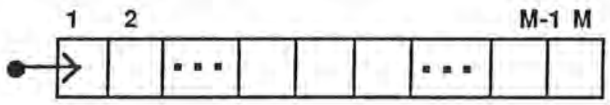
$$u = \begin{pmatrix} 10000 \\ 11100 \\ 00100 \\ 00111 \\ 00001 \\ 00111 \\ 00100 \\ 11100 \\ 10000 \end{pmatrix}$$

A *state* $s[1..M]$ of a manufacturing line (M, N, u, w_0) is an array of non-negative integers of size M. An array element $s[i]$ indicates the number of jobs at stage i. A particular state $s_0 =< 0, ..., 0 >$ is called the *initial* state. No jobs are in the manufacturing line when it is in the initial state. We denote the number of jobs in state s by $|s|$. Note that $|s| = \sum_{i=1}^{M} s[i]$. For example, in state $s_1 =< 1,0,0,0,0,0,2,0,0 >$, one job is in stage *in-conveyor*, and two jobs are in stage *out-storage*.

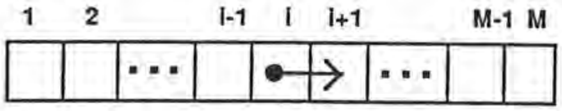
The array $h_s[1..N]$, each of whose element $h_s[j]$ indicates the number of resource units of type j being used by the jobs in state s, is defined as follows: $h_s[j] = \sum_{i=1}^{M} s[i] * u[i,j]$. A state s is called *legal* if $h_s \leq w_0$. [1] Let state $s_1 =< 0,1,0,0,1,0,1,0,0 >$. Then $h_s =< 1,1,2,0,1 >$, and s_1 is a legal state since $h_s \leq w_0$.

A state s' is a *next* state of state s ($s \rightarrow s'$) if both s and s' are legal states and if one of the following conditions hold.

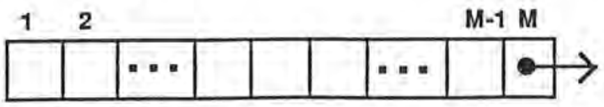
A1. s' is a state after a new job is introduced to s: $s'[1] = s[1] + 1$, and $s'[i] = s[i]$ for every i such that $2 \leq i \leq M$.



A2. s' is the state that results when a job in some stage i in s is advanced by one stage: for some i such that $1 \leq i \leq M-1$, $s'[i] = s[i] - 1$ and $s'[i+1] = s[i+1] + 1$, and $s'[k] = s[k]$ for $1 \leq k \leq i-1$ and $i+2 \leq k \leq M$.



A3. s' is a state after a job in s leaves the manufacturing line: $s'[i] = s[i]$ for $1 \leq i \leq M-1$ and $s'[M] = s[M] - 1$.



[1]Let $x[1..N]$ and $y[1..N]$ be arrays of the same size. Then $x \leq y$ if and only if $x[i] \leq y[i]$ for every i such that $1 \leq i \leq N$. When $x \leq y$, we also say that y covers x.

The *closure* of the relation $\rightarrow$ is indicated by $\xrightarrow{*}$. A state s is called *reachable* if $s_0 \xrightarrow{*} s$. A state s is *safe* if $s \xrightarrow{*} s_0$, otherwise it is *unsafe*. A state s such that $|s| > 0$ is a *deadlock* state if for no state s', $s \rightarrow s'$. A deadlock state is always an unsafe state.

A deadlock involving a set of two or more jobs occurs when these jobs, while holding some resources, issue requests for the resources held by other jobs. Fig. 3 shows all the cases of deadlocks that can occur in the manufacturing line given in Fig. 1. Consider the case depicted in Fig. 3a. The job on the conveyor cannot proceed because the storage space is not available. On the other hand, the jobs in the storage cannot proceed because the conveyor space is not available.

A *Petri net* is often used as a modeling tool because it can encode the state of a system by *tokens* on it. It has been used for modeling a variety of systems including computer hardware and software [MURA-84, PETE-81]. It has also been effectively used to design and implement a sequence controller [HURA-87].

A Petri net is a bipartite directed graph. It contains two kinds of nodes, i.e., *places* drawn as circles and *transitions* drawn as line segments, and directed arcs are used to connect places and transitions. We represent a sequence controller by a Petri net as follows. A *stage place* is provided for each manufacturing stage, and a *resource place* is provided for each resource type. A token in a stage place represents a job in the manufacturing stage associated with that stage place, and a token in a resource place represents a unit of free resource of the resource type associated with that resource place. The Petri net in Fig. 2 models the manufacturing line given in Fig. 1.

The state transition diagram $G = (S, T)$, where S is the set of states and T is the set of edges representing state transitions, for a manufacturing line (M, N, u, w_0) can be obtained by constructing S and T as the minimum closures satisfying the following rules:

B1. The initial state s_0 is in S.

B2. If $s \rightarrow s'$ for some s in S, then s' is in S and (s, s') is in T.

We label an edge in a state transition diagram by the transition that the edge represent. A state transition diagram is a convenient tool for explaining safe, unsafe, and deadlock states. There is a path from a node representing a safe state to the node representing s_0, and there is no such path from a node representing an unsafe node. A node representing a deadlock state has no outgoing edges. An abbreviated state transition diagram for the manufacturing line shown in Fig. 1 is given in Fig. 4.

3 Deadlock Prevention

In Section 2 we discussed examples of deadlocks that can occur in a manufacturing line. Without deadlock prevention or avoidance, a Petri net-based sequence controller that guarantees only mutually exclusive accesses to serially reusable resources may cause deadlocks as discussed in Section 2.

We prevent deadlocks by disallowing every state transition that leads to an unsafe state. For this purpose we define *unsafe entry states.* An unsafe state s' is an unsafe entry state if for some safe state s, $s \rightarrow s'$. It is obvious that if every unsafe entry state is prevented from occurring, a deadlock will never occur.

Assume that s' is an unsafe entry state of a manufacturing line $L = (M, N, u, w_0)$. Then s' can be prevented from occurring as follows. Provide a new *dummy* resource type D with $|s'| - 1$ units, and require every job to reserve one unit of D when it enters any stage i such that $s'[i] \neq 0$. Since there are only $|s'| - 1$ units of D, s' cannot occur as it involves $|s'|$ jobs using $|s'|$ units of D.

More formally, L can be modified to $L' = (M, N + 1, u', w_0')$ as follows:

C1. (New dummy resource type)

$w_0'[j] = w_0[j]$ for $1 \leq j \leq N$ and $w_0'[N+1] = |s'| - 1$.

C2. (Reserving dummy resources)

$$u'[i, N+1] = \begin{cases} 0 & \text{if } s'[i] = 0 \\ 1 & \text{if } s'[i] \neq 0, \textit{and} \end{cases}$$

$$u'[i, j] = u[i, j] \text{ for } 1 \leq i \leq M, 1 \leq j \leq N.$$

When an unsafe entry state s' is prevented from occurring as above, some other state s'' may also be prevented from occurring. This happens if $|s'| \leq |s''|$, and if the stages of non-zero components of s' are also non-zero components of s'' (if $s'[i] \neq 0$, then $s''[i] \neq 0$). If s'' is a safe state, new deadlock states that do not occur in L may occur in L', although such deadlocks are very rare in practice. Therefore we should prevent one unsafe entry state at a time until every unsafe entry state is removed. When all the unsafe entry states disappear, all the unsafe (and hence deadlock) states will also disappear.

We obtained the Petri net-based sequence controller shown in Fig. 5 by adding three dummy resource types *Dummy1*, *Dummy2*, and *Dummy3* to the sequence controller given in Fig. 2. The dummy resources of type *Dummy1* are used to prevent the unsafe entry state $s' = < 1,0,0,0,0,0,1,1,0,0 >$ (see Fig. 4 also). Since $|s'| = 3$, two units of type *Dummy1* resources are provided, and they are used in stages *in-conveyor*, *unloading2*, and *out-storage*. The dummy resources of type *Dummy2* are used to prevent the unsafe entry state $s' = < 0,1,1,0,1,0,0,0,0 >$, and the dummy resources of type *Dummy3* are used to prevent the unsafe entry state $s' = < 1,0,0,0,0,1,1,0,0 >$.

In implementing our deadlock prevention method, all the unsafe entry states must be prevented from occurring. Since the deadlock prediction problem for our model is *NP-complete* [MINO-82], there is most likely no algorithm that can solve this problem efficiently. Therefore, we basically exhaustively search for unsafe entry states. However, the following properties allow us to eliminate certain legal states from our search.

D1. A state covered by a safe state is also a safe state.

D2. A state covering an unsafe state is also an unsafe state.

In the following section, we discuss how some heuristic search strategies can effectively utilize these properties.

4 Enumerating Unsafe Entry States

A program that assists us to implement the deadlock prevention method described in Section 3 was written in C. We developed four algorithms for enumerating unsafe entry states, and each of them was tested with several test cases to determine their effectiveness. In this section, those four algorithms are described. Their effectiveness is compared in the next section.

Algorithm 1

In order to detect unsafe entry states, the first algorithm enumerates all the legal states and classify them into safe states and unsafe states. In expanding the reachability tree, Algorithm 1 performs depth-first search starting from the initial state and stores safe and unsafe states in lists.

Each time a new state node s is enumerated, procedure $safeTest(s)$ compares it with other nodes in the $SafeStates$ and $UnsafeStates$ lists to examine whether it is already known to be safe or unsafe. If safety of s is not known yet, the procedure $safeTest(s)$ generates all the next states ss such that $|ss| \leq |s|$. If any such node ss is safe, s is safe. Otherwise, s is unsafe. When s is safe, the next state obtained by adding a new job to stage 1, if it is legal, is added to the queue of unresolved states. When s is unsafe, such a state is unsafe.

Matching a newly generated state with those in the $SafeStates$ and $UnsafeStates$

```
/*  main procedure for Algorithm 1 */

main()
  s, ss          : state;
  SafeStates     : set of (safe) states;
  UnsafeStates   : set of (unsafe) states;
  Unresolved     : queue of (unresolved) states;
  {
    SafeStates = {s0};           /* s0 = <0,...,0> */
    UnsafeStates ={};
    Unresolved = {};

    nextState(s0, 0, ss);        /* s1 = <1,0,...,0> */
    insertTail(ss, Unresolved);  /* safety of s0 must be resolved */

    while (Unresolved != {})      /* perform recursive safety test */
    { removeTail(Unresolved, s);  /* for every unresolved state    */
      void safeTest(s);
      if (unsafeState(s))        /* if s is unsafe, it is unsafe entry */
        s->UnsafeEntry = TRUE;   /* since it was unresolved state and  */
    }                            /* unresolved state is reachable from */
  }                              /* safe state                         */
```

Fig. 6. Procedure main() of Algorithm 1.

lists is time-consuming. In Algorithm 2, we will use hashing for this purpose. Algorithm 1 is intended to be the base case since hashing cannot be used in Algorithms 3 and 4 that will be discussed later.

The procedures *main*() and *safeTest*(*s*) of Algorithm 1 are given in Figs. 6 and 7.

Algorithm 2

Algorithm 1 is inefficient because it consumes most of its time comparing a new state with those in the *SafeStates* and *UnsafeStates* lists. In order to eliminate this problem, Algorithm 2 uses hash tables instead of lists to store safe and unsafe states. The hash

```
/*  safeTest procedure for algorithm 1 */

boolean safeTest(s)
  s, ss, sss         : state;
  UnsafeLocalQueue   : queue of states;
  status             : boolean;
  {
    if safeState(s) return(TRUE);    /* already known to be safe   */
    if unsafeState(s) return(FALSE); /* already known to be unsafe */

    UnsafeLocalQueue = {};
    status = UNSAFE;                 /* s is not safe yet          */
    for (i=M ; i>=1; i--)            /* i=M for A3, others for A2  */
    { nextstate(s, i, ss);       /* generate next state ss      */
      if (ss != NIL)
      { switch(status)
        { case  UNSAFE:              /* s is not safe yet          */
              if safeTest(ss)        /* if ss is safe, s is safe   */
              { status = SAFE;
                while ((sss = remove(UnsafeLocalQueue)) != NIL)
                    sss->UnsafeEntry = TRUE;   /* s safe, sss unsafe */
              }  else              /* ss is unsafe, reachable from s */
                insert(ss, UnsafeLocalQueue);
          case  SAFE:              /* s is already known to be safe */
              insertFront(ss, Unresolved);
        }
      }
    }

    nextState(s, 0, ss);                       /* add a job to stage 1 */
    if ((ss != NIL) && (status == SAFE) insertTail(ss, Unresolved);

    if (status == SAFE)                      /* s was found to be safe */
    { insertSafeState(s);
      return(TRUE);
    } else
    { insertUnsafeState(s);                  /* s is unsafe            */
      return(FALSE);
    }
  }
```

Fig. 7.   Procedure safeTest(s) of Algorithm 1.

tables are arrays of pointers to lists of nodes representing states. A hashing function is applied to the binary value representing a state.

Algorithm 3

In order to reduce the amount of memory space required for storing the safe and unsafe states enumerated, Algorithm 3 keeps only the safe-cover states (*safe-covers*) and unsafe-basis states (*unsafe-bases*) instead of safe and unsafe states. Thus, not only the amount of memory space used storing state information is reduced, but also the execution time of the program is reduced. This is because a newly generated state need be checked against fewer state nodes.

Algorithm 4

Enumerating unsafe entry states means determining the boundaries between safe states and unsafe states. Algorithm 4 tries to reach this boundary quickly by bimodal search. Assume that safeness of a state s must be resolved. In its first phase, Algorithm 4 tries to obtain a largest legal state s' reachable from s by adding jobs to the current state. Before a job can be added, some jobs may have to be moved. Once a largest legal state s' is reached, the algorithm enters its second phase. In its second phase, the algorithm tries to determine the safeness of s' by only reducing and moving jobs. Algorithm 4, like Algorithm 3, uses the $SafeCovers$ and $UnsafeBases$ lists. However, the queue of unresolved states is organized as a priority queue according to the number of jobs in each state. Each time when the algorithm selects a new node from the $Unresolved$ queue, it always selects the state with the largest number of jobs in it in order to find the largest safe cover quickly.

The earlier large safe covers and smaller unsafe bases are found, the shorter the $SafeCovers$ and $UnsafeBases$ lists will be. Smaller sizes of these lists mean shorter comparison time.

5 Comparisons

In order to determine the relative efficiency of the four algorithms described in the preceding section, they were subjected to the following test cases.

Case A. The system contains eight stages and eight resource types. Each stage i is associated with a particular resource type R_i, and only one unit is provided for each resource type. A job uses one unit of R_i at each stage i. Since at every stage a job can proceed to the next stage as soon as the next stage becomes empty, every legal state is safe.

Case B. Case B is identical to Case A, except that Case B consists of nine stages and nine resource units.

Case C. Case C is identical to Case A, except that Case C consists of 10 stages and 10 resource units.

Case D. The system consists of five stages and three resource types, and several deadlock states can result.

Case E. Case E is identical to Case D, except that one dummy resource is added to Case E in order to prevent a deadlock state.

Case F. Case F is identical to Case D, except that three dummy resources are added to Case F in order to prevent all the deadlocks states.

Case G. The system consists of nine stages and five resource types as shown in Fig. 2.

Case H. Case H is identical to Case G, except that in order to prevent all the deadlock states, three dummy resource types are added to Case G as shown in Fig. 5.

Case I. Case I is rather complex. It contains 15 stages and eight resource types, and many deadlock states can result.

Table 1 show the the amount of CPU time required for enumerating all the unsafe entry states. Programs were executed on Sequent Balance 21. Several conclusions can be drawn from the results shown in Table 1 and some other results.

1. Algorithms 1 and 2 generate all the legal states, and they are inefficient especially when most states generated are safe as in Cases A - C.

2. If we generate all the states, hashing as used in Algorithm 2 is a good way to match states.

3. Safe-covers and unsafe-base states can certainly reduce the number of states generated. In Case C, the maximum number of safe states generated by Algorithms 1 and 2 is 1024 while the number of safe-cover states generated by Algorithm 3 is 48.

4. Algorithm 2 is comparable to Algorithm 3. Time required for matching a state against stored states by hashing seems to be comparable to that required for comparing a state against safe-cover states and unsafe-base states stored in lists.

5. Algorithm 4 performs better than Algorithm 3 when most states are safe as in Cases A - C. This fact shows that the enumeration order of states affects the number of safe-cover and unsafe-base states generated. For example, in Case C, Algorithm 4 generates only 10 safe-cover states, while Algorithm 3 generates 58.

Overall, Algorithm 4 is the best algorithm among those which we investigated.

6 Conclusion

A simple deadlock prevention method for a Petri net-based sequence controller for a linear manufacturing line was presented. This method prevents deadlocks by disallowing any state transition that leads to an unsafe entry state, and the method can be implemented by simply adding some dummy resources. A program that enumerates all the unsafe entry states was written, and four algorithms were developed for this program. These algorithms were tested with several test cases in order to determine their effectiveness.

Algorithm 4, which performed best for most cases, tries to find safe/unsafe boundary states as quickly as possible. In order to save storage space, safe states and unsafe states were reduced into safe cover states and unsafe base states, respectively.

Even in a more complex manufacturing system that includes multiple manufacturing lines, deadlocks can be prevented if all the unsafe entry states are prohibited. It is possible to further extend our program to handle such a case.

Since the problem we tried to solve is in general NP-complete, there is a fundamental limit to the maximum number of stages allowed, which it somewhere between 15 to 20. However, as it is often not easy to figure out manually a deadlock prevention method for a manufacturing system even in this complexity range, the method presented will be useful in many real systems. A more complex system must be decomposed into subsystems of manageable sizes. This problem is left for future research.

References

[KOMO-84] Komoda, N., Kera, K., and Kubo., T. An autonomous, decentralized control system for factory automation. *IEEE Computer 17*, 12 (Dec. 1984), 73-83.

[MAIM-83] Maimon, O.Z., and Nof, S.Y. Activity controller for a multiple robot assembly cell. In *Control of Manufacturing Processes and Robotic Systems*, ASME, 1983, pp. 267-284.

[MART-84] Martinez, J., and Silva, M. A language for the description of concurrent systems modelled by coloured Petri nets: Application to the control of flexible manufacturing systems. Proc. IEEE Workshop on Languages for Automation, 1984, pp. 72- 77.

[MINO-82] Minoura, T. Deadlock Avoidance Revisited. *JACM 29*, 4 (Oct. 1982), 1023-1048.

[MURA-84] Murata, T. Modeling and analysis of concurrent systems, In *Handbook of Software Engineering*, Van Nostrand Reinhold, 1984, pp. 39-63.

[PETE-81] Peterson, J.L. *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, 1981.

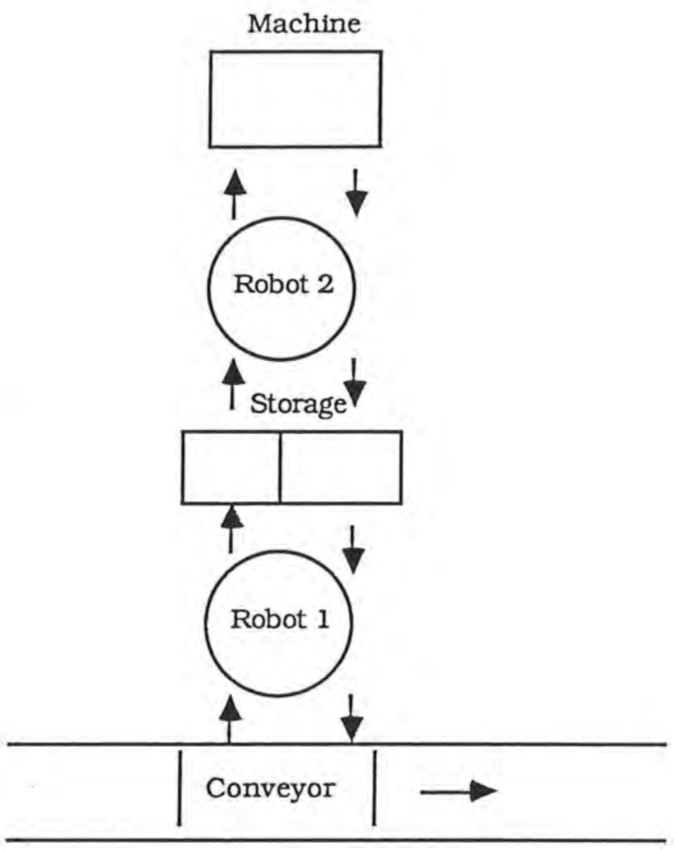


Fig.1. A manufacturing line.

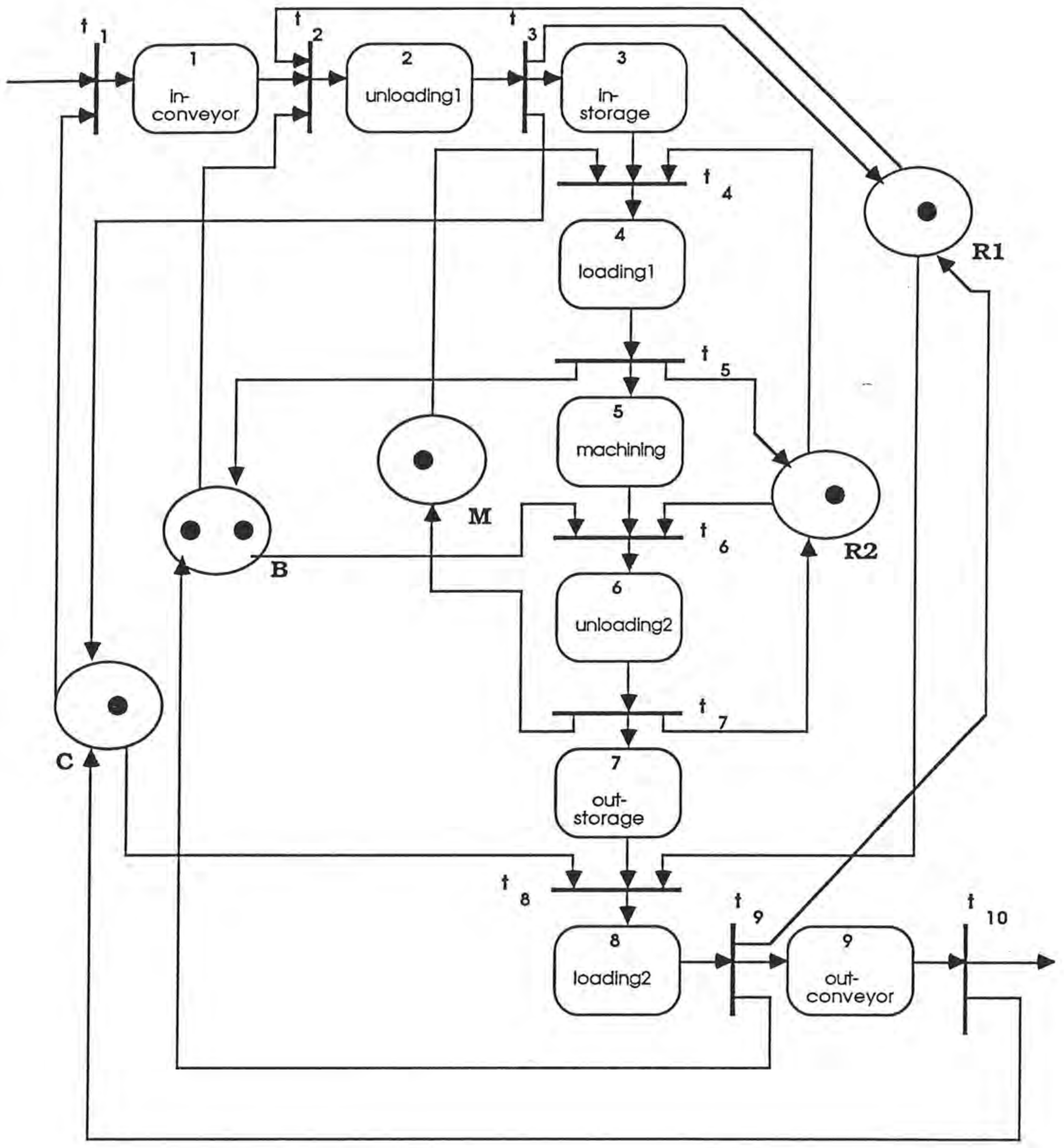


Fig. 2. The Petri net for the manufacturing line in Fig.1.

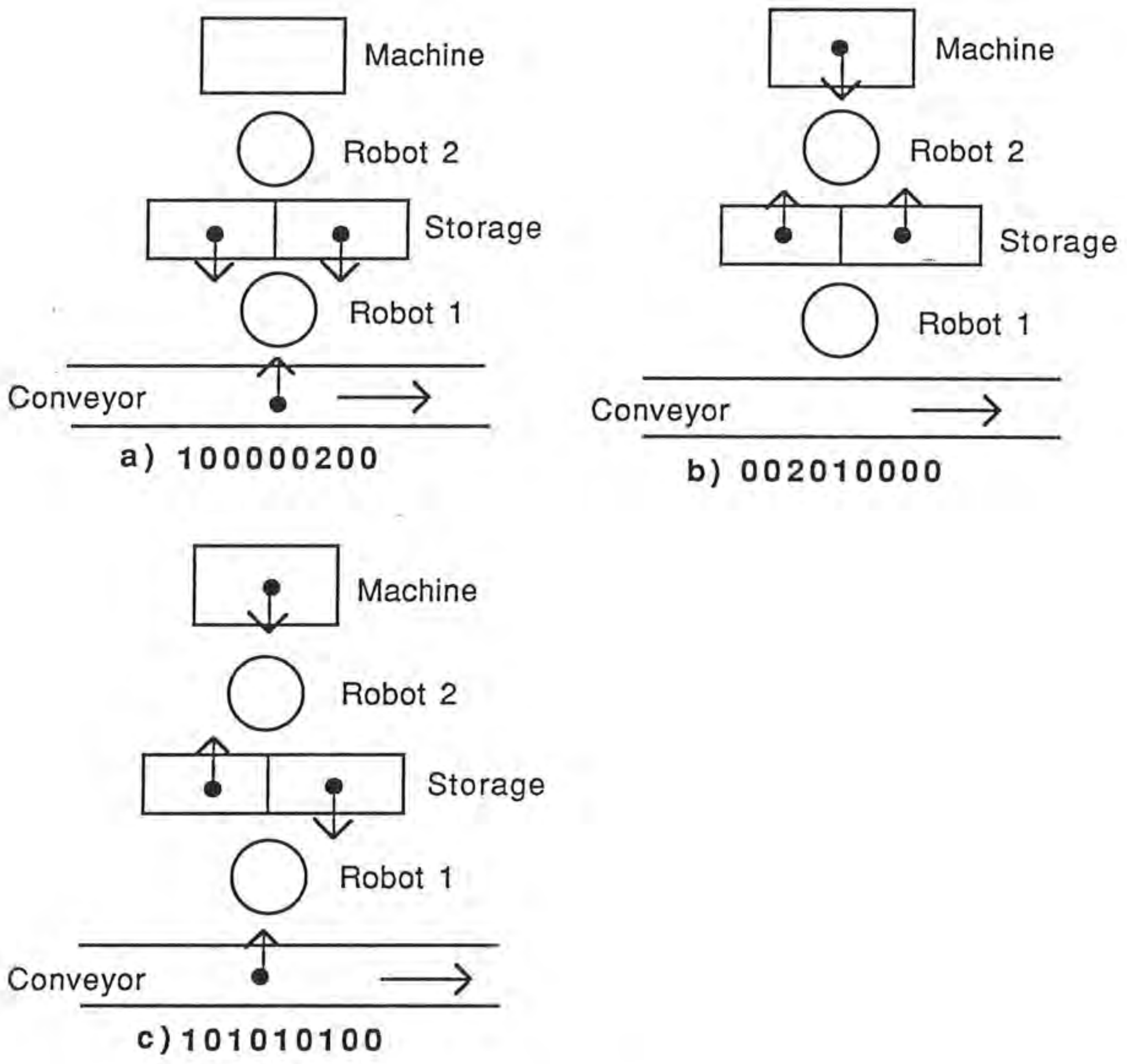


Fig. 3. Deadlocks in a manufacturing line.

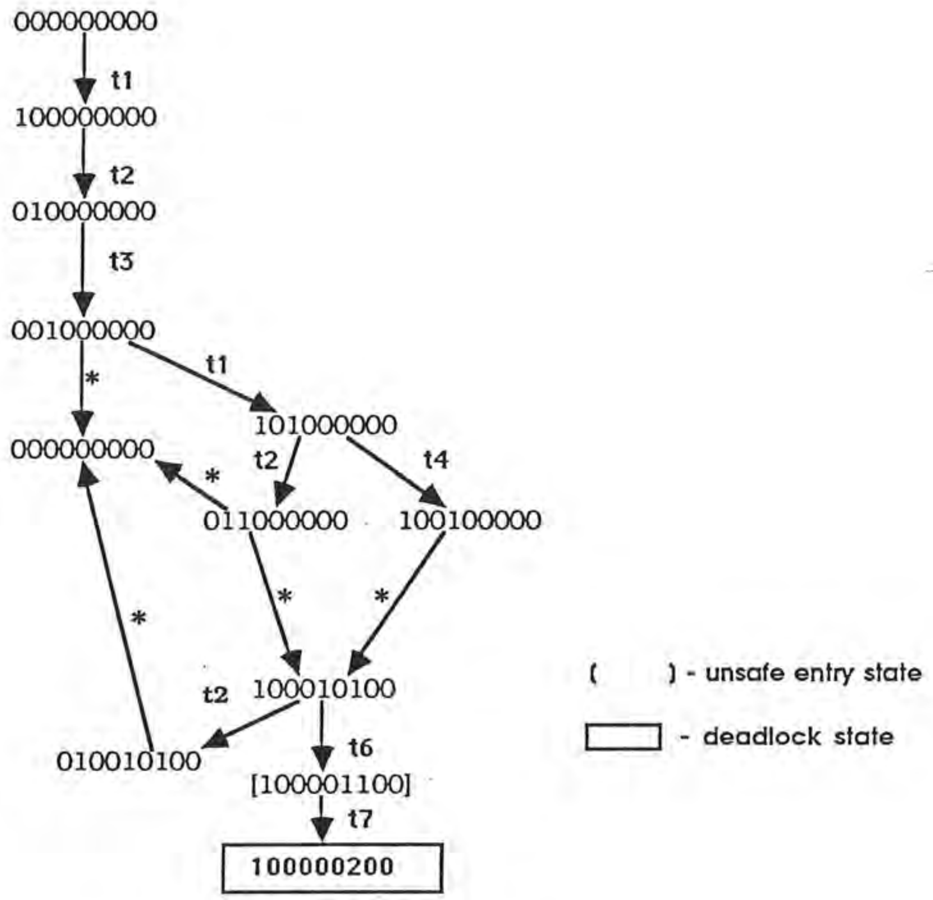


Fig. 4. The state transition diagram (part).

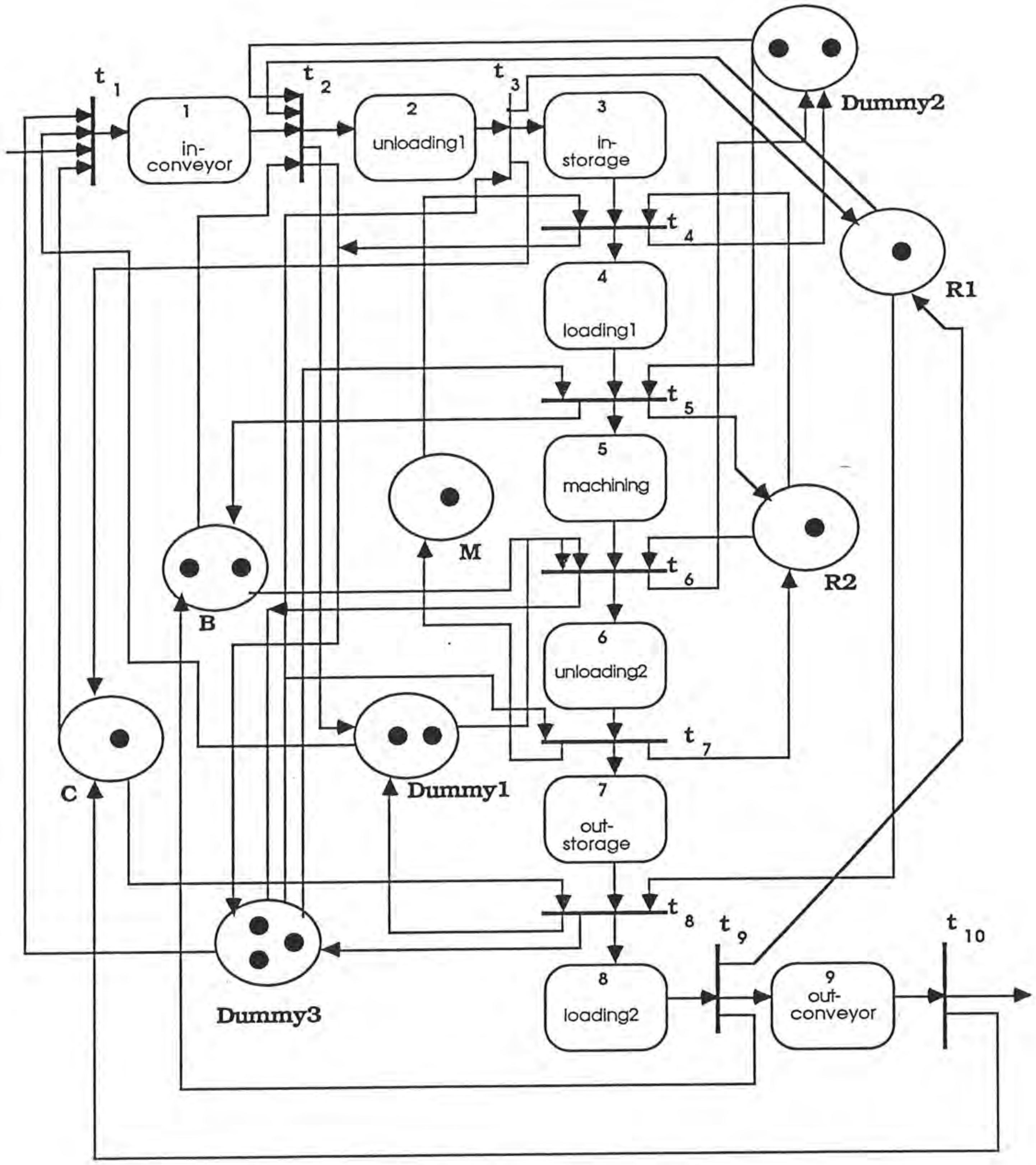


Fig. 5. The Petri net-based sequence controller with three dummy resource types.

		Algorithms			
		1	2	3	4
Test Cases	A	22.1	8.0	8.6	1.1
	B	93.2	28.6	31.2	2.1
	C	388.9	99.3	110.7	3.1
	D	0.2	0.1	0.1	0.2
	E	0.2	0.1	0.2	0.2
	F	0.3	0.2	0.2	0.3
	G	1.6	0.9	1.0	1.1
	H	1.7	1.0	1.0	1.1
	I	1225.4	249.5	122.2	82.5

Table 1. Test results of four algorithms (run times in secs.).