Polite Rescheduling: Responding to Schedule Disruptions in a Multi-Agent Manufacturing System

by

Thomas Kaeppel Tsukada

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1996

Doctoral Committee:
Professor Kang G. Shin, Chair
Professor James C. Bean
Associate Professor Edmund H. Durfee
Assistant Professor Michael P. Wellman

UMI Number: 9712110

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

ACKNOWLEDGEMENTS

I am especially grateful to my graduate advisor, Dr. Kang Shin. As a student and as a researcher, I have benefited from his insights, his encouragement, his support, and his guidance during my course of study and research here.

I would also like to thank:

- the members of my doctoral committee, Professor Durfee, Professor Wellman, and Professor Bean, for their suggestions and advice, both individually, and as a committee;
- my colleagues from the Real-Time Computing Laboratory, for their friendship, their useful feedback, and their occasional commiseration;
- my family, for their understanding and their patience.

TABLE OF CONTENTS

CHAPTERS

LIST OF FIGURES

Figure

CHAPTER 1

Introduction

Over the past two decades, as the focus of manufacturing has shifted away from mass production of a homogeneous product mix to lower volume production of a heterogeneous product mix, the *flexible manufacturing system* (FMS) has attracted great interest. While the assembly line takes advantage of economies of scale and a very predictable manufacturing environment and demand, the flexible manufacturing system is designed to respond rapidly in an unpredictable environment, in which smaller quantities of more customized products are produced for a quickly changing and uncertain demand. While promising many opportunities, providing flexibility also presents many challenges.

Two important fields of research in FMS are organizational decentralization and scheduling of manufacturing processes. Decentralized organization allows more flexible design and control of manufacturing systems, through use of hierarchy and modularity. Scheduling seeks an efficient use of resources in the execution of varied production tasks. Both of these fields present new and important problems: decentralization poses the problem of coordination, while scheduling is itself a problem of efficient allocation. For both these fields, the problem of handling disruptions is of great importance, as flexibility demands being able to cope with an unpredictable environment.

This dissertation considers the problem of distributed schedule revision, the response to schedule disruptions in a distributed manufacturing system. This problem brings together aspects of those posed by decentralization, scheduling, and disruption handling. We will propose and describe an approach, called "polite rescheduling," in which global consequences of local actions are considered, and negotiation is used, in order to coordinate schedule revision. This approach attempts to limit the cost of scheduling disruption by rescheduling to limit the propagation of the disruption throughout the rest of the schedule. We will apply polite rescheduling to the domains of tool management and job shop scheduling, and will show that, under certain circumstances, polite rescheduling using only local knowledge gained through negotiation performs close to good or optimal methods using global knowledge.

1.1 An Illustration

To illustrate the distributed schedule revision problem, let us consider a controller for one part of a decentralized manufacturing system (e.g., a controller for a manufacturing cell). This controller is responsible for overseeing the execution of tasks performed at this part of the manufacturing system, tasks that usually involve the processing of parts; machining tasks, for example, would involve cutting a part into a specified shape. The controller has under its control numerically-controlled machines and robots with which to execute these tasks. The controller's view of the manufacturing system may appear as in Figure 1.1.

Let us suppose that this controller knows which tasks it has been assigned for the foreseeable future, and has been given a detailed schedule describing when it is to perform these tasks, what resources it is allowed to use, when it may use those resources, and where parts that have been processed should be sent next. An example of such a schedule is shown in Figure 1.2. This schedule is called a *local schedule*, as it describes only what is to take place under that controller; a schedule describing the execution of tasks at every controller would be a *global schedule.*

Let us imagine that a machine under the controller unexpectedly breaks down. The controller's local schedule may well be thus disrupted: tasks assigned to the broken-down machine may not be executable as scheduled. In the example of Figure 1.2, the starred tasks are unexecutable when Machine A breaks down. An intelligent controller should determine some response to this schedule disruption. One simple response may be merely to inform a higher level authority that a disruption has occurred and that certain tasks will not be executed. A more sophisticated response may be to revise the local schedule so that all tasks can be executed; perhaps tasks assigned to the broken-down machine can be performed by other machines supervised by the controller.

This kind of response requires solving a scheduling problem of allocating still available

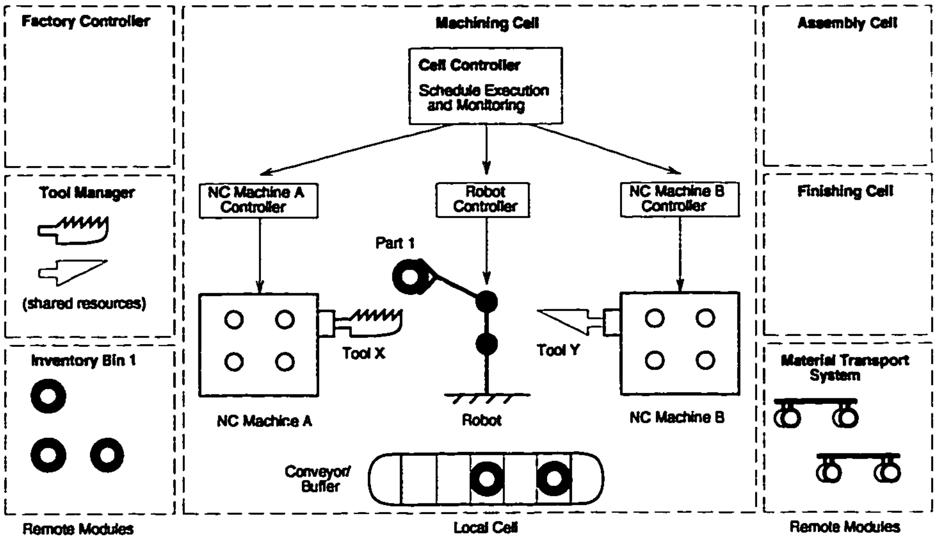


Figure 1.1: Machining Cell Controller's View of the System.

7:00am	*	set up Machine A for processing Part 1 using Large Blade (Shared Tool X)
7:05am	*	begin processing Part 1 on Machine A
7:05am		set up Machine B for processing Part 2 using Small Blade (Shared Tool Y)
7:10am		begin processing Part 2 on Machine B
7:15am	*	complete processing of Part 1 on Machine A
		send Part 1 to Assembly Cell
		return Large Blade
7:20am	*	set up Machine A for processing Part 3
7:25am	*	begin processing of Part 3 on Machine A
7:30am		complete processing of Part 2 on Machine B
		send Part 2 to Finishing Cell
		(keep Small Blade)
7:35am	*	complete processing of Part 3 on Machine A
		send Part 3 to Inventory Bin 1

Figure 1.2: Example: local schedule for cell controller.

6:50am	set up Machine B for processing Part 2 using Small Blade (Shared Tool Y)
6:55am	begin processing Part 2 on Machine B
7:15am	complete processing of Part 2 on Machine B send Part 2 to Finishing Cell return Small Blade
7:20am	set up Machine B for processing Part 1 using Large Blade (Shared Tool X)
7:25am	begin processing Part 1 on Machine B
7:35am	complete processing of Part 1 on Machine B send Part 1 to Assembly Cell return Large Blade
	(Part 3 remains unprocessed)

Figure 1.3: Example: revised schedule after loss of Machine A.

Scheduling Change	External Implication
Part 2 processed earlier	Part 2 may have to be delivered to Machining Cell earlier
Shared Tool Y used earlier	Shared Tool Y must be made available earlier and may be unavailable for other scheduled use
Part 2 delivered earlier	Finishing Cell may be able to process Part 2 earlier
Shared Tool X used later	Shared Tool X must be kept later and may be unavailable for other scheduled use
Part 1 delivered later	Assembly Cell may have to wait longer to begin processing of Part1
Part 3 left unprocessed	Another cell may be required to process Part 3

Figure 1.4: Scheduling changes and implications external to the local cell.

resources to local tasks. However, because the controller and its area of responsibility are integrated with the rest of the manufacturing system, decisions it makes in response to disruptions locally may affect other parts of the system not under its control. In order to revise the schedule of Figure 1.2, for example, the controller may choose the alternate schedule of Figure 1.3. Besides the changes required of the controller itself, this new local schedule may have implications for the rest of the system, external to the controller's area of responsibility, as shown in Figure 1.4.

Thus, a local controller in a decentralized manufacturing system may have to make decisions in response to local needs while considering the external implications of those decisions. Local schedule revision, while necessarily taking into account local scheduling goals and local resource availability and capacity, may also have to account for precedence constraints, shared resource capacity constraints, task prioritization, and others ways through which the local controller's area interacts with the rest of the system. In this dissertation, we will consider several of these issues in the domain of tool sharing and job-shop scheduling.

1.2 Motivation

The problem illustrated in the previous section would be far more straightforward if decisions in the manufacturing system were made by one top-level controller with perfect information about the whole system. It would also be an unimportant problem if there were no worries about schedule disruption. This section suggests why problem solving in a decentralized manufacturing environment is relevant, describes the problem of coordination in a decentralized system and presents background of the field of distributed artificial intelligence that deals with such problems, and describes the scheduling problem and why schedule revision is an important problem.

1.2.1 Organizational Decentralization

Decentralization is an increasingly important concept and reality in FMS, with a great deal of research into new control models for manufacturing system organization, emphasizing organizational flexibility, and modularity and simplicity of design [36, 15, 72]. This trend is motivated in part by advances in computer-integrated manufacturing (CIM) and the use of distributed computing in manufacturing systems. Decentralization of computerized control takes advantage of the often decentralized nature of the manufacturing process, and commonly allows better fault-tolerance, easier modifiability, and exploitation of computational parallelism.

The hierarchical control model for automated manufacturing systems is a very familiar tree-shaped command/feedback organization [36]. Flow of control is vertical; modules execute procedures specified by higher level modules, give commands to lower level modules, and send status feedback to higher level modules. While hierarchical organization is required in almost all manufacturing environments, it does have several drawbacks. Structurally, if only vertical communication is possible, each higher level module may become a communication and processing bottleneck as well as a single point of failure. In addition, Duffie et al. [15] argues that the organization of hierarchical systems becomes fixed in the early stages of design; such systems are difficult to maintain and modify, and fault tolerance is hard to provide. Thus, such systems are less flexible and modular in practice than they are in theory.

Another drawback of hierarchical control, when intelligent control is important, is the distributed nature of the manufacturing environment. Information that is needed by a higher level module may be distributed throughout its lower level 'children'. While information is gathered through status feedback from lower level modules, collecting complete information may require both a great deal of communication and intense processing of the data thus gathered. Additionally, if the environment is very dynamic, relevant information may be constantly changing, thus making it more difficult for a high level controller to have up-to-date information.

These drawbacks are the result of the fact that only vertical communication (between higher and lower level modules) is possible in the basic hierarchical model. Horizontal communication (or peer communication), between modules on the same level of the hierarchy, can remedy some of these drawbacks. In [15], Duffie et al. propose a *heterarchical* control model, based upon horizontal rather than vertical communication. The relationship of this model to other control models is illustrated in Figure 1.5, adapted from [15]. The modules in this control model cooperate through communication to pursue system goals.

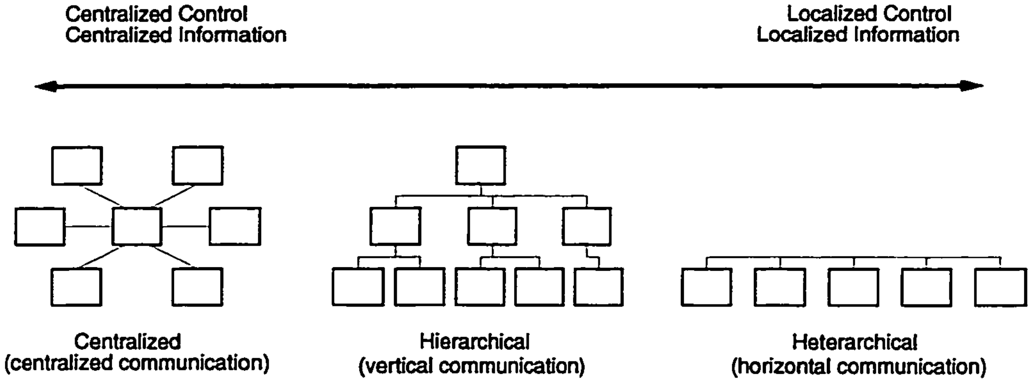


Figure 1.5: Heterarchical Control Model.

While the heterarchical model will not supplant the hierarchical model, because manufacturing systems are naturally hierarchical, it does provide the framework for horizontal communication within a hierarchical structure. If modules on the same hierarchical level, and with the same higher level 'parent', can cooperate among themselves to achieve some common goal, less work is required from the higher level module, less reliance is placed upon the higher level module, and less vertical communication is needed, as illustrated in

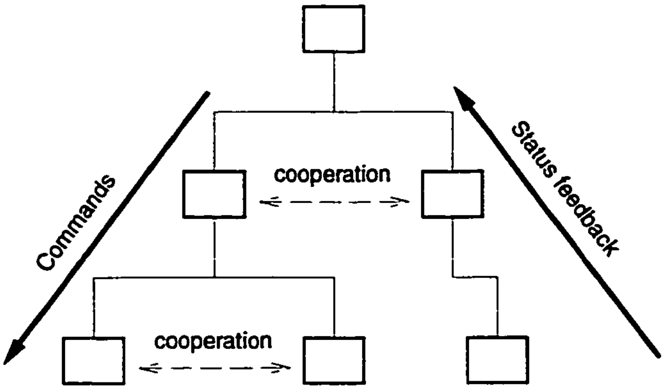


Figure 1.6: Horizontal Communication in a Hierarchy.

Figure 1.6.

1.2.2 Coordination, Negotiation, and Distributed Problem Solving

While horizontal communication can be beneficial, it requires sophisticated coordination. Peer relationships are usually harder to define and design than relationships between supervisors and underlings. In a system of interacting intelligent entities, or *agents*, two main issues are how an agent recognizes in what ways it can interact with other agents, and how several agents can cooperate in order to solve a problem.

These issues have been the focus of research in distributed artificial intelligence (DAI), an important and growing field of AI [6, 27]. Distributed problem solving (DPS) is a subfield of DAI that deals with how several agents cooperate to solve a problem. Several agents may be required for solution because no one agent has sufficient knowledge to solve the problem individually, or because different agents are better able to solve certain portions of the problem than other agents. Among the important issues in distributed problem solving are how the problem is to be decomposed and distributed among the agents, and what form of organization among the agents is to be used to coordinate problem solving efforts.

The important pioneer work in problem decomposition and organization was done by Davis and Smith, with the classic *contract net* protocol [13], and by Steeb and others in the domain of air traffic control [65, 8]. The contract net protocol is a well-known distributed problem solving protocol that explores these issues, and introduces the important concept of negotiation protocols. In the contract net protocol, agents serve as managers and contractors for subproblems. The subproblems are decomposed and distributed among the agents by negotiation. A manager responsible for a subproblem that cannot be solved locally requests bids from a group of potential contractors. The potential contractors submit bids based upon their own evaluation of how well they can solve the subproblem, and responsibility for the subproblem is awarded by the manager to the contractor with the best bid. Some form of communication among agents in a distributed problem solving environment is obviously necessary; such negotiation protocols allow efficient and coordinated exchange of relevant knowledge. This and subsequent work on negotiation deal with such issues as with whom an agent should negotiate, over what issues agents should choose to negotiate, what information agents should exchange through negotiation, and how an agent interprets information about other agents obtained through negotiation.

Organization in the contract net protocol is reflected by the roles agents play. Each agent can be a manager and a contractor for different subproblems, depending upon the problem solving context. Steeb and others explore distributed organizational issues in the domain of air traffic control [65, 8]. Their work includes discussion of *task centralization*, by which agents with a common goal select one 'coordinating agent' among them to solve most of the problem, and *task sharing*, by which agents cooperate more closely to find a solution. In task centralization, the coordinating agent may be selected because it is the least constrained agent, because it is the most knowledgeable agent, or for some other appropriate reason. Durfee and others [16] have explored the problem of organization among agents with interacting goals in a dynamic environment, in the domain of distributed vehicle monitoring. Their work introduces the concept of partial global plans (PGP's), which are evolving high-level multi-agent plans for agents that have common or interacting subgoals. Coordination is achieved through a meta-level organization, in which agents that are most capable of extending PGP's are responsible for doing so. PGP's are extended through exchange of each agent's local plans via communication.

When an agent in a multi-agent system needs a new plan, potential interaction among agent plans must be considered. Important work in multi-agent planning includes Lansky's GEM model of multi-agent domains [41]. The division of the problem into regions allows more efficient search through *localized planning*, in which only constraints relevant to the region in question need be considered. Pope's elaboration of this model for distributed planning with DCONSA [58] also considers inter-region constraints, where agents plan for their own assigned regions, and then deal with the interactions among the region plans.

Reasoning about inter-agent constraints is also the topic of research by Conry and others on multistage negotiation for distributed planning [10, 9]. In the multistage negotiation paradigm, agents trying to construct a plan for a common goal exchange knowledge about subgoals and subplans, detecting and resolving subgoal conflicts. Agents determine local subplans based upon primary goals (determined locally) and secondary goals (communicated from other agents). Negotiation reveals which of these subplans conflict with those of other agents, coordinating the distributed search of the solution space. The information thus gathered is used by agents to derive an *exclusion set* for each local plan fragment, that reflects how that plan fragment may affect other agents.

There is a great deal of current DAI research that is directly relevant to our distributed schedule revision problem. This includes work on distributed job-shop scheduling [54, 67, 42, 39], on distributed meeting scheduling [20, 62], on distributed constraint satisfaction problems [51, 73, 44], and on constrained intelligent action [19, 33]. This work will be discussed in Chapter 2 with respect to our approach to the problem.

1.2.3 Scheduling and Schedule Revision

Whether in a distributed context or not, the problem of scheduling, allocating resources over time for the execution of tasks, is an important problem for FMS. Good scheduling allows a manufacturing facility to use time and resources efficiently. The introduction of numerically-controlled machines with wider ranges of abilities, and the decreasing importance of assembly line manufacturing models, have made scheduling a harder and more important problem in manufacturing.

Scheduling has long been an important subject for operations research [2, 30, 25, 55, 57]. Scheduling problems are generally intractable; all but the most simplistic problems are NP-hard. Thus, much OR research has focussed upon determining which problems are tractable, finding faster solution search methods for non-polynomial-time-solvable problems, and developing heuristics to find good solutions for intractable problems. Scheduling has more recently become a focus of interest in the artificial intelligence community [50, 76], which has addressed scheduling problems with constraint satisfaction and knowledge-based methods. The pioneer AI work in scheduling is that of Fox [24, 23] and Smith [64], which emphasizes reasoning about constraints and analysis of resource capacity.

Recently, there has been growing concern about the differences between theoretical scheduling models and real-world scheduling problems [7, 55, 57]. An important and until recently overlooked aspect of scheduling is the problem of schedule execution and revision. Oftentimes, the real situation on the shop floor is different from that assumed in the scheduling process. Machines may break down, new unexpected jobs may arrive, release dates may change, etc. Such unexpected events may render a preschedule infeasible. Thus, if there is no provision for dealing with such events, even an optimal scheduling method may be completely irrelevant. OR approaches to the possibility of unexpected events and schedule revision include stochastic scheduling [57], and the match-up scheduling approach of Bean et al. [4]. Important AI approaches to schedule revision include work by Minton [46] and Zweben [75] in the domain of space applications. These approaches and how they relate to distributed schedule revision will be discussed in more detail in Chapter Two.

1.3 Polite Rescheduling

In these contexts of organizational decentralization, distributed problem solving and negotiation, and scheduling and schedule revision, this dissertation attempts to address the problem of recovering from a schedule disruption in a distributed manufacturing system. The disruption of a schedule may be costly, not only because of the task of finding a recovery plan itself. When a factory schedule is disrupted, commitments based upon the original schedule, dealing with material transport or personnel, may have to be reorganized. At worst, guarantees made to a customer about delivery times may be violated. Thus, when unexpected events can occur, one goal is to handle disruptions with as little change to existing schedules as possible. Our approach is therefore called "polite rescheduling", in which the affected agent attempts to solve locally the problem of finding a response to the disruption, in such a way that it will be least disruptive to other agents. This approach avoids the costs of making the local problem into a global problem, while it remains in a cooperative framework by attempting to isolate the effects of the disruption. More importantly, by avoiding complete rescheduling of the system, and by attempting to isolate disruptions, it attempts to retain as much of the distributed schedule as possible.

In order to find a response that is least disruptive to other agents, the affected agent must have some information about how its actions will affect those other agents. The agent should be able to reason about, and have some knowledge about, the general effects of rescheduling. Because an individual agent does not have global knowledge about the system, some form of negotiation should be very useful as a means of gathering information about other agents. The disrupted agent searches for the least disruptive response by negotiating with other agents that could possibly be disrupted by its actions, exchanging information about schedule features that are relevant to schedule interaction among agents, and using this information to reason about how local schedule revision affects other agents. In particular, if the disrupted cell cannot prevent propagation of the disruption, it should try to control the propagation so that other agents may easily handle its effects. Through investigating polite rescheduling, we hope to provide a realistic means to deal with the problems of real-world factory scheduling, and to provide a general method for dealing with disruption handling and plan revision in a distributed system of loosely-coupled agents, that may have applications in many domains other than scheduling.

In general, our research attempts to determine whether and under what circumstances local knowledge of local scheduling constraints can be used to obtain a good local schedule revision, when the goal of schedule revision considers its effect upon the global system. Thus, we present a way in which a local agent in a multi-agent system, without global knowledge of the schedule, can respond to a local schedule disruption, when limiting the propagation of schedule disruption is a goal for rescheduling. The main contributions of this work are:

- We propose a new approach to schedule revision in a distributed environment, a problem that hitherto has not been treated in much depth.

- We show experimentally on a set of tool sharing models that polite rescheduling using local knowledge of portions of tool schedules, performs close to optimal methods using global knowledge of tool schedules for a tool scheduling problem, and close to good methods using global knowledge for a tool borrowing problem, especially when task rescheduling is treated as a cost.

- We show experimentally on a set of job-shop scheduling models that different levels of local knowledge of schedule constraints, deriving from precedence constraints, allow significantly different rescheduling performance, and that use of local knowledge of schedule constraints in a flowshop-like job shop allows rescheduling performance close to that of optimal methods using global knowledge of the schedule.

- We describe PRIAM, an architecture for rescheduling in a multi-agent job shop environment, investigate issues in determining scheduling priorities and negotiation strategies, and show that polite rescheduling isolates schedule disruptions better than other methods using only local knowledge.

1.4 Dissertation Outline

The dissertation is organized as follows:

Chapter Two describes in detail the distributed schedule revision problem and a formal model thereof, reviews related research, and presents our polite rescheduling approach, discussing both local rescheduling issues and negotiation issues.

Chapter Three presents a polite rescheduling approach to the domain of tool sharing, presents results for a simple tool scheduling problem, and a more complicated tool schedule revision problem, and compares this approach with good or optimal methods using global knowledge, in terms of task acceptance and task rescheduling.

Chapter Four examines rescheduling issues for a decentralized job shop particularly with regard to precedence constraints, focusing upon rescheduling to minimize the schedule makespan, shows the utility of different levels of knowledge of local schedule constraints, and shows that using of such local knowledge allows performance close to optimal methods using global knowledge.

Chapter Five investigates the polite rescheduling approach to job shop scheduling, focusing upon schedule revision and negotiation for limiting the disruption propagation through the global schedule, describes the job class scheduling problem and a solution approach, presents the PRIAM polite rescheduling architecture, and shows the advantage of polite rescheduling over other methods using local knowledge.

Chapter Six examines the problem of schedule revision to limit disruption propagation when only precedence constraints are known only probabilistically, presents some simple analytical results, and considers the problem of schedule proposal generation.

Chapter Seven summarizes the results of this work and reviews its contributions to field. While each of four preceding chapters present ideas for possible future work, Chapter Seven presents a more integrated consideration of the future direction of this research.

CHAPTER 2

Polite Rescheduling

Etiquette, whether old-fashioned or high-tech, is not merely a matter of style or superficial vanity. It serves an important social role, simplifying social interactions by allowing individuals to have usually accurate expectations about the behavior of other individuals, even when the goals and plans of others are unknown. Etiquette also allows individuals to pursue their own personal goals in a common venue without having to spend all their time worrying about being interfered with by bothersome neighbors.

In order to be polite, one needs to know common modes of polite behavior, and one needs to reason about how one's actions may affect others. While the determination of good rules of etiquette, or *conventions*, in social systems is an important and interesting problem that has attracted attention lately in the DAI field [63, 70], our focus is on the second aspect of politeness, that of being considerate, how an individual determines how to achieve personal goals in a social system without making it unduly difficult for others to pursue their own goals. The aspect of politeness is important for maximizing "social utility", i.e., some measure of how well the goals of the whole system can be met. Two particular aspects of this "social utility" are "fairness", that it is somehow better that every individual is allowed to pursue personal goals, and "stability", that it is somehow better to avoid frequent events that force individuals to change or re-evaluate their plans. "Social utility" is also more explicitly defined when all individuals are acting together to achieve some common set of goals.

For agents in a multi-agent system, etiquette and convention would probably suggest that a global disruption, an unexpected event adversely affecting several agents, be handled with some form of cooperative response. However, a local disruption, an unexpected event adversely affecting and perhaps recognized by only one agent, can often be handled individually by that agent. Handling a local disruption locally, without invoking some cooperative response involving distributed problem solving and negotiation, may be the more efficient alternative, as negotiation, while useful and sometimes indispensable, can be both as complicated and as time-consuming in a DAI context as it is in a human one. Local handling of local disruptions, if possible, also promotes stability of the "distributed plan", avoiding the requirement that other agents change or reevaluate their own plans; more of the "distributed plan" of the system may be retained this way.

The important issues for handling local disruptions politely include: determining possible responses to the disruption; considering how such responses possibly affect other agents; determining whether there are any good responses that leave other agents undisturbed; and deciding what to do in case there are no such responses that leave other agents undisturbed. In this last case, important issues include deciding which other agents to consult, how to negotiate efficiently about possible effects of local decisions, what kinds of effects are acceptable, and what to do if negotiation fails. The remainder of this chapter explores these issues in a generic distributed problem solving context, discusses what form these issues take in the context of our distributed schedule revision domain, and then compares our approach with related work in the distributed scheduling and schedule revision fields.

2.1 Plan Revision in a Distributed Environment

The problems that we are investigating are those in which one agent in a system of loosely-coupled agents needs to recover from a local disruption. By "loosely-coupled", we mean that the agents are not necessarily cooperating closely on any particular task, but they may affect one another, and in particular, the actions of one may hinder another from achieving its goals. The revision problem we address is distinguished by the fact that it involves agents that already have plans, where one agent must revise its local plan in the context of other agents' plans.

The nature and frequency of disruptions are obviously important when determining what kind of response is appropriate. If the disruptions that occur only affect one agent, then the response may involve only that agent, with no communication or cooperation with others. If, however, disruptions usually directly affect many or all the agents in the system, then a centralized approach, or an intensively cooperative approach, may be unavoidable. Likewise, if disruptions are occurring so frequently that agents are constantly revising their plans, and are revising plans simultaneously, then some form of reactive planning or global response may be better suited than explicit plan generation or negotiation about plan revision. The types of problems we consider are somewhere in the middle of these extremes. We will consider disruptions that are relevant mainly to one agent, but the response to which may affect other agents. We will also assume that the disruptions that occur are infrequent enough that only one agent will be directly affected by a disruption at any given time, though several other agents may have to revise plans simultaneously due to this agent's response.

Prior to a disruption, we can consider the set of plans of agents in the system as a solution to a distributed constraint problem, that includes the constraint problems of all the agents. Once a disruption occurs, this set of plans is no longer a feasible solution; a new solution must be found. Finding a new solution from a position with an infeasible old solution is very similar to the problem of backtracking, which is a particularly hard problem in DPS. This problem can be formalized as the distributed constraint satisfaction re-assignment problem.

2.1.1 The DCSP Re-assignment Problem

Constraint Satisfaction Problems

Scheduling problems, as well as many other resource allocation problems, are often cast as constraint satisfaction problems (CSP's). CSP's are a well-studied area in AI literature [40]. A CSP is defined by a set of *variables* ($X = \{x_1, \ldots, x_m\}$), a *domain* for each variable x_i ($V_i = \{v_{i1}, \ldots, v_{in_i}\}$), and a set of *constraints* ($C = \{c_1, \ldots, c_q\}$). An *instantiation* of a variable is the assignment of one value from its domain to the variable, and the instantiation of a set of variables is an instantiation for each variable in the set. Each constraint c_k involves a set of variables, the *variable set* of c_k (denoted by VS(c_k)), and specifies the allowable instantiations for this set. That is, c_k is a subset of the Cartesian product $\prod_{x_i \in \mathrm{VS}(c_k)} V_i$, that contains all the allowable instantiations for this set of variables for this constraint. An instantiation for the entire variable set that satisfies every constraint is a solution to the problem. For some constraint c_k, we will write $c_k(x_1, x_2)$ to indicate that $\mathrm{VS}(c_k) = \{x_1, x_2\}$.

In solution methods for resource allocation CSP's, constraints are often *relaxed* when a solution cannot be found. A constraint c_k is relaxed by replacing it with a different constraint c'_k, so some previously disallowed instantiations of $\mathrm{VS}(c_k)$ are allowable, that is, $c'_k \cap c_k \neq c'_k$. There will often be a cost associated with the relaxation of a constraint. For example, in Zweben's work on space shuttle scheduling [75], the cost of a solution is a function of the number of original constraints that are violated; these violated constraints must be relaxed (and possibly removed) in order for the solution to be allowable. [1]

Distributed Constraint Satisfaction Problems

We here consider a CSP in the context of a multi-agent system (a distributed CSP, or DCSP), in which there is a set of *agents* ($A = \{a_1, ..., a_s\}$), each agent a_i being responsible for a subset of variables Y_i, such that $Y_1, \ldots, Y_s$ form a partition of the variable set X. For an agent a_g, its view of a constraint c_k would distinguish the *local variable set* of the constraint ($\mathrm{LVS}_g(c_k) = \{x_i \mid x_i \in \mathrm{VS}(c_k), x_i \in Y_g\}$), that is, those variables in $\mathrm{VS}(c_k)$ for which the agent is responsible, and the *remote variable set* ($\mathrm{RVS}_g(c_k) = \{x_i \mid x_i \in \mathrm{VS}(c_k), x_i \notin Y_g\}$), those variables in $\mathrm{VS}(c_k)$ for which other agents are responsible. Thus, the constraint can be expressed as a subset of the Cartesian product of two Cartesian subproducts:

$$c_k \subseteq \left(\prod_{x_i \in \mathrm{LVS}_g(c_k)} V_i \right) \times \left(\prod_{x_j \in \mathrm{RCS}_g(c_k)} V_j \right).$$

For an agent a_g and an instantiation I, let E^I_g be the *environment* for agent a_g, the variable instantiations in I for variables not in Y_g. Likewise, let L^I_g be the *local instantiation* for agent a_g, the variable instantiations in I for variables in Y_g. For an agent a_g, given an instantiation I for all the variables not in Y_g, we can consider $c^g_k(I)$ a *local constraint* for the variable, for which an instantiation of $\mathrm{LVS}_g(c_k)$ is allowable if, given E^I_g, the constraint c_k is not violated. In fact, we can pose a CSP subproblem $\mathcal{P}_g(I)$ for agent a_g, given I, consisting of its own variables, their domains, and a local constraint $c^g_k(I)$ for each constraint $c_k \in C$, $C^g(I)$ being the set of these local constraints. ($\mathcal{P}_g(\emptyset)$ is the subproblem for agent g without any instantiation of other agents' variables, the least constrained version of agent g's local subproblem). A solution J for the whole CSP contains within it solutions for each agent's

[1] It can be argued that relaxing constraints simply produces a different problem, or that if costs are to be considered, an optimization formulation is necessary. In practice, however, a CSP formulation with relaxation may often be useful (though imperfect) expressions of an underlying (but possibly intractable or even indescribable) ideal optimization problem.

CSP subproblem.

There have recently been several research efforts into solution methods for DCSP's [9, 51, 44, 42]. Most of this research shows that both backtracking and search termination are very difficult in a distributed environment. Yokoo et al. [73] and Nishibe et al. [51] simplify backtracking by imposing a strict order on agents. The work of Kambhampati et al. on "distributed hybrid planning" briefly discusses the the problem of avoiding "ripple effect" propagation of plan changes when backtracking in distributed design [37]. Luo et al. [44] suggests that distributed algorithms are inefficient and perhaps ineffective, unless they are the only available algorithms for the given problem. We believe, however, that the problem we consider, that of finding a new solution for a DCSP that has been solved and then modified, is restricted enough for a distributed approach to be useful, and the manufacturing resource allocation domain is one in which a distributed approach is most convenient.

Re-assignment and Negotiation

A CSP P can be modified into a similar problem $\mathcal{P}'$ by the addition of a new constraint c_d.[2] A solution J for $\mathcal{P}$ may not be a solution for $\mathcal{P}'$ because of the new constraint; if this is so, we can called c_d a *disruption* for J and $\mathcal{P}$. Values must then be re-assigned to variables in order to find a solution for $\mathcal{P}'$.

We consider this problem in the previous multi-agent context, and assume that a solution J has been found. For an agent a_g and the new constraint c_d, if $\mathrm{VS}(c_d) = \mathrm{LVS}_g(c_d)$, then the new constraint is *local* to a_g. That is, the agent a_g can recognize if a violation has resulted simply by examining the instantiation of its own variables against the new constraint. [3] As will be seen, common disruptions for resource allocation problems will result in new constraints local to one agent. The agent for which a disruption is local is called the disrupted agent.

If a new constraint is disruptive, there are several possible methods for solution. The simplest approach is simply to relax the new constraint until it is no longer disruptive. In a problem of allocating resources to tasks, this approach is equivalent to refusing a

[2] A new constraint may also involve the addition of new variables, but we can always suppose that any new variable actually existed in the original problem as an unconstrained variable.

[3] We do not consider the case in which both $\mathrm{LVS}_g(c_d)$ and $\mathrm{RVS}_g(c_d)$ are non-empty for some agent a_g (when the constraint is not local to any agent). Conflict detection in a distributed environment is an important problem, but not one we consider here.

newly requested resource to a new task. While this approach may often be used in real domains, it is completely inflexible and consequently of little interest. Another approach is to solve the new global CSP, using whatever method was used for the original solution. While this approach may be best for some situations, it may be inefficient. In a distributed environment, information collection for a centralized solution method may be costly and unwieldy, while a distributed method may be inefficient. In the approach we explore here, the disrupted agent will attempt to solve its local CSP subproblem with the new constraint, negotiating with other agents when necessary.

The disrupted agent will first try to solve its sub-problem $\mathcal{P}_g(J)$ by re-instantiating its own variables in order to satisfy the constraints in $C^g(\mathrm{E}_g^J)$, including the disrupting constraint. If such a local solution can be found, then the global CSP has been solved, because no constraint in C can have been violated by this re-instantiation. Also, the local solution has not required re-instantiation of any variable belonging to any other agent.

If such a local solution cannot be found, then the disrupted agent must relax some constraint or constraints of its subproblem. One way of doing so is to relax one of the original constraints in C, or the new constraint c_d. However, the constraints of a_g may also be relaxed by changing the instantiation of some other agent's variable. For some constraint c_k, for which $\mathrm{VS}(c_k)$ includes variables of both a_g and some other a_h, agent a_g's local constraint $c_k^g(\mathrm{E}_g^J)$ may be relaxed if one of the variables in $\mathrm{LVS}_h(c_k)$ is re-instantiated with a different value. Thus, an agent may be able to solve its new subproblem (and thus handle its disruption), through negotiation with other agents over their variable instantiations.

An Illustration

Here is a simple illustration, for a CSP $\mathcal{P}$ in which two jobs, j_1 and j_2, managed by agents a_1 and a_2 respectively, both require a resource T. Agent a_1 is responsible for variable $x_1 \in \{1, 2, u\}$; j_1 can be scheduled during time period 1 or time period 2, or remain unscheduled. Likewise, a_2 is responsible for $x_2 \in \{1, 2, u\}$. Constraints $c_1(x_1) = \{(1), (2)\}$ and $c_2(x_2) = \{(1), (2)\}$ express the requirement that j_1 and j_2 much each be scheduled during one of the two time periods. Constraint $c_3(x_1, x_2) = \{(u, u), (u, 1), (u, 2), (1, u), (1, 2), (2, u), (2, 1)\}$ expresses the requirement that T cannot be used by both j_1 and j_2 at the same time.

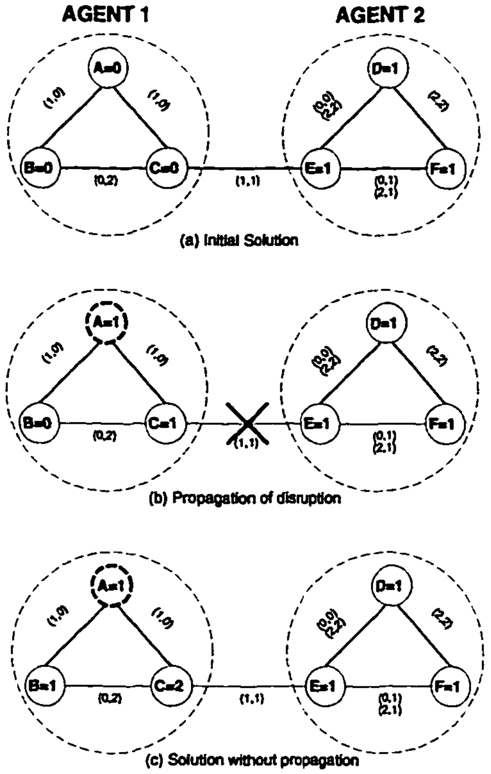
Solution J for this problem is $(x_1, x_2) = (1, 2)$. Given this solution as a schedule, suppose that, before the schedule is executed, a new constraint $c_4(x_2) = \{(1)\}$ is added. This may reflect that j_2 suddenly has a deadline requiring that it finish before the start of time period 2. The subproblem $\mathcal{P}_2$ for a_2, given E_2^J, now includes constraints $c_2^2(x_2) = \{(1),(2)\}$, $c_4^2(x_2) = \{(1)\}$, and $c_3^2(x_2) = \{(2)\}$, and is clearly insolvable. To solve by relaxation, c_2^2 may be relaxed to include (u), allowing j_2 to be abandoned, and c_4^2 may be relaxed to include (2), allowing j_2 to miss its new deadline. Both these relaxations involve relaxing an original constraint of $\mathcal{P}$. However, if a_1 changes the instantiation of x_1 to 2, then c_3^2 is relaxed to allow (1). Then the problem can be solved without relaxing any of the original constraints.

2.1.2 Polite Replanning: Model and Approach

Local problems for an agent can thus be relaxed when other agents change their solutions to their own problems. However, requiring another agent to change its solution is not necessarily solving the disruption-handling problem; it may simply be asking someone else to solve it. Thus, depending upon local responses to disruption handling, a disruption may propagate throughout the system. An example of disruption propagation in a DCSP is shown in Figure 2.1. Here there are two agents, each with three variables. Each variable has the domain $\{0, 1, 2\}$. In this case, the arcs in the graph represent value assignments that are constraint violations (i.e., the arc between A and C indicates that A cannot equal 1 when C equals 0). The initial solution is shown in (a). In (b), variable A has its domain reduced to 1. As this conflicts with C's assignment of 0, there must be a re-assignment. In (b), the variables of agent 1 have been re-assigned so that they do not conflict with one another, but this has caused a conflict between variable C of agent 1 and variable E of agent 2. Thus, the disruption has been propagated to agent 2; in order for agent 1's local solution in (b) to be feasible, agent 2 must change its assignments, to relax the local problem for agent 1. A solution that does not propagate a disruption is shown in (c).

Formal Model

In order to discuss disruption propagation and disruption handling in a multi-agent environment, we propose the following model. Given a global CSP $\mathcal{P}$ as described previously, and a solution I, let $\mathcal{P}_g(I)$ be the local CSP at agent a_g given environment E_g^I. Let $S_g = \{s_{g1}, \ldots, s_{gm}\}$ be the set of all possible complete instantiations of the variables in Y_g; i.e., $S_g = \prod_{x_i \in Y_g} V_i$. A disruption is the addition of a constraint c_d such that the original solution I no longer solves the altered CSP $\mathcal{P}'$. For a disruption local to agent a_g, L_g^I no



(a) Initial Solution

(b) Propagation of disruption

(c) Solution without propagation

Figure 2.1: DCSP "replanning" example.

longer solves the local problem $\mathcal{P}'_g(I)$.

While we are interested in how one agent recovers from a disruption, it is important to consider what happens if several agents are trying to recover from different disruptions at the same time. In general, interactions between agents' actions would be very hard to analyze. We will instead consider *rules* restricting the set of possible recovery methods, so that the recovery methods interact in only very limited ways when every agent follows the rules. Let $\mathrm{RA}_g^R(\mathcal{P}',I) \subseteq S_g$ be the set of local instantiations that solve $\mathcal{P}'_g(\emptyset)$ and are allowable under some rule R given instantiation I; this is called the set of *recovery actions* for the disruption under rule R. Let $\mathrm{NA}_g^R(\mathcal{P}',I) = \{s \in \mathrm{RA}_g^R(\mathcal{P}',I) : \mathcal{P}'_j(E_g^I \cup s)$ is solved by L_j^I for all $a_j \in A\}$ be the set of *non-disrupting* recovery actions, that do not render any other agent's local instantiation under I infeasible, given the new problem $\mathcal{P}'$. When one agent is trying to recover from a disruption while no other agent is changing its local instantiations, a non-disrupting recovery action will not cause a disruption of another agent.

Let $\mathrm{GA}_g^R(\mathcal{P}',I) = \{s \in \mathrm{RA}_g^R(\mathcal{P}',I) : \mathrm{RA}_j^R(\mathcal{P}'_j, E_g^J \cup s) = \mathrm{RA}_j^R(\mathcal{P}'_j, J)$ for all $a_j \in A$ and all instantiations J $\}$ be the set of *guaranteed-safe* recovery actions, that will not change the set of recovery actions for any other agent under rule R. Given rule R for

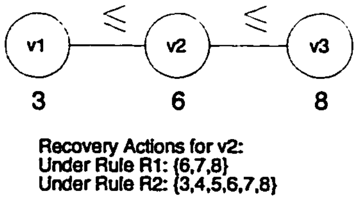


Figure 2.2: Recovery Actions: A Simple Illustration.

selecting recovery actions, if an agent chooses a guaranteed-safe recovery action, it will not reduce any other agent's set of allowable recovery actions under the same rule R. Thus, if several agents each select a guaranteed-safe recovery action using the same rule for determining the recovery action set, none will disrupt any other no matter in what order the recovery actions are taken. For agent a_g and recovery action $s \in \mathrm{RA}_g^R(\mathcal{P}', I)$, let $M_g(s, \mathcal{P}') = \{a_h \in A : h \neq g,\ \exists v_i, v_j \in \mathrm{VS}(c), v_i \in Y_g, v_j \in Y_h$ for some constraint c in $\mathcal{P}'\}$ be the set of remote agents which could possibly be disrupted by agent a_g taking recovery action s. Clearly, if $s \in \mathrm{GA}_g(\mathcal{P}'_g, I)$, then $M_i(s) = \emptyset$.

An Illustration

Consider the simple example of Figure 2.2, in which there are three agents, each responsible for a different variable, and the constraints are $v_1 \leq v_2 \leq v_3$. One possible rule, R_1, requires that any new value for any variable must be greater than or equal to its current value, and less than or equal to the value of any other variable with which it has a $\leq$ constraint relation. Under R_1, for example, v_2 could take a value in $\{6, 7, 8\} = \mathrm{RA}_2^{R_1}(\mathcal{P}, I)$. This set is also a set of guaranteed-safe recovery actions under R_1, for if each agent is choosing a new value according to this rule, no constraint violation will result.

Another possible rule, R_2, requires that any new value for any variable must be greater than or equal to the value of any other variable with which it has a $\geq$ constraint relation, and less than or equal to the value of any other variable with which it has a $\leq$ constraint relation. Under R_2, v_2 could take a value in $\{3, \ldots, 8\}$. These are non-disrupting actions. They are not guaranteed-safe actions under R_2, however. If v_2 is given value 5, then $\mathrm{RA}_1^{R_2}$ will no longer include 6. Under this rule, if v_1 and v_2 are changing values simultaneously, v_1 can get 6 while v_2 gets 5, and a constraint violation will result.

Disruption Propagation

Disruptions can be classified by how much they result in propagation of disruptions. Consider an initial problem $\mathcal{P}$ and solution I. Given and rule R, we call the local disruption for agent a_g a disruption of *type 0* if $\mathrm{NA}_g^R(\mathcal{P}', I) \neq \emptyset$. By taking a recovery action $s \in \mathrm{NA}_g^R(\mathcal{P}')$, agent g can recover from a type 0 disruption without disrupting other agents. Likewise, we call it a type 1 disruption if there is a recovery action $s \in \mathrm{RA}_g^R(\mathcal{P}')$ such that $\mathrm{GA}_h^R(\mathcal{P}', E_g^I \cup s) \neq \emptyset$ for all agents $a_h \in A$. That is, there is a recovery action for the disrupted agent that may disrupt other agents, but for which any disruption of another agent can be resolved by that agent with a guaranteed-local recovery action.

A disruption of type 1 can be handled without the disruption propagating more the one level. Likewise, a disruption is of type l for a rule R, where $l > 0$, if there is a recovery action $s \in \mathrm{RA}_g^R(\mathcal{P}')$ such that, if $M_g(s, \mathcal{P}') \neq \emptyset$, then there is an agent k such that $\mathrm{GA}_h^R(\mathcal{P}', E_g^I \cup s) \neq \emptyset$ for all agents $a_h \in M_g(s, \mathcal{P}'), h \neq k$, and such that the disruption for agent k is of type $l - 1$. Thus the disruption can be handled without the disruption propagating more than l levels, and without agents' recovery actions interfering with one another. If a disruption is not of any type, then it cannot be handled without agents' recovery actions interfering with one another.

Outline of Approach

As mentioned in the previous chapter, one of the goals in distributed schedule revision is to minimize the global "disruptedness" resulting from a local disruption. Thus, in response to such a disruption, we try to limit the propagation of the disruption to other parts of the system, isolating the disruption to the initially disrupted agent and perhaps to a few of its neighbors (i.e. those with which it shares constraints). After describing this approach, we will address the important issues raised by it.

Limiting propagation involves determining rules by which agents may choose recovery actions. When an agent experiences a disruption, it tries to determine whether this disruption is of type 0. If it determines this, then it takes a non-disrupting recovery action. If not, then it tries to determine through negotiation with other agents whether the disruption is of type 1. If it determines this, it takes the action that results in a propagation of the disruption of at most one level. Here we do not go beyond disruptions of type 1; in our cellular manufacturing domain, there are not a large number of agents, so that at greater levels of propagation, the whole system is affected. This approach can be extended to disruptions of type l in systems of greater numbers of agents. In such a case, a small group of agents may cooperate more fully to prevent the disruption from propagating beyond that group.

For a problem $\mathcal{P}$ with an initial solution I, consider a disruption local to agent a_j. We do not assume that agent a_j already knows its full set of possible recovery actions, and their effects on other agents. Instead, agent a_g uses some heuristic G to try to find a non-disrupting action $s \in \mathrm{NA}_g^{R_l}(\mathcal{P}', I)$. If it can find such an action, it will take that action. If not, some communication is necessary for the selection of a good recovery action, either because a non-disrupting action does not exist and other agents will necessarily be disrupted, or because additional information is needed from other agents to find a non-disrupting action. Thus, agent a_g uses some heuristic H to select a recovery action $s' \in \mathrm{RA}_g^{R_l}(\mathcal{P}', I)$ that seems likely, given local information, not to be very disruptive to other agents. Agent a_g then sends a proposal message to all members of $M_g(s', \mathcal{P}')$, proposing action s'. The local recovery rule R_l need not be very restrictive, as we assume that initial disruptions will be rare, and will not occur more than one at a time.

When an agent a_h receives a proposal message proposing action s', it first determines whether action s' will cause a disruption at a_h. If not, then it returns an ok-0 message. Otherwise, it tries to determine whether the disruption caused by s' will be one of type 0, that can be handled locally. To do this, it must find a guaranteed-safe action $s_h \in \mathrm{GA}_h^{R_r}(\mathcal{P}', I)$ to handle the disruptive effect of s', as other agents may also be trying to determine similar recovery actions. If all these agents are using the remote recovery rule R_r, then if they all find guaranteed-safe actions, the disruption will be isolated. If so, it returns an ok-1 message. Otherwise, it will return a not-ok message, perhaps along with some domain-specific information J that can be used by the disrupting agent's heuristic H to propose a better solution.

When disrupted agent a_g receives replies to its proposal, if all replies are ok-0 messages, then it takes the proposed action. If all replies are either ok-0 or ok-1 messages, then the agent can take the proposed action. In either case, the disruption will be isolated. If there is a not-ok reply, then the controller knows that action s' will not isolate the disruption to the agents in $M_g(s', \mathcal{P}')$, so, with whatever information has been gathered, it uses heuristic H again to propose a new recovery action, unless it determines that further negotiation will not be useful.

Issues

Our approach finds a solution local to the part of the system experiencing the initial disruption. Isolating the disruption to the local agent and its neighbors simplifies the problem, and limits the search space, as in Lansky's localized planning. In our case, however, limiting the search space in this manner may mean that the best solution is not found. For example, depending upon the measures being used, the best solution might entail propagating a little disruption throughout the entire system. Because we are considering domains in which the information needed for a search of the entire space is not centralized at any agent, our approach is justified. Nevertheless, in evaluating this approach, we will compare its performance against methods that use global information.

Another important tradeoff is that between measures of disruptedness and other more standard measures of plan quality, including those used in the construction of the initial plan. The goal of limiting disruption and other goals may often be compatible, as limiting disruption allows the retention of some of the original plan and presumably some of its quality with respect to the initial measure. This is the motivation behind Bean's matchup scheduling. Nevertheless, as we will show in our applications of this approach to various scheduling problems, limiting disruption often incurs a cost according to the initial measure.

The approach described above is of course only a simple outline of an algorithm for handling this problem. The real issues are what kinds of heuristics G and H are, what kinds of information J is to be exchanged, what rules ensure that guaranteed-safe recovery actions can be found, and what to do when no proposal is acceptable to the other agents. At least some of these answers are domain dependent, and cannot be more fully described in this very general model.

2.2 Polite Replanning in the Scheduling Domain

Polite rescheduling is the application of polite replanning to distributed schedule revision. In order to discuss issues particular to polite replanning in the scheduling domain, we first give a very brief overview of the scheduling domain and its aspects relevant to the problem in question.

2.2.1 The Scheduling Domain

Scheduling generally is the allocation of limited resources to tasks over time. In a manufacturing context, the scheduling problem usually concerns the determination when a production task (or *job*) will be processed, the machine (or other processing unit) which will process it, and the other resources which will be used during processing. There are many different types of scheduling problems; we will limit ourselves to single and parallel machine problems using due dates, and job-shop scheduling problems.

The single machine scheduling problem is usually a sequencing problem; given n jobs, where each job j has a processing time p_j, a sequence is to be determined according to some objective function. The starting time of a job j in a schedule is denoted as s_j, and the completion time is denoted as C_j. A job j may have a release time r_j, before which it may not begin processing. A job may also have a due date d_j, which represents a desired completion time; a due date is not a hard deadline for job completion, but there is often a penalty associated with missing a due date. Common objective functions (or measures) for the single machine scheduling problem are *total flowtime* ($\sum C_j$), the sum of job completion times; the *makespan* ($C_{\max}$), the completion time of the latest job to complete processing; *maximum lateness* ($L_{\max}$), the worst due date violation among the jobs; *total number of tardy jobs* ($\sum U_j$); and *total tardiness* ($\sum T_j$), where the tardiness of a job is 0 if the job is completed before its due date, and $C_j - d_j$ otherwise. Measures with different weights for different jobs are also often considered; e.g., $\sum w_j C_j$ is the *weighted flowtime* measure.

Some single machine problems have simple solutions. When jobs have identical release times, total flowtime is minimized by the shortest-processing-time-first (SPT) dispatching rule, and maximum lateness is minimized by the earliest-due-date-first (EDD) rule. Non-identical release times make scheduling problems much more difficult, as do tardiness-related measures and non-identical job weights. A parallel machine scheduling problem is identical to the single machine problem except that jobs are distributed among and sequenced upon several machines. Parallel machine problems are also hard, as they usually have bin-packing problems as special cases.

One scheduling problem of particular interest is the job shop scheduling problem. In this problem, a set of jobs is to be scheduled on a group of non-identical machines. Each job consists of a set of operations that must be processed in a predetermined order, represented by *precedence constraints*. Often these operations represent the processing of a work-piece that is transported among the machine, and the precedence constraints represent physical constraints; for example, the task of putting a bolt through a hole in a widget can only be performed after the hole has been drilled into the widget. Often, the operations of a job at one machine or cell are simply called jobs.

2.2.2 Polite Rescheduling

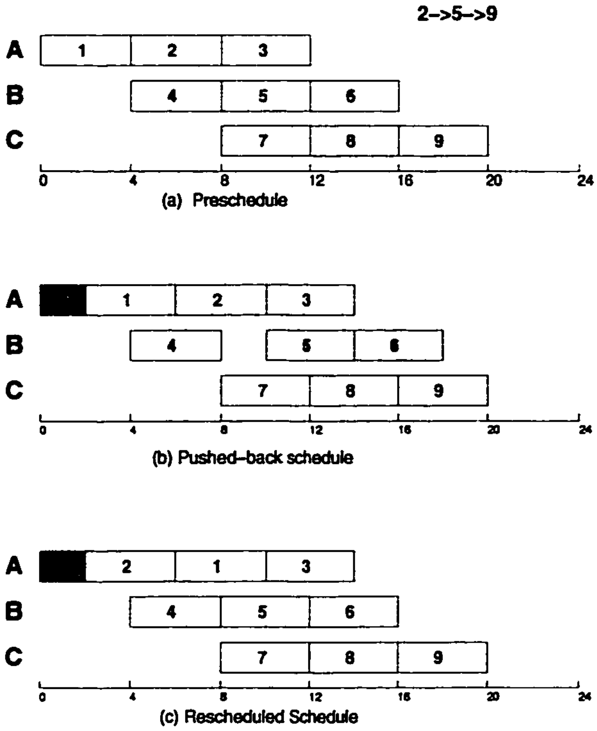
We can view a job shop scheduling problem as a distributed resource allocation problem. Operations are distributed among workcells according to their requirements and workcell capabilities and resource requirements. Obviously, traditional approaches to production scheduling include decomposing the problem into smaller scheduling problems for different non-interacting parts of the manufacturing system. Such scheduling subproblems can then be solved in isolation from one another. However, even an integrated scheduling problem, i.e., a problem that cannot be decomposed into several isolated problems, can still be viewed as a collection of separate but inter-related subproblems.

Local schedule revision can affect other agents' schedules through interactions between the local schedule and other schedules. There are several types of interactions among agents' local schedules:

- *Resource sharing*: If agents are sharing a common resource, then the schedule for the use of that resource is determined by the schedules of those agents using it. If, due to some schedule disruption, one agent reschedules its use of the resource, then other agents' uses of the resource may be affected. An agent may relax another agent's local resource allocation problem by allowing that agent to borrow the resource.

- *Task prioritization*: Agents may have different knowledge about how important different tasks are. Agents may in fact be able to set the global priorities of some tasks. Thus, an agent may discover through negotiation that it needs to reevaluate its view of task priorities. Constraint relaxation takes place when another agent is able to lower the priority of a local task.

- *Precedence constraints*: Agents' schedules are often interlinked via the delivery of parts from one agent to another. An agent's local schedule may indicate when tasks are required to complete so that parts may be transported; the late completion of a local task may disrupt the schedule of another agent waiting for the part in question.



(a) Preschedule

(b) Pushed-back schedule

(c) Rescheduled Schedule

Figure 2.3: A simple example.

Local scheduling problems are relaxed when expectations about when tasks must be completed are relaxed.

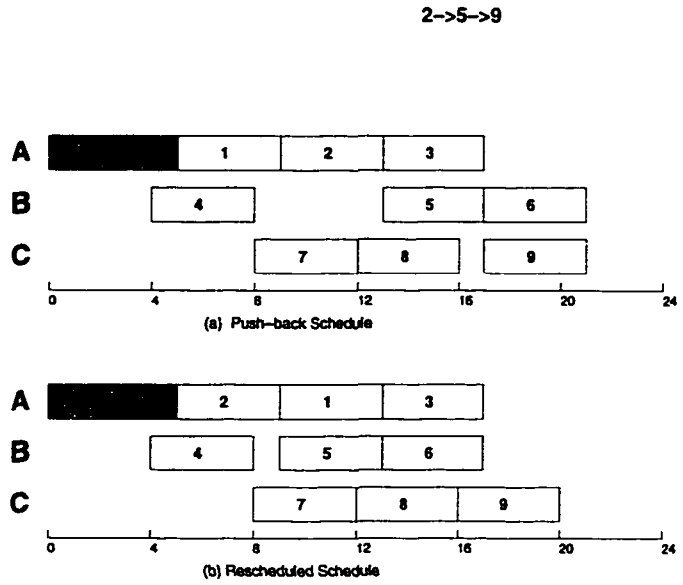
- *Task assignment*: More than one agent may be able to perform a given task. An agent's scheduling problem may be relaxed if another agent agrees to process some tasks originally scheduled locally. The utility of this task migration among agents may depend upon agent workloads and abilities.

These are the interactions about which an agent must reason when determining whether its local rescheduling actions will affect other agents, and how. In the very simple example in Figure 2.3, schedules at different cells interact via precedence constraints. Here there are three cells with one machine per cell. Job 2 has job 5 as a successor, which in turn has job 9 as a successor. The initial schedule is shown in (a). In (b), due to a machine disruption, the machine at cell A is unable to process any job from time 0 to time 2. Cell A's schedule has been pushed back, disrupting the schedule at cell B because of the late processing of job 2.

The algorithm we propose is based upon the outline described in Section 2.1. When a disruption is identified at a cell, that cell will try to reschedule itself without disrupting schedules at other cells; such rescheduling would be a non-disrupting recovery action. It will thus try to find a new schedule in which jobs with successors complete processing before their successors are scheduled to begin processing (in the preschedule). If such a non-disrupting schedule can be found, then the cell will attempt to implement a good non-disrupting schedule, one for example which minimizes some measure not associated with disruptedness (e.g. the measure used for the construction of the initial schedule). While the cell searches for a good non-disrupting schedule, there is a tradeoff between such measures and measures of disruptedness. Our approach would favor a non-disrupting schedule that might score poorly according to the initial measure, over a disrupting schedule that scores better. The work of Bean et al. in matchup scheduling suggests that a local sacrifice of this nature may allow the entire schedule to remain of higher quality according to the initial measure. This is not always the case in the problems we investigate, and this tradeoff must be considered when evaluating under what circumstances this method is appropriate.

If such a schedule cannot be found, then the cell will try to find a schedule that is likely to be least disruptive to other cells. It then will make a proposal to the cells which may be affected by this new schedule, indicating the changes relevant to those cells. Each of these other cells will either accept this proposal, if it determines that it can find a guaranteed-safe response to any disruptions caused by the proposed changes, or reject this proposal, if it cannot determine this. If all of these cells accept the proposal, then the originally disrupted cell will implement it, and the cells disrupted will find and implement new guaranteed-safe schedules that deal with the disruptions caused by the proposed changes.

In the simple example described before in Figure 2.3, while the pushed-back schedule in (b) resulted in a schedule disruption at cell B, the schedule in (c) reschedules cell A without disrupting cell B. In our algorithm, cell A would try to find such a non-disrupting schedule before beginning any negotiations with any other cells. Had the machine of cell A been down from time 0 to time 5 instead, as in Figure 2.4, then cell A first would try to find a non-disruptive schedule, and would fail because none exists. It then would try to find a schedule least likely to be disruptive to cell B. It would then propose this schedule. Were it to propose the pushed-back schedule in (a) of Figure 2.4, cell B would not accept the proposal, as it would be unable to avoid disrupting the schedule at cell C. The schedule in (b) of Figure 2.4, if proposed by cell A, would be accepted by cell B, as it can find a non-disruptive schedule to address the late completion of job 2. Simply delaying jobs, if no



(a) Push-back Schedule

(b) Rescheduled Schedule

Figure 2.4: A simple example (cont'd).

precedence constraints are violated, is a guaranteed-safe action as it cannot cause conflict if other cells similarly delay jobs.

Chapter Three will address the problem of rescheduling when schedules interact via resource sharing, in the domain of tool sharing and borrowing for FMS. Chapter Four briefly considers task prioritization, and Chapters Four, Five, and Six deal mainly with precedence constraints. Task migration is also an important subject, but is not one that we treat in this dissertation.

2.3 Related Work

There has been little work in either the OR or AI communities directly on distributed schedule revision for manufacturing systems. Nevertheless, our work is based upon ideas in the schedule revision and DAI fields, and there is a great deal of research related to our topic in these fields. Here we discuss current research efforts in distributed problem solving, distributed scheduling and schedule revision, and their relation to our problem area and approach.

2.3.1 Distributed Problem Solving

While our particular focus is on the problem of local revision in a distributed schedule, it is important to place this problem in the context of distributed problem solving and multi-agent planning, that was discussed in the previous chapter. Our problem of local revision is one in which a conflict arises, potentially affecting several agents, but about which only one agent is initially aware. Thus the task of responding to the new conflict is centralized at the disrupted agent. This centralization is unlike Steeb's task centralization paradigm [65], which centralizes the task at the least constrained agent, as the disrupted agent is potentially the most constrained agent. However, if this agent must disrupt other agents, determining the least constrained agent or agents is a basis for finding a polite solution.

The distributed scheduling domain is a loosely-coupled domain, in which an agent exists for each workcell. Thus the regions (as in Lansky's GEM model [41]) are well-defined: each workcell is a region. The inter-region constraints are also easily defined, as in the list of potential schedule interactions presented earlier. However, the nature of actual conflicts may not be known by any given agent. An agent may know, for example, that it is expected to deliver a part to another agent by time T, but it might not know how late the delivery may be without affecting the other agent's schedule, or whether the other agent may adjust its schedule to accommodate the late delivery. Thus, while centralizing the problem solving task at the local agent simplifies the solution space, reasoning and communication about inter-agent constraints is required, as in the work of Conry [10] and Pope [58].

Under Conry's multi-stage negotiation framework [9], our **ok-0** reply to a proposal would indicate that the replying agent's goal of remaining undisrupted is not in the exclusion set of the proposal, while an **ok-1** reply would indicate that the replying agent's goal of isolating the new disruption is not in the exclusion set of the proposal. However, because the remote agent is responding to proposed disruptions created by the initially disrupted agent, the remote agent does not know the disruption to which it is rescheduling until that disruption is proposed. Thus, the focus of problem solving is at the initially disrupted agent.

2.3.2 Distributed Scheduling

There has recently been interest in distributed approaches to job shop scheduling. Some are concerned with distributing the computation for various scheduling algorithms [32, 39]. Approaches more related to our distributed rescheduling problem are concerned with decomposing the problem among the various natural actors in the manufacturing system (i.e., jobs, machines, resources, etc.).

The most straightforward of these approaches is that of Parunak's YAMS [54]. This system takes advantage of the hierarchical structure of the manufacturing system by using a contract net methodology. Schedule execution in YAMS is actually a run-time elaboration of a coarse high-level schedule using manager-contractor bidding protocols. Thus a traditional detailed schedule allocating jobs to machines does not exist before run-time. Other contract net-based approaches to similar scheduling problems include Balasubramanian and Norrie's work on design-for-manufacturability systems [3], and the work of Fischer et al. on transportation scheduling [21]. Because contract net systems solve problems very reactively, they may be good for dynamic environments in which traditional scheduling is impractical, but for the same reason they are not adept at optimization, which is the focus of traditional scheduling.

Much similar work has been done in the field of scheduling real-time tasks on a distributed computing system [17, 60]. In this computing task scheduling, a main goal is some form of load sharing, such that some nodes do not remain underutilized or idle while others are heavily-loaded. The main action used to achieve this load sharing is the transfer of tasks from heavily-loaded to lightly-loaded nodes. In most real-time distributed load sharing schemes, the transfer of tasks only occurs when a node has become overloaded (i.e. cannot guarantee that an arriving task can meet its deadline). When a node becomes overloaded, it determines which other nodes are likely to be able to accept the overloading task, through negotiation protocols (e.g. bidding) or through some other information exchange policy. It then sends the overloading task to the remote node which is judged to be most likely to accept it.

Another approach that deals more directly with the problem of constraint satisfaction in scheduling is the work of Sycara on distributed constrained heuristic search [67], in which different agents are responsible for scheduling different sets of tasks, or for monitoring resource reservations. This approach casts scheduling in terms of a DCSP problem. Agents make decisions based upon aggregate resource demand estimates obtained through communication, and coordinate through local resource demand information and resource reservation information at monitoring agents. Backtracking is an important issue, as an agent's current resource reservations must be cancelled if it determines it cannot secure reservations for all required resources. Related work includes Liu and Sycara's work on job shop scheduling by a society of reactive scheduling agents [42].

Distributed meeting scheduling is a similar problem addressed by Sen and Durfee [62], in which a time and place for a meeting must be determined for several agents with different scheduling constraints. An interesting feature of this problem is that local schedule information of individual agents cannot be communicated due to privacy concerns. The responsibility of scheduling the meeting falls to a host, which must negotiate with other concerned agents via proposals and counter-proposals. Work by others on this problem include the economic market-based approach of Ephrati et al. [20].

Relation to Distributed Schedule Revision

Contract net approaches are good for dynamic environments, but because they are reactive, they do not deal with schedule revision per se (as no schedule is constructed to be revised). However, the contract net paradigm is a useful one for disruption handling. In our polite rescheduling approach, the task of finding a 'polite' solution to a disruption is centralized at the disrupted agent, but this agent "contracts" other potentially affecting agents to determine how disruptive the proposed solution may be. In the problems we explore, tasks are not transfered from one agent to another, as in the load sharing problem, but negotiation focuses upon workload states of other agents, as in load sharing, rather than general capabilities of other agents, as in YAMS.

Our distributed rescheduling problem is similar to the backtracking problem in Sycara's DCSP approach. Backtracking is considered very troublesome, and many steps in DCSP approaches are taken to avoid or simplify it (ignoring some of the search space as a result). In our problem, "backtracking" from a previously feasible solution because of a new constraint is unavoidable, but our problem is simplified by the fact that only one agent is concerned with backtracking, and that it is starting from a relatively good point in a partially explored search space, rather than starting from scratch. Resource demand and reservation information used in Sycara's model is also important for an agent's rescheduling decisions.

Sen's work on meeting scheduling has touched upon the rescheduling issue. It is concerned mainly with determining which meetings may be cancelled when a new constraint is added, and uses priority factors and utility measures. In this case, unlike the local schedule disruption that we consider, rescheduling a meeting requires the active participation of all agents involved, as they are all rescheduling the meeting. Thus the solution approach is very similar to the initial scheduling approach.

Part of our approach to distributed schedule revision is related to work on social conventions. In order to determine guaranteed-safe recovery actions, we need to design rules under which actions are guaranteed not to interfere with one another. A relatively new field in DAI studies the design of "social laws", or conventions, by which agents interact in common situations in predictable and coordinated ways, without explicit reasoning about these interactions. Shoham and Tennenholtz [63] describe the off-line design of such rules, while Walker and Wooldridge [70] discuss the on-line development of conventions. While the rules we propose keep agents from mutual interference, Goldman and Rosenschein [29] propose rules by which agents go out of their ways to help one another, using "cooperative state-changing rules" in an effort to be sociable. Our work does not explore very deeply the design of rules for distributed schedule revision; the work being done in this area, however, may provide greater insight into this design.

2.3.3 Schedule Revision

As noted earlier, schedule revision is a topic of increasing importance for scheduling research. There have been several approaches to scheduling for systems that may experience unexpected events. On one extreme is the dynamic scheduling approach, in which no preschedule is constructed. All scheduling decisions are made at run-time, by list processing or dispatch rules [53, 5]. A similar approach involves executing a preschedule until unexpected events render it infeasible, then resorting to dynamic scheduling. Probably the simplest such method, and the most common in every day life, is merely to push the schedule back. The advantages of dynamic scheduling are that it is computationally easy, and that it is robust in unpredictable environments. The big disadvantage is that it is myopic, unable to deal with long-term problems because it concentrates upon immediate decisions.

Another approach to dealing with the possibility of unexpected events is to construct robust schedules which can tolerate a certain amount of unpredictability. For example, idle time on machines may allow a task to use extra processing time if needed, or may allow the processing of a new unexpected task. Likewise, spare machines may allow processing to continue when a busy machine breaks down. However, there is a tradeoff between the goal of having a robust schedule and that of maximizing the utilization of time and equipment. Robust scheduling has been a topic in the stochastic scheduling field, notably by Pinedo [57] and by Daniels and Kouvelis [12], in which expectations about processing times and machine up-times are used to maximize expected schedule quality. Similarly, the *"just-in-*

case" scheduling of Drummond, Bresina and Swanson [66, 14] uses contingency schedules for points at which the schedule is most likely to break.

An approach that combines aspects of the rescheduling and robust scheduling approaches is the matchup scheduling approach of Bean et al. [4]. In this approach, when unexpected events disrupt the preschedule, the scheduler attempts to schedule production so that the system can return to ("match up with") the original preschedule. Thus, the good preschedule need not be discarded when disruptions occur. The preschedule must be robust enough so that the system can return to it in a reasonable amount of time, and the system must be rescheduled efficiently until it matches up with the preschedule. The matchup paradigm reduces the problem of recovering from disruption to the narrower problem of returning to the preschedule.

Several AI approaches to rescheduling treat a disrupted schedule as simply a point on a search path for a regular scheduling problem. Smith's OPIS [64] treats unexpected events and intermediate solution states identically. Likewise, Zweben's GERRY rescheduler for space shuttle scheduling "constructs" a schedule by starting with an infeasible schedule, and then trying to resolve constraint conflicts using iterative heuristics or simulated annealing [75]. The quality of the schedule for GERRY is determined by the number of constraint violations, an over-constrained problem always being the assumption in that particular domain. Similar approaches include Johnston and Minton's work on "local repair" heuristics [35] and Miyashita and Sycara's case-based CABINS rescheduler [47]. Both approaches use local variable reassignment in order to reduce the number of constraint violations. CABINS also relaxes constraints that otherwise over-constrain the problem.

Rescheduling is also related to the topic of replanning in AI. Early work on replanning dealt with "patch planning", in which small already existing contingency plans are inserted into a plan whenever an unexpected event occurs for which a contingency plan exists. Execution monitoring and replanning in Wilkins' SIPE [71] is one of the first sophisticated approaches to the problem of replanning. Particularly relevant to our problem in which one agent needs to replan while taking into account other agents is Ephrati and Rosenschein's work on "constrained intelligent action" [19], that considers the problem of "non-absolute" control among intelligent agents, involving a "supervisor" agent giving orders to a "subordinate" agent. This work deals with how the subordinate agent modifies plans given to it by the supervisor. Such modification may be necessary if the subordinate has access to information that the supervisor doesn't. In this case, the subordinate must reason about how much the original plan should be modified. Approaches proposed include reasoning about differences in costs of plans, differences in knowledge bases of each agent, and actions with irreversible consequences. Somewhat similar issues are considered by Huber and Durfee in their work on "observation-based" coordination, in which agents must infer other agents' plans in the absence of communication [33].

Relation to Distributed Scheduling Revision

Our work follows the matchup scheduling approach described above in that it tries to retain as much of the existing schedule as possible. Whereas matchup scheduling deals mainly with returning to the original schedule after a certain interval of rescheduling, we try to retain the original distributed schedule along the dimension of space by avoiding the disruption of cells remote from the disruption. Our approach differs somewhat from the several AI rescheduling approaches, in that we generally try to construct a new schedule for the disrupted agent, rather than trying to repair it. Constructing a new local schedule is no requirement for polite rescheduling, however, and repair heuristics may well be used where appropriate.

While we assume that communication with other agents is possible, unlike the constrained intelligent action case, we do also assume that negotiation is expensive. Any proposals made by the disrupted agent should be guided by reasoning about other agents' likely objectives. Communication-poor coordination becomes even more important when good information about other agents' schedules is not forthcoming through negotiation, as is the assumption in the meeting scheduling field. These issues will be discussed in chapter six, when we consider simple probabilistic approaches to proposal formulation.

Finally, while the problem of creating robust schedules is an important one, useful schedules always have the potential to be disrupted. Tradeoffs between robustness and resource utilization, and between robustness and contingency planning requirements for time and computation, suggest that there will be a point at which "robustifying" a schedule (to use Drummond's term) will cost more than it provides, but at which schedule disruption is still a non-negligible possibility. Our focus is on ways of dealing with disruptions when they occur.

CHAPTER 3

A Distributed Approach to Intelligent Tool Management

In this chapter, we will explore the application of polite rescheduling in the domain of tool management, and propose possible approaches to the problem of tool allocation. We will consider the problem of sudden unexpected tooling requirements, in which tool management may require a re-allocation of tools, in the context of a simple tool scheduling problem, and a more realistic tool rescheduling problem. Specifically, we will show that polite scheduling and rescheduling of tools, with only local knowledge, has performance close to optimal solutions for maximizing the number of tasks scheduled, with significantly less disruption to the overall schedule. Thus, where collection of global information is not convenient, polite scheduling and rescheduling is a good approach when these performance measures are relevant. We propose polite negotiation protocols by which a cell attempts to solve local tooling problem without causing new problems for other cells, despite inter-cell tooling constraints. In this context, we will also consider some basic issues involving distributed approaches to resource allocation and re-allocation.

3.1 Tool Management in FMS

Tool management, the allocation and scheduling of tools, is an important problem in FMS. A machine tool is an implement usually specialized for cutting, drilling, or shaping metal or other matter. It is often separate from the machine using it, so that a tool used at one machine can be removed and transferred for use on another machine. Tools are often expensive, and the efficient use of tools may thus require sophisticated strategies for tool allocation. Tool management obviously has much to do with scheduling. The standard scheduling problem formulation involves the assignment of jobs to machines given various job and machine constraints. This formulation can easily be extended to include the problems of tool allocation and scheduling.

3.1.1 Schedule Execution and Tool Management

If there were no unexpected changes during the production cycle, schedule execution and tool management during execution would be a straightforward matter. However, unexpected events (or disruptions), such as the arrival of a new job, changes in job priorities, the breakage of a tool, or the breakdown of a machine, may pose problems for schedule execution. The current schedule could of course be discarded, and a new one constructed, taking into account the new requirements imposed by the disruptions. However, as mentioned previously, this re-scheduling can be costly, not only because of the re-scheduling task itself, but also commitments based upon the original schedule, dealing for example with material transport or personnel, may have to be reorganized. At worst, guarantees made to a customer about delivery times may be violated. Thus, when unexpected events can occur, one goal is to handle disruptions with as little change to existing schedules as possible.

The controller at the machine or work-cell affected by a disruption is at first responsible for responding to the disruption. To solve the schedule execution problem posed by one of these disruptions, it needs to consider not only the original job, machine, and tool constraints, but also schedule constraints imposed by the current schedule. Tool and machine availability for example are now greatly restricted.

We focus on the problem caused by unexpected tooling requirements. In order to find a solution, some of the constraints of the scheduling problem may have to be relaxed. For example, the processing of a lower priority job may have to be postponed or cancelled. However, negotiation may allow some of the schedule constraints to be relaxed more easily. Tool availability may be less restrictive if a required tool, assigned to another machine, may be borrowed. Simply seizing the required tool, regardless of where it is assigned according to the current schedule, may be another way of solving the local problem, but this may cause another disruption at the machine which expected the tool to be available. Thus, negotiation should ensure that this does not happen, and we term such an approach *polite*. We examine this type of approach in the context of common tool strategies, in a distributed manufacturing model.

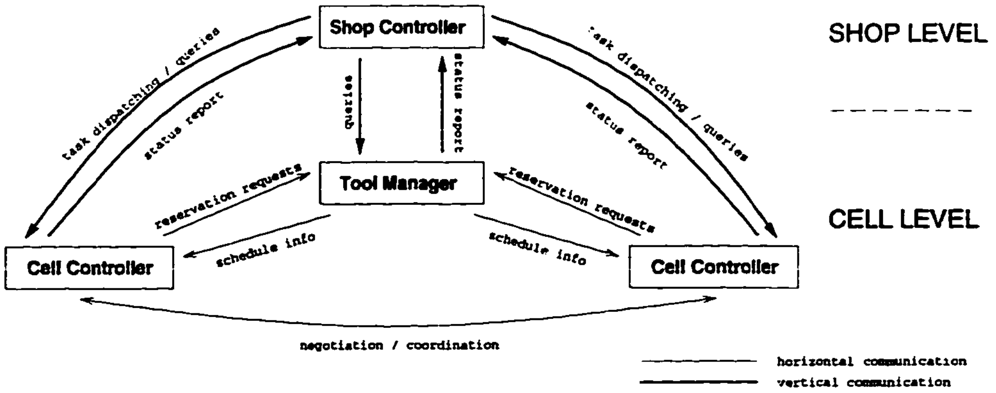


Figure 3.1: A Distributed Manufacturing System Model.

3.1.2 Common Tool Strategies

Common tool strategies include *mass exchange*, *tool sharing*, and *tool migration* [43]. In the *mass exchange* strategy, each work cell has all the tools required for any task it may ever perform. While this strategy is simple, it is not very efficient if different tools are required for the different tasks one cell can perform; particular tools may often go unused. In the *tool sharing* strategy, each work cell has every tool required for every task it is to perform in the next production cycle. Between production cycles, tools may be moved from one cell to another. Thus, while the management of the tools is somewhat more complicated, tools can be used more efficiently. In the *tool migration* strategy, tools can be moved from cell to cell during the same production cycle, so that a tool, that has been used but is no longer needed at one cell, can be transported to another cell, where it is needed. Thus, even more efficient use of tools is possible.

Tool sharing and tool migration offer greater flexibility, but are clearly harder to implement. Tool sharing requires information about the locations of the tools and how they will be allocated during the next production cycle. Tool migration requires this information, and information about which tools are to be moved from one cell to another, and when the transfer is to take place. An important issue in both of these strategies is tool scheduling, the allocation of tools to work cells over time.

3.1.3 Distributed Manufacturing System Model

In our model of a distributed manufacturing system, information required for tool management may be distributed. For example, as illustrated in Figure 3.1, a shop level supervisory module may have only general information about tooling and job status; a cell controller may have detailed information about cell status and the status of jobs at that cell, while a tool manager module may keep track of tool status (e.g., age, capabilities, etc.) and scheduling. Tool management may require detailed information about cells, jobs, and tools, that may not be centralized in any one module. This module is similar to the contract net-based multi-agent shop-floor control model of Balasubramanian and Norrie [3].

3.2 A Simple Tool Scheduling Problem

We begin our investigation of using polite re-allocation methods in the tool management domain by considering a very simple tool scheduling problem. In this problem, we consider the allocation of time slots for use of a tool among tasks which require use of that tool. We consider a system in which there are tool manager agents and task manager agents. For each tool there is a tool manager agent, which knows the current schedule for the tool (that is, which time slots have been allocated for use by which tasks). For each task there is a task manager agent, which knows the tool requirements of the task.

The constraints for this problem specify, for each task, not only which tools are needed, but also during which time slot windows these tools may be used. This is a simplified version of the time window scheduling problem, in which tasks may be scheduled only during certain time windows. These window constraints may reflect other commitments that may have already been made regarding the task; they are local to the agent which manages the task. A resource capacity constraint requires that only one task may use a given tool during a given time slot. A task may use one or more tools during any given time slot, and to complete processing it must have use of each of its required tools during one of its time slots (not necessarily the same one). The task managers may communicate with tool managers in order to request reservations or scheduling information, and may communicate with one another to coordinate their actions, as illustrated in Figure 3.2.

3.2.1 Problem Description

One obvious goal of scheduling these tasks (allocating tool time slots to tasks) is to maximize the number of tasks that are allowed use of each of their required tools. Given this goal, a backtracking algorithm can be used to find an optimal solution, or some heuristic can be used to find a 'good' solution. Other possible goals conjunction with this goal might to be to consider when tasks will be able to complete processing.

However, these tasks may arrive individually over time, rather than all at once. Because

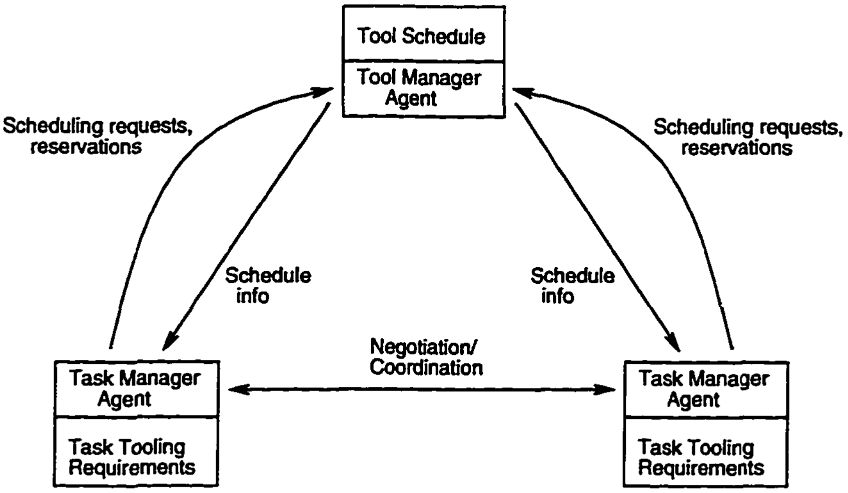

Tool Schedule
Tool Manager Agent
Scheduling requests, reservations
Scheduling requests, reservations
Schedule info
Schedule info
Negotiation/ Coordination
Task Manager Agent
Task Tooling Requirements
Task Manager Agent
Task Tooling Requirements

Figure 3.2: Negotiation for Tool Scheduling.

a schedule represents a commitment to those executing the schedule, and perhaps to a customer, we would like to avoid rescheduling a task once it has already been scheduled. Thus, instead of rescheduling every current task anew each time a new task arrives, our concern is how best to allocate tool use to tasks incrementally. In other words, we will consider ways in which the schedule may be modified, rather than completely rescheduled, to accommodate each task arrival. When a new task arrives, its task manager is responsible for reserving the required time slots on the appropriate tools. If time slots cannot be reserved, then the task must be rejected. Each task requires the use of its required tools for one time slot each. For each required tool, the task has a number of possible time slots during which it may use the tool.

An example of such a tool scheduling problem is shown in Figure 3.3, in which three tasks have arrived and have been scheduled. When the new task arrives, either it must be rejected, or one of the already scheduled tasks must be dropped or rescheduled. Figure 3.4 shows the original schedule, along with two possible schedules that allow the new task to be scheduled with the previous three. While both schedules include all four tasks, schedule 1 changes the schedule time for all three of the originally scheduled tasks, while schedule 2 changes only the time for task A, and is therefore less disruptive.

As mentioned previously, the responsibility for scheduling a new task belongs to that task's manager. It tries to schedule the task by requesting time slots from the appropriate tool managers. If possible time slots for each tool are available, then the task manager can reserve use of the required tools during appropriate time slots, and the task is thus

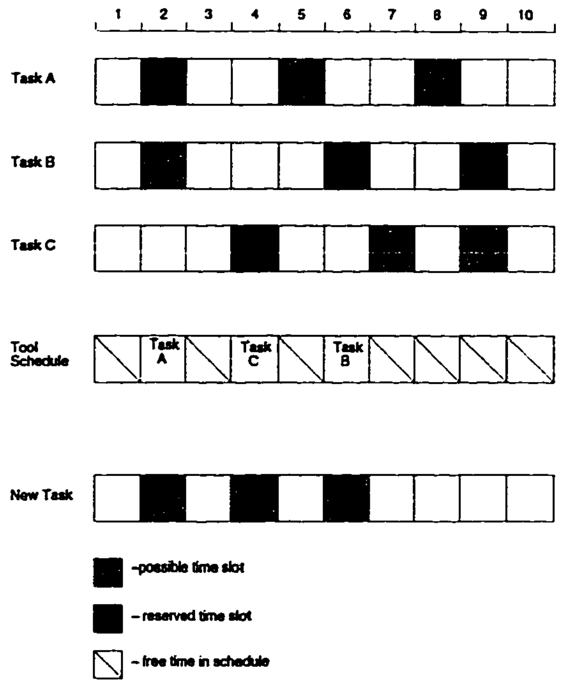


Figure 3.3: A Simple Tool Scheduling Problem.

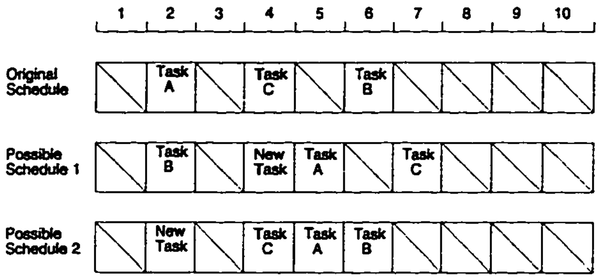


Figure 3.4: Possible Tool Schedules.

accepted. It is clear that reserving an available time slot is a non-disrupting action.

However, one or more of the tools may be unavailable during all of the possible time slots. In this case, the environment of the current schedule and inter-agent resource constraints make scheduling the new task impossible. The task manager may handle tool unavailability in one if the following ways. It can decide that its task must be *rejected*, grab the tool during one of the task's possible time slots (thus denying it to the task originally scheduled to have the tool for that time slot), request of the tool manager that the tool be *completely rescheduled*, or *negotiate* with other task managers, to relax local constraints stemming from inter-agent resource constraints. In the traditional bidding approach of the contract net [54, 3], the task would be rejected, as the tool manager would be unable to satisfy the request. When task agents may negotiate with one another, however, a task agent with a tool reservation required by another can become a "contractor", offering use of the tool as if it were a tool manager, if it determines that it can obtain another replacement reservation. We will show that, when complete rescheduling is costly because the collection of global information is not convenient, negotiation to relax local constraints has performance close to optimal in terms of number of tasks accepted, while being less costly than complete rescheduling in terms of the number of tasks that have to be rescheduled.

For example, in the problem in Figure 3.3, once the task manager discovers that the task cannot get the tool given the current tool schedule, it can ask the task manager for task A whether task A can use the tool in a different time slot. Because task A can find out from the tool manager that it can also use the tool during time slot 5, its manager would reply to the new task manager that it can change time slots. Thus, the new task can gain use of the tool without changing the tool use plans of tasks B or C. The resulting tool schedule would be the second possible schedule shown in Figure 3.4. If task A had not been able to use a different time slot, then the task manager of the new task could then have asked the task managers of task B or task C.

Problem Statement and Solution Methods

In these simulations, there are s tasks which arrive separately, all requiring use of the same set of m tools. There are s time slots for which tools may be reserved; during one time slot, a tool may be reserved for only one task. Each task has n randomly chosen possible time slots; obtaining use of a required tool for any one of these time slots will satisfy that task's requirement for that tool. In order to simplify the simulation, the first task will not start processing before the last task arrives. As each task arrives, its task manager attempts to schedule it. If it can reserve time slots on each of the required tools, it is accepted. Otherwise, it is rejected. The problem at the ith task arrival is described as follows. Given the time slots, the m tools, $j \leq i - 1$ previous successfully scheduled tasks and their tooling requirements and possible time slots, the existing reservations for the j jobs, and the ith job and its possible time slots and tooling requirements, the problem is to find a set of reservations for all $j + 1$ jobs by which each job will have reservations for each of its required tools.

We compare two methods, a simple scheduling method, and a polite scheduling approach. In each method, the task manager of a new task first contacts the tool managers to enquire whether any of the possible time slots are available. If an appropriate time slot is available for each tool, then the task is scheduled. If not, in the simple scheduling method, the task is rejected, while in the polite scheduling method, the tool manager informs the task manager which tasks have already reserved those requested time slots. For each unavailable tool, the task manager then contacts the manager of each of those tasks one at a time, to ask whether its tool reservation can be rescheduled. The attempt to reschedule by a task manager is in fact a guaranteed-safe rescheduling action, because at any time, only one task manager will be trying to reschedule a given tool. This would not be true if tasks had to reserve all their tools during the same time slot, as task managers trying to reschedule might be trying to change reservations on the same tool, allowing the possibility of deadlock. The tool allocation algorithms for a task t requiring one time slot on one tool can be described simply as follows:

n = the number of possible time slots for task t.
p_i^t = ith possible time slot for task t.
k_t = the number of tools required by t, where $k \leq m$.
Q_t = the set of k_t tools required by t, where $Q_t = \{q_1^t, \ldots, q_{k_t}^t\}$.
$\mathrm{sch}_q(j)$ =reserve if time slot j for tool q is already reserved,
    hold if it is tentatively reserved, free otherwise.
T_j^q = the task which has reserved time slot j for tool q, if one exists.
r_t^q = the time slot reserved for task t on tool q, if any; none otherwise.
h_t^q = the time slot tentatively reserved for task t on tool q, if any.

```
subprocedure *make-tentative-reservation* $(t, q)$
0. $i := 1$.
1. If $\mathrm{sch}_q(p_i^t)$ = free, then goto 3; else $i := i + 1$; If $i \leq n$ then goto 1; else goto 2.
2. Set $h_t^q$ := none. Report failure. End.
3. Set $\mathrm{sch}_q(p_i^t)$ :=hold and set $h_t := p_i^t$. Report success. End.
```

```
subprocedure *tentatively-change-reservation* $(t, q)$
0. $i := 1$.
1. If $p_i^t \neq r_t$ and $\mathrm{sch}_q(p_i^t)$ =free, then goto 3; else $i := i + 1$; if $i \leq n$ then goto 1; else goto 2.
2. Report failure. End.
3. Set $\mathrm{sch}_q(p_i^t)$ :=reserve. Set $h_t^q := p_i^t$ and $r_t^q := p_i^t$; Report success. End.
```

```
subprocedure *confirm-reservations* $(t)$
0. For all $q \in Q_t$,
    0.1. Set $\mathrm{sch}_q(h_t^q)$ = reserve;
    0.2. If $r_t^q \neq h_t^q$,none, set $\mathrm{sch}_q(r_t^q)$ := free;
    0.3. Set $r_t^q := h_t^q$.
```

```
1. End.
```

```
subprocedure cancel-reservations ($t$)
0. For all $q \in Q_t$,
    0.1. If $h_t^q \neq$ none, then set $\text{sch}_q(h_t^q) :=$ free;
    0.2. Set $r_t^q :=$ none;
1. End.
```

```
procedure simple-schedule ($t$)
0. For all $q \in Q_t$, make-tentative-reservation($t, q$); If failure, goto 2;
1. confirm-reservations($t$); report success; End.
2. cancel-reservations($t$); report failure; End.
```

```
procedure polite-reschedule ($t$)
0. For all $q \in Q_t$,
    0.1. make-tentative-reservation($t, q$); if failure, goto 0.2; else goto 0.5;
    0.2. $i := 1$.
    0.3. Contact task $T^q_{p_i^t}$ and request that it tentatively-change-reservation($t, q$);
    if successful, then goto 0.5; else $i := i + 1$, if $i \leq n$ then goto 0.3; else goto 2;
    0.4. make-tentative-reservation($t, q$);
    0.5. Continue;
1. confirm-reservations($t$); End.
2. cancel-reservations($t$); End.
```

We compare these heuristics, which use only local information about who has a reservation for a particular time slot, with optimal methods which use global information about tool schedules to construct schedules with each new task arrival. One optimal method simply maximizes the number of tasks accepted. Another uses a cost factor c for each task that is moved to a different time slot; for each new schedule, this method maximizes $t - cr$, where t is the number of tasks accepted, and r the number of tasks rescheduled. These optimal methods require not only global information, but also a great deal of search, even when bounds are used; the search tree has depth equal to s (the number of task requests), and a branching factor n^m (where m is the number of tools required). We will also compare results with a good upper bound, based upon total demand characteristics, for problems that are too big for optimal solutions.

It must be noted that the optimal methods use perfect information about current tool schedules, but not about future task arrivals. If future task arrivals can be perfectly predicted, no task rescheduling would be required, as tasks are only rescheduled to allow

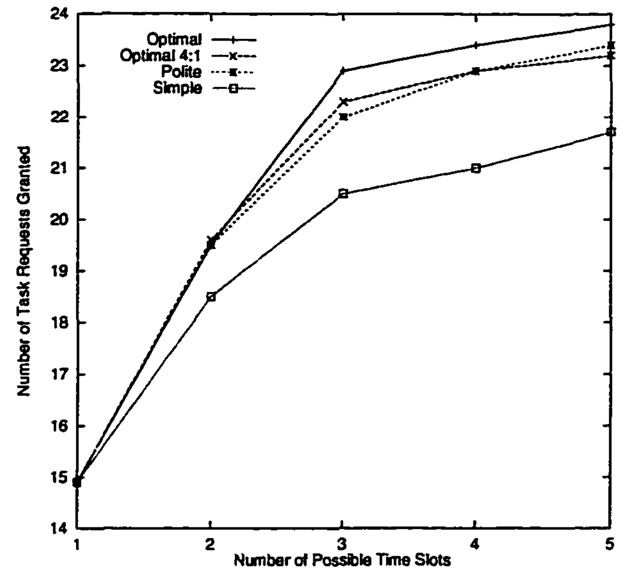


Figure 3.5: Simulation Results: Task Scheduling with Two Tools.

scheduling of other tasks. If some information about probabilities of future task characteristics were known, tasks could be scheduled in order to decrease the expected need for future reschedulings, using some robust scheduling method as discussed previously. In our simulations, however, tasks have uniformly random tool-time slot requirements, so no such advantage can be taken.

Simulations and Results

Each of our simulation figures shows the results averaged from 20 simulations. Figure 3.5 shows the number of tasks successfully scheduled versus n, the number of possible time slots for each task for each tool. In this case, one tools is required, and 24 task requests arrive, one at a time ($s = 24$). Here, Optimal maximizes the number of tasks accepted, while Optimal 4:1 has $c = \frac{1}{4}$. As would be expected, more tasks are scheduled as n increases and the problem becomes less constrained. The polite scheduling method also schedules about 9% more tasks than the simple method, and comes within 5% of the optimal method. Figure 3.6 shows the number of tasks accepted (here with only 16 task requests and time slots) versus the number of tools required, with n set to 3. As the number of required tools is increased, it becomes harder to schedule tasks, as would be expected because the problem is more constrained, and the polite method performance degrades relative to the optimal solution. Figure 3.7 shows the number of tasks accepted versus the size of the problem,

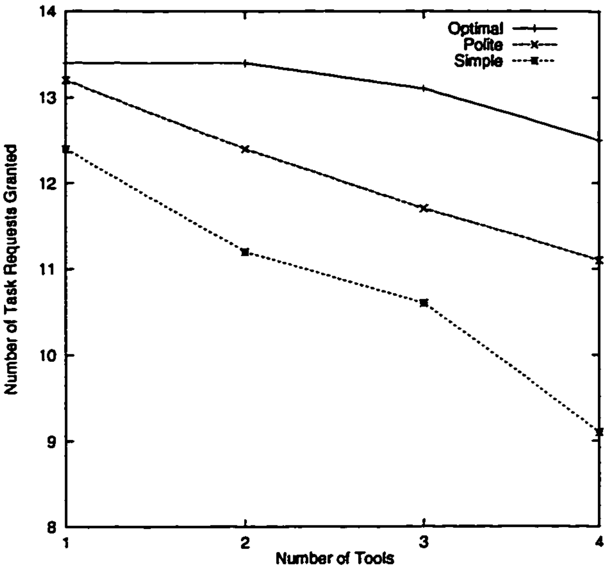


Figure 3.6: Simulation Results: Task Scheduling vs Number of Tools.

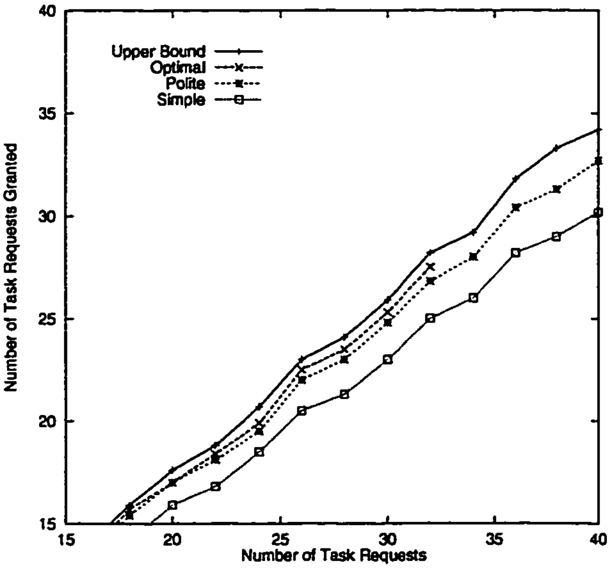


Figure 3.7: Simulation Results: Task Scheduling vs Problem Size.

with one tool and $n = 2$.

While the polite scheduling method is able to schedule more tasks (and thus increase tool utilization), it also imposes additional costs. One of these costs is the rescheduling of certain tasks during the negotiation process. We have mentioned previously that rescheduling is costly, because it often requires changing other commitments associated with the task. Figure 3.8 shows the number of tasks rescheduled versus n, where there is one required tools. This number increases initially when n increases, as it becomes easier to reschedule tasks. However, it soon begins to drop, as an increasing n makes the initial scheduling of tasks easier, thus reducing the need for rescheduling. Here, Optimal 10:1 is an optimal method with $c = \frac{1}{10}$, that performed as well as Optimal in terms of number of tasks accepted. The figure clearly shows that the polite method requires fewer task reschedulings than the optimal methods, even when the optimal methods take into account task rescheduling. As c is increased, the optimal method reschedules fewer tasks but accepts fewer tasks, and as c is increased above $\frac{1}{4}$, the task acceptance performance becomes worse than that of the polite method.

Simple Negotiation Strategies

Another cost associated with the polite approach is the communication required by negotiation between task managers. While with increases in network speed, this communication itself might not cause a significant delay, there can be significant costs in network congestion from message traffic, and in computation time from the processing of incoming and outgoing messages. Thus, an important part of negotiation strategies is trying to reduce the number of messages generated.

Two basic ways of reducing message traffic are to contact first those agents most likely to help, and to avoid communicating with agents known to be unhelpful. In order to contact first those agents most likely to help, an agent needs information about which agents are likely to help, and which are not. In this simple tool scheduling problem, the task manager has such information by virtue of the fact that, when a task is initially scheduled, it reserves the earliest possible time slot. Thus, a task with an earliest time slot on a tool is more likely to be able to reschedule than a task with a later time slot, because the later task is more likely to have been unable to schedule earlier and thus is likely to have fewer alternative possible time slots. Sen's work in distributed meeting scheduling shows how search bias

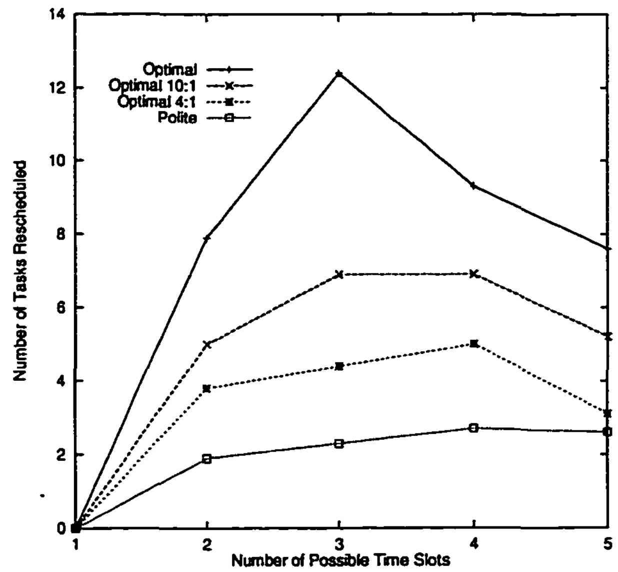


Figure 3.8: Simulation Results: Task Rescheduling.

determining scheduling preferences can affect communication costs and iterations required for a solution [62]. Here, the same is true when agents are competing for individual time slot reservations, rather than trying to find a common acceptable time slot for a common meeting objective. This early-first preference is equivalent to Sen's *Linear early* search bias, and depends upon agents having common initial scheduling preferences.

In order to avoid communicating with agents known to be unhelpful, an agent needs information about which agents are not helpful. One possibility is to decide that any agent that was unhelpful in the past will be unhelpful in the future. While this assumption may not hold when agents' situations are frequently changing, in this simple problem, it is a very good assumption. If a task cannot currently change its time slot on a tool, it is unlikely that it will ever be able to do so in the future. Thus, the tool manager for each tool can keep track of which task managers were unable to reschedule on that tool, and inform a requesting task manager only of those other tasks managers that have not yet been unable to reschedule. In this scenario, the tool manager can accomplish this without extra communication; once a task manager has requested a reserved time slot from the tool manager with the intention of negotiating with the holder of that reservation, and that reservation has not been relinquished by the reservation holder, then that reservation holder can be marked as unhelpful. Otherwise, an unsuccessful negotiator could send a message to the tool manager reporting that a reservation-holding task manager has been unhelpful.

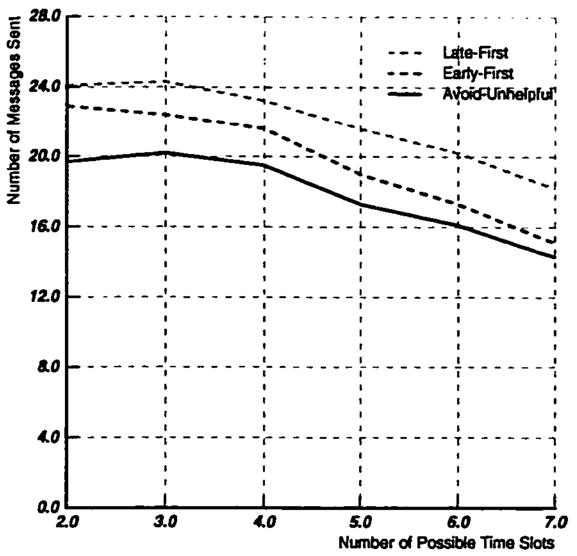


Figure 3.9: Simulation Results: Task Rescheduling.

Some communication with these unhelpful task managers can thereby be avoided.

Figure 3.9 shows the advantages of these methods of reducing message traffic. The graph shows the number of messages between task managers versus n, where two tools are required by each task. In the late-first method, a negotiating task manager contacts other task managers in latest-first order, with regard to the time slot in question. The early-first method does so in earliest-first order. The avoid-unhelpful method does so in earliest-first order, and avoids contact with unhelpful task managers as described above. The figure shows the success of these two ways of reducing message traffic. It also shows that the number of messages falls as n is increased, even though, with a greater n, each task manager can negotiate with more task managers; the explanation is that the problem is less constrained and requires less rescheduling with a larger n. Likewise, the advantage of avoiding unhelpful task managers is less with greater n, because fewer task managers are likely to be unhelpful when the problem is less constrained. In these simulations, each method performed equally well with respect to the number of task requests granted.

3.2.2 Discussion

These results show that, in the incremental scheduling of tasks, polite methods using only local knowledge gained through negotiation with a small subset of agents has performance close to that of optimal for maximizing the number of tasks accepted, at lower cost in terms of the number of tasks rescheduled. This is particularly true when inter-agent constraints are limited to the sharing of one tool; as more inter-agent constraints are added with the sharing of more tools, the polite method performance in terms of the number of tasks accepted degrades relative to optimal methods, the problem size for optimal solutions grows exponentially with the number of tools. This problem, while simple, provides a simple domain in which to study aspects of polite replanning and negotiation as applied to scheduling. Our results also provide a basis for our investigation of a more realistic tool allocation and scheduling problem discussed in the next section.

3.3 A Tool Borrowing Problem

In the problem of the previous section, one simplifying assumption is that tool transport costs need not be considered. Any task could reserve any tool at any time, regardless of where that task was being performed. While this assumption is valid for a stationary resource, it is not valid for portable tools. A more realistic approach would consider the time required for transport of a tool from one manufacturing workcell to another. If a tool is frequently moved among several cells, much potential utilization of the tool will be wasted in transport, given non-negligible transport times. Tool setup times should similarly be considered, as should the resources required for moving the tool. Thus, some balance should be maintained between moving tools among cells to allow sharing, and keeping the tool in one place to allow efficient utilization.

The *tool sharing* strategy described previously, in which tools are exchanged between (but not during) production cycles, provides this balance. This strategy is similar to using batch scheduling to reduce machine setup costs. In this strategy, a tool is allocated to only one cell during a production cycle (an 8-hour shift, for example). The operations at that cell which require use of the tool are scheduled during one such production cycle. For example, Figure 3.10 shows a schedule in which three cells share one tool, cell A getting the tool during shift 1, cell B during shift 2, and cell C during shift 3.

Tool sharing, however, does not address the problem of unexpected tooling requirements, due to a new job arrival, longer-than-expected processing of existing tasks, or machine breakdown. Due to such circumstances, a cell may require a tool during a production cycle during which the tool has been allocated to another cell. For example, suppose cell A in Figure 3.10 receives an unexpected job that must be scheduled during either shift 2 or shift 3, and that requires use of the tool. Because the tool is not scheduled to be at cell A during

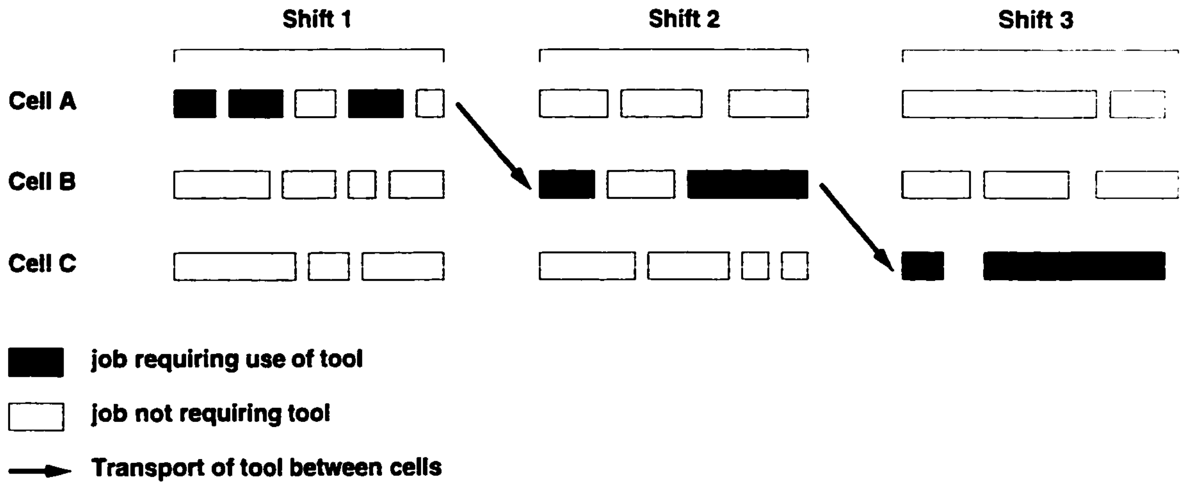


Figure 3.10: Tool Sharing Example.

either of those shifts, cell A would have to reject this new job, under the simple tool sharing strategy.

Allowing *tool migration* (i.e., exchange of tools during production cycles) during such exceptional circumstances may permit handling of these unexpected events. In the example, cell A could use the tool during shift 2 or shift 3, and then send the tool to the cell for which it is scheduled. Using a tool when it is scheduled for use at another cell, however, should not be done haphazardly. Here again negotiation can be useful; the cell which needs the tool can negotiate with the cells which will have the tool, and determine whether the tool can be obtained, and from which cell it should be obtained, if there is a choice. For example, in Figure 3.11, cell A from the previous example can "borrow" the tool from cell B during shift 2, without changing cell B's scheduled output for shift 2.

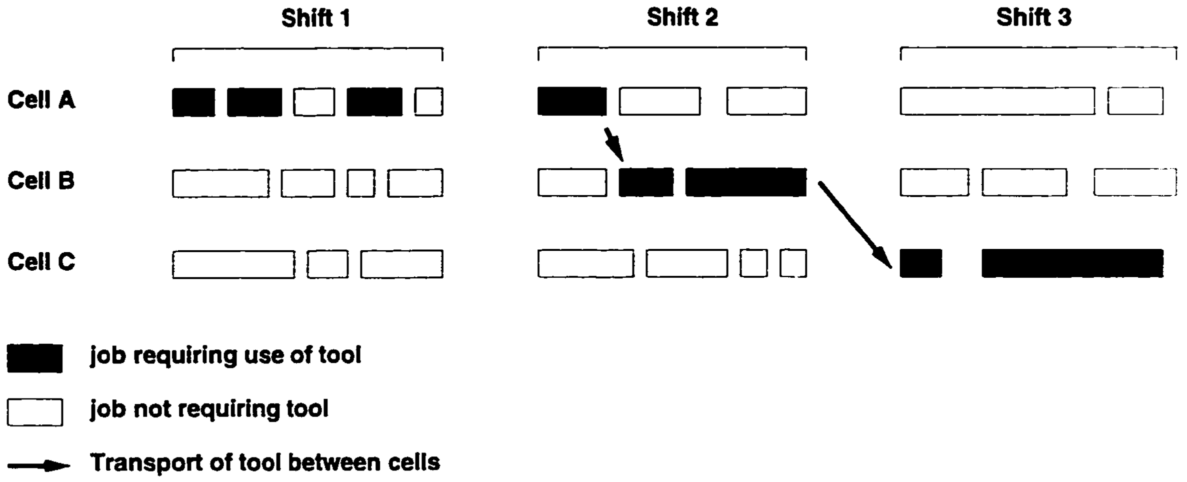


Figure 3.11: Tool Sharing Example, cont'd.

3.3.1 Problem Description

We shall investigate how negotiation can allow useful tool migration, or *tool borrowing*, when unexpected tool requirements arise, but first we should discuss what kind of schedules and assumptions are involved in this problem. For simplicity, we assume here that there are a fixed number of manufacturing cells that share a small set of tools; here we shall consider just one tool. A similar approach should effectively handle situations in which more than one tool is being shared, but will require more complex handling of job sequencing.

Each of these cells has a set of jobs to process. This job set changes as jobs arrive or are completed. A job may require use of the tool, and can be of high priority (e.g., it is being processed to fulfill a particular order) or of low priority (e.g., it is being processed to build inventory). We assume that the production cycle in this problem is a shift, and that jobs have release times and due times that indicate during which shift the jobs can begin processing and by which shift the job must complete processing, respectively. Here we assume jobs have no precedence constraints, and each job can thus be scheduled any time between its release time and its due time.

A schedule is constructed for a *scheduling horizon*; shifts beyond the scheduling horizon are not considered in the schedule. Given a scheduling horizon, a schedule is a set of assignments: an assignment of the tool to one cell for each shift within the scheduling horizon, and for each cell, an assignment of its jobs to a shift within the scheduling horizon. In this problem, we are unconcerned how the jobs are sequenced within a shift; we only require that every job assigned to a shift can be processed during that shift. Unassigned jobs in a schedule can be processed after the scheduling horizon, or may be rejected.

We are interested in finding an effective and efficient way to respond to unexpected tooling requirements during schedule execution. Given a schedule for a particular job set and scheduling horizon, and an unexpected demand for the tool (e.g., an unexpected "rush" priority job that requires the tool), the objective is to meet the new tooling requirement if possible without completely rescheduling all the cells, without requiring use of a centralized scheduling agent, with as little disruption as possible to cell schedules that must be changed, and without having to reject other already scheduled priority jobs.

Before examining possible approaches to this problem, we should first investigate how a schedule is initially constructed. How a schedule is constructed may give insight into how to proceed when it is to be modified.

3.3.2 Constructing the Initial Schedule

In order to determine an initial schedule, we first must determine how the tool is to be allocated among the cells. It would be very hard to construct a schedule for a cell without knowing during which shifts the cell is going to have possession of the tool. Furthermore, the tool is clearly a bottleneck resource, and the results of ISIS [23] and OPIS [64] suggest that bottleneck resources be scheduled first. Thus, it makes sense first to decide which cells have the tool during which shifts, and second to schedule each cell given this allocation of the tool.

Scheduling the tool first and the cells second illustrates an example of how a scheduling problem can be distributed. Once the tool has been scheduled, each cell can be scheduled independently of the others, and no centralized scheduler is needed. If, however, jobs may have precedence constraints, then interaction between cells' schedules may require a centralized scheduler or some form of cooperation.

The scheduling process is as follows:

1. A schedule for the tool is determined by considering the potential demand for the tool for each shift by every cell. This approach is motivated by the work on the "texture"-based resource allocation of Fox [22] and Sycara *et al.* [67], which also suggests how this allocation can be accomplished in a distributed way. Here, however, we assume that the tool allocation will be determined in a centralized fashion, by the tool manager:

   (a) the aggregate potential demand for the tool is determined for each shift.

   (b) for each shift (in least-demanded-first order), the tool is awarded to the cell with the highest demand for that shift; the aggregate demand is then adjusted appropriately ,by removing the demand that has just been satisfied.

2. A schedule for each cell is determined separately, by allocating jobs to shifts using a simple capacity constraint propagation algorithm. Here we try to schedule jobs close to their due dates, though trying to schedule jobs soon after their release dates would be very similar. Thus, each cell is scheduled as follows. First, each job is tentatively assigned to the shift of its due date. Then, for each shift, in latest-to-earliest order:

   (a) jobs are assigned to the shift using a simple knapsack packing algorithm, giving greater preference to more important jobs (as described below), and taking into consideration whether the tool is available during the shift;

(b) the capacity constraints are propagated by assigning to the next earliest shift the remaining jobs, which did not fit in this shift.

Jobs which are propagated beyond the earliest shift (i.e., those which could not be assigned to any possible shift) are rejected.

In this method, job importance is determined by priority, due time, and tool requirements: high priority jobs with due times outside the scheduling horizon are of lowest importance, followed by jobs with due times within the scheduling horizon, in the following order (lowest to highest): low priority jobs without tool requirements, low priority jobs with tool requirements, high priority jobs without tool requirements, and high priority jobs with tool requirements. No other jobs are considered. Tool jobs are considered more important when scheduling a shift because they may be scheduled in fewer shifts than jobs without tool requirements.

As stated previously, our assumption is that tool sharing will be used for routine scheduling. We evaluate our capacity constraint propagation algorithm in Figure 3.12, which shows its performance versus a computationally intensive optimal method. Here, six shifts on one cell are scheduled with each shift being the deadline for an average of two jobs, while the mean job length is varied. This method provides a schedule close to optimal, far more quickly than the optimal method.

3.3.3 Handling "Rush" Jobs

In order to investigate how unexpected tooling requirements may be handled, we consider how to handle unexpected "rush" jobs, which arrive at a cell after the schedule has been constructed. Such a job has a due date sometime within the horizon of the schedule; thus, if this new job is to be accepted, either a new schedule must be constructed, or the new job must be fit into the initial schedule. As mentioned previously, making a new schedule from scratch is undesirable, as other important decisions may already have been made based upon the original schedule. Here we assume that the rush job can only be processed by the cell at which it arrives.

A rush job that requires use of a tool may be harder to fit into the schedule, because tool availability is limited. As in the initial scheduling method, the rush job is scheduled

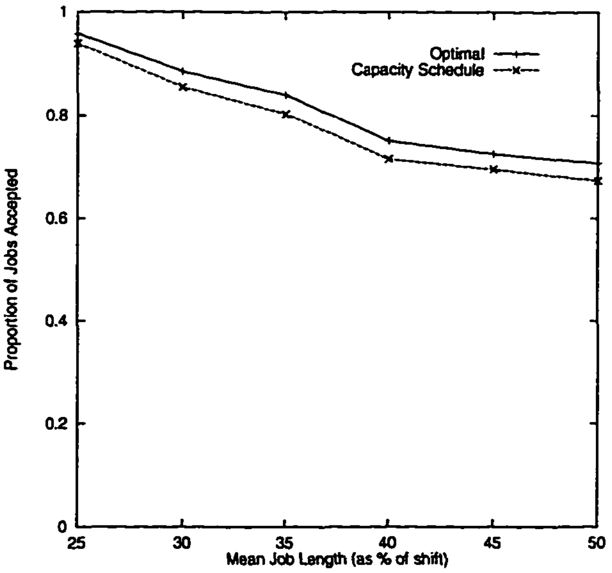


Figure 3.12: Capacity Constraint Scheduling Performance.

during the latest shift before its due date, as the scheduler prefers schedules in which jobs are scheduled close to their due dates. If there is no tool borrowing, the new job can only be scheduled during shifts before its due date and during which the tool is allocated to the cell. For handling the rush job locally (without tool borrowing), we consider three different non-disrupting methods, among which there is a trade-off between accepting the new job, and modifying the initial schedule as little as possible:

- LOCAL A, in which the job is scheduled during the latest possible shift in which there is enough idle time to accommodate the new job without rescheduling anything else. It is rejected if no such shift exists.

- LOCAL B, in which the job is scheduled during the latest possible shift in which there is enough idle time to accommodate the new job without considering the processing time requirements of low priority jobs. Low priority jobs may be rescheduled if necessary to make room for the new job. These are rescheduled using the same capacity constraint propagation that produced the original schedule, and may be removed if necessary. High priority jobs may not be moved.

- LOCAL C, in which the job is scheduled during the latest possible shift for which rescheduling does not result in any high priority jobs being removed. Rescheduling may move high priority jobs, and move and remove low priority jobs, but accepting the new job must not cause another high priority job to be rejected.

If tool borrowing is allowed, the cell may ask other cells if it may borrow the tool, if it determines that the job cannot be handled without borrowing the tool. The cell can determine for each shift whether the job can be scheduled in that shift, were the tool available, and then can ask to borrow the tool from the cell which has the tool during that shift. A request to borrow the tool should indicate how long the tool will be needed, including any tool transport time needed. For the cell from which the tool is requested, we consider three possible ways of handling these requests, that are similar to the above handle-local methods, and that have the same tradeoff:

- BORROW A, in which the tool is lent if the tool idle time during the shift is greater than or equal to the requested time. Thus, lending the tool during this time will not require any rescheduling at the lending cell.
- BORROW B, in which the tool is lent if the tool idle time, not counting process time requirements of low priority jobs, is sufficient for the requested time. Low priority jobs may be rescheduled as above (except that a multi-capacity knapsack packing algorithm is used, in which both tool capacity and shift capacity are considered). Low priority jobs may be moved and removed; high priority jobs may not be moved.
- BORROW C, in which the tool is lent if the cell can give up the tool for the requested time, and reschedule without having to remove any high priority jobs. Low priority jobs may be moved and removed; high priority jobs may be moved.

When tool borrowing is allowed, we will assume that similar local-handle and borrow methods will be used together; for example, when BORROW A is being used, a cell will ask to borrow the tool if LOCAL A is unsuccessful in scheduling the job without tool borrowing. These tool borrowing request handling methods are all guaranteed-safe methods, as the tool to be lent is only scheduled on the requested cell during the duration of the loan. Obviously more sophisticated rules would be needed for guaranteed-safe methods when jobs have precedence constraints.

A final option for handling a rush job is to reschedule all cells completely, using global knowledge of all job requirements, with tool migration allowed between two cells for one shift (as in the tool borrowing case). In this method, all scheduled jobs and unscheduled, the rush job is added to the job set, and the tool is reallocated. Then all possible tool migration between two cells sharing one shift evenly are explored, and the initial capacity constraint propagation method is used to assign jobs to shifts. This does not provide an optimal solution; any meaningful problem in this domain is too large for an optimal solution, but this method does use the initial scheduling method of assigning jobs to shifts, that was shown to perform well in the previous section. This GLOBAL method would not require cooperation per se, but it would require all cells to submit to rescheduling at the same time.

3.3.4 Simulations

We performed several simulation experiments to investigate the performance of these methods. For each simulation, a random job set was created using the following parameters: the mean number of jobs per shift per cell (λ), the probability that a job required use of the tool (p), the probability that a job was a priority job, and the mean job processing time. The actual number of jobs for each shift of each cell was a Poisson random variable, which determined the number of jobs for which that shift was the due date; this was not necessarily the number of jobs scheduled at that shift, as a job can be scheduled before its due date. Processing time for each job was a uniform random variable. Each simulation point on the graphs represents the results averaged from 100 job sets. Four separate rush job simulations were run for each job set. One more parameter for rush job simulations was the latency of the rush job, how many shifts in advance it was due.

The simulations were for a system of four cells with a scheduling horizon of eight shifts, and sharing the use of one tool. For each of these simulations, the probability that a job was a priority job was set at 0.5, and the mean job processing time was set at 0.25 × the length of a shift. The time required for tool transport from one cell to another in the middle of a shift was arbitrarily fixed at 0.05 × the length of a shift. If a shift is 8 hours, then this tool transport time is 24 minutes. Unless otherwise specified, each rush job had a latency of 3 shifts. Where not specified, $\lambda = 4$ and $p = 0.20$.

Results

Figures 3.13 and 3.14 show the performance of the initial scheduling algorithm for different values of λ and p. In Figure 3.13, λ is varied while $p = 0.20$. In Figure 3.14, p is varied while $\lambda = 4$. While we are not primarily concerned with evaluating this initial sched-

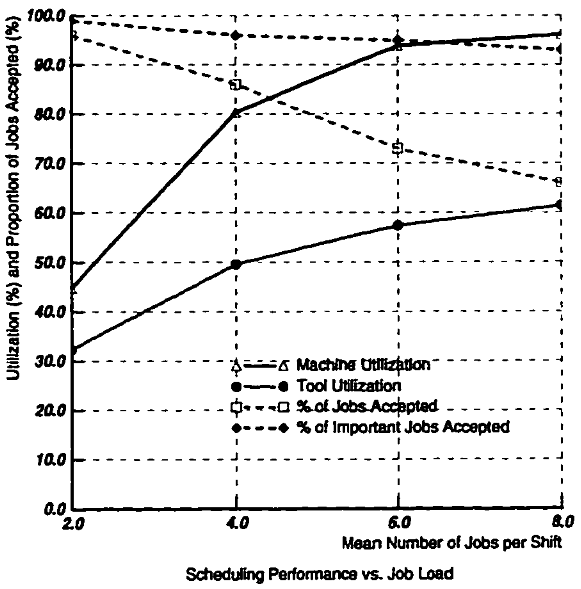


Figure 3.13: Scheduling Performance.

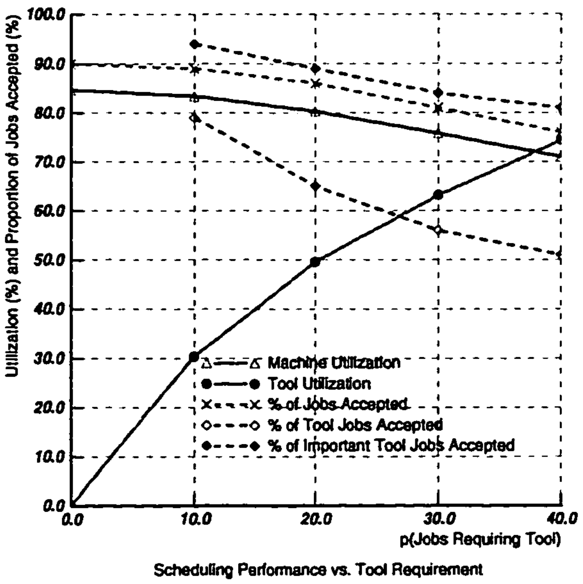


Figure 3.14: Scheduling Performance (cont'd).

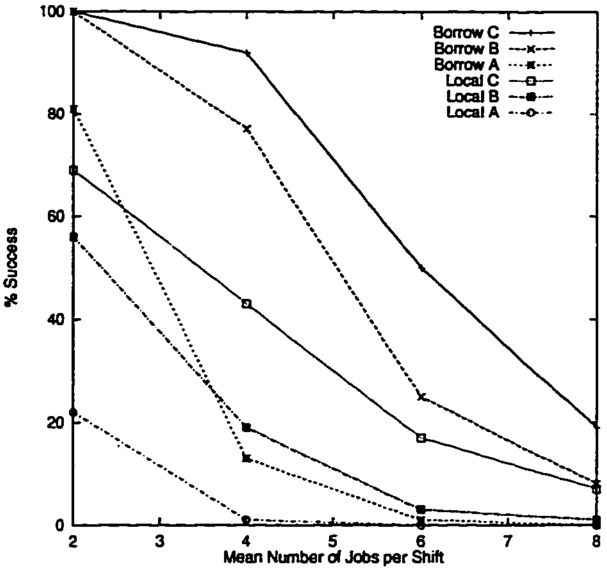


Figure 3.15: Rush Job Handling vs Load.

ule construction algorithm, these results should provide a context for the other simulation results, showing how busy a machine or a tool is under given conditions.

Figures 3.15, 3.16 and 3.17, show the performances of the various methods for handling rush jobs. The success axis indicates how often each method was able to schedule the rush job without having to remove any other important jobs. Figure 3.15 demonstrates that the requirement not to move any job is very constraining; both LOCAL A and BORROW A, which have this requirement, perform much worse than their less restrictive counterparts. It should also be noted that the flexibility of BORROW C, which is allowed to move important jobs, allows it greater advantage as the problem becomes more constrained.

Figures 3.16, 3.17 and 3.18 show the performance of the most effective BORROW and LOCAL methods versus the GLOBAL method which constructs a new schedule using global information. As the problem becomes more constrained, with more jobs per shift or more jobs requiring tool use, the tool borrowing method degrades relative to the GLOBAL method. Nevertheless, when the problem is constrained little enough to allow a good chance of success for the GLOBAL method, the BORROW method using only local information gathered through tool borrowing requests performs close to the GLOBAL method. Figure 3.18, showing rush job handling versus rush job latency, also indicates the advantage of tool borrowing as the problem is less constrained. In all cases, the BORROW

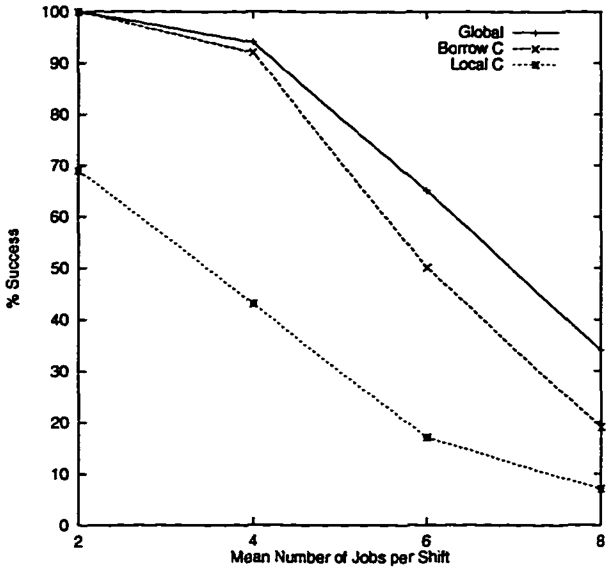


Figure 3.16: Rush Job Handling vs Load (cont'd).

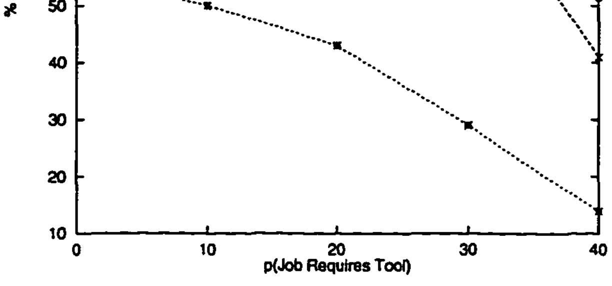


Figure 3.17: Rush Job Handling vs Tool Requirements.

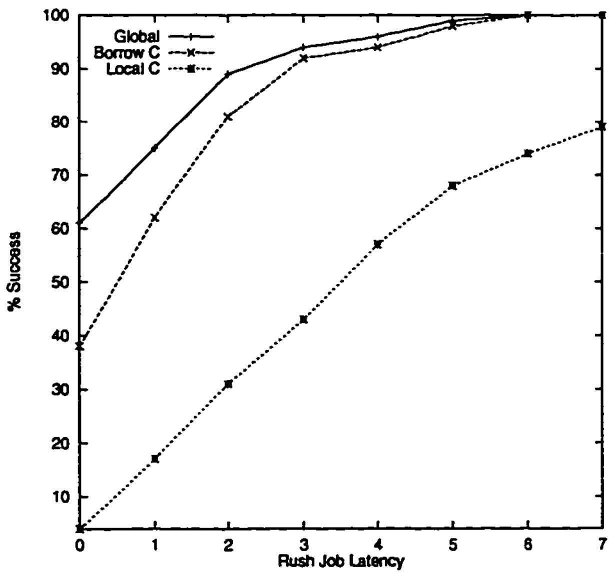


Figure 3.18: Rush Job Handling vs Rush Job Latency.

method greatly outperforms the LOCAL method.

As in the previous tool scheduling problem, two measures of the cost of handling an unexpected job are the number of jobs that have to be rescheduled in order to accommodate the new job, and the number of message exchanges that are required to borrow the tool. Figure 3.19 shows the number of jobs that are moved to different shifts when an urgent job is successfully scheduled. (The LOCAL A and BORROW A methods do not move any jobs.) These results show that, while the flexibility to move important jobs (as in LOCAL C and BORROW C) has a higher cost, the flexibility of tool borrowing does not seem to have a significantly higher cost in terms of rescheduling jobs. LOCAL C and BORROW C move roughly the same number of jobs per rush job acceptance. Thus, tool borrowing itself does not necessarily entail greater disruption, when measured by jobs moved, than local handling methods. Using the GLOBAL method, however, results in many more jobs being moved to different shifts, as might be expected, due to potentially different tool allocations.

Figure 3.20 shows the number of message exchanges required by the three tool borrowing algorithms per success, including those exchanges that do not result in successful scheduling of the rush job. The most successful algorithm, BORROW C, also requires the fewest message exchanges, as its flexibility allows both more successful local scheduling, and more successful negotiations. The two other negotiation algorithms require more communication

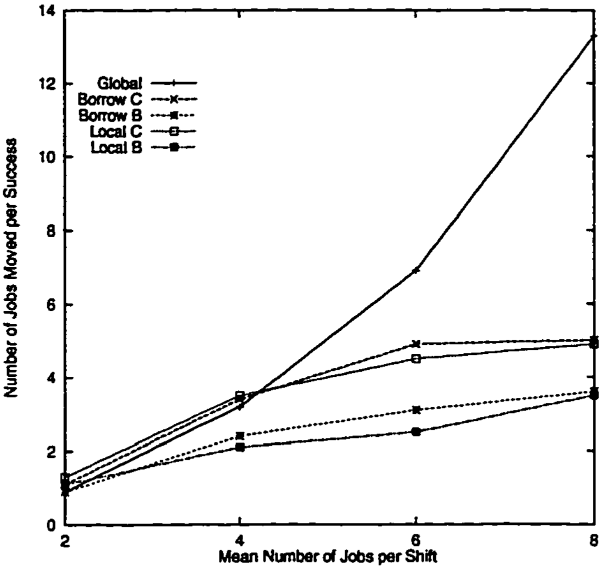


Figure 3.19: Jobs Moved to Handle Rush Job vs Load.

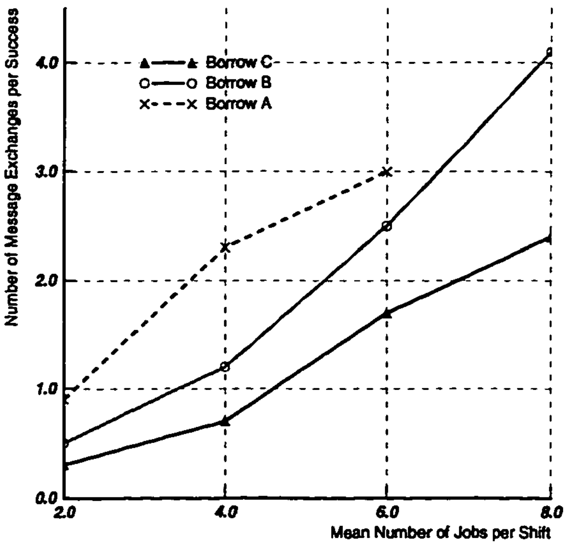


Figure 3.20: Number of Message Exchanges vs Job Load.

because negotiation with any given cell is less likely to be successful.

Thus, the tradeoff between the local handling and tool borrowing methods is between message exchange (which local methods do not require) and rush job acceptance, while the tradeoff between the tool borrowing methods and the method using global information is between disruptiveness and rush job acceptance. Where current workload is such that rush job acceptance is likely using the GLOBAL method, the most flexible borrowing method accepts the rush job almost as often as the GLOBAL method, moving fewer jobs from their scheduled shifts, and without need for global knowledge. This borrowing method also far outperforms non-borrowing local method, without moving more jobs than its corresponding most flexible non-borrowing local method. Thus, where collection of global information is not convenient, and especially where tool sharing (rather than tool migration) is the desired policy for routine tool scheduling, local rescheduling in response to unexpected tooling requirements is a good method, given the performance and cost measures.

3.4 Summary

In this chapter, we have proposed and evaluated tool borrowing protocols, that require an agent to reason about its priorities and capacities when considering a request from another agent to borrow a locally-reserved tool. We have shown that, for unexpected tooling requirements, polite scheduling and rescheduling, using only local information gained through negotiation with a small subset of agents, has performance close to good or even optimal methods using global information, in terms of accepting tasks to be scheduled. Such polite methods also incur smaller costs in terms of the rescheduling of already scheduled tasks. In addition, polite methods show a clear performance advantage over local methods that do not use negotiation. This is particularly true when the scheduling problem is not very constrained; polite methods performed much closer to global methods when only one tool was being shared, or when rush job acceptance was likely for global methods (though not for local methods).

These methods allow many disruptions to be handled locally by one agent, in communication with others, without requiring an appeal to a higher level authority, more wide-spread distributed problem solving techniques, or beginning the resource allocation problem from scratch. Thus, where collection of global information is not convenient, polite scheduling or rescheduling of tools and tasks is a good approach given the performance and cost measures discussed in this chapter.

We would like, as future work, to integrate the problem of reallocating tools and rescheduling jobs. As mentioned previously, when more than one tool is shared, sequencing of jobs within a shift must be considered when tools can migrate. Likewise, precedence constraints require more sophisticated management of job sequencing. Adding job constraints and allowing multiple tools will necessarily make the reallocation problem harder; however, we are encouraged by the results presented here that this approach to reallocation can be realistically useful.

CHAPTER 4

Rescheduling in a Decentralized Job Shop

This chapter focuses upon scheduling in a job shop in response to a disruption where the objective function is to minimize the makespan. Knowledge about the initial undisrupted schedule may be useful when fast rescheduling methods are required. We will show that, for local schedule revision, knowledge of local scheduling constraints allows scheduling performance almost as good as that of methods using global knowledge of the whole schedule, given the goal of minimizing the global makespan. We use a simple three cell manufacturing model for evaluating the utility of different levels of knowledge, and the utility of schedule constraint relaxation, when one cell is to be rescheduled in response to a break-down disruption. Then we examine a more general job shop rescheduling problem, and propose and evaluate methods for rescheduling one cell.

4.1 Initial Observations

The introduction of a new disruptive constraint into an already existing schedule may significantly alter the schedule. We focus upon the delay introduced by the break-down of a machine. Though repair times may be known from experience, if operation starting times are not altered, capacity constraints may be violated due to machine unavailability. If starting times of operations assigned to the broken-down machine are pushed back beyond the repair time, other temporal or precedence constraints may be violated, or additional costs may be incurred with respect to the measure of the schedule.

For common measures, there are upper bounds for the extra costs associated with such a delay of known duration. For example, if a delay of length d pushes back a machine's schedule of n operations, neither the total weighted flow time $\sum w_j F_j$ nor the total weighted

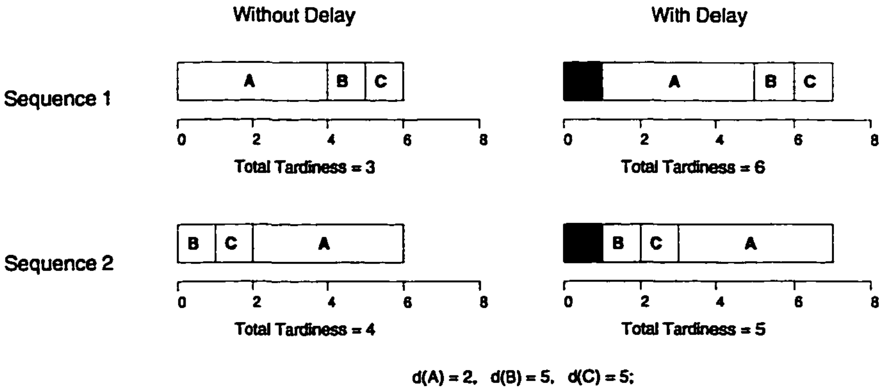


Figure 4.1: Delay and Total Tardiness.

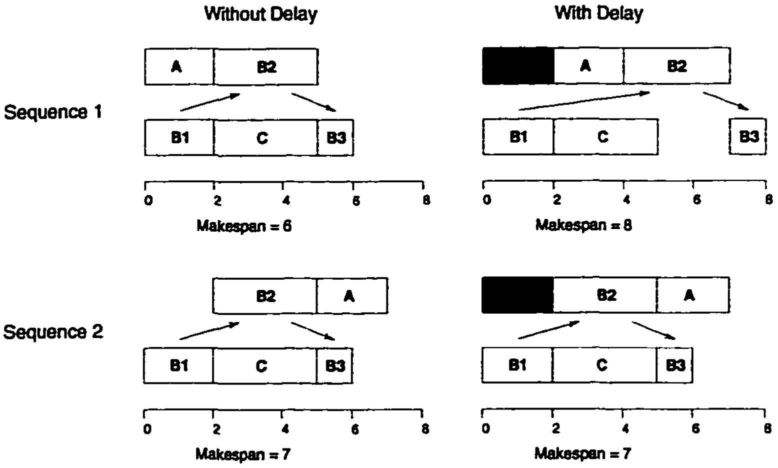


Figure 4.2: Delay and Makespan.

tardiness $\sum w_jT_j$ may increase by more than $\sum dw_j$, and neither the maximum lateness (Lmax) among the operations nor the makespan for the machine may increase by more than d. There is no upper bound other than n for the increase in the number of tardy jobs.

For many measures, there is no guarantee that an optimal schedule will remain optimal after the introduction of a delay. Figure 4.1, in which the measure is total tardiness, sequence 1 is optimal and better than sequence 2 without delay, while with even the smallest delay sequence 2 becomes optimal and sequence 1 suboptimal. Likewise, in Figure 4.2 where makespan is the measure, sequence 1 is optimal without delay, sequence 2 with delay. However, optimal single machine schedules for minimizing flow time or maximum lateness, when release times are identical, will always remain optimal despite a delay of any duration at the start of the schedule.

4.2 Job Shop Scheduling and Precedence Constraints

As mentioned previously, we can view a job shop as a distributed set of workcells. The distribution of operations among workcells is determined by the capabilities of the machines at those workcells. While in the tool sharing problem considered previously, workcells interacted via tooling resource requirements, workcells in a job shop also interact via the precedence constraints among the operations of a job. A precedence constraint imposes an ordering on two or more operations. For an operation j, we call an operation which must precede j a *predecessor* operation of j, and we call an operation which has j as a predecessor a *successor* operation for j; $\mathrm{prec}(i, j)$ indicates that operation i is a predecessor for operation j.

Through precedence constraints, schedules at different cells are linked. A cell which is to process a work-piece may have to wait until another cell has finished processing that work-piece. In this way, given a global schedule, the local schedules of the individual cells are linked together. A change in one cell's local schedule may affect the local schedules of other cells, by delaying the delivery of a work-piece from the cell, or by demanding the early delivery of a work-piece to the cell.

Because we are interested in how a cell responds to local schedule disruptions, we consider the local scheduling problem seen at one cell, and how schedule constraints may be expressed. Using the DCSP terminology, the global schedule S for the job shop provides the environment E_g^S for the scheduling problem at the local cell g. Precedence constraints may link the schedule at one cell indirectly to every other job in the system. Thus, complete knowledge about how local schedule changes affect the global schedule may require global knowledge about the schedule. Here, we will show how merely local schedule constraints may be expressed. In the rest of the chapter, we will show how the knowledge of local constraints is almost as good as having global knowledge.

4.2.1 Precedence Ready Times and Deadlines

The environment E_g^S imposes various schedule constraints on top of the initial capacity and temporal constraints that define the original problem. Some of these schedule constraints may be represented as *precedence ready times* and *precedence deadlines.* We define the *precedence ready time* ρ_j of an operation j in a job shop schedule as the time before which the operation cannot start due to the current scheduling of predecessor jobs. Sim-

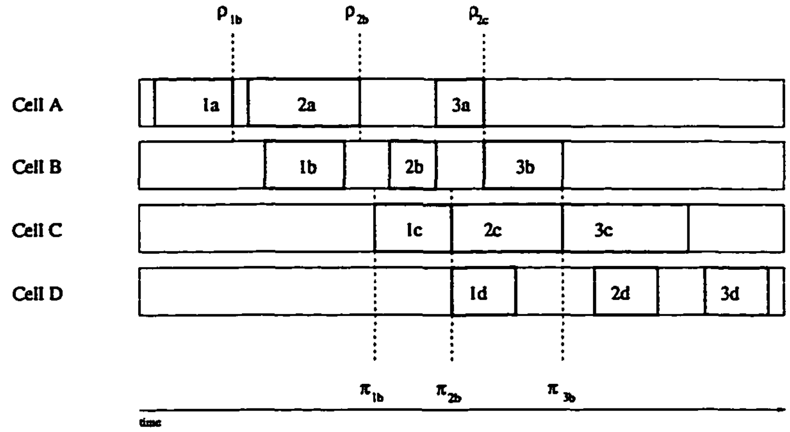


Figure 4.3: Precedence Ready Times and Deadlines for Cell B.

ilarly, the *precedence deadline* (π_j) of an operation j in a job shop schedule is the time by which the operation must finish processing so that any successor operation may begin processing according to the schedule. We will refer to precedence ready times and deadlines as *precedence constraint times.* [1] If the transport of work-pieces from machine to machine is disregarded, $\rho_j = \max C_i : \text{prec}(i,j)$, and $\pi_j = \min s_i : \text{prec}(j,i)$. It is clear that, when rescheduling local operations due when other cells already have schedules, if every operation's new starting time s' is such that $\rho_j \leq s'_j \leq \pi_j - p_j$, then the resulting local schedule is a non-disrupting rescheduling action; neither will an operation begin before its predecessors finish, nor will it end before its successors begin. Figure 4.3 shows the precedence constraint times for cell B, where jobs move through cells A, B, C, and D in order.

Related Concepts

Ideas similar to precedence constraint times have been used in methods for constructing initial schedules, in which constraints imposed by a partial schedule upon unscheduled operations are used to extend the partial schedule. For example, precedence constraints may be used to define due dates for individual operations, given the due date of the entire job. *Backwards scheduling* in the widely-used MRP (Materials Requirement Planning) system defines the due date of an unscheduled operation as the starting time of that operation's already-scheduled successor. Likewise, in the shifting bottleneck procedure of Adams et

[1] We assume that material transport is not a concern unless otherwise noted. When transport is a concern, then we can conceive of *transport ready times* and *transport deadlines*, which indicate when the relevant materials awaiting processing are scheduled to arrive at the local cell, and when the processed result is scheduled to be transported away from the local cell. Given the precedence ready times and due times, these transport constraint times may be relaxed if the material transport system can deliver earlier or later than planned. If the time to transport materials is itself non-negligible, precedence constraint times can be adjusted to reflect the time required for transport.

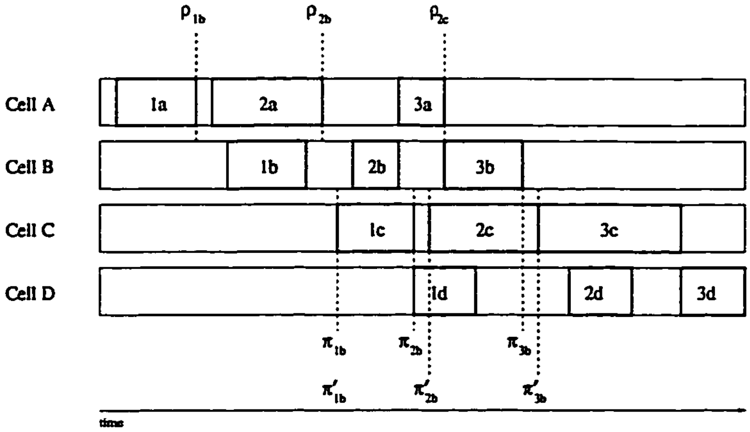


Figure 4.4: Relaxed Precedence Deadlines for Cell B.

al.[1] for minimizing the makespan in a job shop, an individual machine is scheduled by assigning due dates to its operations based upon precedence constraints and the makespan for the already scheduled machines. In Liu and Sycara's work on distributed scheduling by a society of agents [42], an agent scheduling a job accounts for its precedence constraints by an *exclusion-off constraint bunch*, which is similar to precedence constraint times.

4.2.2 Relaxed Precedence Deadlines

These precedence constraint times may sometimes be too constraining for rescheduling. For example, in Figure 4.3, operation 3b cannot be moved at all in the schedule, given that $\rho_{3b} = s_{3b} = \pi_{3b} - p_{3b}$. While the precedence constraint times at a cell represent the constraints imposed upon that cell by a global schedule, these constraints may be relaxed if parts of the global schedule are modified. If predecessor jobs at other cells are moved earlier, or successor jobs moved later, then precedence ready times and deadlines may be relaxed. However, scheduling a job j using the rule $\rho_j \leq s'_j \leq \pi_j - p_j$ may prevent another cell from rescheduling successfully in the same way. For example, if j with lone predecessor i is rescheduled so that $s'_j < s_j$, then i's precedence deadline will be moved forward to s'_j. Thus rescheduling using this rule may not be a guaranteed-safe action. [2] For polite rescheduling, the initially disrupted cell may use this rule to find a non-disrupting action, but if it cannot, the other cells it negotiates with may have to use a more restrictive rule in order to find a solution that isolates the disruption.

[2] If the system is a *flowshop*, then this rule may allow guaranteed-safe rescheduling in particular cases. In a flowshop, jobs are processed by a group of machines, each job visiting every machine in the same fixed order (i.e., machine 1, machine 2, etc.). If each cell is one stage of the flowshop, then the structure of the system allows this rule to be used for the cell immediately preceding the disrupted cell, and the cell immediately succeeding the disrupted cell. It does not allow guaranteed-safe rescheduling of two adjacent cells, however.

If the disrupted cell needs remote cells to reschedule in order to relax its own local schedule constraints, restricting remote cells to revisions in which each operation j at a remote cell is scheduled such that $s_j \leq s'_j \leq \pi_j - p_j$ allows rescheduling to be a guaranteed-safe action. A similar rule restricting s'_j to $\rho_j \leq s'_j \leq s_j$ also allows rescheduling to be guaranteed-safe. In this chapter we will restrict ourselves to the first rule, though the second is equally valid, and though one can easily imagine scenarios in which some cells would use one rule, and some the other, in order to reschedule. Using this first rule may result in each operation at the disrupted cell having a new precedence deadline $\pi'_j \geq \pi_j$; later deadlines usually make a problem easier, so altering the schedule environment in this way will relax the local constraints seen at the cell. We will call any precedence deadline that has been moved into the future a *relaxed precedence deadline.* Figure 4.4 shows the schedule from Figure 4.3, and the relaxed precedence deadlines for the operations at cell B, given the schedule change at cell C. The original precedence deadline for operation 1b is π_{1b}, and the relaxed precedence deadline is π'_{1b}.

In this example, while we have tried to push back the operations at cell C to their own precedence deadlines (imposed by the schedule at cell D), note that the relaxed precedence deadlines at cell B are not determined merely by cell C's precedence deadlines and operation processing times. Capacity constraints at cell C are also involved; for cell C, operations 2c and 3c cannot both be pushed back to their precedence deadlines, as the machine at cell C can only process one job at a time. Thus, the determination of relaxed precedence deadlines may require information about the current schedule, as well as precedence constraints and ultimate due dates. The relaxed precedence deadline may be the same as the unrelaxed, or simple, precedence deadline, as is the case with 1*b* in the example. Note also that a different modification of the schedule at cell C may result in a different set of relaxed precedence deadlines for cell B.

4.3 Using Precedence Deadlines

Precedence deadlines at a cell describe in part how modifying the schedule at that cell will affect the schedule for the rest of the job shop. We have described rules for non-disrupting and guaranteed-safe rescheduling actions which use these times. Even when a disruption cannot be isolated, however, these precedence constraint times may be useful. For example, the extent by which a precedence deadline is missed may be an estimate of how much another cell will be disrupted. Likewise, the extent by which a relaxed precedence deadline is missed may represent how much of the disruption is being propagated. In this section, we will show that different levels of knowledge about precedence constraint times produce significant differences in local rescheduling performance.

In order to investigate how useful precedence deadlines may be, we examine rescheduling due to disruption in a simple three cell flowshop-like system. In the system considered here, a job does not necessarily visit every cell, but the cells it does visit must be visited in order, with no skipping. Thus there are four types of jobs, jobs processed at cell 1, then cell 2, then cell 3; jobs processed at cell 1, then cell 2; jobs processed at cell 2, then cell 3; and jobs processed at only one cell (1, 2, or 3). Cells may have one or more identical machines working in parallel. Jobs have identical release times.

For this experiment, the measure of quality for schedules will be the makespan of the third cell. [3] The job sets are generated using the parameter p, which is the mean proportion of operations at cells 1 and 2 which have successors. The mean proportion of operations at cell 1 which have ultimate successors at cell 3 is thus p^2, which we call the *schedule tightness.* An initial schedule is constructed from a job set using a method to minimize total makespan (which effectively minimizes the makespan for cell 3). Cell 1 is then disrupted by having one of its machines break down for a set duration at the beginning of the schedule.

The problem stated formally is as follows. Given a set of cells, each having an identical number of identical machines, a set of jobs with processing times and each belonging to one particular cell, precedence constraints of the type described previously, a complete feasible schedule assigning jobs to starting times and machines, and a disruption of known duration of one machine in cell 1 at the beginning of the schedule, the problem is to reschedule the jobs of cell 1 so that none are scheduled on the disrupted machine during the duration of the disruption, with the objective of minimizing the makespan at cell 3 once the disruption has been propagated as far as necessary. The disruption is propagated when cells 2 and 3 have to push back their schedules due to delays in the processing of predecessor jobs. Given this problem, we consider four levels of knowledge at a cell about how local scheduling decisions affect the global objective:

- At the *Utterly Ignorant* level, the cell is unaware even of which operations have suc-

[3] We can imagine that jobs finishing at cells 1 and 2 are for inventory, while those proceeding to cell 3 are for delivery.

cessors. Thus, the cell reschedules using as due dates the operation completion times from the initial schedule.

- At the *Locally Informed* level, the cell is aware of which operations have successors. For those operations, initial schedule completion times are used as due dates, while successorless jobs are given no due dates (i.e.,their due dates are ∞).
- At the *Precedence Deadline* level, the cell uses each operation's precedence deadline as its due date, successorless operations having no due dates.
- At the *Relaxed Precedence Deadline* level, the cell uses relaxed precedence deadlines as due dates. The relaxed deadlines are obtained by pushing the operations at cell 2 back as far as possible without pushing any beyond its precedence deadline.

At each level, the cell will use the due dates in scheduling to minimize the maximum lateness, the assumption being that a late job will cause a delay at the next cell, and the later the job, the greater the delay. These due dates implicitly assign priorities to operations at cell 1. While the Utterly Ignorant level assigns roughly the same priorities to every operation, the Locally Informed and Precedence Deadline levels assign higher priority to operations with successors at cell 2, and the Relaxed Precedence Deadline level assigns highest priority to those operations that either ultimately have successors at cell 3, or precede such operations.

Different levels of knowledge, however, imply different amounts of effort, particularly regarding how this knowledge is to be obtained. The Push Back and Utterly Ignorant methods require no knowledge that is not always already available at the cell. The Locally Informed method requires knowledge about which operations have successors; such knowledge should be available when the operation is scheduled, but may become out-dated if successor operations can be added to the schedule after initially successorless operations have been scheduled. Knowledge of precedence deadlines requires some knowledge of the global schedule, that may or may not be available to the cell when the operation is scheduled. This knowledge may also become outdated if the schedules at other cells change. The relaxed precedence deadline method requires more specific knowledge, about how successor cells may be rescheduled. While the Locally Informed and Precedence Deadline methods may require some communication to keep knowledge up-to-date, the relaxed precedence

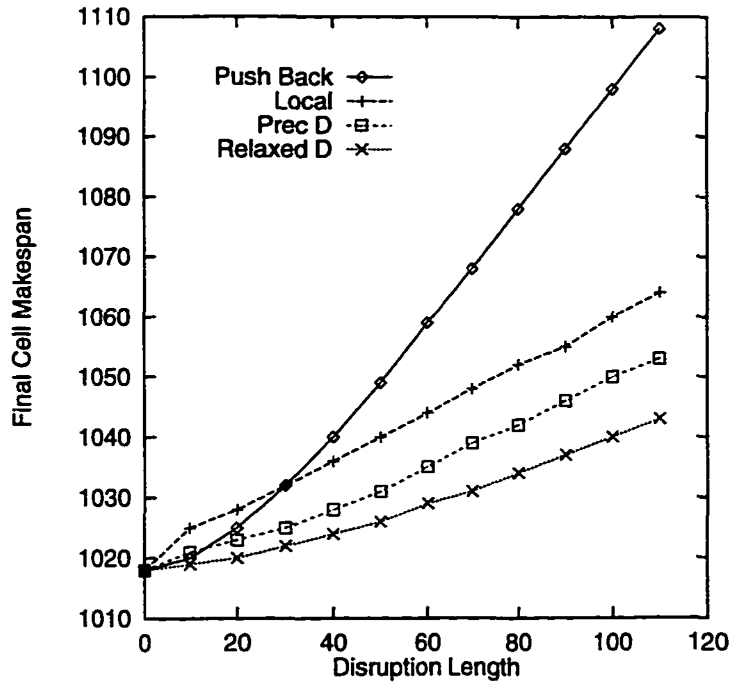


Figure 4.5: Results for $p^2 = 0.8$, two machines per cell.

deadline method requires closer communication and coordination with other cells. Thus maintaining more useful knowledge may also have greater costs.

4.3.1 Results

Figures 4.5 through 4.7 show results for several rescheduling methods using these different levels of knowledge. In each graph, each point shows the averaged results for 100 different job sets, in which each cell has twenty operations per machine, and the average processing time for an operation is 50. The methods compared include that using locally-informed knowledge (Local), that using precedence deadlines (Prec D), that using relaxed precedence deadlines (Relaxed D), and a method simply pushing back the schedule (Push Back). The method using the Utterly Ignorant level of knowledge was found to perform exactly as well as the Locally Informed method, so results of that method are not shown; the initial schedule was almost always constructed with successorless operations at cell 1 scheduled last anyway, so the knowledge of which operations have successors was not useful.

Figure 4.5 shows results for job sets with schedule tightness $p^2 = 0.8$, each cell having two machines. For very short disruptions, push back performs better than the Locally Informed method; it may sometimes be better to ignore small disruptions than to respond to them without the proper information. While the precedence deadline method always

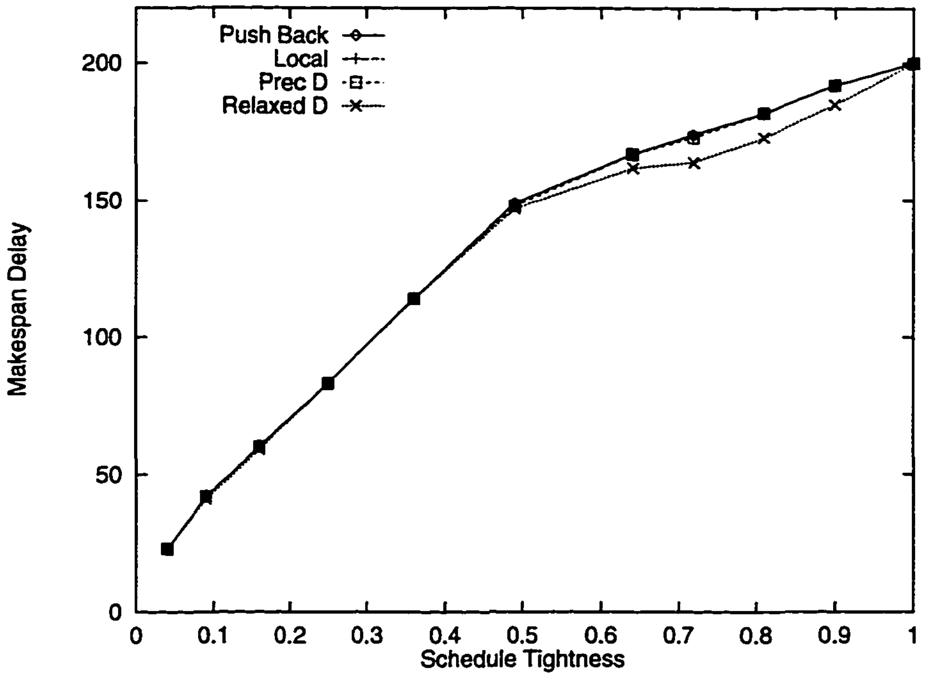


Figure 4.6: Results with disruption length = 200, one machine per cell.

outperforms the Locally Informed method, it provides no additional improvement when the disruption length increases past a certain point (here about 40). Likewise, the relaxed precedence deadline method provides no additional improvement after a further point (here about 100). This may be explained as follows. Using precedence deadlines provide the cell 1 schedule with extra room, or slack, in which to reschedule. After a certain point, this slack has been used up, so no additional improvement can be obtained without further relaxing the problem.

Figures 4.6 and 4.7 show the same methods compared over different job sets. Each point represents 100 job sets; here the disruption length is held at 200, and a different set of job sets is used for each schedule tightness value. Figure 4.6 shows results when each cell has only one machine. Only the relaxed precedence deadline method provides any improvement over simply pushing the schedule back. Only when cell 2 is also rescheduled can any resequencing of operations at cell 1 be useful. However, in Figure 4.7, in which each cell has two machines, the precedence deadline method provides significant improvement over the Locally Informed method, and the relaxed precedence deadline method produces further improvement. Both figures illustrate the obvious point that the disruption delay is harder to contain as the schedule tightness increases.

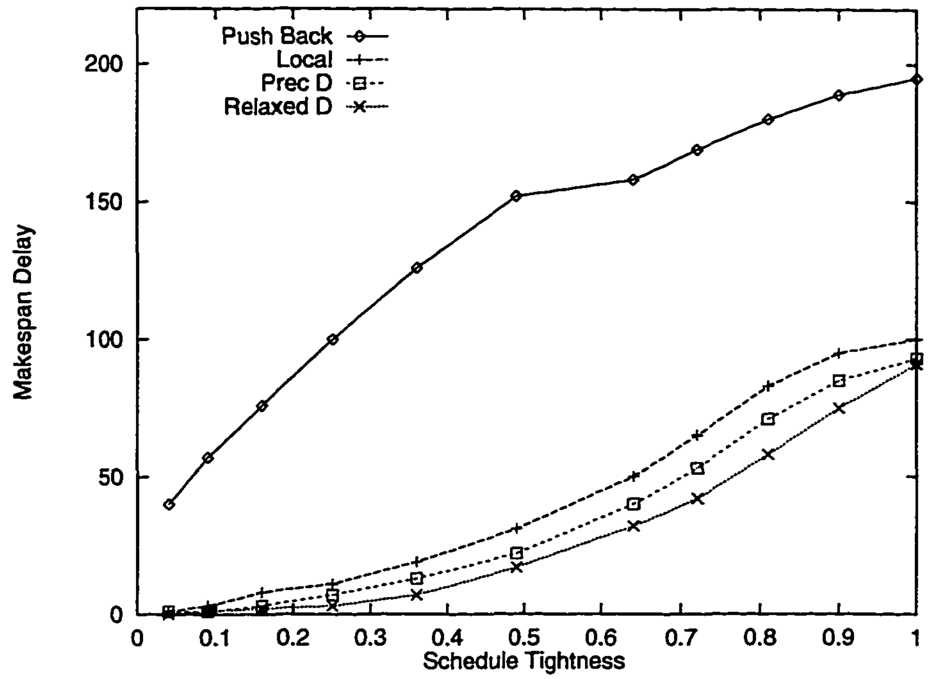


Figure 4.7: Results with disruption length = 200, two machines per cell.

Bottleneck Cells

In the job sets considered so far, the operations at each cell have the same mean processing time. In practice, however, different types of operations may require different ranges of processing times. Often in a manufacturing system, there is one stage which is the bottleneck, at which operations take much greater time to execute. Scheduling at such stages is most critical for scheduling the whole system. Thus, the existence of a bottleneck may affect efforts to reschedule. The rescheduling problem for a bottleneck cell, for example, should usually be much more constrained than that for a non-bottleneck cell. Likewise, a non-bottleneck cell should expect little help from a bottleneck cell when the non-bottleneck cell is trying to relax its schedule constraints.

Figures 4.8 through 4.10 show results for the same three cell system as in Figure 4.5, with two machines per cell and $p^2 = 0.8$, but in each of these figures, one cell has become the bottleneck. While each cell still has twenty operations per machine, the bottleneck cell's operations have mean processing time 75, while those for other cells have mean processing time 50. Figure 4.8 shows the performance of the rescheduling methods when cell A (the disrupted cell) is the bottleneck. Here, the Locally Informed method performs exactly as well as the precedence deadline method. Because cell A is the bottleneck, cell B's machines

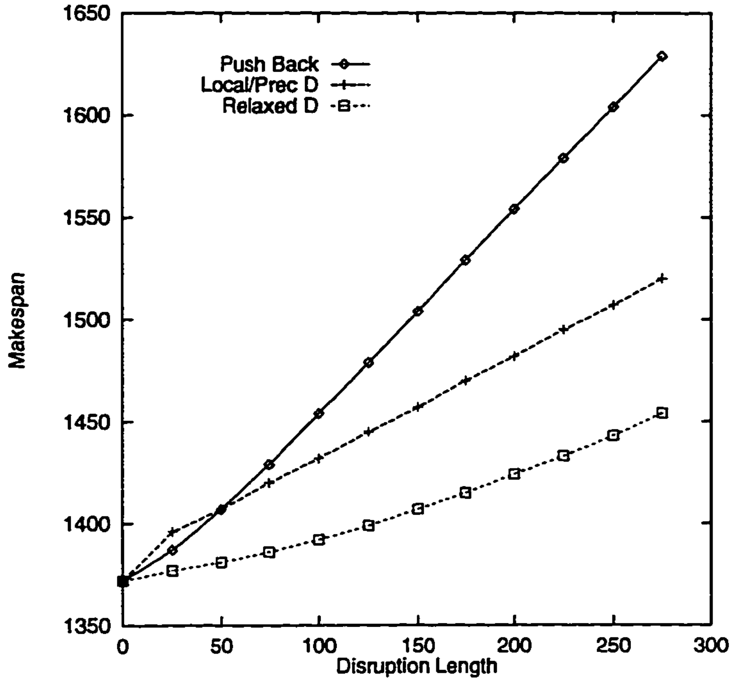


Figure 4.8: Performance when Cell 1 is the Bottleneck.

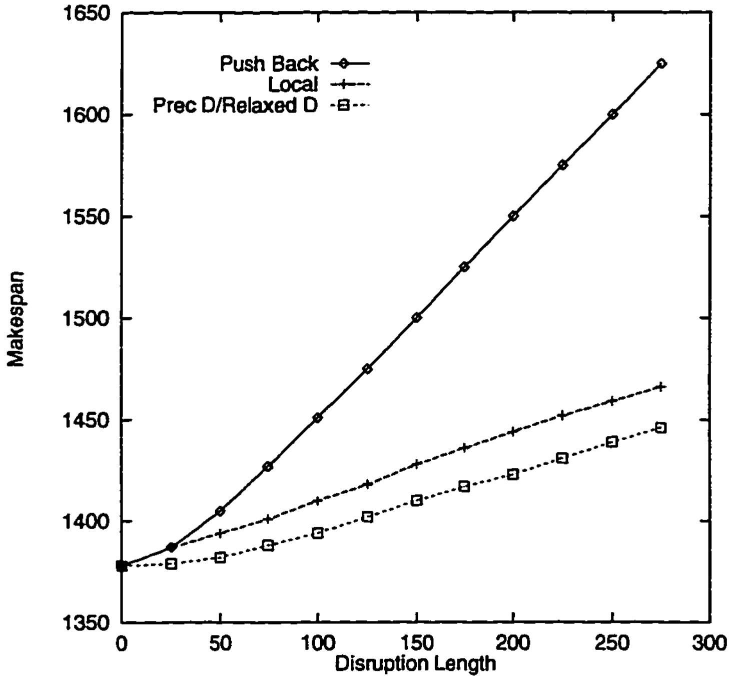


Figure 4.9: Performance when Cell 2 is the Bottleneck.

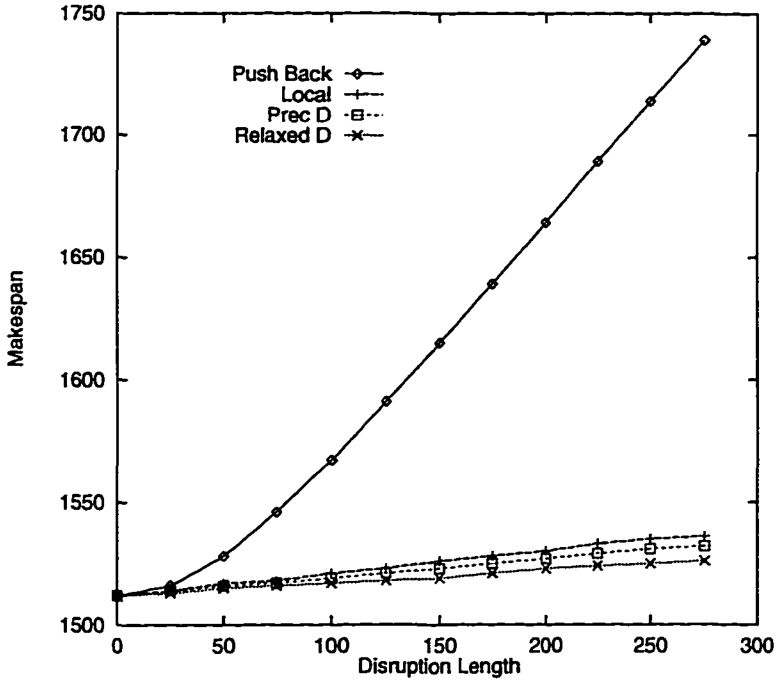


Figure 4.10: Performance when Cell 3 is the Bottleneck.

are often idle, waiting for deliveries from cell A. Precedence deadlines for an operation at cell A therefore will usually be its actual completion time; knowledge of precedence deadlines thus does not improve upon Locally Informed knowledge. In this case, the improvement provided by the Locally Informed method over the simple push back method is also less than in the non-bottleneck case of Figure 4.5. However, knowledge about cell B's schedule, in the form of relaxed precedence deadlines, is very useful.

Figure 4.9 shows results when cell 2 is the bottleneck. In this case, the relaxed precedence deadline method provides no improvement over the precedence deadline method. As in the previous case, the machines at the bottleneck cell's successor usually must wait for deliveries. Thus, there is no relaxation of cell 1 precedence deadlines. The precedence deadline method, however, provides a greater advantage over the Locally Informed method than in the non-bottleneck case. Because cell B's operation are longer, the precedence deadlines for cell A operations are likely to be further from their completion times than in the non-bottleneck case. Thus, knowledge of these precedence deadlines is likely to be more useful.

Finally, Figure 4.10 shows results when cell 3 is the bottleneck. While knowledge of precedence deadlines and rescheduling cell 2 to relax precedence deadlines is still relatively useful, all these rescheduling methods are effective for containing the cell 1 delays from cell 3. Cell 3 is so busy that delays of deliveries from cell 2 are not likely to delay cell 3's schedule by much.

Discussion

In this section, we have shown how different levels of knowledge of local schedule constraints allow significantly different performance for local rescheduling to minimize global makespan. Precedence constraint times require some knowledge of local schedule constraints, but not global knowledge about the whole schedule. The results show that knowledge about precedence constraint times and relaxed precedence constraint times allows the local cell to reschedule its operations more effectively. These results also show how heterogeneity among cells may affect efforts to reschedule, depending upon which cells are bottlenecks. The next section will compare the usefulness of this local knowledge against that of global knowledge.

4.4 Rescheduling in a General Job Shop

In this section, we examine rescheduling in a more general job shop. We investigate how local knowledge of schedule constraints, in the form of precedence constraint times, can assist the local rescheduling of a cell in response to a disruption in the form of a machine breakdown. We will show that use of precedence constraint times, under certain circumstances, allows performance close to that of optimal local rescheduling using global knowledge of the whole schedule. These results suggest how and for which kinds of systems polite rescheduling will be useful.

A common measure for scheduling in a job shop is the makespan for the entire shop. While this may not seem like a very realistically useful measure, as the shop will never actually complete all of its work as new work constantly arrives, minimizing the makespan for available jobs is a good way of maximizing utilization of the available machines. We have suggested earlier that a good initial solution to a problem is often useful when finding a solution when the problem has been modified. For the job shop makespan problem ($Jm//Cmax$), one good method for an initial solution is the "shifting bottleneck procedure" of Adams et al. [1]. Good dispatching rules include the *largest processing time first* (LR) rule, which chooses that operation whose processing time summed with the processing times of all its ultimate successors is the greatest of those operations available for scheduling. This rule is generally the best among general dispatching rules for minimizing job shop makespan.

4.4.1 Rescheduling the Entire Job Shop

While our research has focused upon local rescheduling at one cell, in this first subsection we briefly examine ways of rescheduling the entire job shop in response to disruption. While this examination provides some insight into how to reschedule locally, and some perspective for the local rescheduling results later in this chapter, it also shows how cells may reschedule in the absence of communication and knowledge about the schedule.

The problem of rescheduling the entire job shop may be described formally as follows. Given a set of cells, with an identical number of identical machines, a set of jobs, each consisting of an ordered set of operations, a complete feasible schedule assigning operations to machines and starting times, and a disruption of known duration at one machine of one cell, the problem is to find a new schedule for the whole shop in which no operation is scheduled on the disrupted machine during the duration of the disruption, and in which none of the original constraints are violated, with the objective of minimizing the overall makespan. While obvious approaches include any method by which an initial schedule is constructed, such as the shifting bottleneck method of the LR rule, we would like to investigate the performance of fast heuristics that use information from the initial undisrupted schedule. Such information consists mainly of how the operations have been sequenced in the original schedule.

Figure 4.11 shows various approaches to rescheduling a job shop due to break-down delay at one machine, when the initial schedule has been constructed by the shifting bottleneck procedure. Here, each point represents averaged results for three different one-machine disruptions for each of ten job sets; each job set includes twenty jobs, each with one operation for each cell. Operations have mean processing time 50. There are ten cells, and here each cell has one machine. The figure compares four methods for rescheduling in response to the disruption: the same shifting bottleneck procedure that was used to construct the initial schedule (Shifting Bottleneck), simply pushing back the schedule without resequencing operations (Push Back), using the operation starting times from the initial schedule as the basis for dispatching at every cell (Original Start Dispatch), and using the LR dispatching rule at every cell (LR Dispatch). While the shifting bottleneck procedure provides the best result, it requires an order of magnitude more computing time than the others, and it requires a central scheduler with global knowledge. The other methods do not require a

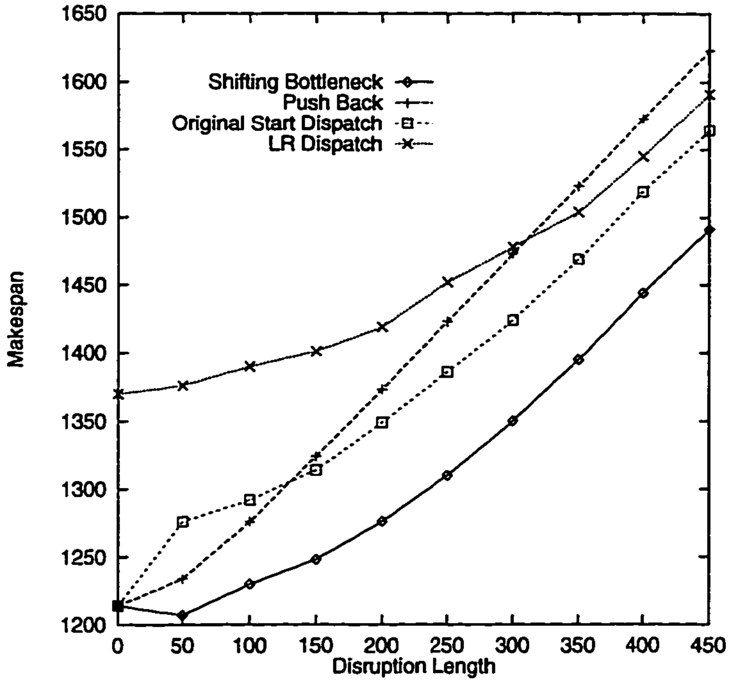


Figure 4.11: General Job Shop Rescheduling Performance.

central scheduler nor any knowledge about the rest of the schedule. Thus, they require no communication, as rescheduling is based upon scheduling information already available at the cell. Among the other methods, merely pushing back the schedule performs better than both dispatching methods for small disruptions, and outperforms the LR dispatching rule for even longer disruptions. The original start dispatching method, which uses values derived from the initial schedule generally performs the best among these three fast methods, though as the disruption length increases, it approaches the LR rule performance.

The greater performance of the original start rule, and of the push back method for short disruptions, over the LR dispatching method demonstrates the utility of using the original schedule as a basis for rescheduling. While using the initial method provides better performance, when only a quick method is an option, methods using information about the good initial schedule provide better results than the best general dispatching method. Figure 4.12 shows these same results with the average machine utilization measure. Machine utilization is calculated as the total processing time at a machine divided by the individual machine makespan.

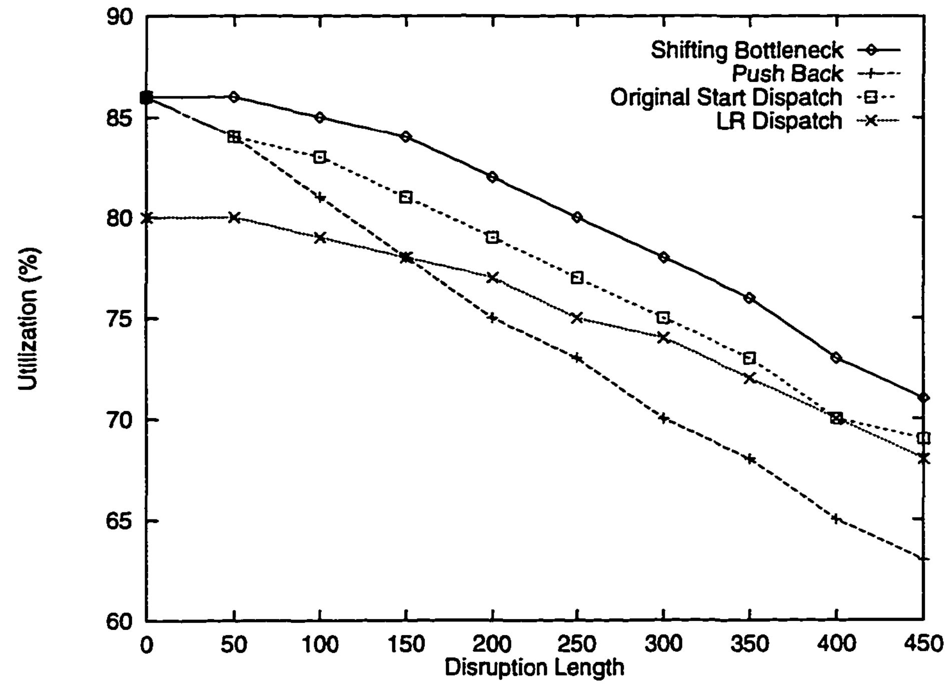


Figure 4.12: General Job Shop Rescheduling Performance cont'd.

4.4.2 Rescheduling at the Disrupted Cell

In the previous figures, the whole job shop is rescheduled in response to a disruption. However, we are chiefly interested in how one cell may be rescheduled in order to minimize the effects of a disruption on the entire system. As suggested previously there may be reasons not to reschedule operations at other cells, that aren't taken into account by the makespan objective. In the case in which there was only one machine per cell, as in the previously examined three cell system when each cell has one machine, constraints are very inflexible, so that little gain can be achieved over simply pushing back. However, when there is more than one machine per cell, there is sufficient flexibility that rescheduling a disrupted cell may have some benefit.

This rescheduling problem may be formally described as follows. Given a set of cells, with an identical number of identical machines, a set of jobs, each consisting of an ordered set of operations, a complete feasible schedule assigning operations to machines and starting times, and a disruption of known duration at one machine of one cell, the problem is to reschedule the disrupted cell so that no operation is scheduled on the disrupted machine during the duration of the disruption, with the objective of minimizing the overall makespan after the disruption has been propagated as far as necessary. The disruption is propagated,

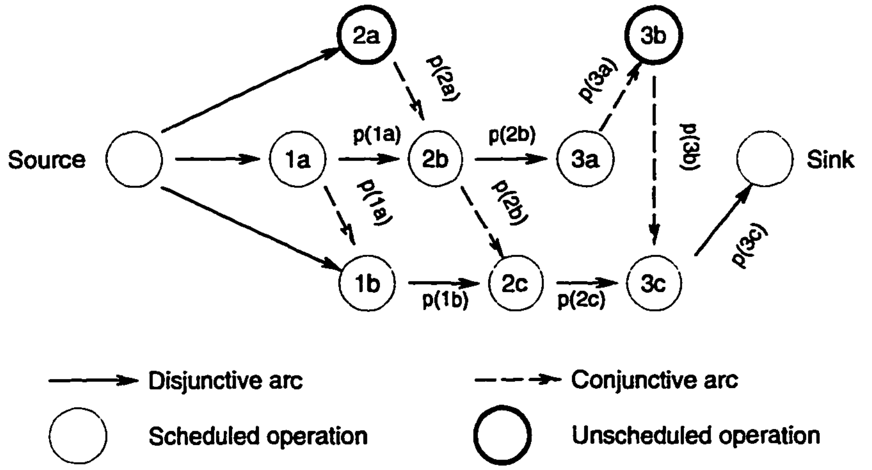


Figure 4.13: Disjunctive graph for scheduling.

as in the previous three-cell experiment, by pushing back the schedules of other cells due to delays in predecessor operations. We would like to investigate the performance of fast local rescheduling methods that use information from the original undisrupted schedule.

Local Rescheduling with Global Knowledge

An optimal local rescheduling method for minimizing global makespan using global knowledge is suggested by the shifting bottleneck procedure. As mentioned previously,the shifting bottleneck procedure uses the partially constructed schedule to derive release times and due dates for operations to be scheduled. This is done by representing the partial schedule as a *disjunctive graph*, as in Figure 4.13, similar to a PERT graph. Each operation is represented by a node. Conjunctive arcs indicate precedence constraints, while disjunctive arcs indicate previous scheduling decisions. A disjunctive arc goes from a scheduled operation to the operation scheduled next on the same machine; thus, in the figure, operations 1a, 2b, and 3a are already scheduled in that order on one machine, while operations 1b, 2c, and 3c are scheduled in that order on another machine. The length of each arc is the processing time of the node it leaves. The longest path L(source,sink) of this directed acyclic graph is a lower bound for the total makespan, given the partial schedule. An unscheduled operation a, given the partial schedule, cannot be scheduled to begin before L(source,a), and if it is scheduled to begin after (L(source,sink)− L(a,sink)), the resulting partial schedule will necessarily have a larger makespan lower bound. Thus, the shifting bottleneck procedure assigns operation a a release time L(source, a), and a due date (L(source,sink) − L(a,sink) - p(a)). All operations for an unscheduled machine are then scheduled using these release

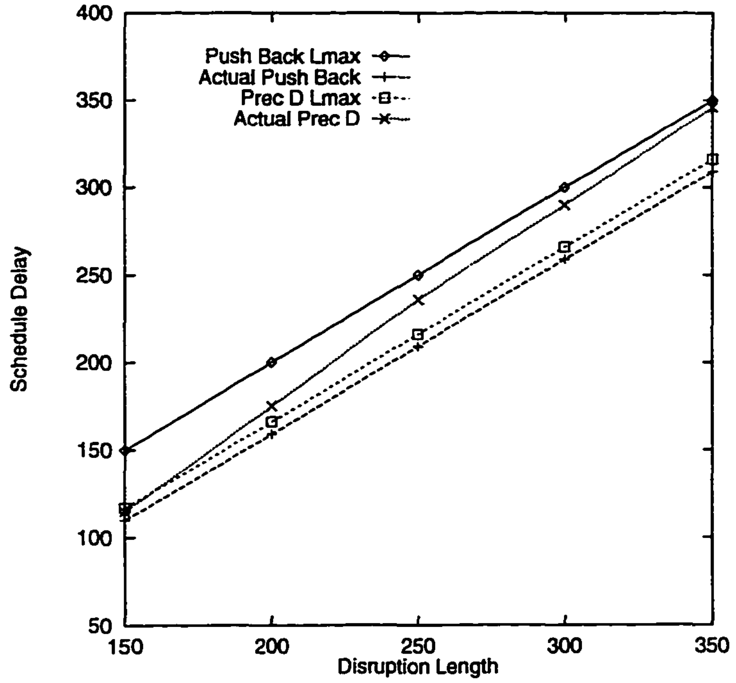


Figure 4.14: Precedence Deadline Problems.

times and due dates.

Rescheduling a disrupted machine is somewhat like scheduling an unscheduled machine to extend a partial schedule. The partial schedule in the rescheduling case is the original schedule at the non-disrupted cells. The release times and due dates thus derived using global knowledge of the schedule will provide an optimal solution for local rescheduling to minimize the global makespan when maximum lateness at the local cell is minimized. Thus, this method provides a benchmark against which to compare methods that do not use global knowledge. [4] In the following figures, this method, which uses PERT-related characteristics, is labeled PERT-Times.

Precedence Deadlines

Precedence constraint times can also be used to reschedule a disrupted cell. Precedence ready times and deadlines are used to minimize maximum lateness. However, in a general job shop, precedence constraint times are not always informative or useful. In an experiment for a generic job shop, the method using precedence constraint times as due times and

[4] If the initial schedule has been constructed using the shifting bottleneck procedure, then these release times and due dates should be available for every operation, because the last stage of the procedure, after a complete schedule has been constructed, is to de-schedule each cell by itself, and reschedule it after those times have been calculated. Thus, using these PERT-related characteristics of the initial schedule may be a way of using the initial schedule when a cell needs to be rescheduled.

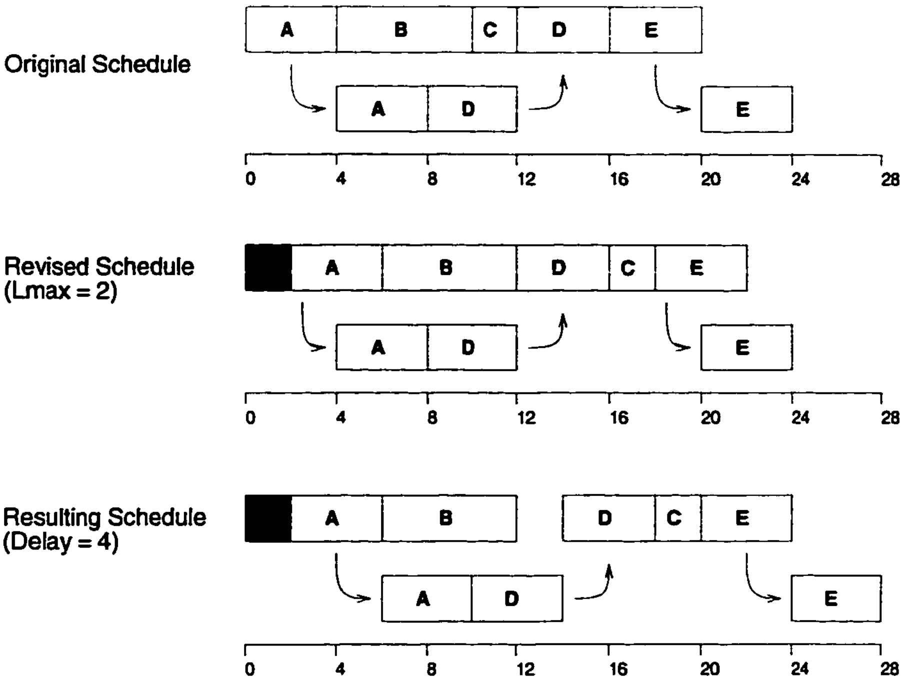


Figure 4.15: Added Delay using Precedence Deadlines.

release times performed worse than the Original Start (Local) method, in which the local cell is resequenced according to starting times in the original schedule. The reason for the lesser usefulness of precedence constraint times is the tightly-coupled nature of the job shop. In the simple three cell system of the previous section, if an operation is completed past its precedence deadline, it cannot add more to the final makespan than the extent to which it is late. This is not true in the general job shop. Figure 4.14 shows results for the same job shops of Figure 4.11, where each cell has one machine. Actual Prec D shows the actual performance when rescheduling the disrupted cell with precedence constraint times. Prec D Lmax shows the worst extent to which an operation at the disrupted cell missed its precedence deadline (the maximum lateness at that cell). In the three cell system, this would serve as an upper bound for the actual delay. Actual Push Back shows the actual push back performance, while Push Back Lmax shows the maximum delay for the disrupted cell's operations (which cannot be greater than the disruption delay itself). The figure shows that the actual precedence deadline rescheduling method produces a greater delay than its maximum lateness, and that it even performs worse than push back (for which the actual delay is somewhat less then the maximum lateness).

The reason for this phenomenon is that delaying an operation at a cell may ultimately

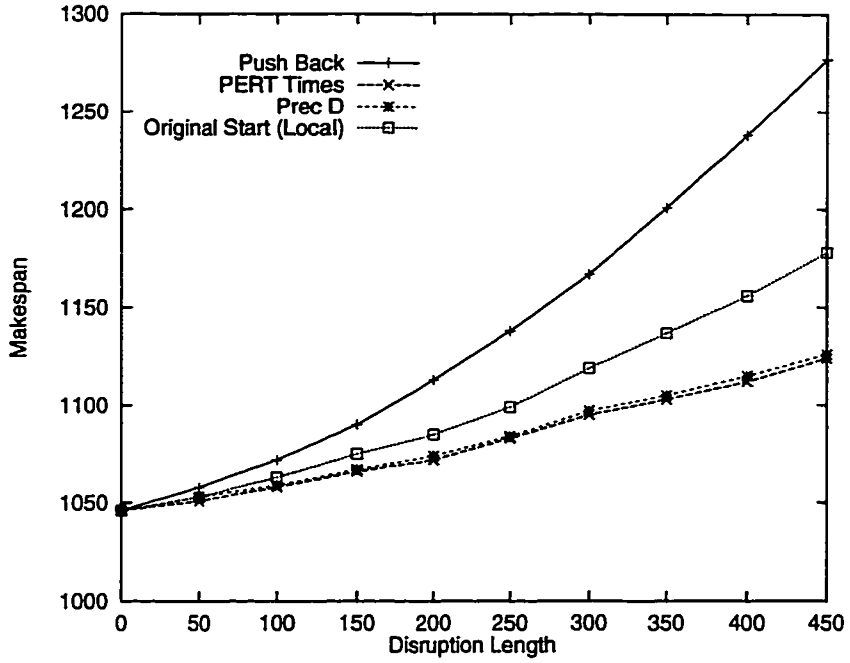


Figure 4.16: Rescheduling One Cell in a Flowshop-like System.

delay a later operation at that same cell via precedence constraints and schedule constraints at other cells, even if a job can visit a cell only once (no recirculation). This is illustrated in Figure 4.15. While in the revised schedule, no operation misses its precedence deadline by more than 2, the resulting schedule has a delay of 4, greater than the initial disruption itself.

Other possible methods for rescheduling the disrupted cell may have the danger of creating deadlock in the schedule. Recognizing deadlock in a distributed system is an important problem not considered here. The push back method is guaranteed not to create deadlock. The PERT-times, original start, and precedence deadline methods are not guaranteed to avoid deadlock, but are good at avoiding it, as they use information about the structure of the original schedule. In none of the simulations did any of these methods result in deadlock. Methods which did not use such information, such as rescheduling the disrupted cell using the LR dispatching rule, sometimes did produce deadlock.

In job shops which are like flowshops, in which work-piece delivery flows in one direction, the added delay problem for precedence deadlines disappears, as delay cannot propagate from a cell back to itself. Deadlock cannot occur in such systems. Such systems need not be strict flowshops; they can also be systems in which cells are strictly ordered, but where jobs may skip cells. In such job shops, the use of precedence deadlines is much more effective, as shown in Figure 4.16. Here the precedence deadline method performs almost as well as the optimal PERT-times method, and significantly better than the original start method.

4.5 Summary

In this chapter, we have investigated how knowledge about local schedule constraints can be used for local rescheduling with the objective of minimizing global makespan. We have shown that different levels of knowledge about local inter-cell schedule constraints, in the form of precedence constraint times and relaxed precedence constraint times, allow significantly different performance. We have also shown that use of precedence constraint times for local rescheduling in flowshop-like scheduling problems provides performance close to that of optimal methods using global knowledge, for local rescheduling to minimize global makespan. Use of these precedence constraint times requires only knowledge about local scheduling constraints, that is available either at the local cell or through communication with directly interacting cells.

The results of this chapter also suggest where local rescheduling is likely to be useful. When a cell has only one machine, a breakdown of this machine often produces a scheduling problem too constrained to allow local scheduling methods to be very useful. Having multiple machines at a cell allows a more pressing operation to be moved to an undisrupted machine during the duration of the disruption. Likewise, in general job shops, cells are too tightly-coupled for precedence constraint information to be very useful, because delays can propagate back to the originally disrupted cell; deadlock is also a concern. Precedence constraint times are much more useful in systems where precedence constraints are more particularly defined, such as those in which work tends to flow in one direction.

We have also suggested ways in which precedence constraint times can be used as non-disrupting rescheduling actions and guaranteed-safe scheduling actions. While in this chapter we were chiefly concerned with minimizing the value of the objective function for which the original schedule was constructed (i.e., the makespan), the next chapter will focus specifically upon using precedence constraint times for polite rescheduling and minimizing the propagation of disruptions. The results of this chapter's experiments, however, should provide insights for the problems considered in the next chapter.

Possible future work includes the consideration of other forms of information about cells' schedules. Precedence times are very useful and obvious ways of representing aspects of the global schedule for the local cell. However, less obvious information about the structure of the global schedule, as represented for example by original start times and PERT-related deadlines, can also be useful for determining how local schedule changes may affect the global schedule.

CHAPTER 5

Limiting Disruption Propagations in a Distributed Job Shop

Objectives for traditional scheduling methods consider when jobs are to be completed, or how efficiently available machines and tools are utilized. Certainly these objectives are important when schedules are revised; the original objective would often seem to be the most obvious objective for rescheduling. The previous chapter examined rescheduling in this case.

However, once an initial schedule has been constructed and is being implemented, the objectives may change. The focus may shift from constructing good schedules, to executing the current schedule. As mentioned previously, decisions based upon this initial schedule may have been made, such as material transport scheduling, workcell setups, and customer guarantees. Likewise, once a schedule is being implemented, a different authority may be responsible for its execution; while the production planning department may have constructed the schedule, the managers of the various production units may be responsible for completing the tasks themselves. Thus, objectives may be very different during schedule revision, and limiting the spread of disruptions is likely to be very important.

Some initial measures may reflect somewhat how much the system has been disrupted. For example, in the simple three cell problem of Section 4.2, the makespan of cell 3 (the initial objective) will be increased only if the disruption at cell 1 has propagated through cell 2 to cell 3, through execution delays of predecessor operations. Figure 5.1 shows the proportion of the time that cell 3 is disrupted versus disruption length, using the same simulation results as Figure 4.5. The performance of the methods evaluated in Section 4.2 with regard to disruption propagation is similar to their performance with regard to cell 3 makespan.

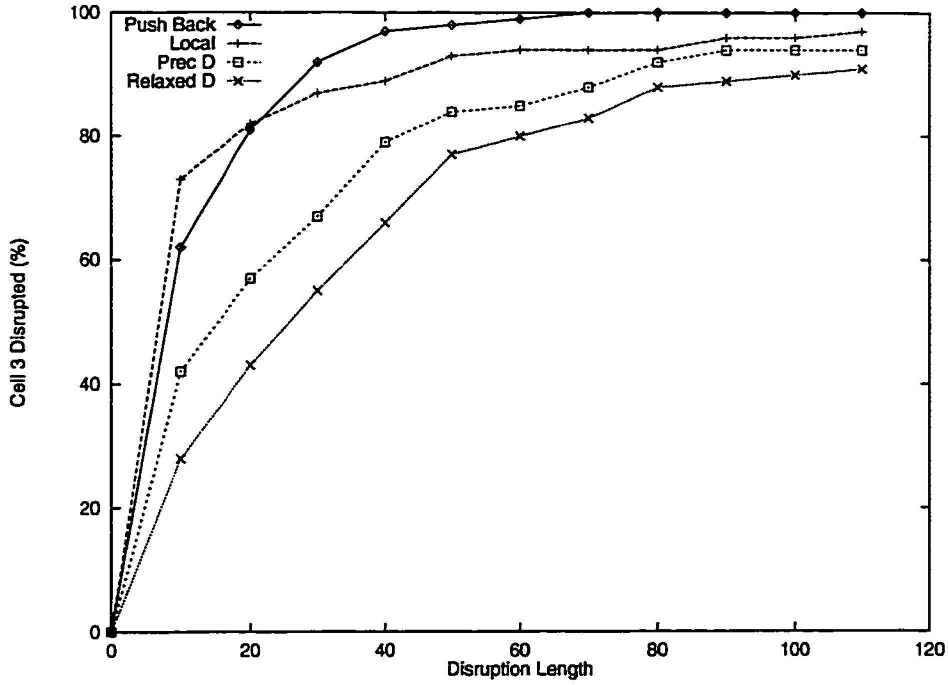


Figure 5.1: Three Cell System: Disruption Propagation.

However, in more general systems, the extent of "disruptedness" will not necessarily be reflected by the makespan, or by any other initial objective. In this chapter, we will investigate the problem of limiting the propagation of disruption, and propose a distributed rescheduling controller architecture, PRIAM, for rescheduling to avoid disruption propagation. First, using the number of cells as a simple measure of disruptedness, we will describe the *job class scheduling problem.* We will then discuss other possible measures for disruptedness. Then we will describe PRIAM, and evaluate the polite approach against other methods that also rely upon local information.

5.1 The Job Class Scheduling Problem

In order to consider the problem of limiting disruption propagation for handling schedule disruptions, we consider the problem of rescheduling one cell to minimize the number of other cells directly disrupted by the new local schedule. A remote cell is disrupted if any local job with a successor at that remote cell misses its simple precedence deadline. To define a formal scheduling problem that has the number of disrupted cells as a measure, we will assume to begin with that cells will only be disrupted by the rescheduling cell; i.e., here we will consider only disruptions that cannot propagate beyond one level. We will also assume that jobs will have at most one successor.

5.1.1 Problem Definition and Observations

The *job class scheduling* problem is a scheduling problem for scheduling jobs on one cell, and is defined as follows: the set of jobs for that cell is partitioned into job class sets $K_1, \ldots, K_l$, jobs being in the same class if their successors are at the same remote cell, each job class corresponding to exactly one remote cell. Each job has a due date equal to its simple precedence deadline. We call a job class tardy if any of the jobs in the class is tardy, i.e., for job class K_k, $\Upsilon_k = 1$ if $\exists j \in K_k : c_j > d_j$, 0 otherwise. The objective is to maximize the number of job classes for which every job is on time, that is, to minimize $\sum \Upsilon_k$, the number of tardy job classes. The measure minimizes the number of remote cells disrupted. Here, whether all jobs in a class are tardy, or only one, is irrelevant; any tardy job in a class will disrupt the corresponding cell's schedule.

This problem is very similar to the problem of minimizing the number of tardy jobs, $\sum U_j$. The simple machine scheduling problem to minimize $\sum U_j$ when all jobs have the same release time is solvable in polynomial time by the well-known forward algorithm by Moore [48]. This algorithm schedules jobs one by one in EDD order, removing the scheduled job with the greatest processing time if at any stage a scheduled job misses its due date. The more general problem of minimizing $\sum w_j U_j$, when jobs have weights, has as a special case the knapsack problem and is thus NP-hard. Minimizing the number of tardy jobs with non-identical release times is also a hard problem, treated by Kise et al. [38].

Despite the similarity, the problem of minimizing the number of tardy job classes with identical release times on one machine is more complicated. The multiple due dates for one job class makes job class scheduling more difficult. While we have not proven this problem to be NP-hard, it is not solved by the Moore algorithm or any other familiar rule, nor does it have a structure that would suggest a common technique. There are different assumptions that do allow polynomial-time solutions to this problem. If all jobs in the same class have the same due time, then the problem can be converted into a regular $\sum U_j$ problem, with one job replacing each job class, its processing time equal to the sum of the processing times of the jobs in that class, and its due date equal to the common due date of the job class. A simple pair-wise interchange argument proves that there exists a solution to any $\sum \Upsilon_k$ problem in which all jobs of any class form one uninterrupted block in the schedule, the ordering of jobs within the same class being irrelevant. Converting the problem and solving using the Moore algorithm thus will provide a solution.

Likewise, a similar solution can be found when jobs in the same job class have different due dates, but when a job in one class having a later due date than a job in another class implies that all jobs in the first class have later due dates than all in the second class. That is, if, for any job classes K_a and K_b, $d_h < d_k$ for $h \in K_a$, $k \in K_b$, implies that $d_i \leq d_j \; \forall d_i \in K_a, d_j \in K_b$ then an optimal solution exists with all jobs in any class forming one uninterrupted block in the schedule. Another assumption, that all jobs have the same deadline, has the fairly obvious solution to order jobs by the increasing total aggregate processing time of of their job classes.

5.1.2 A Branch-and-Bound Solution

While we do not have a polynomial time algorithm to solve this problem, the observations of the previous section can help guide the search for a solution. Given a scheduling problem to minimize $\sum \Upsilon_k$, we can find an upper bound for the solution by converting it into a $\sum U_j$ problem, as in the last section, one job for each job class, but with the due date equal to the latest due date among the jobs of that class. Solving this aggregate job problem will at least account for any capacity constraint violations, given job processing times and latest job class due dates. The solution to the aggregate job problem is certainly an upper bound, as greater due dates cannot be more constraining. However, this bound can be arbitrarily bad. While a job's due date may not be greater than or equal to the sum of all jobs' processing times (any job for which this is true may be effectively removed from the problem), many tight due dates, not taken into account by this bound, may make impossible the scheduling of more than one job class, given any number of job classes to be scheduled. Nevertheless, for many problems it may be useful. Figure 5.1.2 shows a branch-and-bound algorithm that uses this bound.

5.2 Limiting Disruption Propagation

While the job class scheduling problem does consider the spread of disruptions for one level of propagation, it does not consider the effects beyond that level. While a solution method may find a new schedule that disrupts only one other cell, this cell may be disrupted to such an extent that it inevitably propagates the disruption to other cells. A different solution, while not optimal in terms of the job class scheduling problem, may disrupt more

Notation:	For subproblem $\mathcal{P}$ and job classes $C = \{C_1, \ldots, C_k\}$, $S_{\mathcal{P}}$ = the set of job classes chosen for scheduling, $R_{\mathcal{P}}$ = the set of job classes rejected (tardy), $n_{\mathcal{P}} = \max i : C_i \in S_{\mathcal{P}}$, $b_{\mathcal{P}}$ = the upper bound for number of tardy job classes.
Initialization:	$\mathcal{P}_0$ is the first subproblem, $S_{\mathcal{P}_0} = \emptyset$, $R_{\mathcal{P}_0} = \emptyset$, $n_{\mathcal{P}_0} = 0$.
Branch:	From subproblem $\mathcal{P}$, make $k - n_{\mathcal{P}}$ new subproblems. For subproblem $\mathcal{P}_i, i = 1, \ldots, k - n_{\mathcal{P}}$, if all the jobs in the classes in $S_{\mathcal{P}}$ and the jobs in class $C_{n_{\mathcal{P}}+i}$ can be scheduled on time, then $S_{\mathcal{P}_i} = S_{\mathcal{P}} \cup \{C_{n_{\mathcal{P}}+i}\}$, $R_{\mathcal{P}_i} = R_{\mathcal{P}}$, and accept $\mathcal{P}_i$. Otherwise reject $\mathcal{P}_i$.
Bound:	For subproblem $\mathcal{P}$, let the upper bound $b_{\mathcal{P}}$ = the number of tardy jobs for the solution of the aggregate scheduling problem, using Moore's algorithm, using only job classes in $C - R_{\mathcal{P}}$

Figure 5.2: Branch-and-Bound Algorithm for Tardy Job Classes Problem.

cells more moderately, so that the disruption does not propagate beyond the group of cells.

Likewise, even if the disruption propagation is restricted to one level, as assumed in the job class scheduling problem, a solution may not consider how badly disrupted the disrupted cells are. It may be preferable to have a schedule in which two job classes are tardy with just one tardy job each, instead of a schedule in which only one job class is tardy, but all of the jobs in that class are tardy. On the other hand, rescheduling when only disruptions to jobs are considered (e.g., number of jobs delayed), or considering only the initial objective, may neglect the negative effects of causing even a slight disruption to another cell. Thus, there is a tradeoff between considering disruptive effects for cells, and those for jobs. Determining how to schedule to limit propagation of disruptions requires reasoning about this tradeoff. Determining a correct balance would also depend upon the domain.

Possible Measures

There are various possible measures that may reflect the extent of a disruption to a manufacturing system. A simple one already mentioned is the *total number of cells disrupted* (i.e., the number of cells which must perform rescheduling). Other job-based measures include the *total number of operations rescheduled* due to rescheduling (i.e., operations whose starting times or designated machines are changed) and a similar measure, the *total*

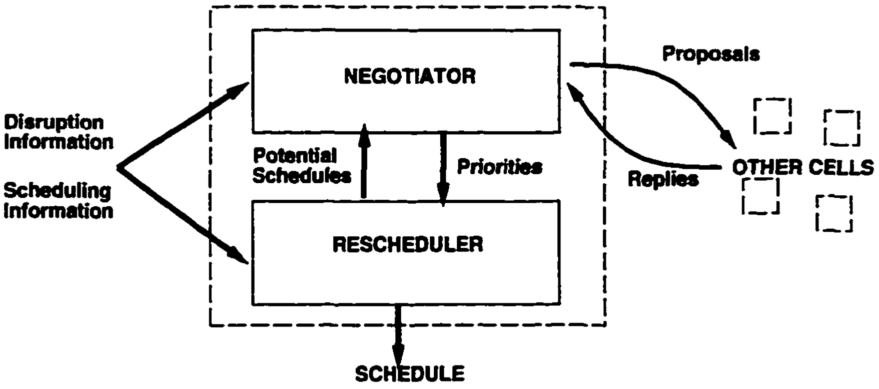


Figure 5.3: The PRIAM Architecture

number of operations moved to other machines or cells due to rescheduling.

Other measures may include the time by which the original schedule can be resumed (as in the match-up scheduling case), the time required to complete the rescheduling process, and some measure of how much the revised schedule causes deviations from inventory planning or customer expectations. This last possibility is obviously harder to gauge than the others. All of these measures are necessarily imperfect approximations of some intangible measure of how much the revised schedule imposes additional net costs.

While these measures may only be approximations of real objectives, the local agent may itself be forced to use imperfect estimates of these measures, depending upon what knowledge it has of how its actions affect other cells. Using the knowledge levels of section 4.3, at the *Utterly Ignorant* level, the agent may have to use the *number of local operations delayed* as an estimate of total disruptedness. At the *Locally Informed*, *Precedence Deadline*, and *Relaxed Precedence Deadline* levels, the agent may have more accurate estimates of the number of remote cells disrupted at the first level of propagation. Through negotiation, an exact count of the number of cells disrupted may be obtained, if the disruption can be limited to this first level. While the agent may need to communicate in order to attain these levels of knowledge, communication with affected cells may allow a better estimate of the total number of cells disrupted, if disruption cannot be limited to the first level. Communication is also necessary to estimate the total numbers of operations rescheduled or moved.

5.3 Implementation

We are implementing our approach to distributed schedule revision using a rescheduling module architecture called PRIAM (Polite Rescheduler for Intelligent Automated Manufacturing), illustrated in Figure 5.3 In this scheme, each cell controller will have a local PRIAM module, which determines when rescheduling is necessary and how to reschedule, interacting with PRIAM modules of other cells when necessary. Because there is a natural division between reasoning about scheduling priorities and actually generating schedules, each PRIAM module consists of two separate but interacting modules, the *negotiator* module and the *rescheduler* module. The division of an intelligent system into separate modules with different responsibilities is a common practice in AI system architecture, based upon the differing applicability of knowledge sources to different problems. In PRIAM, the negotiator module is responsible for determining scheduling goals and job priorities and for negotiation, while the rescheduler module is responsible for producing schedules for given scheduling goals and job priorities.

5.3.1 The Negotiator Module

The negotiator uses local information and information obtained through negotiation to determine scheduling goals, and local job priorities and effective due dates and release times. [1] In order to reason about priorities, it must determine what information is useful, and how this information may be obtained, if it is not locally available. Thus it must use effective negotiation strategies, an issue we have discussed in Chapter 3 in the domain of tool management.

When the rescheduler module reports that a local disruption cannot be handled without constraint violation, the negotiator must determine whether local constraints should be relaxed in order to simplify the problem, and whether negotiation is needed so that changes in other cells' schedules (or additional information about other cells' schedules) may simplify the local problem. If negotiation is to be used, the negotiator determines what proposals to make, and how to interpret the replies to these proposals. Likewise, when a proposal is received from another cell, the negotiator determines what new constraints are to be given to the rescheduler so that a reply can be determined. These new constraints should reflect

[1] Effective due dates and release times are those to be used for scheduling, though not necessarily the same as the official due dates and release times of the corresponding job order.

```
Given $n$ high priority predecessor jobs, $m$ low priority predecessor jobs, and $s$ non-predecessor jobs,
and a preferred dispatch rule $D$:
1. Schedule the high-priority predecessor jobs by earliest precedence deadline, and label
   these jobs $1, 2, \ldots, n$;
2. For each scheduled high-priority predecessor job $i$ in order,
      2.1. Define the slack time $s_j$ for job $j$, $i \leq j \leq n$ as the idle time in the current
      schedule between the completion time of job $i - 1$ and the precedence deadline
      for job $j$;
      2.2. If there is an unscheduled low-priority predecessor job which has processing time less than
      $\min_{i \leq j \leq n}(s_j)$, insert into the schedule before job $i$ the job which has the
      smallest precedence deadline of such jobs, and go to 2.1.
3. Schedule the remaining unscheduled low-priority predecessor jobs by earliest precedence deadline.
4. Relabel the scheduled predecessor jobs in order of starting time $1, 2, \ldots n + m$;
5. For each scheduled predecessor job in order,
      5.1. Define the slack time $s_j$ for job $j$, $i \leq j \leq n + m$ as the idle time in the current
      schedule between the completion time of job $j - 1$ and the precedence deadline
      for job $j$;
      5.2. If there is an unscheduled job that has processing time less than
      $\min_{i \leq j \leq n}(s_j)$, insert into the schedule before job $i$ one such job chosen by $D$,
and go to 5.1.
6. Schedule the remaining unscheduled jobs by $D$.
```

Figure 5.4: Priority scheduling algorithm for polite scheduling

rules that allow the generation of guaranteed-safe schedules. We examine some issues of proposal generation in Chapter 6.

As an example of a simple negotiation policy (used for the simulations described in the following section), we can respond to a negative (*not-ok*) reply from a cell to a local schedule revision proposal by raising the priority level of all local jobs that have successors at that cell. This strategy will assign a higher scheduling priority to jobs with successors at bottleneck (or otherwise relatively busy) cells. In order to enforce the generation of guaranteed-safe schedules in response to another cell's proposed schedule revision, local jobs' starting times can be used as effective release times, as mentioned previously. However, certain configurations of production cells may allow the use of precedence ready times as effective release times, as is the case in the simulations of the following section.

5.3.2 The Rescheduler Module

The rescheduler module maintains the existing local schedule, and when necessary is responsible for producing appropriate schedules for new scheduling goals, and job priorities and effective due dates and release times. Thus it uses a library of scheduling methods that can be used when appropriate. It evaluates a newly generated schedule using the appropriate measure, and reports the result to the negotiator module.

The rescheduler uses priorities for jobs, both implicitly (e.g., through due dates) and explicitly. In the latter case, scheduling algorithms that deal with job priority levels are needed. On such algorithm, which we propose (and use in the following section) is shown in Figure 5.4. It uses three priority levels: high priority predecessor jobs, low priority predecessor jobs, and non-predecessor jobs (lowest priority). As suggested previously, a high priority job may be one whose successor is at a bottleneck cell. In this algorithm, highest priority jobs are scheduled in earliest-predecessor-deadline-first order, and jobs of successively lower priority levels are inserted into the schedule in a way that does not cause any already scheduled job to miss its precedence deadline. Slack in the schedule and between each scheduled job's completion time and precedence deadline is used to determine where an unscheduled job can be inserted. The objective of this algorithm is that jobs meet their precedence deadlines; it is not necessarily optimal, but is useful because jobs may have non-identical release times.

The rescheduler also receives new constraints from the negotiator, and propagates these new constraints through the existing schedule to determine whether any constraint violations occur due to this new constraint. Thus, the rescheduler provides the negotiator with information about whether an incoming proposal (proposing new constraints) should be accepted, and what kind of reply to give. In this case, a copy of the original schedule must be maintained, and not altered until a proposal has been both accepted and then implemented by the proposing cell.

5.4 Evaluation

5.4.1 Simulation Model

We evaluate priority scheduling algorithms in PRIAM through simulation of disruptions in a generic manufacturing system. In these simulations, a preschedule is constructed for a generic manufacturing system illustrated in Figure 5.5, that consists of four groups of three cells each: a set of machining cells, two sets of subassembly cells, and a final assembly cell. Each cell has two identical machines, and a job may be processed only at one specified cell.

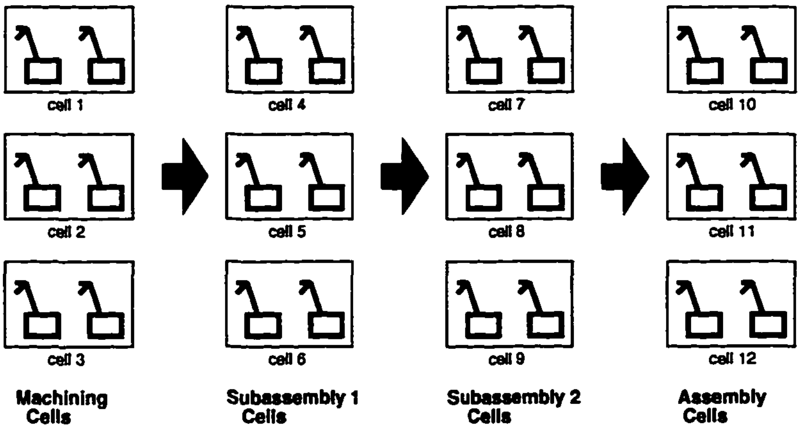


Figure 5.5: Generic Cellular Manufacturing System

Jobs may have precedence constraints: a job at a machining cell may have a successor at a subassembly 1 cell, a job at a subassembly 1 cell may have a successor at a subassembly 2 cell, and a job at a subassembly 2 cell may have a successor at a final assembly cell. But there may be jobs at any cell that do not have successors or predecessors.

The preschedule is generated from a randomly generated set of 192 jobs (16 per cell). One parameter in the generation of the job set is p, the probability that any given job is the predecessor of some other job (excepting final assembly jobs). By varying p, job sets with different levels of precedence constraints are generated. In each of the job sets, a job may have at most one successor, but may have several predecessors. We assume, for simplicity, that setup times are not sequence-dependent, and can be ignored. In each simulation, one of the machines at a machining cell is disabled for a given interval, and the system is rescheduled using each of the rescheduling methods described above.

We are chiefly concerned with how disruptive the rescheduling process is to the manufacturing system. Our primary measure of disruptiveness is the number of cells that are affected by the disruption. Other measures that we consider are the total number of times cells need to reschedule, the number of jobs whose scheduled completion times are changed, and the number of jobs that are rescheduled on a machine different from that on which it was originally scheduled. Our secondary measure is the makespan, the completion time of the last job to finish. Makespan is the measure used in constructing the original preschedule, and it is a measure of the quality of the resulting schedule.

We consider three different negotiation strategies in these simulations. The negotiation strategies that can be used depends upon what kind of information can be obtained from other cells. For the polite negotiation algorithm for these negotiations, we assume that, when a disrupted cell proposes a new schedule to a remote cell, that remote cell will reply **ok-1** if it can reschedule without disrupting other cells; otherwise, it will include in its reply the identities of the cells it will disrupt if the proposed schedule is implemented. From this information, the disrupted cell will have an estimate of how many cells will be disrupted by a proposed schedule. In the first polite negotiation algorithm (POL-NEG1), the disrupted cell will propose a small number of possible schedules generated from different priorities, and will decide to implement the first proposal that elicits only **ok-1** replies from other cells. Otherwise it will implement the proposal causing fewest disruptions. In the second polite negotiation algorithm (POL-NEG2), the disrupted cell will also propose a small number of possible schedules and will decide to implement the proposal causing fewest disruptions. The third negotiation algorithm (POL-NEG3) is the same as POL-NEG2, except that the number of proposals is smaller. Only the originally disrupted cell uses these polite negotiation algorithms; if a cell is disrupted only by the late completion of a predecessor job, it will not use negotiation.

In these simulations, we compare the results from our polite negotiation algorithms with the results from a polite algorithm (POL) using the previously described priority scheduling algorithm but without negotiation. In this algorithm, the disrupted cell generates a small number of possible schedules and tries to limit the number of disrupted cells, but does so without negotiation with other cells. We also compare these results with the results from two similarly fast algorithms that do not consider how the rescheduling of one cell may affect another and that rely only upon local knowledge: the push-back algorithm (PB), in which schedules at disrupted cells are simply pushed back, and the largest-remaining-processing-time-first dispatch rule (LPT), that is used to achieve a low makespan, but does not consider the problem of disrupting other cells. These algorithms do not include optimization techniques. Such techniques usually consume much time and computation, and our assumption is that schedule revision at a cell will not take place on powerful computing platforms dedicated to execution of long intensive tasks, as the cell controller is responsible for other local management tasks.

5.4.2 Results

Figures 5.6 through 5.17 show results for three simulations for each of twenty job sets

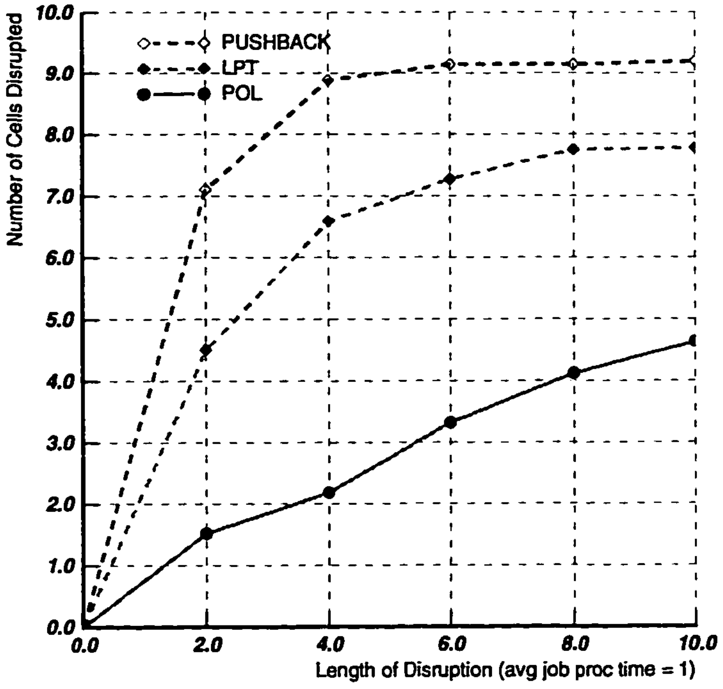


Figure 5.6: Number of cells disrupted

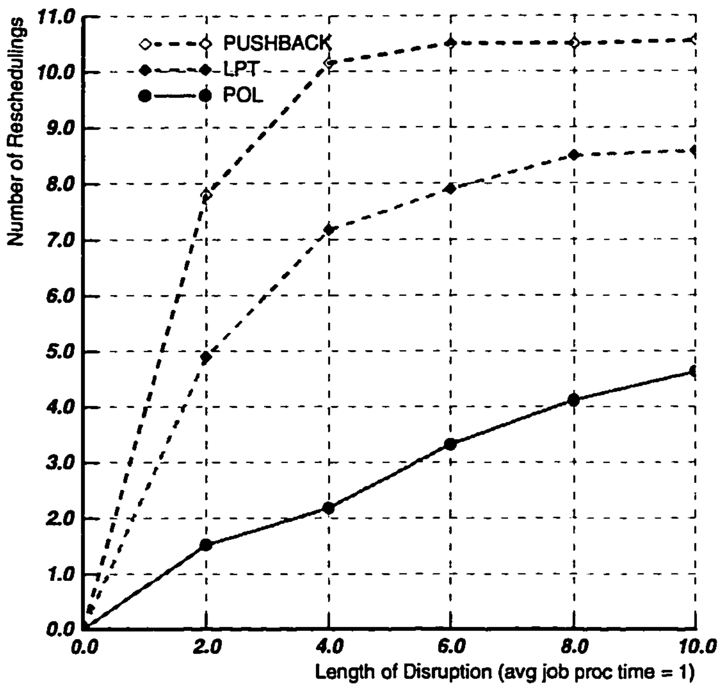


Figure 5.7: Number of reschedulings

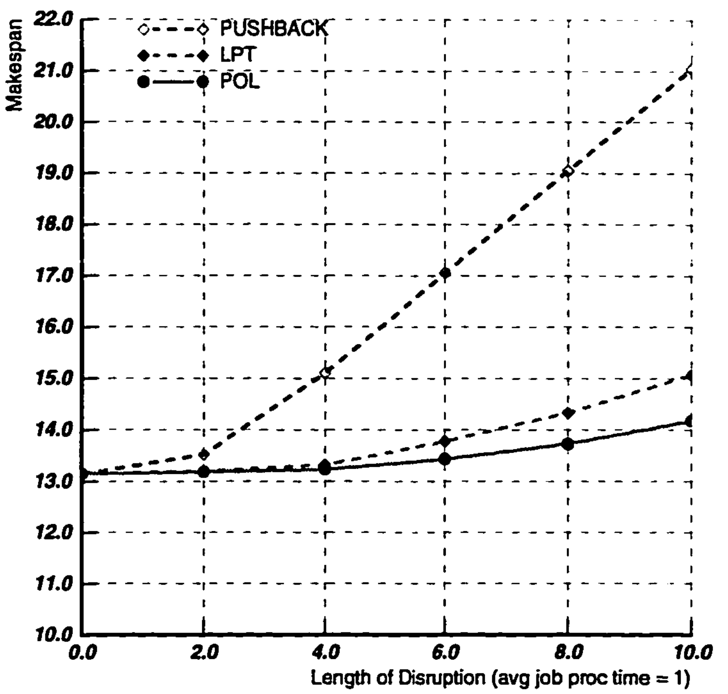


Figure 5.8: Makespan

with the constraint parameter $p = 0.6$. First we consider only the PB, LPT, and POL algorithms. Figure 5.6 shows the number of cells eventually disrupted from the propagation of one machine disruption, versus the length of the original disruption. These results show that the POL algorithm isolates disruptions much more than the two other non-negotiation rescheduling methods. Figure 5.7 shows the number of times cells eventually have to reschedule. Again, the POL algorithm is much better for preventing other cells from having to reschedule. Figure 5.8 shows the makespan after all rescheduling is finished. The POL algorithm is slightly better than the LPT algorithm at keeping the makespan from being affected by the disruption.

Figures 5.9 through 5.15 show simulation results for the polite negotiation algorithms. Figures 5.9 and 5.10 show that for short disruptions, negotiation does not seem to provide an advantage; for longer disruptions, both polite negotiation algorithms show a significant advantage over the POL algorithm without negotiation. There seems to be little difference in how well the POL-NEG1 and POL-NEG2 algorithms prevent the spread of disruptions, while Figure 5.11 shows that the POL-NEG1 algorithm requires fewer message exchanges. Figure 5.12 shows how many levels of propagation are caused by the original disruption. Figures 5.13 through 5.15 compare the POL-NEG2 and POL-NEG3 algorithms, and show

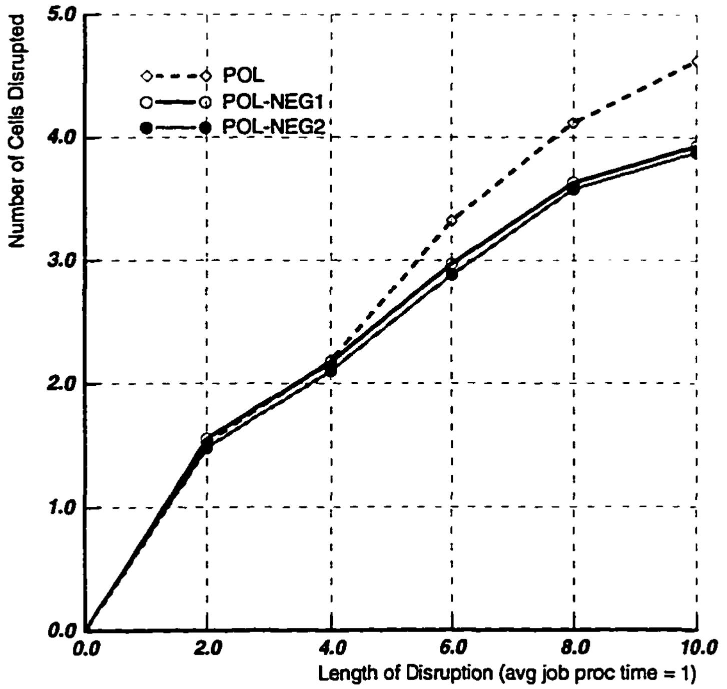


Figure 5.9: Number of cells disrupted

that while fewer proposals reduce the number of messages required, it also reduces the advantage of using negotiation. Figures 5.16 and 5.17 show the performance of the various algorithms using the measures of the number of jobs that are rescheduled, and the number of jobs that are moved to a different machine during rescheduling. These show that, when the disruption length is low, the polite algorithms reschedule and move many fewer jobs than the LPT algorithm. When the disruption length becomes large, it is much harder to avoid rescheduling jobs.

5.4.3 Discussion

While any use of negotiation in distributed manufacturing systems is necessarily very domain-dependent, our generic simulations allow us to make some observations about the usefulness of the polite approach and polite negotiation. These simulations show that polite rescheduling provides an advantage for a distributed system, allowing a cell to respond to a schedule disruption while reducing the spread of this disruption to other cells. However, they also indicate under what conditions polite rescheduling is likely to be useful. When the constraints of the rescheduling problem are light, as when the initial machine disruption is short, the use of negotiation does not provide much advantage, because the disruption is not

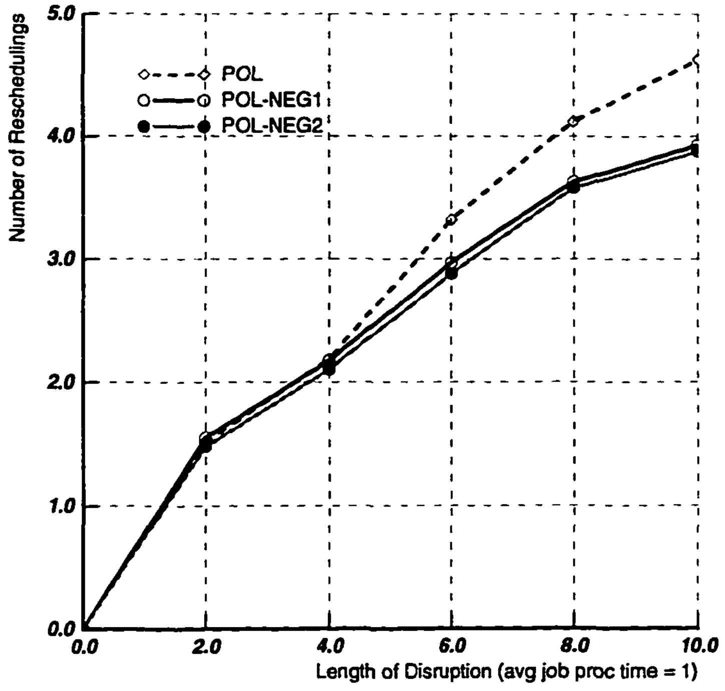


Figure 5.10: Number of reschedulings

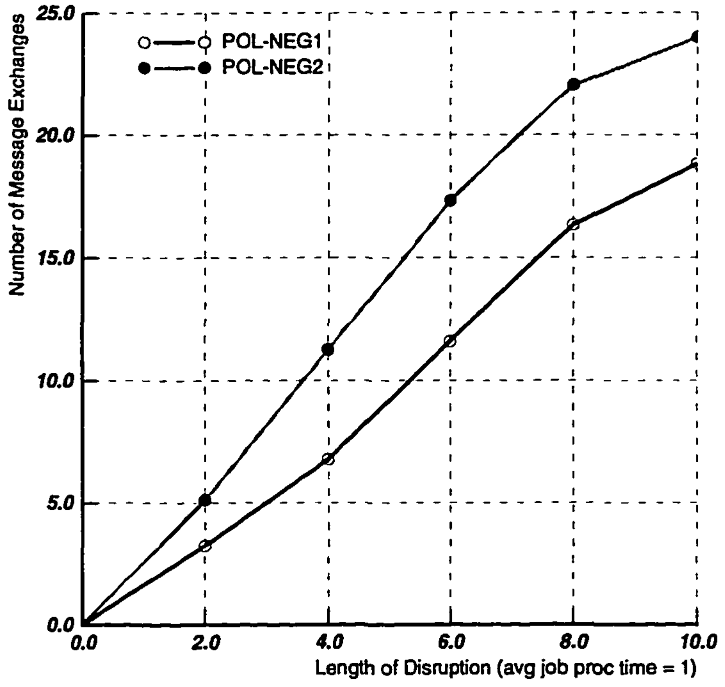


Figure 5.11: Number of message exchanges

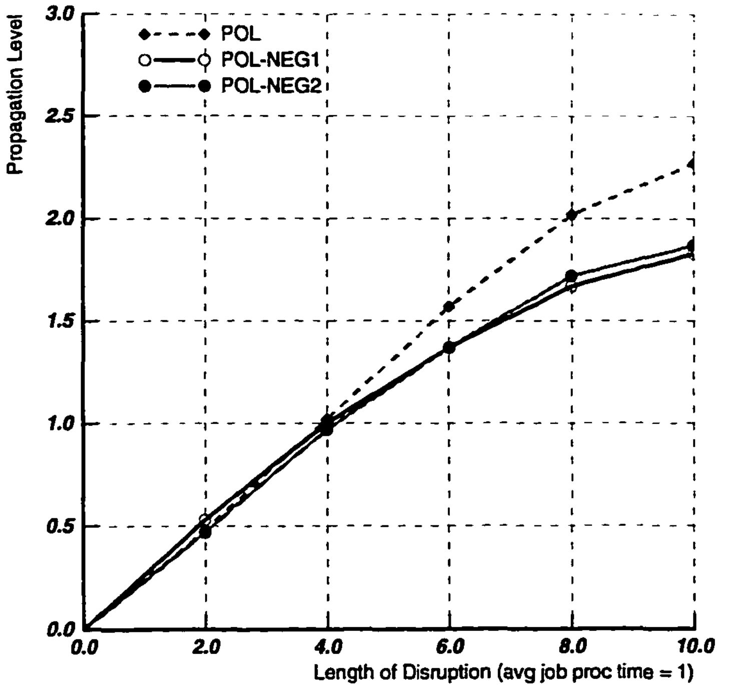


Figure 5.12: Propagation level

likely to be propagated even if the possibility of propagation is not considered. Likewise, other simulation results not presented here indicate that when the scheduling problem itself is less constrained (because there are fewer precedence relations among jobs), the polite approach with and without negotiation provide less significant advantages, once again because the disruption is much less likely to be propagated.

When the rescheduling problem is more constrained, as when the initial disruption is longer, the polite strategies that use negotiation provided a much more significant advantage over those strategies not using negotiation. Under these conditions, a cell that does not have global information is not likely to have enough information about how its actions will affect other cells, and thus communication allows the cell to make better-informed decisions. However, our other simulation results suggest that when the scheduling problem is much more constrained (because there are more precedence relations among jobs), the polite approaches provide less of an advantage over other approaches. Under these conditions, the many constraints of the problem almost ensure that any disruption will be propagated throughout the system, and attempts to respond to the disruption using negotiation will merely confirm that the propagation is almost unavoidable. Thus, there is a "window" of usefulness for our polite approach; it is most useful when the problem is constrained

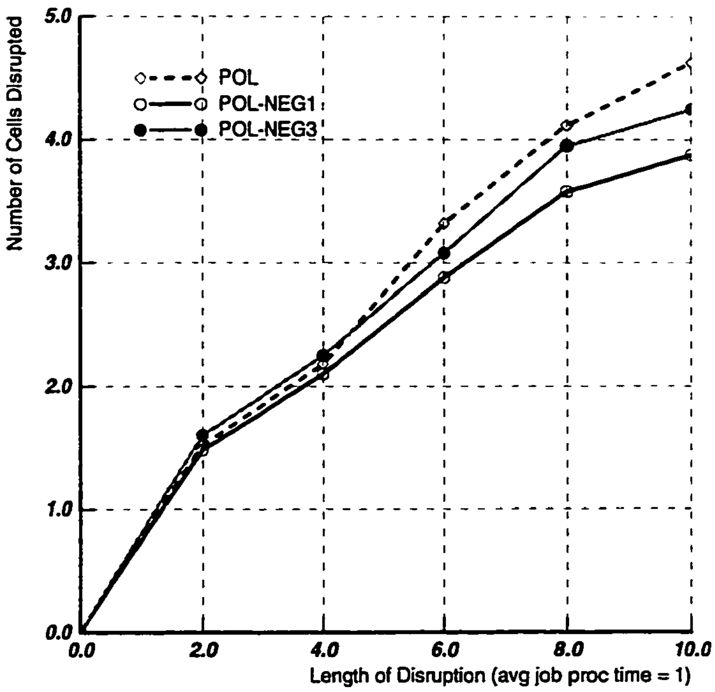


Figure 5.13: Number of cells disrupted

enough that a cell cannot have sufficient information without communication, but not so constrained that propagation of disruptions cannot be avoided. The size of this window is obviously very domain-dependent.

Another important issue for evaluation of the polite approach is cost. There are at least two dimensions for cost in this type of problem. The first involves the quality of the resulting schedule based upon some scheduling measure disregarding disruption. In these simulations, we considered the makespan measure, which is a single 'global' measure rather than an aggregate one. Using this measure, our polite methods performed well, because preventing disruption propagation also prevents delays in the completion times of jobs with predecessors. However, in previous simulation results that we have reported in [69], we considered the average tardiness measure, which is an aggregate measure. When this measure was considered, our polite methods produced schedules of less quality (higher tardiness) than other approaches that allowed greater disruption propagation, because the polite methods trade off increased tardiness at the disrupted cell against the spread of disruptions. Thus, there can be a tradeoff between the measure for *scheduling* (e.g., tardiness) and the measure for *rescheduling* (e.g., number of cells disrupted). The other dimension of cost is the computation and communication overhead for the polite approaches. This overhead

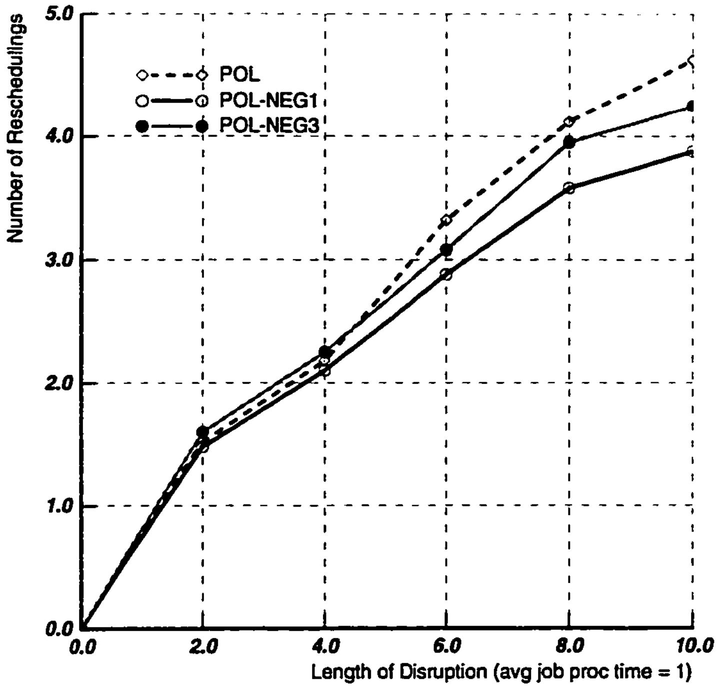


Figure 5.14: Number of reschedulings

is an important consideration for any use of negotiation, especially if communication and processing resources are limited, as might be true for a cell controller computer. Our simulations show that the communication costs (in number of message exchanges) grow as the size of the disruption grows (because more negotiation is needed to find a good rescheduling solution), and that additional gains in preventing disruption propagation come at the cost of additional communication (because more proposals are required to find a better solution).

In these simulations, for simplicity, we have used the same job characteristic parameters for the job sets for each cell. Greater heterogeneity among cells is more realistic, as different cells may process different classes of tasks. Our ongoing work investigates how such heterogeneity can determine the utility of different negotiation strategies.

5.5 Summary

In sum, this chapter has investigated several aspects of the problem of limiting disruption propagation in a distributed job shop. We have proposed a new scheduling problem formulation, the job class scheduling problem, that considers the problem of minimizing the number of cells disrupted directly by local scheduling decisions. We have introduced various measures of disruptedness with which schedule revision decisions can be evaluated.

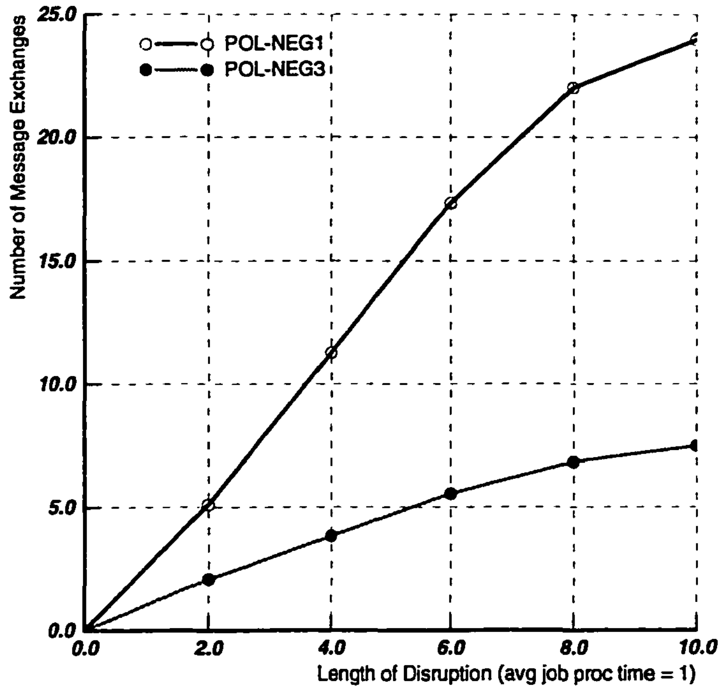


Figure 5.15: Number of message exchanges

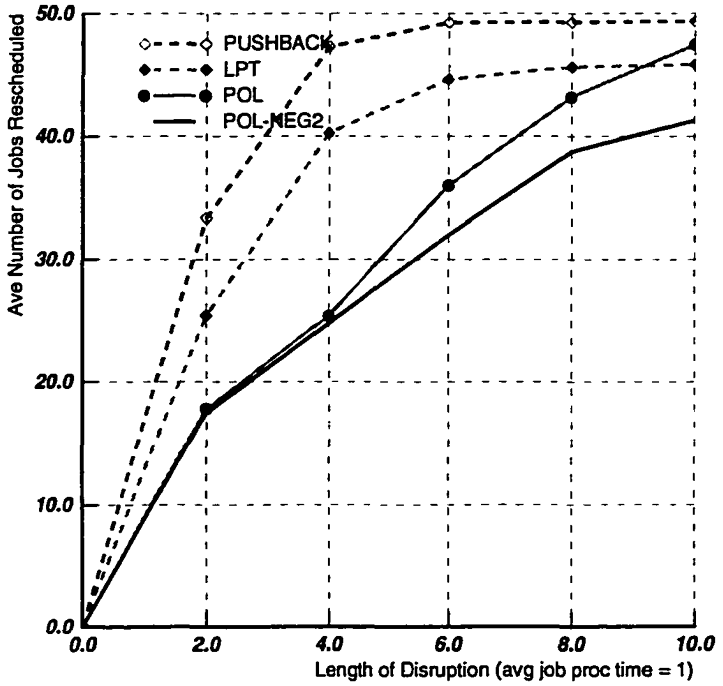


Figure 5.16: Number of jobs rescheduled

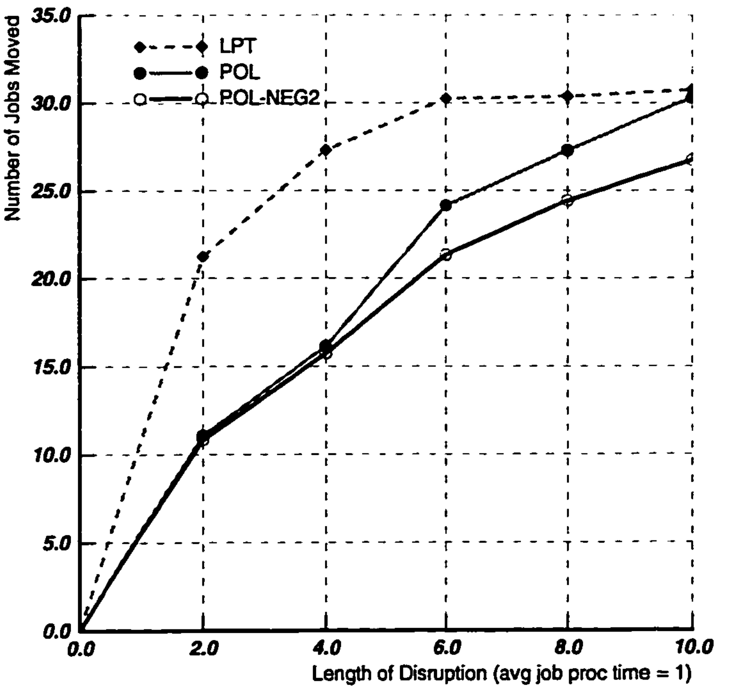


Figure 5.17: Number of jobs moved

We have proposed a distributed rescheduling architecture, PRIAM, which makes negotiation decisions and rescheduling decisions in separate but interacting modules. We have evaluated the polite rescheduling approach for a plausible distributed job shop, simulating PRIAM's decision-making activities and using appropriate measures of disruptedness. Our simulation results for PRIAM show the advantages of using a scheduling algorithm that emphasizes precedence constraints over other scheduling considerations, and demonstrate the advantages of negotiation.

There are a great many implementation issues for PRIAM that will determine the future direction of this research. These are described more fully in Chapter 7. Briefly, we would like to have more sophisticated reasoning in the negotiator module, to allow it to model the states at other cells more explicitly. We would like the rescheduler to have a larger and more varied collection of scheduling methods, and to consider tools and other resources shared among cells. Finally, we believe that the user interface for PRIAM will be of great practical importance, as most real scheduling systems are often in advisory rather than decision-making roles.

CHAPTER 6

Polite Rescheduling with Uncertain Precedence Constraints

When trying to reschedule politely, an agent reasons about the effects its scheduling decisions are likely to have on other agents. However, sometimes information about those effects is not available locally, and there may be reasons to speculate about effects before asking other agents about them. One reason is that an agent should construct a schedule proposal likely to be good when it begins negotiation, before information is exchanged. Another reason may be that detailed information about other agents' schedules may not be forthcoming due to concerns about privacy.

Using good proposals may reduce the number of negotiation cycles needed to produce an acceptable solution. This reduction is always desirable in negotiation. Besides the communication and communication processing costs mentioned previously, human intervention may be required during each negotiation cycle if the scheduling system is an advisory system rather than a controlling one. [1] Thus, it is important to use limited available information to decide among possible proposals and to guide the negotiation process.

Lack of full communication about agents' schedules is a concern in distributed meeting scheduling, in which agents' schedules are to be kept private. Such privacy may not seem reasonable in a single manufacturing system, but it may be more plausible when production is the joint or related efforts of different firms or different organizations within one large firm. When full information is not forthcoming, two important issues are trust, and anticipation of agents' responses to negotiation. How agents establish trust among one another is an important problem in DAI that has been addressed by others [28, 20, 18]. Here, we will simply assume that agents will not lie, but that they will not tell "the whole truth". We

[1] Issues concerning interaction between human users and scheduling systems is discussed in [31, 52, 59].

instead focus upon rescheduling when only probabilistic information about other agents' schedules is known.

6.1 Stochastic Scheduling and Some Observations

Stochastic models for scheduling have long been a subfield of OR scheduling, most likely because of their similarity to queuing models. Most work has focussed upon problems with uncertain processing times, uncertain processing times and uncertain due dates, or uncertain machine break-down. When revising a local schedule, however, an agent will know the processing time of the jobs it must schedule, but may not know the effects of its scheduling actions on other agents, due to a lack of information about other agents' states. Thus, we investigate scheduling problems in which processing times are known, but due dates (representing simple precedence deadlines) are treated as random variables. As we are interested in limiting disruption propagation, we consider the measures minimizing the number of tardy jobs, and minimizing the number of tardy job classes. Unless otherwise specified, release times are assumed to be identical and equal to 0, and that random variables are independent. The random variable due date for job j will be written D_j, with distribution function F_j and density function f_j.

6.1.1 Expected Number of Tardy Jobs

While minimizing the number of tardy jobs is easy in the deterministic case (using Moore's algorithm), it has not been proven easy for the stochastic case, in which the expected number of tardy jobs ($\mathrm{E}[\sum U_j]$) is to be minimized. Crabill and Maxwell [11] have shown that, when due dates and processing times are stochastic, the probability that every job is early is maximized if the jobs are ordered so that their due date distributions are stochastically ordered. [2] Pinedo [56] has shown that the expected number of tardy jobs, when jobs have exponentially distributed processing times and identically distributed due dates, is minimized by ordering jobs by expected processing time. This ordering is optimal also when jobs have a common random due date.

These theorems remain true when processing times are known. Knowing processing times also allows us to show the following:

Theorem 1 *If jobs have identical processing times $= p$, and can be ordered $j_1, \ldots, j_n$ such*

[2] Two random variables, X and Y, are called stochastically ordered if their distribution functions, F_X and F_Y, satisfy $F_X(z) \leq F_Y(z)$ for all z.

that their due dates are in increasing stochastic order, and such that, over the interval $[0, np]$ due dates are in decreasing density function order (i.e., if $i < j$, $f_i(z) \geq f_j(z)$ for all $z \in [0, np]$), this order will minimize the expected number of tardy jobs.

Proof: The proof is by an interchange argument. Assume a schedule S in which two adjacent jobs i and $i+1$ are not in the order indicated (i.e., $D_i > D_{i+1}$, and $f_i(z) < f_{i+1}(z) \forall z \in [0, np]$). Consider the effect of interchanging these two jobs to produce schedule S'. The objective function values for these two schedules are:

$$\begin{aligned}
\mathrm{E}_S[\textstyle\sum U_j] &= \sum_{k \neq i, i+1} F_k(kp) + F_i(ip) + F_{i+1}((i+1)p) \\
\mathrm{E}_{S'}[\textstyle\sum U_j] &= \sum_{k \neq i, i+1} F_k(kp) + F_{i+1}(ip) + F_i((i+1)p). \\
\mathrm{E}_S[\textstyle\sum U_j] - \mathrm{E}_{S'}[\textstyle\sum U_j] &= (F_{i+1}((i+1)p) - F_{i+1}(ip)) - (F_i((i+1)p) - F_i(ip)) \\
&= \int_{ip}^{(i+1)p} (f_{i+1}(x) - f_i(x))dx \\
&\geq 0.
\end{aligned}$$

Thus S' will be at least as good S, and repeated interchanges will result in the specified ordering. □

An Example

Consider the following three jobs.

jobs	1	2	3
p_j	2	2	2
D_j uniform over	[0,6]	[0,8]	[0,12]

The conditions of the theorem hold, and thus the sequence 1-2-3 is optimal with $\mathrm{E}[\sum U_j] = 8/6$. If however the processing times are doubled, then the decreasing density function order condition does not hold. In this case, the sequence 1-2-3 has $\mathrm{E}[\sum U_j] = 16/6$, while the optimal sequence 2-3-1 has $\mathrm{E}[\sum U_j] = 13/6$.

6.1.2 Expected Number of Tardy Job Classes

As in the deterministic case, it is harder to prove the optimality of sequences in the job class scheduling problem when due dates are random and the objective is to minimize the expected number of tardy job classes $\mathrm{E}[\sum \Upsilon_k]$. Because no job classes are tardy if and only if all jobs are early, ordering jobs so that due dates are in increasing stochastic order, if possible, will minimize the probability that any job class is tardy. We can also show the following.

Theorem 2 *If jobs have identical processing times $= p$, and identically distributed due date random variables, scheduling jobs in increasing order of job class set size will minimize the expected number of tardy job classes.*

This theorem is equivalent to the deterministic case in which scheduling jobs in the job classes with least aggregate processing time first will minimize $\sum \Upsilon_j$ when all jobs have identical due dates.

Proof: The proof is again by interchange. Assume a schedule S with two job classes K_a and K_b, $|K_a| < |K_b|$, whose jobs are not in the specified order. Let us label their jobs $1, \ldots, m$ in their scheduled order in S, where $m = |K_a| + |K_b|$, and denote their starting times in schedule S by $s_1, \ldots, s_m$. Given the common due date distribution function F, the probability that a job starting at time s_j is on time is $\overline{F}(s_j)$. Define sets $A = \{j \in K_a : j \leq |K_a|\}$ and $A' = \{j \in K_a : j > |K_a|\}$, and define sets $B = \{j \in K_b : j > |K_a|\}$ and $B' = \{j \in K_b : j \leq |K_a|\}$. Note that $|A'| \leq |B'|$ (and thus $|A| < |B|$), and that, for any $i \in A'$ and $h \in B'$, $s_i > s_h$.

Now consider the schedule S' in which the jobs from class K_a are scheduled at times $s_1, \ldots, s_{|K_a|}$, while the jobs from class K_b are scheduled at times $s_{|K_a|+1}, \ldots, s_m$. In schedule S', the jobs of A' have exchanged places with jobs of B'. Then, denoting the expected number of on-time job classes as O, we have:

$$
\begin{aligned}
O_S &= \sum_{k \neq a,b} \mathrm{E}[\Upsilon_k] + \prod_{j \in A} \overline{F}(s_j) \prod_{j \in A'} \overline{F}(s_j) + \prod_{j \in B} \overline{F}(s_j) \prod_{j \in B'} \overline{F}(s_j). \\
O_{S'} &= \sum_{k \neq a,b} \mathrm{E}[\Upsilon_k] + \prod_{j \in A} \overline{F}(s_j) \prod_{j \in B'} \overline{F}(s_j) + \prod_{j \in B} \overline{F}(s_j) \prod_{j \in A'} \overline{F}(s_j). \\
O_{S'} - O_S &= \prod_{j \in A} \overline{F}(s_j) \left(\prod_{j \in B'} \overline{F}(s_j) - \prod_{j \in A'} \overline{F}(s_j) \right) - \prod_{j \in B} \overline{F}(s_j) \left(\prod_{j \in B'} \overline{F}(s_j) - \prod_{j \in A'} \overline{F}(s_j) \right).
\end{aligned}
$$

Because $s_i \leq s_h \; \forall i \in A, h \in B$ (and thus $\overline{F}(s_i) \geq \overline{F}(s_h)$), and because $|A| < |B|$,

$$\prod_{j \in A} \overline{F}(s_j) - \prod_{j \in B} \overline{F}(s_j) \geq 0.$$

Likewise, because $\overline{F}(s_h) \geq \overline{F}(s_i)$ for all $i \in A', h \in B'$, and because $|A'| = |B'|$,

$$\prod_{j \in B'} \overline{F}(s_j) - \prod_{j \in A'} \overline{F}(s_j) \geq 0.$$

Thus $O_{S'} \geq O_S$, and schedule S' has a lower expected number of tardy job classes. By repeated interchange, the specified ordering is achieved.□

6.2 Multiple Proposals

Thus far, in this chapter, we have considered how to use limited available information about remote agents for constructing one proposal which is most likely to be acceptable. However, when human intervention is required for negotiation, each cycle of the negotiation process may require a relatively long time, not due to proposal processing but to user availability. If this is the case, then using multiple proposals for each negotiation cycle may be useful for reducing the time requires to reach a solution.

Sen and Durfee have investigated the question of multiple proposals in the distributed meeting scheduling domain[61], where communication cost is a main concern, and where there is a tradeoff between the number of iterations required for agreement and the number of proposals communicated during each iteration. Here, we assume that reducing the number of iterations is a much greater concern than the actual communication cost, because each iteration may require a relatively long time. In either case, for a given number of proposals to be communicated during one iteration, it is very desirable to maximize the chances of mutual acceptance of at least one of the proposals by all relevant parties.

Using multiple proposals for one negotiation cycle is fundamentally different from using one proposal for each of several negotiation cycles. In the latter case, later proposals make use of knowledge gained through replies to earlier proposals. Multiple proposals at the same negotiation stage are determined using the same knowledge. Also, while a single proposal should be the proposal most likely to be accepted, multiple proposals should not necessarily include a set of proposals most likely to be accepted. Consider an example with three proposals $\wp_1$, $\wp_2$, and $\wp_3$, where the event that $\wp_i$ is acceptable is A_i. The probability that at least one of the three is acceptable is

$$P(A_1 \text{ or } A_2 \text{ or } A_3) = P(A_1) + P(A_2|\overline{A}_1) + P(A_3|\overline{A}_1\overline{A}_2).$$

When proposals are made so that jobs will complete before uncertain deadlines, then the random variables A_i are likely not to be independent. While three very similar proposals may each have a high probability of being acceptable, proposing all three may not be as good as proposing one of the three, and two others less likely to be acceptable but more different from the first and each other.

6.2.1 Possible Methods

We are interested in simple methods for producing a good set of proposals to be considered in the same negotiation cycle. Given a set of operations, their processing times precedence deadline distribution functions, and one possible proposal, Bayesian analysis might be used to determine the precedence deadline distributions on the condition that the first proposal is not accepted. These distributions might then be used for creating a new proposal which maximizes the probability of acceptance in the case that the first proposal is not accepted. However, even if the original distributions are independent, these new distributions necessarily are not independent. Thus, using such probabilities may be very hard.

We consider some much simpler alternatives. These involve using random guesses using available information. These methods are simple, and thus appropriate for situations in which proposals are needed cheaply and quickly. However, future work will investigate more systematic methods of determining proposal sets.

Proposals without Any Useful Information

First we consider the case in which no useful information is available about the precedence requirements of other cells (other than that they exist). Local operations have identical processing times, and have independent and identically distributed precedence deadline variables. Assuming the problem is a sequencing problem for one machine, all sequences are equally likely to be acceptable (that is, each sequence is just as likely to meet all of its precedence deadlines). Thus, if only one proposal is needed, picking among possible proposals at random is as good a method as any.

If several proposals are needed, picking several possibilities at random may be a good method. However, two proposals chosen at random may be very similar. Thus, we also consider a method that tries to choose a set of proposals that are different from each other, described as follows. If n proposals are needed, one sequence is randomly chosen. The

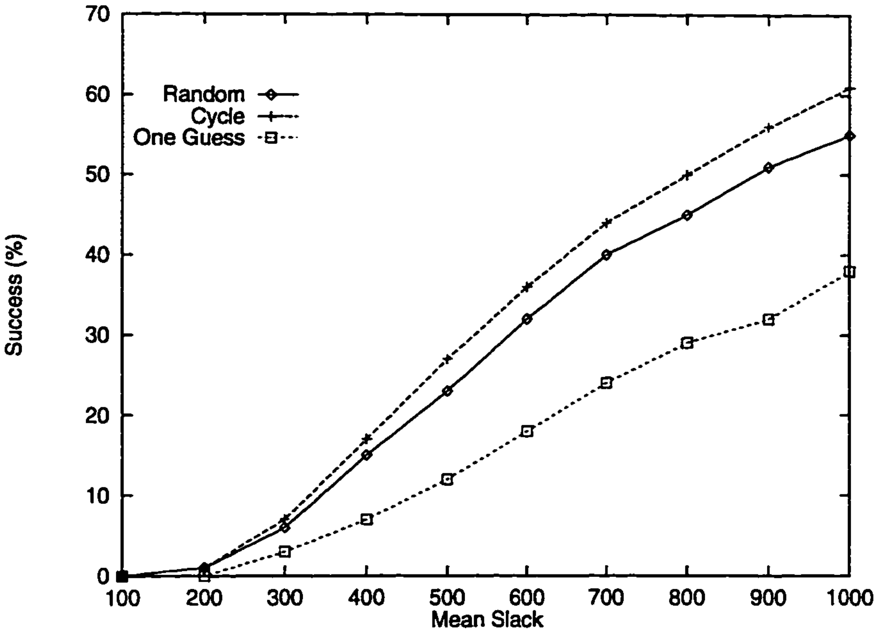


Figure 6.1: Success of Sequence Proposals vs. Slack Time.

set of operations is then divided into n disjoint subsets, $S_1, \ldots, S_n$, for which, all of the operations in S_i are earlier in the initial sequence than all of the operations in S_{i+1}, for $i = 1, \ldots, n$. The other $n - 1$ proposed sequences are then produced by 'cycling through' the initial sequence; i.e., the ith sequence will schedule operations from S_i first, then those of $S_{i+1}, \ldots, S_n, S_1, \ldots, S_{i-1}$.

Figures 6.1 and 6.2 shows the performance of these two methods for a large number of simulations. For these simulations, slack time random variables are uniformly distributed, where slack time is the difference between the actual precedence deadline and the processing time of an operation. Success is achieved when, for at least one proposed sequence, each operation is completed before its precedence deadline. Figure 6.1 shows success versus mean slack time, where the number of guesses for each method is three. Figure 6.2 shows success verses the number of guesses, where the mean slack time is 1000. The cycling method (Cycle) performs somewhat better than the random method (Random); the advantage is significant but not dramatic. In each figure, the One Guess curve shows how successful one random guess is.

Proposals with Uncertain but Useful Information

Now we allow some information that can be used to differentiate among the operations. Processing times are no longer identical, and precedence deadline variables are not iden-

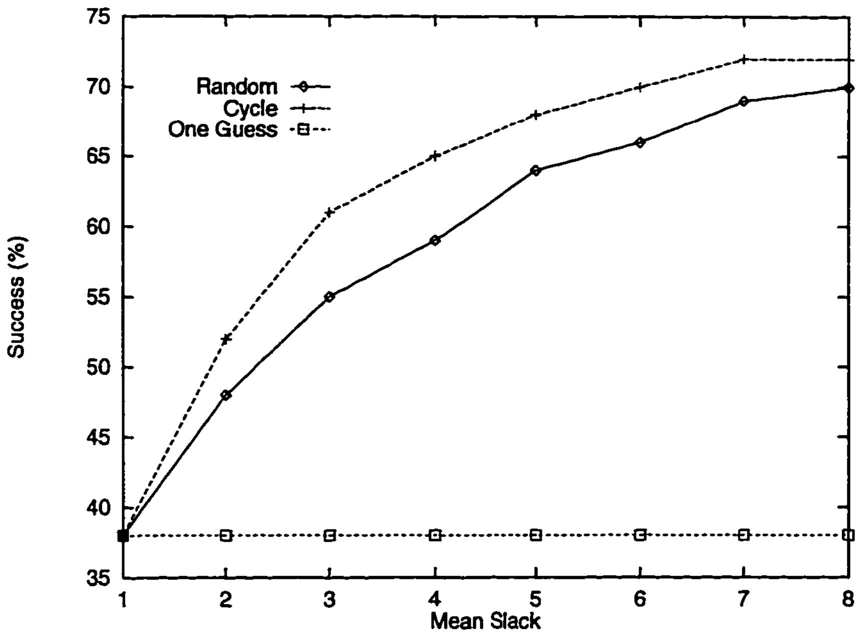


Figure 6.2: Success of Sequence Proposals vs. Number of Guesses.

tically distributed. While precedence deadline variables are not necessarily stochastically ordered or density-function ordered, we can still use the results of Crabill and Maxwell, and of Theorem 1, to choose a good initial proposal. Expected precedence deadlines can be used as operation due dates, and EDD ordering can be used to construct a good initial sequence proposal.

If several proposals are needed, the random or cycle methods previously described can be used to produce additional proposals. These methods, however, do not take advantage of the additional information about operation processing time and precedence deadline distributions. The cycle method in particular may make very little sense if certain jobs should always appear early in the schedule. A method that uses the available information and also may produce sufficiently different proposals is, for each proposal, to randomly select operation precedence deadlines using the known distributions. In a new proposal, after the first best guess proposal, each operation is assigned a due date randomly, using the probability distributions of that operation's precedence deadline variable. EDD ordering is then used to construct a sequence proposal.

Figures 6.3 and 6.4 show the performance of this method (Random 2) versus the previously described random sequencing method (Random 1), the one-proposal best guess method using expected precedence deadline values (Best Guess), and the actual proportion of sched-

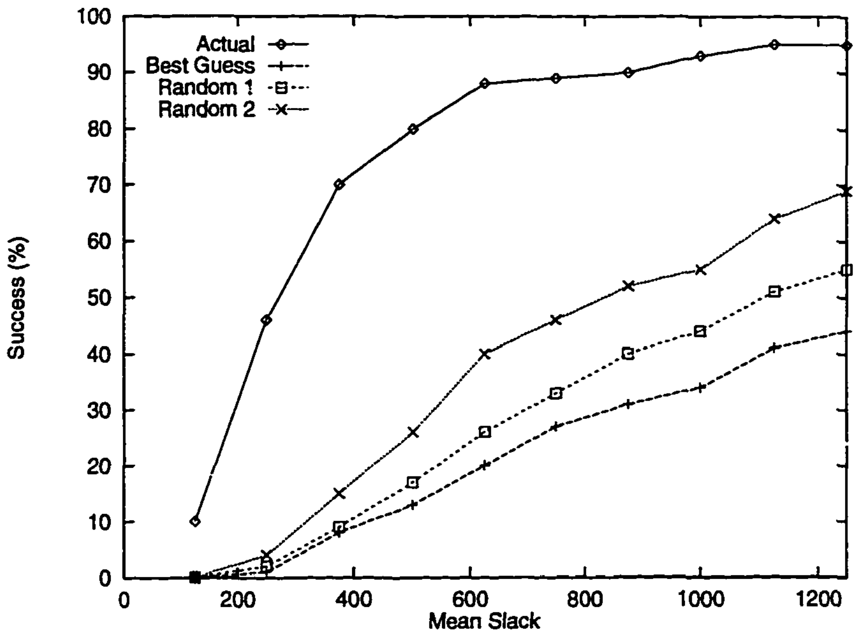


Figure 6.3: Success of Sequence Proposals vs Slack Time.

ules which have solutions (Actual). Mean operation processing time is 10; the slack time mean for each operation is uniformly distributed over 0 to twice the overall slack time mean, and operation slack time is uniformly distributed over 0 to twice its slack time mean. Figure 6.3 shows the proportion of successes versus the overall mean slack time, where the Random 1 and Random 2 methods each use 10 guesses. Figure 6.4 shows the proportion of successes versus the number of random guesses, where the overall mean slack time is 750 (Actual and Best Guess do not change with the number of guesses). These results show the utility of using available information for random guessing, and the advantage of multiple proposals over on best-guess proposal.

6.3 Summary

In sum, this chapter has investigated issues about negotiation for polite rescheduling, when effects of local actions on remote agents is uncertain. We have presented original observations and theorems about the maximization of expected acceptability and expected number of remote cells disrupted by local rescheduling decisions. We have also investigated methods for producing multiple proposals for one negotiation cycle, when local information is incomplete, and when the goal is the greatest probability that at least one proposal is acceptable. The results of this chapter can be used to design a more sophisticated negotiator module for PRIAM, that can make more intelligent decisions when local information about

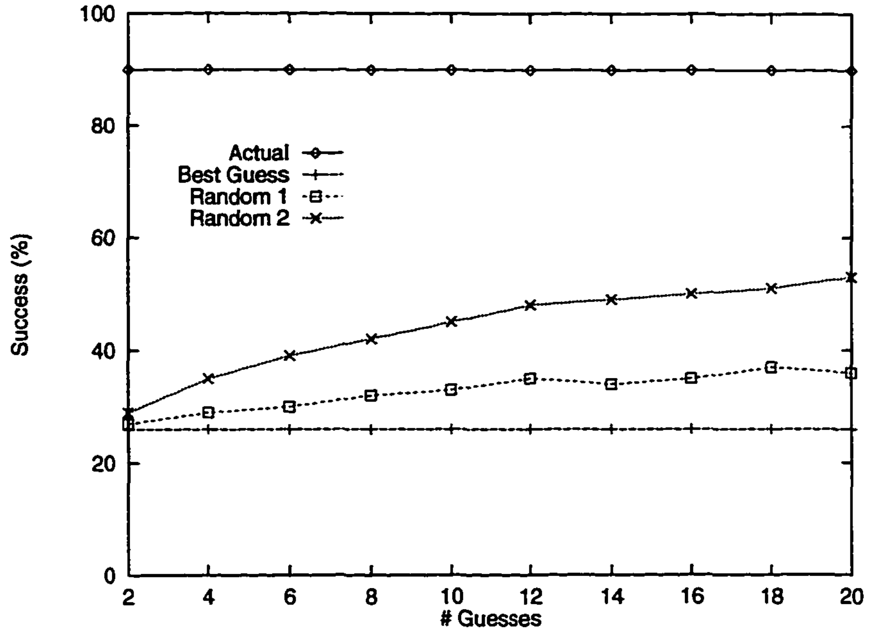


Figure 6.4: Success of Sequence Proposals vs. Number of Guesses.

precedence deadlines is incomplete, and when the environment for negotiation favors having multiple proposals for each negotiation round.

In order to pursue these problems further, we would like as future work to consider more sophisticated ways of generating proposals. These might require a measure of "diversity" in sets of proposals, gauging how different proposals are from one another. However, diversity among proposals may not necessarily be good if useful information about proposal quality is not considered. We would also like to consider how information gathered through negotiation can be used by a negotiator to build models of other cells. For example, if a certain cell has recently been rejecting all proposals, we might assume that it is currently very busy, and that future proposals are likely to be rejected. Such information is likely to be collected anyway, for performance evaluation of the manufacturing system; making use of such information will likely allow for more efficient negotiation.

CHAPTER 7

Conclusion

Distributed schedule revision is an important but relatively unexplored field that is relevant for any domain in which multiple agents with separate schedules interact. Work on this problem can be applied to computer-integrated manufacturing and other resource allocation problems. In this dissertation, we have presented polite rescheduling, a new approach to a distributed schedule revision in response to schedule disruptions in a distributed manufacturing system. Our approach takes into consideration the possibility that responding to a disruption in one part of the system may cause disruptions in other parts of the system. It thus attempts to respond to disruptions local to one manufacturing cell so that other cells are disrupted as little as possible.

We have applied this approach to various problems of schedule revision, including that of tool management and scheduling, job shop schedule revision with the goal of minimizing the total makespan, job shop schedule revision with the goal of minimizing and containing disruption propagation, and job shop scheduling when information about remote cell requirements is uncertain. Our simulation results show the advantages of using a scheduling algorithm that emphasizes interaction with other cells over other scheduling considerations, and demonstrate the advantages of using negotiation for determining priorities in local scheduling decisions.

Our proposed polite rescheduling architecture, PRIAM, investigates ways to implement polite rescheduling. We have examined methods to be used by the negotiator module for determining priorities, and for negotiating efficiently. We have also proposed methods to be used by the negotiator module for using these priorities and other scheduling constraint information to construct useful schedules.

7.1 Contributions

The main contribution of this work is the investigation of the field of distributed schedule revision, the formulation of the polite rescheduling approach to this problem, and the application of this approach to the problems of managing the reallocation of shared resources and the rescheduling of jobs in a multi-agent manufacturing system. In this context, contributions include:

- a new approach to schedule revision in a distributed environment, a problem that hitherto has not been treated in much depth;
- an application of polite rescheduling in the tool management domain, in which we show that polite rescheduling using local knowledge of portions of tool schedules, performs close to optimal methods using global knowledge of tool schedules for a tool scheduling problem, and close to good methods using global knowledge for a tool borrowing problem;
- an application of polite rescheduling in the job-shop scheduling domain, in which we show that different levels of local knowledge of schedule constraints allow significantly different rescheduling performance, and that use of local knowledge of schedule constraints in a flowshop-like job shop allows rescheduling performance close to that of optimal methods using global knowledge of the schedule;
- a description of a PRIAM, an architecture for rescheduling in a multi-agent job shop environment, that both provides new schedules in response to new scheduling constraints, and determines what goals and priorities are important for schedule revision, through use of information about other cells and negotiation with other cells;
- an investigation of issues in determining scheduling priorities and negotiation strategies, and a consideration of the problem of generating schedule proposals when scheduling information is uncertain.

7.2 Future Directions

We have presented the distributed schedule revision problem, and have shown that polite rescheduling is an appropriate and useful approach to this problem. We have briefly discussed future work in each technical chapter. While we feel that future work in this area is best pursued in the context of a real and appropriate domain, there are certain general future directions we believe are important. We will discuss these in the context of the PRIAM architecture.

- The negotiator in our simulations uses a very simplistic method for determining priorities and generating proposals. It is important that the negotiator have a more sophisticated representation of conditions at other cells, and perhaps more sophisticated knowledge about the structure of the schedule (even when global scheduling information remains distributed throughout the system). An appropriate implementation of the negotiator may be a *Truth Maintenance System* (TMS), in which knowledge about the global schedule in general and about other cells in particular is represented as a collection of facts, either believed or not believed. Rules determine how information available locally or obtained through communication support or undermine belief in various facts. There has been research on the subject of distributed TMS's [45, 34]; that work suggests a possible structure for the negotiator module. As mentioned in Chapter 6, the negotiator also may need efficient ways for generating multiple proposals for one negotiation round, that consider how likely at least one proposal will be acceptable. If, as seems likely, analysis of the actual probabilities is too hard, this may require consideration of how diverse a proposal set is.

- We would like the rescheduler module to be able to consider both jobs and shared resources when scheduling, integrating the work we have done on tool scheduling (in Chapter 3) and job scheduling (in Chapters 4 and 5). Likewise, the rescheduler will be more useful with a wider range of scheduling methods. Because we assume that time for rescheduling and local computational resources may be limited, we have considered only fast heuristic rescheduling methods. However, the time available for rescheduling may range over different values; the negotiator may provide as a scheduling parameter the time allowed for the rescheduling task. Thus, the rescheduler may need to reason about which scheduling methods are appropriate to use. Reasoning about similar tasks has been investigated in the "Design-to-Time" scheduling of Garvey and Lesser [26], and in Musliner's CIRCA [49], and any-time or otherwise iterative scheduling algorithms have been proposed by Zweben [74] and other researchers on scheduling repair.

- We have not discussed issues relating to the user interface of an agent in a multi-agent schedule revision system, but these issues are very important. Many practical scheduling systems are advisory rather than decision-making systems, and the goal of scheduling automation is often to increase the effectiveness of the human scheduler, rather than to replace this person. In a distributed manufacturing system, an automated scheduling system can assist the user not only by notification of scheduling constraint violations and presentation of various scheduling options and the explanations behind them, but also by suggesting ways in which negotiation can be more efficient. Mediation agents for negotiation between human agents is considered by Sycara's PERSUADER [68] and elsewhere; in PRIAM, we would desire the negotiator modules to act as a mediating buffer among the human users, which attempt to identify the important issues in a particular negotiation session in order to facilitate its quick resolution.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] J. Adams, E. Balas, and D. Zawack. The shifting bottlenect procedure for job shop scheduling. *Management Science*, 34(3):391–401, 1988.

[2] K. R. Baker. *Introduction to Sequencing and Scheduling*. Wiley, 1974.

[3] S. Balasubramanian and D. H. Norrie. A multi-agent intelligent design system integrating manufacturing and shop-floor control. In *Proc. of the First International Conference on Multi-Agent Systems*, pages 3–8, 1995.

[4] J. C. Bean et al. Matchup scheduling with multiple resources, release dates and disruptions. *Operations Research*, 39(3):470–483, May-June 1991.

[5] J. H. Blackstone, D. T. Phillips, and G. L. Hogg. A state-of-the-art survey of dispatch rules for manufacturing job shop operations. *International Journal of Production Research*, 20(1):27–45, 1982.

[6] A. H. Bond and L. Gasser. An analysis of problems and research in DAI. In A. H. Bond and L. Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 3–35. Morgan Kaufmann, San Mateo, 1988.

[7] Geoff Buxey. Production scheduling: Practice and theory. *European Journal of Operations Research*, 39:17–31, 1989.

[8] S. Cammarata et al. Strategies of cooperation in distributed problem solving. In A. H. Bond and L. Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 102–105. Morgan Kaufmann, San Mateo, 1988.

[9] S. E. Conry et al. Multistage negotiation for distributed constraint satisfaction. *IEEE Trans. on Systems, Man, and Cybernetics*, 21(6):1462–1477, November 1991.

[10] S. E. Conry, R. A. Meyer, and R. P. Pope. Mechanisms for assessing nonlocal impact of local decisions in distributed planning. In L. Gasser and M. N. Huhns, editors, *Distributed Artificial Intelligence Volume 2*, pages 245–258. Morgan Kaufmann, San Mateo, 1989.

[11] Thomas B. Crabill and William L. Maxwell. Single machine sequencing with random processing times and random due-dates. *Naval Research Logistics Quarterly*, 16:549–554, 1969.

[12] R. L. Daniels and P. Kouvelis. Robust scheduling to hedge against processing time uncertainty in single-stage production. *Management Science*, 41(2):363–376, 1995.

[13] R. Davis and R. G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20:63–109, 1983.

[14] Mark Drummond, Keith Swanson, and John Bresina. Robust scheduling and execution for automatic telescopes. In Monte Zweben and Mark S. Fox, editors, *Intelligent Scheduling*, pages 341–370. Morgan Kaufmann, 1994.

[15] N. A. Duffie et al. Fault-tolerant heterarchical control of heterogeneous manufacturing system entities. *Journal of Manufacturing Systems*, 7(4):315–328, 1988.

[16] E. H. Durfee, V. R. Lesser, and D. D. Corkill. Coherent cooperation among communicating problem solvers. *IEEE Trans. on Computers*, pages 1275–1291, November 1987.

[17] D. L. Eager et al. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. on Software Engineering*, pages 662–675, May 1986.

[18] E. Ephrati, M. E. Pollack, and J. S. Rosenschein. A tractable heuristic that maximizes global utility through local plan combination. In *Proc. of the First International Conference on Multi-Agent Systems*, pages 94–101, 1995.

[19] Eithan Ephrati and Jeffrey S. Rosenschein. Constrained intelligent action: Planning under the influence of a master agent. In *Proc. AAAI-92*, pages 263–8, 1992.

[20] Eithan Ephrati, Gilad Zlotkin, and Jeffrey S. Rosenschein. A non-manipulable meeting scheduling system. In *Proc. of the 13th International Distributed AI Workshop*, pages 105–125, 1994.

[21] Klaus Fischer et al. A model for cooperative transportation scheduling. In *Proc. of the First International Conference on Multi-Agent Systems*, pages 109–116, 1995.

[22] M. S. Fox et al. Constrained heuristic search. In *Int'l Joint Conf. Artificial Intelligence*, pages 309–315, 1989.

[23] Mark S. Fox. ISIS: A retrospective. In Monte Zweben and Mark S. Fox, editors, *Intelligent Scheduling*, pages 3–28. Morgan Kaufmann, 1994.

[24] Mark S. Fox and Stephen F. Smith. ISIS - a knowledge-based system for factory scheduling. *Expert Systems*, 1(1):25–49, 1984.

[25] S. French. *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*. Wiley, 1982.

[26] Alan J. Garvey and Victor R. Lesser. Design-to-time real-time scheduling. *IEEE Trans. on Systems, Man, and Cybernetics*, 23(6):1491–1502, November 1993.

[27] L. Gasser and M. N. Huhns, editors. *Distributed Artificial Intelligence Volume 2*. Morgan Kaufmann, San Mateo, 1989.

[28] P. J. Gmytrasiewisz and E. H. Durfee. Truth, lies, belief and disbelief in communication between autonomous agents. In *Proceedings Eleventh Internat'l Wrkshp on DAI*, pages 107–125, 1992.

[29] C. V. Goldman and J. S. Rosenschein. Emergent coordination through the use of cooperative state-changing rules. In *Proc. of the 12th International Distributed AI Workshop*, pages 171–185, 1993.

[30] S. C. Graves. A review of production scheduling. *Operations Research*, 29(4):646–675, 1981.

[31] Khosrow C. Hadavi. ReDS: A real time production scheduling system from conception to practice. In Monte Zweben and Mark S. Fox, editors, *Intelligent Scheduling*, pages 581–604. Morgan Kaufmann, 1994.

[32] D. J. Hoitomt et al. Distributed scheduling of job shops. In *Proceedings 1991 IEEE Int. Conf. on Robotics and Automation*, pages 1067–1072, 1991.

[33] Marcus J. Huber and Edmund H. Durfee. Deciding when to commit to action during observation-based coordination. In *Proc. of the First International Conference on Multi-Agent Systems*, pages 163–170, 1995.

[34] M. N. Huhns and D. M. Bridgland. Multiagent truth maintenance. *IEEE Trans. on Systems, Man, and Cybernetics*, 21(6):1437–1445, November/December 1991.

[35] Mark D. Johnston and Steven Minton. Analyzing a heuristic strategy for constraint-satisfaction and scheduling. In Monte Zweben and Mark S. Fox, editors, *Intelligent Scheduling*, pages 257–289. Morgan Kaufmann, 1994.

[36] A. C. Jones and C. R. McClean. A proposed hierarchical control model for automated manufacturing systems. *Journal of Manufacturing Systems*, 5(1):15–25, 1986.

[37] S. Kambhampati et al. Integrating general purpose planners and specialized reasoners: Case study of a hybrid planning architecture. *IEEE Trans. on Systems, Man, and Cybernetics*, 23(6):1503–1518, November 1993.

[38] H. Kise, T. Ibaraki, and H. Mine. A solvable case of the one-machine scheduling problem with ready and due times. *Operations Research*, 26:121–126, 1978.

[39] K. Krishna, K. Ganeshan, and D. Janaki Ram. Distributed simultaed annealing algorithms for job shop scheduling. *IEEE Trans. on Systems, Man, and Cybernetics*, 25(7):1102–1109, July 1995.

[40] V. Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, pages 32–44, Spring 1992.

[41] Amy L. Lansky. Localized representation and planning. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 670–674. Morgan Kaufmann, 1990.

[42] JyiShane Liu and Katia Sycara. Distributed problem solving through coordination in a society of agents. In *Proc. of the 13th International Distributed AI Workshop*, pages 190–206, 1994.

[43] W. W. Luggen. *Flexible Manufacturing Cells and Systems.* Prentice Hall Inc., 1991.

[44] Q. Y. Luo, P. G. Henry, and J. T. Buchanan. Strategies for distributed constraint satisfaction problems. In *Proc. of the 13th International Distributed AI Workshop*, pages 207–221, 1994.

[45] C. L. Mason and R. R. Johnson. Datms: A framework for distributed assumption based reasoning. In L. Gasser and M. N. Huhns, editors, *Distributed Artificial Intelligence Volume 2*, pages 293–318. Morgan Kaufmann, San Mateo, 1989.

[46] S. Minton and A. B. Philips. Applying a heuristic repair method to the hst scheduling problem. In *Proceedings DARPA Wrkshp on Planning, Scheduling and Control*, pages 215–219, 1990.

[47] Kazuo Miyashita and Katia Sycara. Adaptive case-based control of schedule revision. In Monte Zweben and Mark S. Fox, editors, *Intelligent Scheduling*, pages 291–308. Morgan Kaufmann, 1994.

[48] J. M. Moore. An *n* job, one machine sequencing algorithm for minimizing the number of late jobs. *Management Science*, 15:105–109, 1968.

[49] D. J. Musliner, E. H. Durfee, and K. G. Shin. Circa: A cooperative intelligent real time control architecture. *IEEE Trans. on Systems, Man, and Cybernetics*, 23(6):1561–1574, November 1993.

[50] P. A. Newman. Scheduling in CIM systems. In A. Kusiak, editor, *Artificial Intelligence Implications for Computer Integrated Manufacturing*, pages 361–402. IFS Ltd., 1988.

[51] Y. Nishibe et al. Effects of heuristics in distributed constraint satisfaction problems. In *Proceedings Eleventh Internat'l Wrkshp on DAI*, pages 285–302, 1992.

[52] Masayuki Numao. Development of a cooperative scheduling system for the steel-making process. In Monte Zweben and Mark S. Fox, editors, *Intelligent Scheduling*, pages 607–628. Morgan Kaufmann, 1994.

[53] S. S. Panwalkar and W. Iskander. A survey of scheduling rules. *Operations Research*, 25(1):45–61, 1977.

[54] H. V. D. Parunak. Distributed artificial intelligence systems. In A. Kusiak, editor, *Artificial Intelligence Implications for Computer Integrated Manufacturing*, pages 225–251. IFS Ltd., 1988.

[55] H. V. D. Parunak. Characterizing the manufacturing scheduling problem. *Journal of Manufacturing Systems*, 10(3):241–258, 1991.

[56] Michael Pinedo. Stochastic scheduling with release dates and due dates. *Operations Research*, 31:559–572, 1983.

[57] Michael Pinedo, editor. *Scheduling: Theory, Algorithms, and Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[58] R. Pope et al. Distributing the planning process in a dynamic enviroment. In *Proceedings Eleventh Internat'l Wrkshp on DAI*, pages 317–332, 1992.

[59] Michael J. Prietula et al. MACMERL: Mixed-initiative scheduling with coincident problem spaces. In Monte Zweben and Mark S. Fox, editors, *Intelligent Scheduling*, pages 655–682. Morgan Kaufmann, 1994.

[60] K. Ramamritham et al. Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Trans. on Computers*, pages 1110–1122, August 1989.

[61] Sandip Sen and Edmund H. Durfee. A formal analysis of communication and commitment in distributed meeting scheduling. In *Proc. of the 13th International Distributed AI Workshop*, pages 333–344, 1994.

[62] Sandip Sen and Edmund H. Durfee. Unsupervised surrogate agents and search bias change in flexible distributed scheduling. In *Proc. of the First International Conference on Multi-Agent Systems*, pages 336–342, 1995.

[63] Yoav Shoham and Moshe Tennenholtz. On the synthesis of useful social laws for artificial agent societies. In *Proc. of the Tenth National Conference on Artifical Intelligence*, 1992.

[64] Stephen Smith. OPIS: A methodology and architecture for reactive scheduling. In Monte Zweben and Mark S. Fox, editors, *Intelligent Scheduling*, pages 29–66. Morgan Kaufmann, 1994.

[65] R. Steeb et al. Architectures for distributed intelligence for air fleet control. In A. H. Bond and L. Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 90–101. Morgan Kaufmann, San Mateo, 1988.

[66] Keith Swanson, John Bresina, and Mark Drummond. Robust telescope scheduling. In *i-SAIRAS'94 Planning and Scheduling Workshop*, 1994.

[67] K. Sycara et al. Distributed constrained heuristic search. *IEEE Trans. on Systems, Man, and Cybernetics*, 21(6):1446–1461, November 1991.

[68] K. P. Sycara. Multiagent compromise via negotiation. In L. Gasser and M. N. Huhns, editors, *Distributed Artificial Intelligence Volume 2*, pages 119–138. Morgan Kaufmann, San Mateo, 1989.

[69] T. K. Tsukada and K. G. Shin. Polite rescheduling: Responding to schedule disruptions in a distributed manufacturing system. In *Proceedings 1994 IEEE Int. Conf. on Robotics and Automation*, pages 1986–91, 1994.

[70] Adam Walker and Michael Wooldridge. Understanding the emergence of conventions in multi-agent systems. In *Proc. of the First International Conference on Multi-Agent Systems*, pages 384–389, 1995.

[71] D. E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*, chapter 11. Replanning during execution. Morgan Kaufmann, San Mateo, 1988.

[72] D. J. Williams and P. Rogers, editors. *Manufacturing Cells: Control, Programming and Integration.* Butterworth-Heinemann Ltd., Oxford, 1991.

[73] M. Yokoo et al. Distributed constraint satisfaction for formalizing distributed problem solving. In *IEEE Proc. 12th Int. Conf. on Distr. Computing Systems*, pages 614–21, 1992.

[74] M. Zweben et al. Anytime rescheduling. In *Proceedings DARPA Wrkshp on Planning, Scheduling and Control*, pages 251–259, 1990.

[75] Monte Zweben et al. Scheduling and rescheduling with iterative repair. In Monte Zweben and Mark S. Fox, editors, *Intelligent Scheduling*, pages 241–255. Morgan Kaufmann, 1994.

[76] Monte Zweben and Mark S. Fox, editors. *Intelligent Scheduling.* Morgan Kaufmann, San Fransisco, 1994.