A Petri-Net Based Hierarchical Structure for Dynamic Scheduler of an FMS : Rescheduling and Deadlock Avoidance

Y. L. Chen, T. H. Sun and L. C. Fu

Department of Computer Science and Information Engineering
National Taiwan University, Taipei, Taiwan, R.O.C.

Abstract

Flexible manufacturing systems (FMSs) have received considerable attention and evolve to be one of the fastest growing industrial field in the last decade. In these systems, much higher efficiency of manufacturing can be achieved (owning to their intrinsic flexibility) provided a good scheduling policy is adopted. In this paper, we propose a dynamic scheduler with a hierarchical structure to cope with the unavoidable disturbing events in such dynamic systems like an FMS. In particular, we based on our earlier work [6] handle the rescheduling problem as we as the deadlock avoidance problem. The merit of this work is its completeness in considering all possible components in an FMS, including AGV transportation system.

Abstract

Flexible manufacturing systems (FMSs) have received considerable attention and evolve to be one of the fastest growing industrial field in the last decade. In these systems, much higher efficiency of manufacturing can be achieved (owning to their intrinsic flexibility) provided a good scheduling policy is adopted. In this paper, we propose a dynamic scheduler with a hierarchical structure to cope with the unavoidable disturbing events in such dynamic systems like an FMS. In particular, we based on our earlier work [6] handle the rescheduling problem as we as the deadlock avoidance problem. The merit of this work is its completeness in considering all possible components in an FMS, including AGV transportation system.

1 Introduction

An FMS may be viewed as a discrete event dynamic system, and its scheduling problem is known to one of the NP-hard combinatorial problems. For this reason, it can be alternatively tackled by means of heuristic or approximate scheduling procedures, which are usually allowed in most of the cases, to determine optimal or nearly optimal solutions. On the other hand, real-time control operations of an FMS often involves decisions as how to solve occasional problems due to addition of new parts, machine failure, set-up changes, maintenance, etc., which may be viewed as rescheduling. Such a problem also belongs to the class of NP-hard combinatorial problems. So, a good rescheduler is critical to the performance of the system. Recently, there has been much interest in researches on rescheduling that has been focused on how to improve the system performance.

The rescheduling problem is generally more difficult than the scheduling problem, because the former problem needs to be accomplished in real-time and, hence, remains to be a challenging problem. On this regard, the work [3], proposes a prototype of an intelligent real-time rescheduler for a job-shop FMS which is actually a decision support system to help the manager to handle the disturbance based on an expert system scheduling module and a high level Petri-Net simulator module. Another work [2] solves this problem based on the simulated annealing to technique obtain a modified schedule by rescheduling. Others like [6] proposes a rule-based on-line scheduling system for an FMS that generates appropriate priority rules to select a transition to be fired from a set of conflicting transitions. It should be noteworthy that both [6] and [7] include rescheduling in their scheduling method, in which [6] designs a real-time scheduling subsystem to select one randomly among the conflicting transitions whereas [7] uses timed place Petri-Net to solve this problem. The work [8] proposes a production rule base in such way that the dynamic scheduler can read different processing rules as candidates from the rule base and make real-time decisions accordingly.

In flexible manufacturing systems, many kinds of parts are processed by many kinds of machines and are transferred by automated-guided vehicles ($AGVs$). The transferred one may go from a buffer to a specified machine, from a machine to a buffer, or between two different machines. In such a complex environment, lack of proper control will cause the deadlock to occur. But the problem of an FMS deadlock is ignored by most researchers who pay most attention to scheduling and control. So far, several efforts have been focused on this problem. In most of the cases, deadlock prevention and deadlock avoidance methods are used because those methods can make the FMS utilization better. If we want to avoid a deadlock, then deadlock avoidance is a major issue. There have been some existing results on deadlock avoidance pro-

posed in the literature. For example, [4] uses PN-based models to do deadlock prevention by static resource allocation policies, and deadlock avoidance by dynamic policies; [5] uses a deadlock avoidance technique; [9] proposes a systematic method for designing locks and interlocks for deadlock avoidance by using the reachability graph of the Petri-Net model; [1] proposes a destination graph in order to analyze the flow of workpieces in job-shop manufacturing system, and hance achieves the deadlock avoidance by eliminating the pre-deadlock condition.

Section 2 proposes a hierarchical structure consisting of four levels to solve the dynamic scheduling problem. Section 3 introduces the modeling technique used for the system simulator . Section 4 discussed the rescheduling problem and solves it mainly using the A* search technique. In section 5, problem with deadlock is introduced and is solution algorithm is suggest. Finally, conclusion is made in section 6.

2 Hierarchical Structure for a Dynamic Scheduler

Schedules for medium and long term operations of a production system must be generated a *priori*. Unfortunately, the future states of the system cannot be known at the time schedules are generated. Many unpredicted disturbances will arise during production, but they were not accounted for in the original schedule. The typical solution to this problem is to apply standard scheduling methods to reschedule the system on occurrence of such a disruption. Nevertheless, and optimal policy is to perform dynamic scheduling which includes some sort of discrete control of production systems to handle the above-mentioned disturbing events flexibly and effectively. But, except the small sized systems, the problem for general systems is rally an NP-hard problem. In order to reduce its difficulty as well as the complexity as much as possible, hierarchical control methods should be employed.

Generally speaking, a scheduling system may work either on-line or off-line. An off-line scheduling system is relatively easier to design but is more difficult to perform rescheduling, whereas an on-line scheduling system is difficult relatively more to design but is easier to do rescheduling. So, how to balance the trade-offs between those two is critical. To this aim, we propose a hierarchical structure consisting of several levels, such as *scheduler*, *process runner*, *controller* and *physical system*. Now, we define the function of each level and the relations among them.

Scheduler: Scheduler level generates a sequence of actions for achieving a given goal for each task, i.e., generates a complete plan for a given task before any of its operations begins. The specification of the operation sequence concerning a job (task) includes information about operation type and operation time. Usually, we can have two classes of different schedulers. One class is to consider transporting time, whereas the other does not. But roughly speaking, the scheduler level can be viewed as a coarse schedule generator which mainly generates the order of operations concerning the job and seldom considers the transporting time. To sum up, the main function of this level consists of

1) getting jobs information and the layout of the system, and
2) generating a coarse schedule for the whole jobs.

Process Runner: The level called process runner modifies the existing production (process) plan according to the run-time situations when necessary, i.e., do rescheduling when one of the following situations occur: a physical machine breaks down, an urgent task or a new part is added in, unacceptable time delay in execution is observed, or a deadlock is perceived, etc. The function of the process runner is only to modify the originally planned order of operations under abnormal conditions but with the minimum change. For this, an objective function in order to decide the priority of conflict tasks has to be defined. In this level, a simulator and a deadlock detection mechanism are also included. The simulator can evaluate various scheduling rules while analyzing the effects due to these different rules on several performance measures possibly under different conditions by using simulation. Therefore, the process runner can be viewed as a detailed schedule generator which generates the detailed order of jobs operation, which certainly includes the vehicle routing schedule.

In the absence of abnormal condition , the simulation should provide a "better" way of the coarse schedule received from the scheduler, which is readily implementable by the physical system. However, if a deadlock is detected by the deadlock detection mechanism or if some abnormal condition described earlier suddenly takes place, the process runner is supposed to perform rescheduling to avoid deadlock or to appropriately respond to that abnormal condition. To sum up, the function of this level includes

1) generating a detailed schedule for the whole jobs, including the AGV routing schedule;
2) recovering from unpredictable disturbing events in the system;
3) modifying the schedule generated by the scheduler for the reasons described above;
4) simulating the system behavior under different scheduling rules to find a "better" refined schedule;
5) detecting the potential deadlock situation;
6) activating the global rescheduler when necessary.

Controller: This level is the interface between the logical system and the physical system, which directly controls the whole system to execute commands generated by higher level, i.e., it runs the

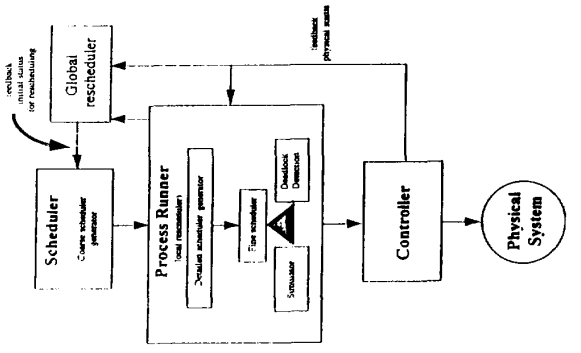

Figure 1: Architecture of a hierarchical dynamic scheduler

physical operations interpreted from the hierarchical dynamic scheduler commands. So, controller executes tasks on the physical hardware based on the time table decided from the higher levels. To sum up, the function of this level includes

1) driving the machine to perform operations;
2) controlling the physical operations onto real world system;
3) feedbacking physical status information to the upper levels.

Physical System: The real world system layout. It contains physical machines, resources, AGV's, Automatic Storage/Retrieval System (AS/RS), etc.

Besides these, there is a global rescheduler which functions only when the process runner can not solve the problem. When this occurs, the process runner trigger the global rescheduler to feedback initial status, i.e., the status to which the current status can be reset, for rescheduling process to scheduler in order to regenerate a new coarse schedule. Finally, the architecture of the whole hierarchical dynamic scheduling system is shown in Fig. 1.

3 Petri-Net Based Modeling

In the level with process runner as has been described previously, detail system modeling is crucial to the success of the functioning of that level. Since the dynamics of an FMS are often very complex and, hence, how to model an FMS is an extremely complicated problem. There exist many ways to model an FMS, like colored Petri-Net, Markov chain processes, queuing networks, etc. Furthermore, there are many system properties in an FMS, such as resource sharing, concurrency, routing flexibility, unexpected changes, and deadlock, etc. It is well-known that Petri-Net is a powerful tool for describing and analyzing asynchronous and concurrent system behavior mainly because it can represent the precedence relations of asynchronous and concurrent systems. Therefore, Petri-Net model is very suitable for modeling and analyzing such dynamic processes.

Petri-Net is also a graphical representation to understand the system. Traditionally, Petri-Net is often used as models of automated manufacturing systems to represent the controlled system operation. This model is, then, used for analysis of the system properties or simply for simulation. In light of this, here we use Petri-Net to model flexible manufacturing systems. Generally, Petri-Net does not include time and precedence relationships in the model, but time, however, is a crucial consideration in our problem. Hence, we need to include time in our model, called timed place Petri-Net, to model our system. Timed place Petri-Net model associates time with places, which represent the periods of time during which the tokens have to stay in the places before they can become available again. Petri-Net based simulation model for a general FMS with multiple task flows and transportation control. Due to shortage of space, the modeling well not be presented here, but the very technique will be adopted in this paper. In our earlier work [10], we have proposed a detailed.

4 Rescheduling

Most of rescheduling methods solve the rescheduling problem by using the original scheduling methods to reschedule the system. But our goal is to reschedule the original schedule by different policy and still retain the schedule efficiency. So, rescheduling is in general a more difficult problem than scheduling problem because the time allowed to be spent is quite limited, i.e., a decision must be made by the rescheduling system based on the some optimization criterion defined therein itself immediately.

Many criteria have been proposed for rescheduling (e.g. makespan, mean flow time, lateness, etc.). We select the makespan here as our criterion, where makespan is defined as the maximum job completion time. To be realistic, we assume job preemption is not allowed. So, the present objective is to find a schedule which gives a sequential order of performing the jobs so that the makespan is minimized. Once the Petri-Net model is constructed, a feasible schedule for these jobs can be obtained by simulating the Petri-Net and searching through the reachability graph. Simply because the search is based on the Petri-Net model, the search space is can be constrained only the feasible schedules.

In this paper, the rescheduling is done by the process runner. The purpose of process runner is to relax the existing schedule generated by the scheduler to a linearly ordered sequence without time consideration. This can drastically reduce the complexity of the original scheduler. At the same time, the process runner can be viewed as an on-line scheduler.

4.1 Timing for Rescheduling

Generally speaking, the time to do rescheduling is when a disturbing event occurs. All the operations in the schedule which have not been completed yet at that time must be rescheduled. Because we assume that jobs are non-preemptive, when the start time of some operation is before the time to perform rescheduling and that particular operation is not yet completed, we will still retain the time schedule for that particular operation in rescheduling process. The appropriate occasions for doing rescheduling are listed below:

1) a physical machine breaks down;

2) an urgent task is added into the system;

3) a new part is added into the system;

4) time lag in execution of the original schedule;

5) a deadlock occurs.

In consequence, we can regard the rescheduler as a real-time scheduler which is going to be processed every time immediately after the disturbing event occurs in the system. For example, when a machine breaks down, the controller senses that event and then place a token to the place, which is interpreted as disappearance or identification, or both of that machine. Or, when a sensor which is controlled by the controller has not sensed the arrival of some AGV at some destination yet along the route determined by scheduler due to possible time delay. All these occasions will ask the rescheduler to perform rescheduling.

4.2 Rescheduling Algorithm

In the following, we propose a method to do rescheduling. First we assume each job consists of a set of operations to be performed by various machines. Below we introduce some assumptions and notations in order to treat the problem.

We assume that the job shop has N jobs in total and M machines, and

$$\begin{cases} J_i : & \text{the } i\text{th job, } 1 \le i \le N \\ M_j : & \text{the } j\text{th machine, } 1 \le j \le M \ , \end{cases}$$

where $J_i = (O_{i,1}, O_{i,2}, \ldots, O_{i,n})$ is the order of operations of the ith job, i.e., the operations are performed in the increasing order of the index n. Sometimes an operation may require a shared resource such as an AGV to transfer the part to its next destination machine to process.

After the Petri-Net model of the problem is constructed, we use a search algorithm to find a solution described below. Before that ,we first introduce some notations. Let the function $c(M_i, M_j)$ be the actual cost of a minimum-cost path from the marking M_i to the marking M_j. Then, the cost of a minimum-cost path from the marking M to some goal marking M_g is given as $c(M, M_g)$. Then, we define $h^*(M)$ as follows:

$$h^*(M) = min\{c(M, M_g) \mid M_g \text{ is a goal marking } \}$$

so that any path from marking M to the goal marking M_g that achieves $h^*(M)$ is an optimal path. Another function $g^*(M)$ is defined as follows:

$$g^*(M) = c(M_0, M), \text{ for any marking } M \text{ reachable from } M_0.$$

Now we define the function f^* as follows:

$$f^*(M) = g^*(M) + h^*(M)$$

The value of $f^*(M)$ at marking M indicates the actual cost of an optimal path from M_0 to M plus the cost of an optimal path from M to a goal marking M_g. So, we let the function f be an estimate of f^*. Then, $f(M)$ is defined as follows:

$$f(M) = g(M) + h(M) \ ,$$

where g is an estimate of g^* and h is an estimate of h^*.

From the above, this algorithm has three functions f, g, and h, which hence constitute the evaluation function for search. These three functions are defined as follows:

$f(M)$: is an estimate of the minimum cost, i.e., the cost from the initial marking M_0 to the goal marking M_g along an optimal path which goes through the marking M.

$g(M)$: is the cost observed so far from the initial marking M_0 to the current marking M.

$h(M)$: is an estimate of the cost from the marking M to the goal marking along an optimal path which goes through the marking M.

The search method which we used to solve the rescheduling problem is the A^* search method. The A^* search algorithm is a minimum-cost graph search algorithm. It can be regarded as a branch-and-bound search algorithm which uses the dynamic programming principle with a cost estimate from the current state to the goal state. It can be guaranteed to find a minimum cost solution path if the heuristic function $h(M)$ from the current marking to the goal marking is a lower bound of $h^*(M)$. Now, we are ready to describe the basic method of our algorithm as follows:

State Description: State is used to represent the marking of the Petri-Net.

Initial State: The initial state is given as $M_0 = (t_1, t_2, \ldots, t_p)$, where:

$$\begin{cases} M_0 & : \text{initial marking} \\ t_k & : \text{the token number at the place } P_k \\ p & : \text{the number of places} \end{cases}$$

Goal State: The goal state is reached when all jobs are completed.

Operator: Each marking is one node of the A^* search algorithm. The node expansion in the A^* search algorithm at every step is according to the evaluation function defined below.

Obviously, the evaluation function will be defined as the objective function. Since our goal is to minimize the makespan for the whole jobs (i.e., the time required to finish all the jobs), the evaluate function here is chosen to be:

$$f(M) = \max\{C_1, C_2, \ldots, C_k\} = \{\text{makespan of } M\}$$

where the notation C_k above is the completion time of the job k. In order to use the A^* search algorithm to find an optimal and feasible solution, it needs to be modified slightly to include the detection deadlock mechanism is as follows:

A^* search include Deadlock-detection algorithm:

INPUT: An optimal routing assignment S obtained from the scheduler.

OUTPUT: An optimal deadlock-free routing assignment with minimum cost routing.

Step 1. Put the initial marking M_0 on the list $OPEN$ and calculate its cost function value. Then, initialize the upper bound on the makespan to be $MAXINT$.

Step 2. If $OPEN$ is empty, terminate with failure.

Step 3. Select a marking from the $OPEN$ list with the minimum cost (i.e., the first marking of the list).

a. Remove the marking M from the $OPEN$ list and put M on the list $CLOSED$. Let the current marking $M_c = M$. If several markings have the same cost value, choose a goal marking if it exists; otherwise, choose among them arbitrarily.

b. Here, we must check if the marking is deadlock-free by the *Deadlock – detection* algorithm to ensure solution feasibility. If the marking is deadlock-free, then accept it; otherwise, abort it and repeat *Step 3* to select the next again.

Step 4. If M is the goal marking, construct the optimal path from the initial marking to the final marking and terminate with success. And, the optimal deadlock-free routing assignment has been found; otherwise, continue.

Step 5. Find all the enabled transitions from the current marking M_c.

Step 6. Generate the next marking, or successor, for each enabled transition, and set pointers from the next markings to M.

Step 7. For every successor M_s of M, do the following:

a. If M_s is already on either $OPEN$ or $CLOSED$ list, direct its pointer along the path yielding the smallest $g(M_s)$.
If M_s is on $CLOSED$ and requires pointer redirection, put M_s on $OPEN$ list.

b. If M_s is neither on $OPEN$ list nor on $CLOSED$ list, calculate $h(M_s)$ and $f(M_s)$, and put M_s on $OPEN$.

Step 8. Reorder $OPEN$ by the increasing magnitude of f of the markings.

Step 9. Go to *Step 2.*

Because of the heuristic rules and constraints, we can substantially reduce the solution space of the problem. It can make the searching process much simpler. Finally, we then apply the A^* search to the Petri-Net based hierarchical dynamic scheduler to find an optimal solution to complete all requirements and operations.

5 Deadlock Avoidance and Deadlock Recovery

System deadlock is a serious problem in a flexible manufacturing system, which is a situation where a set of jobs are waiting indefinitely for one another to release certain resources. In other words, each job in the set is waiting for a resource being held by another job in the set while holding a resource needed by some other jobs in the set. Consequently, the set of jobs are in circular waiting. In an improperly designed FMS, deadlocks may be resolved by clearing of buffers or machines, and by restarting the system from an initial condition known to produce deadlock-free operation under nominal production conditions.

The deadlock problems can be classified into deadlock prevention, detection, recovery and avoidance. Here, for our interest, we only deal with the deadlock avoidance problem, which is to test a request to see if it will cause deadlock. Deadlock avoidance is an important issue for effective control of an FMS. To avoid deadlocks and to allow design flexibility at the same time, we try to disable the events that may lead to deadlock at certain states. So, deadlock avoidance is an attempt to falsify one or more of the necessary conditions in a dynamic way by keeping track of the current state and the possible future conditions (i.e., disable some conditions when a deadlock becomes a possibility in the immediate future).

Now, we are prepared to discuss how to solve a scheduling conflict problem, and propose a method to find deadlocks should they have happened.

5.1 Reasoning for Deadlock

When a deadlock occurs, the following four conditions must be satisfied at the same time (i.e. the necessary conditions for a deadlock to occur): Mutual exclusion, Hold and wait, No preemption, Circular wait [11].

From the conditions described above, we know when requesting a resource, one of the following situations may have happened:

(1) requested without success: For example, a job requests a machine which is busy now and hence the request is unsuccessful.

(2) requested with success: For example, a job requests a machine which is idle now and hence the request is successful.

(3) release of a resource: For example, a job releases a machine which has completed its assigned operation for the job.

In the first two situations, we must check if a deadlock cycle exists because those two situations constraint the system tightly. Situation 3 will not foresse deadlocks because it releases a resource.

5.2 Deadlock Avoidance Algorithm

We propose a method to do deadlock detection as follows. We use the matrix multiplication and *Wait-for* graph to find deadlock situations. From above, we know when a deadlock happens, there must exist at least one cycle in the *Wait-for* graph. One thing we must note is that the cycle is a necessary, but not a sufficient condition for running into a deadlock. Therefore, when no cycle exists in the *Wait-for* graph, the system is in a safe state. But the system is potentially unsafe if there is a cycle in the *Wait-for* graph. The *Wait-for* graph is adapted from *Resource-Allocation* graph.

Formal description of the *Wait-for* graph is given below. We define a directed graph (i.e. digraph) $G = (V, E)$ consisting of a node set V and an edge set E where $V = \{1, 2, \ldots, |V|\}$ and $u, v \in V$, $(u, v) \in E$ is an edge from u to v. In such a graph, nodes correspond to the resources being held or requested, and arcs correspond to the wait relations between the resources. Our goal is to find if any cycle exists in such a graph. A cycle is a path from u to v, where $u = v$, and the length of the path is not equal to zero. A path from u to v is a sequence of nodes $v_0, v_1, \ldots, v_k$, such that $v_0 = u$ and $v_k = v$ and each pair $(v_i, v_{i+1}) \in E$, for $0 \leq i < k$, so that the length of the path is equal to k.

First, we define the directed incident matrix in order to represent the *Wait-for* graph as follows:

$$A = \begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix}$$

where

$$a_{ij} = \begin{cases} 1 & \text{when job i is holding one resource while waiting for the resource held by job j} \\ 0 & \text{otherwise} \end{cases}$$

then, we define:

$$A^n = A^{n-1} * A$$

and

$$A^n_{ij} = \sum_{k=1}^{n} A^{n-1}_{ik} * A_{kj} ,$$

where $i, j = 1, \ldots, n$, so that

$$A^n = \begin{vmatrix} a^n_{11} & a^n_{12} & \cdots & a^n_{1n} \\ a^n_{21} & a^n_{22} & \cdots & a^n_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a^n_{n1} & a^n_{n2} & \cdots & a^n_{nn} \end{vmatrix}$$

where

$$a^n_{ij} = \begin{cases} 1 & \text{when there exists one path with a length equal to n} \\ 0 & \text{otherwise} \end{cases}$$

By the method proposed above, we can calculate the matrix A^n. Any element in matrix A^n that is not equal to zero represents a path with length being equal to n. So, we can find all the circles in the *Wait-for* graph.

Because the complexity order of the matrix multiplication is $O(n)$. Therefore, in order to reduce the cost of calculation complexity, we can make some improvement in matrix multiplication method. This is because one cycle may appear in more than one matrix element. So, to avoid repeated calculation, we only need to calculate it once. For example, 1231 and 2312 represent the same cycle, and we need not calculate it twice. For the new method, A^n_{ij} is generated as follows:

$$A^n_{ij} = \sum_{k=i+1}^{n} A^{n-1}_{ik} * A_{kj}$$

where $i, j = 1, \ldots, n$.

Furthermore, we need not examine all non-zero elements. In fact, we only have to examine the diagonal elements. Only a diagonal element is possible to create a cycle, because the start node of the first edge is the same as the end node of the last edge in a cycle. So, we only have to find out to the diagonal terms which are not equal to zero and backtrack to find the circuit path, which is then a possible candidate for deadlock situations.

Up to now, a simple algorithm that will perform the function described above can be presented in following, whenever one of following occasions occurs.

1) a new part is added into the system;

2) an urgent task is added into the system;

3) a part is requested by a new operation;

4) request of a resource can not be permitted;

Deadlock-detection Algorithm:

Step 1. Find all the *Wait* relations between each pair of the jobs and create the *Wait-for* graph for the current state;

Step 2. Call *Detect-cycle* algorithm;

Step 3. If (a cycle exists) then
Call *Resolve-deadlock*;
else
update the current status of the system to a new status;

Detect-cycle Algorithm:

Step 1. Create the directed incident matrix which represents the status of the wait relations in the *Wait-for* graph;

Step 2. Calculate the matrix by the method proposed previously (matrix multiplication);

Step 3. Check if there is any non-zero element;
If found, then the element is a situation which may potentially cause a deadlock;

To resolve a deadlock, we can select a victim from the cycle according to some special priority and move it to a reserved storage buffer. When some special conditions are satisfied, we can release it and let the operation continue. The algorithm is summarized as follows:

Resolve-deadlock Algorithm:

Step 1. Determine the set of jobs which potentially cause the deadlock cycle;

Step 2. Select a victim from the set of jobs by using priority rules, and move it to a reserved storage buffer;

Whenever there is a deadlock that can not be avoided, then the productivity of the system will be affected considerably or even the production of the whole system will be stopped. Consequently we must avoid the deadlock as much as possible. But if we hope to consider all the status that deadlock may occur, it is impossible to do so because in a real-world system too many unpredictable events may arise. When any of these events happen, it is very likely to lead the system to a deadlock status.

6 Conclusion

This paper proposed a Petri-Net based hierarchical structure for a dynamic scheduler of an FMS, consisting of several levels, including *scheduler*, *process runner*, *controller* and *physical system* modeled by a Petri-Net using bottom-up approach. In particular, we discussed the problems of rescheduling and deadlock avoidance. and applied the A^* search and matrix multiplication methods to solve these problems. A prototype of this scheduler is developed for our experimental FMS in National Taiwan University, but their results are not included due to the shortage of space.

References

[1] Koji Takahashi, "Fundamental Control to Avoid Deadlock for Job-shop Manufacturing Systems", *JAPAN/USA Symposium on Flexible Automation,* Vol.2, pp.1703-1706, ASME 1992.

[2] Yasunori Watatani, and Susumu Fujii, "A Study On Rescheduling Policy In Production System", *JAPAN/USA Symposium on Flexible Automation,* Vol.2, pp.1147-1150, ASME 1992.

[3] Zhiming Wu, "Combining of Expert System and Simulator for FMS Rescheduling", *JAPAN/USA Symposium on Flexible Automation,* Vol.2, pp.1143-1146, ASME 1992.

[4] N. Viswanadham, Y. Narahari, and T. L. Johnson, "Deadlock Prevention and Deadlock Avoidance in Flexible Manufacturing System Using Petri Net Models", *IEEE Trans. Robotics and Automat,* Vol.6, No.6, pp.713-723, Dec. 1990.

[5] Perenc Belik, "An Efficient Deadlock Avoidance Technique", *IEEE Trans. on Computers,* Vol.39, No.7, July 1990.

[6] Itsuo Hatono, Keiichi Yamagata and Hiroyuki Tamura, "Modeling and On-Line Scheduling of Flexible Manufacturing Systems Using Stochastic Petri Nets", *IEEE Trans. on Software Engineering,* Vol.17, No.2, pp.126-132, February 1991.

[7] Doo Yong Lee and Frank DiCesare, "Experimental Study of A Heuristic Function for FMS Scheduling", *JAPAN/USA Symposium on Flexible Automation,* Vol.2, pp.1171-1177, ASME 1992.

[8] Paul G. Ranky, "Intelligent Planning and Dynamic Scheduling of Flexible Manufacturing Cells and Systems", *JAPAN/USA Symposium on Flexible Automation,* Vol.1, pp.415-422, ASME 1992.

[9] N. Viswanadham, T.L. Johnson, and Y. Narahari, "Performance Analysis of Automated Manufacturing Systems with Blocking and Deadlock", *Proceedings of The Second International Conference on Computer Integrated Manufacturing,* pp.64-68. May 1990.

[10] Chin-Jung Tai,and Li-Chen Fu, "A Simulation Modeling for a Flexible Manufacturing System with Multiple Task-Flows and Transportation Control Using Modular Petri-Net Approach", Proc. 9th International Conference on CAD/CAM , Robotics & Factories of the Future, August, 1993.

[11] Andrew S. Tanengaum, Modern Operating System, Prentice-Hall, 1992.