Table of Contents

List of Figures

Abstract

In this paper we address the problem of realizing the benefits of pre-computed schedules in the face of a partially unpredictable execution environment. We focus specifically on the problem of manufacturing production scheduling, where advance planning is crucial to overall factory performance but is, at the same time, confounded by the unpredictability of factory operations. We present a scheduling framework where decision-making responsibility is shared between a *global scheduler*, responsible for establishing and maintaining execution constraints in accordance with overall performance objectives, and a *local dispatcher*, responsible for containing execution within globally imposed constraints and notifying the scheduler when containment is no longer possible. We identify the sources of local executional flexibility that can be expected in a pre-computed schedule, and describe an execution-time scheduler (the dispatcher) capable of exploiting this flexibility.

1. Introduction

In broadest terms, this paper is concerned with realizing the benefits of pre-computed schedules aimed at optimizing global performance objectives in the face of a partially unpredictable execution environment. The specific domain of interest is manufacturing production scheduling, where the problem is one of allocating resources to production activities over time to produce a factory behavior that satisfies product demands in a timely and cost-effective manner. The development of good solutions to the manufacturing production scheduling problem is confounded by two realities: the *combinatorial complexity* of decision-making and the *executional uncertainty* of factory operations. Problem complexity derives from the need to coordinate the simultaneous execution of many production processes, each a specific temporally ordered set of process steps that require use of shared resources, in a manner that tends to optimize the overall performance of the factory. Performance concerns argue strongly for advance development of schedules, providing the opportunity to anticipate important process interactions (in particular resource contention) and impose execution constraints that minimize their degrading effects on performance objectives. At the same time, the unpredictability of factory operations (e.g. process step variability, unexpected resource unavailability, quality control failures) inevitably forces deviations from prescribed actions, and tends to work against any attempt to exploit pre-optimized schedules. An advance schedule will be of limited use unless the guidance it contains can be continually adapted to current circumstances on the factory floor. Moreover, since scheduling/rescheduling takes time, practical solutions to the production control problem must strike an appropriate balance between system performance and system responsiveness objectives.

Ideally, one would like an advance schedule to impose sufficient constraints on execution so as to push the behavior of the factory in the right direction while leaving sufficient flexibility to insure an equitable partition of the computational load between global and local decision-making processes as execution proceeds. From the standpoint of performance, the dominant determinant in domains like production scheduling is efficient resource sharing among production processes. In contrast to the assumptions of most AI planning work (e.g. [15, 5]), there is typically relatively little complexity in determining the structure of the processes themselves (i.e. what activities need to be performed). Rather the complexity is in synchronizing their resource usage to produce a good global behavior (e.g. jobs completed by their due dates, minimization of work in progress, maximization of machine utilization, minimization of process throughput time, etc.). Given these characteristics, there are two basic dimensions along which an advance schedule might remain imprecise: resource assignments and process step timing constraints[1]. It is often the case that a process step can be performed with any of a number of substitutable resources. If these resources are unreliable and/or there is no strong reason to precisely commit (e.g. to minimize resource setup time), the schedule might instead dictate capacity from specific resource groups. Alternatively, a schedule need not commit in advance to specific start and end times for individual process steps, but instead maintain execution windows. Unfortunately, it is difficult to remain imprecise simultaneously along both dimensions without giving up the guarantee of schedule feasibility. Feasibility is important, since recognition of scheduling commitment violations as execution proceeds can then be treated as signifying a broken schedule and the necessity of schedule repair activity.

Historically, scheduling research has tended to focus exclusively on either the optimization or the control part of the problem. One body of work [6] has been concerned with producing optimal solutions under various problem assumptions. From an operational standpoint, such approaches can be viewed as purely predictive. They produce a complete set of commitments in advance of execution, and any surprises encountered during execution require complete resolving of the problem. The computational expense of producing a schedule is often high, but even in the case of an infinitely fast scheduler,

[1] We recognize the additional possibility of remaining imprecise with respect to the order in which the steps of a given process are performed; our target domain does not admit such opportunities so we ignore this issue.

frequent rescheduling might not be practical. A manufacturing organization gains momentum once execution begins and does not usually respond favorably to repeated resetting of goals.

Control oriented research [12] has alternatively focused on the development of local dispatch priority rules. These approaches can be seen as purely reactive. They are relatively insensitive to surprises, since commitments are only made as necessary to continue execution, and typically require only modest computational resources. However, overall factory performance is very much a function of the sensitivity of the local decision rule to the structure and operating conditions of the factory. Unless the dynamics of the factory are well understood and stable over time, such strict reliance on local decision-making leads to otherwise avoidable congestion and suboptimal performance.

More recent research in knowledge-based scheduling has attempted to close the gap between optimization and control perspectives. One approach [3, 4] suggests that combinatorial complexity can be used to solve the problem of executional uncertainty, and, conversely, that executional uncertainty can be used to solve the problem of combinatorial complexity. The part of the system which builds the schedule operates under the principal of least commitment, generating a set of good schedules (rather than just one) with the certainty that as execution proceeds, circumstances will prune the members of the set which are inappropriate. The part of the system which executes the schedule is based on the principal of opportunism, manipulating the set of schedules such that its real time decisions will leave as many options open for the future as possible.

Other knowledge-based scheduling efforts have explored frameworks for integrating predictive and reactive scheduling processes. OPIS [13, 10, 14] defines an incremental scheduling methodology that is focused by repeated analysis of current solution constraints and objectives, which is used as a basis for both generating schedules in advance of execution and revising them as circumstances warrant. The schedule revision strategies studied here [11] are motivated primarily by global performance concerns, attending secondarily to schedule stability and system responsiveness concerns. From the standpoint of executional flexibility, a representation is employed that enables inprecision in resource assignments (i.e. provides an ability to allocate capacity from resource groups without naming individual resources). In SONIA [2], a system which generates and revises schedules according to a similar methodology, a framework for explicitly controlling the extent of constraint propagation performed is proposed as a means of balancing the time spent revising the schedule against the quality of the reaction. Moreover, a schedule representation that retains executional flexibility with respect to timing details is introduced, in contrast to most other schedulers which generate and manipulate temporally "crisp" schedules (i.e. schedules with absolute start and end times). As suggested above, this temporal flexibility is achieved at the expense of an ability to remain flexible in the resource assignment dimension. [8] proposes an alternative framework for remaining temporally imprecise based on fuzzy logic (again at the expense of flexibility in resource assignments). DAS [1] partitions the overall decision-making effort among a hierarchically organized set of scheduling agents, where each agent is given responsibility for making and retracting specific commitments (time assignments on a particular machine, machine assignments in a given work area, due dates of current jobs). Schedule generation and schedule revision are again treated uniformly, in this case as a collective activity based on backtracking search. In reactive contexts, the hierarchical organization provides heuristic structure for incrementally enlarging the scope of the change required to the schedule.

In this paper, we present an approach to exploiting local flexibility in a globally motivated schedule during schedule execution. Our approach assumes an overall scheduling framework where decision-making responsibility is shared between a *global scheduler*, responsible for establishing and maintaining execution constraints in accordance with overall performance objectives, and a *local dispatcher*, responsible for containing execution within globally imposed constraints and notifying the scheduler when containment is no longer possible (and global reaction may be required). The framework is realized through cooperative management of a temporally crisp schedule that is initially produced by the global scheduler, and from which local temporal flexibility is derived and exploited during execution. For purposes of this paper, we assume the existence of a global scheduler [16, 7] and focus on mechanisms for exploiting schedule flexibility during execution. Our approach is influenced by the specific characteristics of the target application domain, and we first briefly consider the manufacturing domain of interest.

2. The Problem Context

The scheduling framework described here is aimed specifically at scheduling facilities which fabricate semiconductor wafers each containing hundreds of die. A typical "fab" runs a few thousand jobs of a few tens of products over a few hundred machines[2] through three or four basic processes each with a few hundred linearly-ordered steps. Any particular machine is capable of running one or more steps, and any given step can run on one or more machines. Each job has a specified due date, and the overall cycle time of a job through the fab is measured in weeks. Furthermore, both machines and process steps employ very advanced technology, and the semiconductor market is quite volatile. Consequently, uncertainty is present both in terms of the health of the process and the machines, requiring preventive and emergency maintenance (called PM and EM respectively), and in terms of the delivery demands.

The overall system of interest is depicted in Figure 1. It includes the factory with its data collection subsystem and its command execution subsystem, and the scheduling system with its global scheduling subsystem and local dispatching subsystem along with their data model, knowledge base, and schedule. The data collector feeds the data model, and the dispatcher feeds the command executor. Both the scheduler and the dispatcher have access to the data model, the knowledge base, and the schedule. The factory with its data collector and control system work continuously. The scheduler is initialized by filling its data model with a whole factory snapshot and with which it builds a schedule for the next time period, twelve hours for example. The dispatcher executes the schedule, issuing control instructions to the factory, watching the data stream coming from the factory, and communicating with the global scheduler as necessary. The dispatcher recognizes problems and tries to either avoid or locally repair them. When it fails, the scheduler is called on to perform more global repairs. The cycle starts over when either the time period ends or the global scheduler determines that complete schedule regeneration is the most appropriate global repair.

For purposes of this paper, we assume that the data collector and the control system function perfectly.[3] Three message rates are important to the reactive component. The first is the rate of "expected" messages from the data collector (i.e. process step or preventive maintenance (PM) started or ended on time, emergency maintenance (EM) ended on time) which occur on average every 5 seconds. The second is the rate of "surprise" messages from the data collector (i.e. process step or PM or EM ended early or late, machine broken and EM started, job broken and placed on hold) which occur on average every 60 seconds. The third is the rate of messages from the dispatcher to the control system (i.e. start process step or PM step) which averages one every 10 seconds.

3. Exploiting Local Flexibility in the Global Schedule

Any machine in the factory spends its time either running production, being maintained, or sitting waiting. One objective of the global scheduler is to try to reduce machine waiting time since this will

[2] While other types of resources are used, only machines are considered here.

[3] Note that the first of these is a gross assumption. In fact the data collector misses some data, delivers some incorrect and/or inconsistent data, and is sometimes late. These problems are being addressed through a related project on intelligent data collection agents.

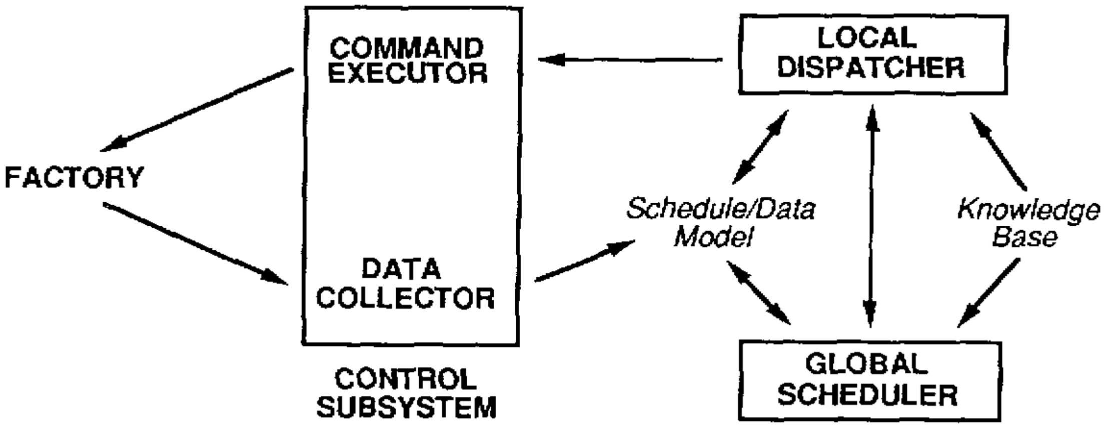

Figure 2-1: The Overall System

improve machine utilization, one of the performance metrics by which the behavior of the factory will be judged. But due to the interactions between activities in the factory, schedules with zero machine waiting time can never realistically be expected. A similar set of comments can be made for jobs. Any job in the factory spends its time either on a machine, on a transporter, on hold, or sitting waiting. The global scheduler also tries to reduce job waiting time since this will improve throughput time, another of the performance metrics by which the behavior of the factory will be judged. However, again due to the interactions between activities in the factory, schedules that completely eliminate job waiting time can never realistically be expected.

Even if "no wait" schedules were achievable, they would not be desirable. Given the uncertainty in the execution environment, any surprise would disrupt the whole schedule. We argue that, in fact, these inescapable waits can be used to hedge against surprises. They can provides a basis for distinguishing between locally containable problems and problems of global import that might develop in the schedule, and, as such, can be used to dynamically download a share of scheduling responsibility to the dispatcher as the factory operates. In the following subsections, we describe a framework for local decision-making based on this perspective. We first consider the representational assumptions underlying our approach.

3.1. Representing the Schedule and the Current Scheduling State

Central to our approach is a "distributable" representation of the current schedule. We represent the current schedule as a set of schedule objects (SOs). The most common type of SO is a production step. A production step has the basic form *<step job mach st et>*, where, for any given production step *s*: *step(s)* is the process step to be performed, *job(s)* is the job being operated on, *mach(s)* is the machine allocated for purposes of performing this production step, *st(s)* is the scheduled start time of the production step, and *et(s)* is the scheduled end time of the production step. The first two attributes of a production step uniquely identify a particular production activity to be performed. The final three represent the decisions of the global scheduler. As execution proceeds and production step *s* moves from pending to inprocess to completed, *st(s)* and *et(s)* are updated accordingly to reflect actual times. Two additional attributes, *mach–pred(s)* and *mach–succ(s)*, are also associated to indicate respectively the previous and next step to use *mach(s)* in the current schedule.

We also associate the following constraints with any given production step *s*:

- *job–pred(s)* = the immediate predecessor production step to *s* or *nil* (implying *s* is the first step of the process). If *job–pred(s)* exists, *et(job–pred(s))* ≤ *st(s)*.
- *job–succ(s)* = the immediate successor production step to *s* or *nil* (implying *s* is the last step of the process). If *job–succ(s)* exists, *et(s)* ≤ *st(job–succ(s))*.
- *dur(s)* = the duration of *s*
- *mg(s)* = the machine group from which capacity is required to perform *s*. We assume that machines are partitioned into mutually exclusive machine groups, and that each machine in a given machine group is capable of performing the same set of process steps in the same times. The precise resource requirement for a given production step *s* is use of one of the machines in the set designated by *mg(s)* for the entire duration of *s*. For a given machine group *MC*, *machs(MC)* indicates the set of constituent machines.

These constraints are imported or derived from generic product/process models when particular production steps are created and are assumed static over the lifetime of a step.[4]

Machine PM and EM steps constitute the other types of SOs in the schedule and are represented similarly to production steps. In these cases, a machine is identified rather than a job and there are no *pred*, *succ*, and *mg* constraints. There are also additional constraints on the scheduling decision variables: For any PM or EM step *s*, *mach(s)* cannot be changed, and, in the case of EM, neither can *st(s)*.

The current scheduling state is represented in part by status information in the data model that is associated with each job and each machine in the factory and is updated by the data collector. For each job *j*, *status(j)* indicates either "busy" executing on a process step, "waiting" for a machine to become available to begin the next process step, "in-transit" from one machine to another, or "on-hold" awaiting repair actions. If the job is "busy", the relevant *current–SO(j)* is designated. For each machine *m*, *status(m)* indicates either "busy" executing a job, "waiting" for a job to become available, undergoing a PM operation, or involved in an EM. Again, if the machine is "busy", the relevant *current–SO(m)* is designated.

To provide a basis for temporally containing local decision-making, we define four additional times relative to a given SO *s*: *pre–slack(s)*, *post–slack(s)*, *pre–idle(s)*, and *post–idle(s)* (see Figure 2.). If *s* is a production step, then *pre–slack(s)* = *et(job–pred(s))* and *post–slack(s)* = *st(job–succ(s))*. If *s* is a PM step, we assume *pre–slack* and *post–slack* is specified externally. If *s* is of either type, *pre–idle(s)* = *et(mach–succ(s))* and *post–idle(s)* = *st(mach–succ(s))*. These four times explicitly represent the residual temporal flexibility implied by the current schedule. The earliest time at which *s* can be started without effecting the *st* or *et* of any other SO in the system is *max(pre–slack(s), pre–idle(s))*. Likewise, *min(post–slack(s), post–idle(s))* designates the latest time at which *s* can end without impacting other SOs.

These earliest starts and latest finishes are of direct interest to the local dispatcher. Given that factories usually have actual throughput times three to five times higher than theoretical, and usually have machine utilizations in the range of sixty to eighty percent, the dispatcher will have substantial pre and post slack and idle to use in managing surprises. The heuristics it uses to perform repairs are explained in a later section, but have guaranteed locality of effect due to their exclusive use of pre and post slack and idle. This avoids contention between the global scheduler and the local dispatcher.

[4] Note that this is not actually always the case. Process step failures can occur, necessitating either the addition of rework production steps or the removal of the job from the system, and resulting in changes to the *job–succ* and *job–pred* constraints of different production steps. We assume, for this paper, that such situations simply result in putting the job on "hold" and informing the global scheduler.

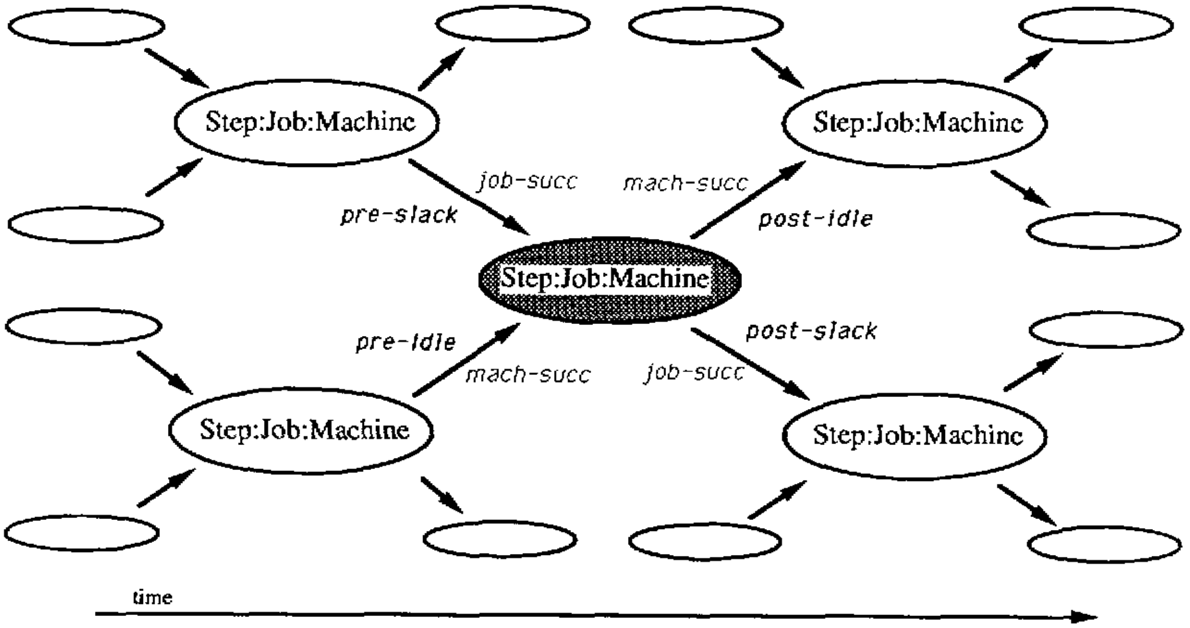

Figure 3-1: Pre and Post Idle and Slack

3.2. Partitioning decision-making responsibility

To mediate interaction between the global scheduler and the dispatcher, the set of all SOs is divided into six subsets, as shown in Figure 3. Two of these subsets, *Unscheduled* and *Scheduled*, are designated as the responsibility of the global scheduler. SOs in either of these subsets are accessible only to the global scheduler. Three other subsets, *Executing*, *OnDeck*, and *InTheHole*, are similarly defined to be the exclusive responsibility of the local dispatcher. Control of the flow of SOs between these subsets and the sixth identified subset (*Finished*) defines the overall operation of the system.

SOs are created in response to production and maintenance requests and loaded initially into subset *Unscheduled*. At this point, SOs have only a name and type, and an assigned job or machine. The main task of the global scheduler is moving SOs from subset *Unscheduled* to subset *Scheduled*, where SOs have all scheduling decision variables assigned. The scheduler is also responsible for SO's that are returned to *Unscheduled* from the dispatcher, indicating the necessity of a global schedule repair.

Within the jurisdiction of the dispatcher, there are three subsets of interest. Subset *Executing* contains all SOs that are currently executing in the factory. The SOs in subset *OnDeck* are the focus of attention of the dispatcher. These are the next to execute, according to the schedule, and the principal task of the reactive scheduler to move SOs from *OnDeck* to *Executing*, making whatever local adjustments are necessary before sending the start command for the SO to the command execution system in the factory. The SOs in subset *InTheHole* are the next to execute after the SO's in *OnDeck*, according to the schedule. They are under the control of the dispatcher to enable computation of *post–slack* and *post–idle* times for OnDeck-SO's without danger of their corruption by the global scheduler. When an OnDeck-SO is moved into *Executing*, InTheHole-SOs are pulled into *OnDeck* and Scheduled-SOs are pulled into

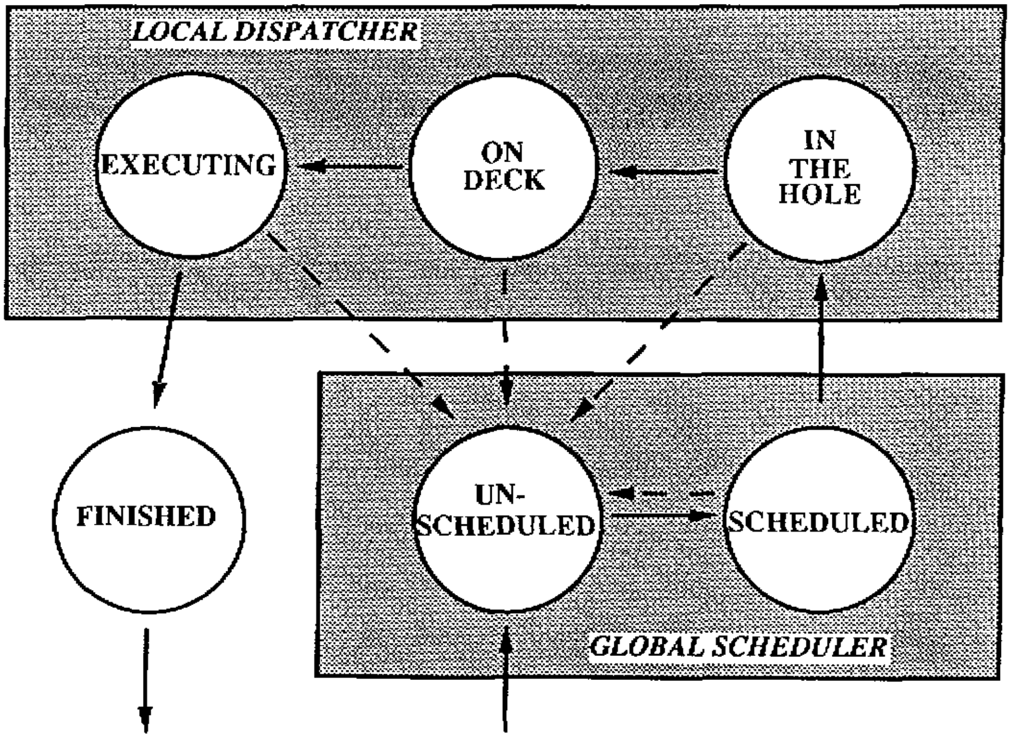

Figure 3-2: Sets of Schedule Objects

InTheHole (the solid arrows in Figure 3.). Failure of an Executing-SO to successfully execute, or an inability to contain the execution of a OnDeck-SO within the constraints imposed by the global scheduler, can lead to movement of SOs in *Executing*, *OnDeck*, or *InTheHole* back to *Unscheduled* (the dashed arrows in Figure 3.).

As SOs successfully finish executing, they flow into Subset Z, providing a record of the actual behavior of the manufacturing system.

3.3. Executing Schedules and Detecting Problems

As indicated above, the local dispatcher focuses on OnDeck-SOs, trying to start execution of each within the constraints imposed by the global scheduler. To decide whether execution within the constraints is possible, the dispatcher has at is disposal a number of relevant times for each OnDeck-SO: a *st* and a *et* (suggested by the global scheduler), a *pre–slack* and a *pre–idle* (reflecting current Executing-SOs), and a *post–idle* and a *post–slack* (reflecting InTheHole-SOs). To start execution of a particular OnDeck-SO *s*, the dispatcher must be sure that two requirements are met - *mach*(*s*) must be available, and *job*(*s*) must be available. A number of circumstances are possible depending upon the order in which the relevant events take place.

There is the possibility that *mach*(*s*) is reported available sometime between *pre–idle*(*s*) and *st*(*s*) but *job*(*s*) is not available until *st*(*s*), or that *job*(*s*) is available between *pre–slack*(*s*) and *st*(*i*) but *mach*(*s*) is not available until *st*(*s*), or that both become available just at *st*(*s*). In any of these cases, the dispatcher can just execute as planned. It is also possible that both *mach*(*s*) and *job*(*s*) are reported available sometime between *pre–idle*(*s*) and *st*(*s*). In this case, the dispatcher can be opportunistic and begin execution of *s* early.

Neither of these cases is especially difficult to resolve. More problematic circumstances arise if either *mach*(*s*), or *job*(*s*), or both become available after *st*(*s*). Here the dispatcher will be forced to try to rearrange execution. If the lateness of *mach*(*s*) or *job*(*s*) pushes *s* past (*minpost–idle*(*s*),*post–slack*(*s*)), it may not be possible for the dispatcher to begin execution at all. Inability to locally recover is almost assured upon notification that an Executing-SO has failed during execution. This implies that either an EM has been started on *mach*(*s*) or *job*(*s*) has been placed on hold, or both. In any of these failure modes, many OnDeck-SOs and InTheHole-SOs (and possibly Scheduled-SOs) will be effected and the dispatcher will most likely have to appeal to the global scheduler to perform a global repair.

As alluded to above, the detection of problems is tied to recognizing inconsistencies between the current state of the factory and the predicted schedule. The issue of when to raise the "alarms", however, can be seen from different perspectives. Clearly some events (e.g. broken machine, misprocessed job) serve themselves as alarms. But when a machine or job has the possibility of simply being late, there are tradeoffs involved in determining when the alarm is sounded and repair attempts initiated. On one hand, alarms can be tied to Executing-SOs. Under this scheme, whenever an actual start time is established, an expected end is computed and propagated to related OnDeck-SOs. This provides the opportunity to respond immediately to anticipated problems at the danger of spending computational resources to resolve problems that may fail to materialize. On the other hand, alarms may be tied to the current scheduled start times of OnDeck-SOs and deactivated if execution can be started before the alarm goes off. This scheme affords less flexibility for local reaction at the potential benefit of less computational expense. These ideas are being evaluated in the prototype.

3.4. Making Local Repairs

Since the dispatcher has access to only Executing-SOs, OnDeck-SOs, and InTheHole-SO's, the space of possible revisions to the schedule is bounded. In particular, all changes are temporally bounded by the *et* of all Executing-SOs (which are currently executing), and the scheduled *et* of all InTheHole-SOs (which guarantees non-interference with the global scheduler). This implies, for example, that InTheHole-SOs can only be moved earlier in time while OnDeck-SOs can be moved earlier or later and Executing-SOs can not be moved at all. Similarly, a OnDeck-SO or InTheHole-SO can only be reassigned to another machine in the same machine group as the machine it is currently assigned to. The recorded *post–idle* and *post–slack* constraints associated with Executing-SOs, the *pre–idle*, *pre–slack*, *post–idle*, and *post–slack* constraints associated with OnDeck-SOs, and the *pre–idle* and *pre–slack* constraints associated with InTheHole-SOs provide guidance in identifying local repairs that leave the *st*s and *et*s of Executing-SOs, OnDeck-SOs, and InTheHole-SOs in a consistent state, and a number of simple heuristics which exploit this information are used by the dispatcher to respond to detected problems and opportunities.

In the case of an opportunity, the *pre–idle* and *pre–slack* can be inspected to see how far the *st* of the OnDeck-SO in question can be shifted earlier. In the case of a problem, *post–idle* and *post–slack* indicates how far the *et*, can be shifted later. These simple "shift-earlier" and "shift-later" repair heuristics, if applicable, fix the problem with minimal disruption to the schedule, and combat the very common situation of "time jitter" in the executional system where events occur a few minutes earlier or later than expected. Another rarer circumstance to which simple shifting seems well suited is that resulting after a misprocessed job has been retracted from the schedule, leaving holes throughout the schedule into which OnDeck-SOs can be pulled.

The dispatcher can also exploit knowledge of equivalent machines in responding to problems. If a machine availability problem is detected in conjunction with a given OnDeck-SO *s*, the dispatcher inspects all of the Executing-SOs, OnDeck-SOs, and InTheHole-SOs contained in machine-group *mg(s)* for an idle interval which spans the interval from *st(s)* and *et(s)*. If such an interval is found, a reassignment of *mach(s)* can be again made without disrupting to any other SO. This simple shift-machine repair heuristic can also be utilized in cases where there are holes left in the schedule by misprocessed jobs.

A third heuristic is a combination of the simple shift-earlier and shift-later heuristics. The dispatcher can use the *mach–succ* link in the problematic OnDeck-SO to determine whether the OnDeck-SO can undergo a shift-later and the identified InTheHole-SO can undergo a shift-earlier. Simply stated, this results in switching the order of two jobs occurring sequentially on the same machine.

Another pair of more complex repair heuristics combine either of the shift-earlier or the shift-later heuristics with the shift-machine heuristic. This allows the local dispatcher to reassign a OnDeck-SO to a different machine at a different time than assigned by the global scheduler. Basically another machine in the same machine group must have an idle period that spans the the problematic OnDeck-SO's duration, and this idle period must fall completely within the OnDeck-SO's *pre–slack* and *post–slack* constraints.

The most complex repair heuristic includes all three of the simple heuristics: "shift-later", "shift-earlier", and "shift-machine". The dispatcher can use this heuristic to manipulate two OnDeck-SOs in such a way that their machine assignments are swapped and their time assignments are moved earlier or later as appropriate.

With regard to the order of application of the above set of heuristics, our current approach is computationally cheapest first. This is based on the observations that any solution found is consistent with the global schedule, and the faster the system can find one the better. As we gain more understanding of the nature of local reactive problems, we expect more knowledgeable ordering strategies to emerge.

4. Conclusions

In this paper, we have presented a framework for manufacturing production scheduling that distributes decision-making responsibility between a global scheduler, concerned with optimizing overall factory performance in the face of a high degree of combinatorial complexity, and a local dispatcher, concerned with robust execution in the face of an uncertain execution environment. The framework is based on the use of existing machine idle time and job wait time in the global schedule as sources of flexibility in managing execution. Assuming the existence of a scheduler capable of effecting global schedule repairs, we have defined a local decision-maker capable of exploiting this flexibility. The approach is currently being implemented for testing in a specific wafer fabrication facility.

One aspect of the framework that has been left underspecified concerns reassignment of responsibility for specific scheduling decisions (i.e. OnDeck-SOs and InTheHole-SOs) to the global scheduler upon failure of the dispatcher to move a given OnDeck-SO *s* into *Executing*. One straightforward approach is to simply remove *s*, and its related InTheHole-SOs *mach–succ(s)*, and *job–succ(s)* from the dispatcher's purview. This implies that the dispatcher must wait for the global scheduler's response (reaction) before making any further decisions involving either *job(s)* or *mach(s)*, and provides the global scheduler with complete flexibility to revise decisions relating to *job(s)* and/or *mach(s)*. On the other hand, there are circumstances when such an approach unnecessarily constrains factory execution. For example, if the failure of OnDeck-SO *s* is due to the fact that *job(s)* was misprocessed during execution of *job–pred(s)* (i.e. its related Executing-SO), then removal of *s* and *job–succ(s)* from dispatcher jurisdiction is appropriate but removal of *mach–succ(s)* may stall execution for no reason (if, in fact, determination by the global scheduler that *mach–succ(s)* is not affected by the problem is time consuming enough to cause an actual delay in executing *mach–succ(s)*). It may be more advantageous to leave responsibility for *mach–succ(s)* with the dispatcher, in essence treating it subsequently as a OnDeck-SO that can be executed as soon as conditions allow. The appropriateness of various responsibility reassignment policies depends heavily on the computational demands that are placed on the global scheduler (i.e. the frequency at which non-locally resolvable problems arise) and the speed at which global schedule repairs can be made.

At a more general level, there are several assumptions made in the model described in this paper that warrant further consideration. One assumption that our present model makes is that unique machine assignments are made in advance of execution by the global scheduler. In some manufacturing environments there may be no strong reason for such advance commitment, and it might be argued that doing so places artificial constraints on the dispatcher. On the other hand, there may be no danger in doing so if the dispatcher's machine swapping heuristics can effectively exploit opportunities to reassign machines. If not, it is straightforward to extend our approach to accommodate deferring of specific machine assignments to execution time. Since the present approach is built around manipulation of temporally crisp schedule, representations that enable enforcement of resource "capacity" constraints are well known (e.g. [9]) and could be easily integrated.

We have also made strong assumptions regarding the amount of lookahead given to the dispatcher. In the dispatcher described here, we have bounded the view of the dispatcher to the maximum extent, giving it access to those scheduling decisions that are only one or two steps into the future from those currently executing in the factory. This was done purposely to minimize computational expense and simplify the local reactive scheduling problem. At the same time, looking further ahead does provides the opportunity to "accumulate" more slack and idle time, and it may be the case that this would lead to a more equitable balance between global and local decision-making. It is certainly the case that looking further ahead will lead to more complicated local scheduling heuristics and longer compute times.

References

[1] Burke, P., and P. Prosser.
A Distributed Asynchronous System for Predictive and Reactive Scheduling.
Technical Report AISL42, Dept. of Computer Science, Univeristy of Strathclyde, October, 1989.

[2] Collinot A., and C. LePape.
Controlling Constraint Propagation.
In *Proceedings 10th International Joint Conference on AI*, pages 1032-1034. Milano, Italy, August, 1987.

[3] Fox B.R. & Kempf K.G.
Complexity, Uncertainty and Opportunistic Scheduling.
In *Proceedings of the Second IEEE Conference on AI Applications*, pages 487-492. IEEE-85, 1985.

[4] Fox B.R. & Kempf K.G.
Reasoning about Opportunistic Schedules.
In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1876-1882. IEEE-87, 1987.

[5] Georgeff, M.P. and A.L. Lansky.
Reactive Resaoning and Planning.
In *Proceedings AAAI-87*, pages 677-682. Seattle, Washington, July, 1987.

[6] Graves, S.C.
A Review of Production Scheduling.
Operations Research 29(4):646-675, July-August, 1981.

[7] Kempf K.G.
Manufacturing Scheduling - Intelligently Combining Existing Methods.
In *Proceedings of the Stanford Spring Symposium*, pages 51-55. AAAI-89, 1989.

[8] Kerr, R.M and R.N Walker.
A Job Shop Scheduling System Based on Fuzzy Arithmetic.
In M.D. Oliff (editor), *Proceedings 3rd International Conference on Expert Systems and the Leading Edge in Production and Operations Management*, pages 433-450. Hilton Head Island, SC, May, 1989.

[9] LePape, C. and S.F. Smith.
Management of Temporal Constraints for Factory Scheduling.
In C. Rolland, M. Leonard, and F. Bodart (editors), *Proc. IFIP TC 8/WG 8.1 Working Conf. on Temporal Aspects in Information Systems (TAIS 87)*, pages 165-176. Elsevier Science Publishers, May, 1987.

[10] Ow, P.S. and S.F. Smith.
Viewing Scheduling as an Opportunistic Problem Solving Process.
In R.G. Jeroslow (editor), *Annals of Operations Research 12*, pages 85-108. Baltzer Scientific Publishing Co., 1988.

[11] Ow, P.S., S.F. Smith, and A. Thiriez.
Reactive Plan Revision.
In *Proceedings AAAI-88*. St. Paul, Minn., August, 1988.

[12] Panwalker, S.S. and W. Iskander.
A Survey of Scheduling Rules.
Operations Research 25:45-61, 1977.

[13] Smith, S.F.
A Constraint-Based Framework for Reactive Management of Factory Schedules.
In M.D. Oliff (editor), *Intelligent Manufacturing.* Benjamin Cummings Publishers, 1987.

[14] Smith, S.F., P.S. Ow, N. Muscettola, J.Y. Potvin, and D. Matthys.
An Integrated Framework for Generating and Revising Factory Schedules.
Journal of the Operational Research Society, to appear, 1990.

[15] Wilkins, D.
Practical Planning.
Morgan Kaufmann Publishers, 1988.

[16] Yu C., Scott G & Kempf K.G.
Artificial Intelligence and the Scheduling of Semiconductor Wafer Fabrication Facilities.
In *Proceedings of the Intel Technology Conference*, pages 135-138. Intel Corporation, 1988.