Planning and Scheduling of Software Manufacturing Projects

Ali Safavi

CMU-RI-TR-91-04

Center for Integrated Manufacturing Decision Systems
The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

March 1991

This research was sponsored in part by DARPA under contract number F30602-88-C-0001.

Table of Contents

List of Figures

List of Tables

Abstract

In today's highly competitive and constantly growing market for software products, planning and scheduling of large software projects has become a bottleneck to increasing production productivity [133]. This work is to investigate the mechanisms required to support software project planning and scheduling (SPPS).

Our approach is to

1. define SPPS as a reactive process that involves human negotiation, and
2. develop a heuristic search model, that is consistent with the negotiation process, to improve an existing schedule by incrementally revising it.

The main contribution of this thesis is that it represents the first major effort in building a *problem solving model* for SPPS that accomodates the dominant characteristics of SPPS. Our problem solving model is based on the previous results in social analysis of computing, operations research in manufacturing, artificial intelligence in manufacturing planning and scheduling, and the traditional approaches to planning in artificial intelligence, and extends the techniques that have been developed by them in dealing with SPPS.

We demonstrate the sufficiency of the model that has been developed on specific test cases that reflect actual software project planning and scheduling circumstances. A program called NEGOPRO that uses our basic model to support SPPS in large software projects has been implemented.

Acknowledgements

In the course of the last 5 years, I relied on many good people, too many for me to mention the names of all here, who supported me while I was working on this dissertation.

I would like to thank Dr. Les Gasser for his constant guidance, support, suggestions, and encouragement. His understanding and outlook on human negotiation helped shape the contributions of this dissertation. I also can not say enough about the impact of numerous mind-expanding discussions with Dr. Stephen Smith, and also the corrections that he made on all earlier drafts of this dissertation.His insight in scheduling and planning are pillars supporting the research reported here.

I would like to thank Dr. Mark Fox for his many invaluable discussions, and also for providing the financial support that was vital to the completion of this research. I would also like to acknowledge the importance of the financial support that Dr. David Wile and Dr. Bob Balzer provided during the earlier years of my research while I was at USC/ISI. The interest and encouragement that Dr. George Bekey and Dr. Dan O'Leary provided were also a significant factor.

The inspiring atmosphere created by the members of the OPIS and CORTES teams at the center for integrated manufacturing decisions systems of CMU (especially Nicola Muscettola, Katia Sycara, and Norman Sadeh) over the years that I stayed there was also a significant factor in the successful completion of my dissertation.

I would like to thank Dr. Walt Scacchi, Dennis Allard, and Don Cohen at USC/ISI for their friendship, and critical evaluation of my ideas during the earlier years of my research.

I am also extremely greatful to my friends Charles Marshal at Digital Equipment Corp and Hamid Nabavi at XEROX for their many mind-expanding discussions throughout my research. I would also like thank Dr. Barry Boehm at TRW for his valuable comments.

Lastly, I would like to acknowledge the moral and financial support, constant encouragement, as well as love, patience, and sacrifices of my parents to whom I dedicate this work.

Chapter 1
Introduction

In today's highly competitive and constantly growing market for software products, planning and scheduling of large software projects has become a bottleneck to increasing production productivity [133]. Although in the recent years, many tools have been developed that improve the productivity of individuals in developing software, little has been achieved in the area of improving the productivity of planning and scheduling of software projects [121]. This is because previous scheduling models that have been developed fail to address the major issues of SPPS. Software projects are planned and scheduled under an array of conflicting technical and organizational constraints including budget constraints, temporal and resource capacity constraints, tool and staff productivity limitations, organizational rules and regulations, and so forth. The task of planning and scheduling is further complicated by the fact that some of these constraints are negotiable (i.e. can be satisfied to varying degrees) and are subject to a diverse set of preferences[1]. SPPS is distributed among multiple agents each with its own set of resource requirements and involves face-to-face human negotiation between them to resolve the scheduling conflicts (i.e. constraints that can not be satisfied) that arise due to differences in goals, technical judgements, etc [48]. Uncertainty in the budget estimates of a project activity (including the time and resources needed to complete that activity) is another source of problem in SPPS [13]. Human errors only add to the inaccuracy in resource requirement estimates by inflating or deflating them to serve their own personal objectives. Last but not the least, the occurrence of frequent unexpected events (e.g. discovery of a major bug in the product, high rate of staff turnover) can make some aspects of the plan/schedule that has been developed obsolete frequently [91].

In the remainder of this chapter, we provide a definition of SPPS, our approach to solving it, and the contributions of this thesis.

1.1. Definition of SPPS Problem

A client organization is an organization that *orders software systems*. Each *order* consists of a product (e.g. a hardware simulator), a set of feature requirements for that product (e.g. deadline, portability, reliability), the degree which each feature requirement has to be met, and a set of budget constraints. A software development organization is an organization that can carry out orders from several clients at the same time. For every order, a project is created to undertake that order.

Assuming that feasibility studies have been conducted earlier, the creation of a project starts with building a plan for meeting the project requirements. Intuitively, a project plan is a specification of *how* the project product

[1] In the case of negotiable constraint, for instance, these preferences might state the importance of meeting each constraint to varying degrees.

can be produced by employing a set of *primitive* resources that are either readily available inhouse or can be acquired from outside (e.g. software libraries, software tools, developers, workstations) within the budget. For each product, the specification includes names, quantities, and the periods (relative to the beginning of production) that the required resources have to be allocated for the production of the product.

Inside a software project organization, the specification of how a project product can be produced from a set of primitive resources is developed in an incremental fashion. This specification is developed incrementally because the project agents who are made responsible for producing a product specify the production of that product *locally* in terms of the resources that they use *directly* during the production. For instance, if they require the use of a graphic interface software, their specification would include how the graphic interface software would be used in the production but would not include how it would be acquired (or produced). This scheme illustrates how the production of a *final product* of a project can be specified recursively in terms of more primitive project products (also called *intermediate project products*). Furthermore, it shows that what is considered to be a *resource request* to one group of agents would appear as production goal (or *responsibility*) to another group of agents. A graphic illustration of the relationship between a *resource request* and a *responsibility* is provided in Figure 1-1.

Figure 1-1: The Relationship of User Defined Constraints and Resource Requests

Since a project product can be developed in more than one way, alternative project plans (or process plans as named in manufacturing literature) are normally generated and evaluated to choose the best one among them. Project plans are evaluated by studying alternative schedules that can implement them. A software project schedule is a specification that for each product in the plan includes the names, quantities, and absolute periods of the resources that have to be reserved for its production. The resource reservation data can be used to assess how well the project can meet the resource requirements of its plan if that plan is chosen to be implemented. In studying SPPS, we consider both the selection of a project plan and the scheduling of that plan, and frequently refer to their combination as *scheduling*.

A major problem with scheduling of software development projects is that there exists a great deal of uncertainty about their budget estimates (i.e. their resource requirement predictions are frequently inaccurate). As a result, the schedule which is originally constructed is at best a rough approximation of what will happen during execution. Some software project managers go as far as saying that it is not unusual to a software project manager if a schedule that is considered 'good' in the morning of one day becomes a 'bad' schedule and has to be scrapped on the afternoon of that same day, and argue that why should anyone build a schedule for a project that contains so much uncertainty. This uncertainty in budget estimates is largely due to the facts that (1) the process of developing software is not yet well understood [69, 75] (thus leading to inaccurate estimates), and (2) unexpected events such as discovery of a bug late in the project (that nullify schedule assumptions) are common. These unexpected events are not just caused by improper implementation of project procedures (e.g. formal reviews), and can be attributed to politics inside the organization, change in customer requirements, failure of a supplier to deliver a resource by a deadline, and so forth as well.

SPPS is more of a schedule revision problem than a problem of schedule generation (which constitutes the traditional perception). The traditional perception of SPPS gives way to obscuring the need to schedule software projects by characterizing it as a problem of schedule generation. To manage the uncertainty in schedules, they have to be continuously revised over the course of project. This is also supported by studies of actual software development organizations, which have indicated that the majority of scheduling time in a software project is spent on revising a pre-existing schedule [48, 121].

SPPS is also a distributed problem solving process involving a set of agents with conflicting interests [75]. Each agent specifies a set of resource requests in order to satisfy his responsibilities and refuses to commit to a compromise (or to a different resource mix) unless he is convinced that no feasible schedule exists which satisfies his requests. This implies that frequently a schedule can not be found until several rounds of negotiation have taken place each resulting in a more refined set of resource requests. Another way of looking at this, which we adopt here, is that an inconsistent schedule (a schedule that contains conflicts) is repeatedly revised to react to the changes in resource requirements as they become available until a consistent schedule emerges from the process.

Software project schedules evolve through human negotiation, which continues through all phases of SPPS. Since negotiation plays a major role in SPPS, a scheduling model for this domain needs to be consistent with the negotiation process. In fact, some scheduling experts believe that they will be better off using human intuition than automated support during scheduling since automated support involves spending huge amounts of time entering the constantly changing scheduling data for rescheduling purposes. However, the belief that scheduling data changes so frequently is justified only if each organizational unit builds its own schedule without considering how it will affect the schedule of other organizational units. While the crux of rescheduling within a large software development organization lies in selecting the local scheduling decisions that do not trigger new changes in the schedule of other agents, it is difficult for human schedulers to analyze the interaction between the schedules of different agents, and to discover the impact of a local scheduling decision on the schedule of the project as a whole. Therefore, it is important to use a scheduler (other than human) to aid the making and evaluation of scheduling decisions.

Even idealized formulations of SPPS, where reactivity, negotiation, and uncertainty are not modeled, are NP-hard in the general case. This is because the number of possible schedules that can be generated for a given SPPS increases exponentially with the number of products that need to be produced (each product is realized by one activity) as well as with the number of types of primitive resources used during the course of a project [62].

1.2. Existing Solutions

When we reviewed the literature, we found that while the research in building problem solving models for SPPS is at its early stages [64, 9, 85], commercially available project management tools [96] have been in the marketplace for a long time. However, we also found out that these tools lack the knowledge representation and specification language that are needed to specify and model SPPS.

Operations research has developed a set of techniques to be used during the scheduling process. A set of commercially available tools have been developed to operationalize these techniques which include PERT/CPM [70], Monte Carlo [127], and visual aids such as WBS (Work Breakdown Structure) and Gantt chart [96]. Although these tools can be used to estimate the duration of a project and figure out how critical the on-time completion of each production is to the project as a whole, they are not capable of resource allocation. Heuristic techniques are often augmented to PERT/CPM systems to provide resource allocation capability but the effectiveness of this synthesis is limited by the type of constraints that can be considered. Linear programming (LP) [96] can consider a broader set of constraints (i.e. those that can be formulated as linear inequalities) and finds optimal solutions, but it is difficult to formulate many constraints as linear inequalities. Moreover, linear programming is not appropriate for reactive scheduling [50].

PMA [64] is a knowledge-based project management assistant prototype tool which integrates a set of conventional visual and activity network analysis aids with an intelligent data base, and assumes the development of software project plans and schedules is very similar to writing software. Although this restrictive assumption prevents PMA from modeling the problem solving process during software project scheduling (because software planning/scheduling has an organizational dimension as much as it has a technical dimension), PMA does provide a wide-domain logic-based plan and schedule specification language (REFINE) to assist the specification of project plans and schedules. REFINE supports high-level specification of task assignments, milestones, and temporal constraints, but it lacks a problem-solving model to use the knowledge base that is created from the specification of a project in REFINE to build schedules.

Bimson and Boehm [9] also provide a knowledge based project management assistant prototype tool that uses a frame-based knowledge representation scheme for representing project knowledge borrowed from Callisto [119]. The main contribution of the Bimson and Boehm's work is in the area of knowledge representation. However, this is not accompanied by a problem solving model for scheduling that accomodates different types of constraints and preferences, supports negotiation, and manages uncertainty.

In summary, although existing models for SPPS [64, 9, 85] provide rich representation schemes and high-level software project specification languages, they rely on human decision making for scheduling.

1.3. Our Approach

In this thesis, we focus on the development of a problem solving framework for scheduling that accomodates the dominant characteristics of SPPS.

Our approach is to

1. define SPPS as a reactive process that involves human negotiation, and
2. develop a heuristic search model, that is consistent with the negotiation process, to improve an existing schedule by incrementally revising it.

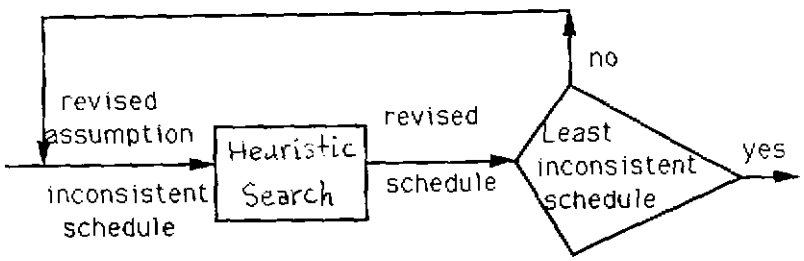


Figure 1-2: Reactive Cycle

Each reactive cycle is triggered by a change in the schedule assumptions which include change in user resource requests or items in the schedule that have deviated from actual execution. In the case of building a schedule for the first time, the assumptions that change (and trigger a reactive cycle) are limited to user resource requests. Each reactive cycle involves finding a schedule for which the penalty of resolving its conflicts[2] (i.e. making it consistent) will be minimal (we will introduce a variable to measure this consistency); see figure 1-2. This is achieved by heuristic search through the space of possible schedules to improve (lower the inconsistency of) a given schedule by incrementally revising it. The space of possible schedules spans along the following dimensions:

1. The available capacity of a resource can vary over time.
2. The same production (activity) can start at different dates.
3. The same production can be carried out with different mixes of capacities of resources. This occurs because some resources are interchangible. For instance, suppose that the allocation of more manpower to a production can reduce the duration of producing a product under certain circumstances. Then two different combinations of capacities of manpower and time can carry out the production of that resource.
4. Different productions might require the same resource. Then, if there is not enough of the resource to satisfy all requirements, a scheduler has to decide which production the resource should be allocated to or whether a product should be preempted from its resource so that the resource can be reallocated to another product.
5. The same product can be developed with different process plans. For instance, a business application software can be developed by acquiring an application generator and hiring a group of specification language experts or by developing the entire application inhouse by hiring high-level language experts and system designers.
6. Product feature requirements can be compromised thus multiplying the number of ways that the requirements of a product can be satisfied. For instance, the testing of a software module might require conducting formal reviews periodically, but it also might require to abandon this requirement if a major cost overrun would occur otherwise.

Once such schedule (a schedule with minimal inconsistency) is found, it can be used to advise the users on the

[2]Conflicts can be relaxed to varying degrees.

constraints that still need to be satisfied in order to make the schedule consistent. The users could then use this advice in conducting negotiation aimed at making the schedule consistent. For instance, they could reconcile the requirements of the constraints that could not be satisfied (to varying degrees), or they could propose alternative process plans under which those constraints are more likely to be satisfied. These changes in turn trigger a new reactive cycle which tries to exploit those changes to develop a less inconsistent schedule (by revising the existing schedule). A new reactive cycle will begin to be executed until a consistent assignment (a schedule

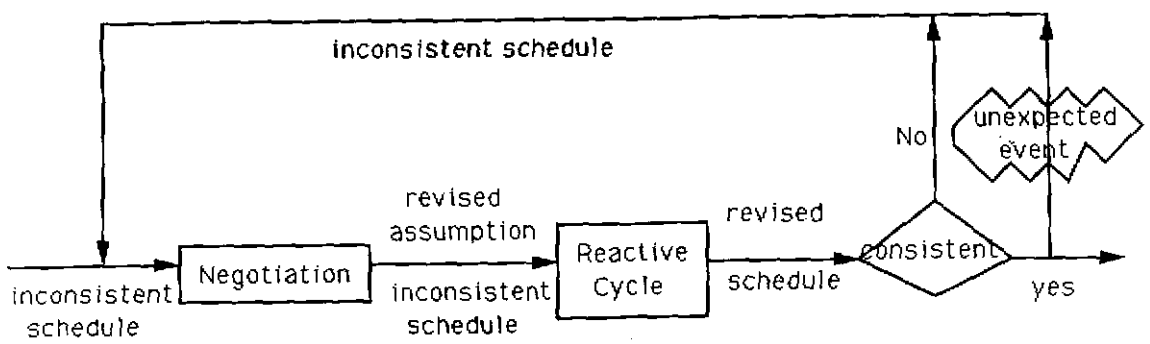


Figure 1-3: Incremental Scheduling Loop

without a conflict) has been found, or until the users stop introducing changes that trigger reactive response 1-3.

The heuristic search model that has been developed is composed of three principle components: a set of *search (revision) operators*, each of which modifies a subset of the commitments comprising the current schedule to produce a new schedule, an *evaluation function* (i.e. a metric on the potential solution space), which provides a basis for comparing the transformations produced by the application of alternative operators to a given schedule, and a *scheduling strategy* , which specifies knowledge relating to use of the search operators and the evaluation function within the search (e.g. conflict prioritization heuristics, operator selection heuristics, termination criteria, etc).

We can contrast our heuristic search model with traditional search-based planning models (e.g. [146]). In these models, the general objective is to find a feasible solution relative to a given set of non-negotiable constraints (i.e. a course of action that brings about a particular goal state in a manner consistent with the physics of the domain) and it is assumed that it is plausible to explore the entire search space (even if knowledge that enables more efficient search is seen as fundamental in practical applications). This assumption, even as a worst case scenario, is unworkable in the context of most scheduling problems, where the crux of the problem is balancing a conflicting set of preferences, each of which can be satisfied to varying degrees (i.e. optimizing within the space of feasible solutions[3]). The search space is too large to ever exhaustively explore, and reliance on heuristic

[3]Feasible solutions are the solutions that satisfy the physics of the problem.

strategy knowledge and heuristic evaluation functions to optimize the search within the time allowed is imperative [115]. Since we have adopted a schedule revision (in contrast to a schedule generation) approach to scheduling, we always keep a copy of the most recently revised schedule. Moreover, since this schedule represents the best schedule that has been developed so far, we can always return the best complete schedule that the heuristic search has produced as soon as the bound on the scheduling time is reached. We have also developed a heuristic to exploit the conflicting nature of problem constraints to optimize the search within the available time, but the utility of this heuristic remains to be tested through a set of experiments. The advantage of such heuristic is that it allows users to balance the quality of a solution with the time that they can afford for finding it.

Although flexibility in specifying the time that users can afford to wait for finding a solution allows search to be optimized with respect to time, this result can be improved if a human scheduler is used as a collaborator. A human scheduler can make qualitative scheduling decisions about selectively abandoning (or alternatively focusing on) search paths when the scheduler's heuristics are insufficient to take the appropriate course of action in specific situations. We define an interactive scheduling framework that supports the collaboration between the machine and human schedulers during scheduling search. To collaborate with computer during search, a human scheduler needs to be familiar with this interactive scheduling framework. Moreover, he should be able to understand the representation that we use for a schedule.

We can also contrast our heuristic search model with manufacturing scheduling heuristic search based models [42, 104]. The overall revision strategy in ISIS [42] (i.e. highest priority order first) dictates a single trajectory through the search space and thus there is no use of a global evaluation function[4]. The OPIS factory scheduling system [104, 128] implements a more sophisticated approach to reactive schedule revision which operates according to an *opportunistic* scheduling strategy. More specifically, a heuristic theory relating the implications of current solution constraints (e.g. important reoptimization needs and opportunities) to the strengths and weaknesses of various revision operators is used as a basis for conflict prioritization and operator selection. In our framework, which also relies on opportunistic scheduling, the use of heuristic strategies is integrated with the use of heuristic search (by virtue of a global evaluation function) [115]. While we have similarly developed a set of heuristics to relate the implications of current solution constraints to the strengths and weaknesses of various revision operators, we also use a global evaluation function to compare revision operators if the heuristics fail to subscribe to one.

This basic heuristic search model has been extended to incorporate uncertainty. Although reactive scheduling addresses the uncertainty in resource estimates, the risk of building inaccurate schedules in SPPS can be reduced by considering the data available about uncertainty during the heuristic search. For instance, if we are not sure about the exact duration of an activity but can estimate its lower and upperbound, then we can make scheduling commitments more accurately if we account for the duration of the activity to be anywhere between its lower and upperbound. We have developed a weighted interval estimate approach (as opposed to point estimate approach which is used in most scheduling systems) to specify the resource requirements of each project activity and have modified our evaluation function to take into account the uncertainty.

---

[4]It should be noted that the "order rescheduling" procedure itself employed a heuristic beam search to locally explore alternative sets of commitments for the order being scheduled, and this search was focused by an evaluation function that reflected scheduling preferences relevant to the order (e.g. meeting the due date, utilizing preferred resources, etc).

It is not possible to eliminate face-to-face negotiation in SPPS entirely because it is simply *impractical* (from the stand point of both *computational* complexity and knowledge) to hypothesize all negotiations and their possible outcomes. Instead, we have to look into how negotiation can be supported in an environment that integrates negotiation and schedule revision. We investigate how our basic heuristic search model can be used to support negotiation by determining and prioritizing the conflicts and resources that need to be negotiated, and also by identifying the agents that have to be involved in the negotiation. This problem is not as easy as it might first look because the resource that should be negotiated to resolve a conflict is not always the same as the resource that has caused the conflict, and the agents who should be involved in the negotiation might be different from those who are directly affected by the conflict.

A program called NEGOPRO that uses our basic heuristic search model to solve SPPS has been implemented. NEGOPRO is a reactive scheduler that reacts to the changes in schedule assumptions (specified by multiple decision-makers) and uses heuristic search to improve a given (e.g. human generated) input schedule. The architecture of NEGOPRO is depicted in figure 1-4).

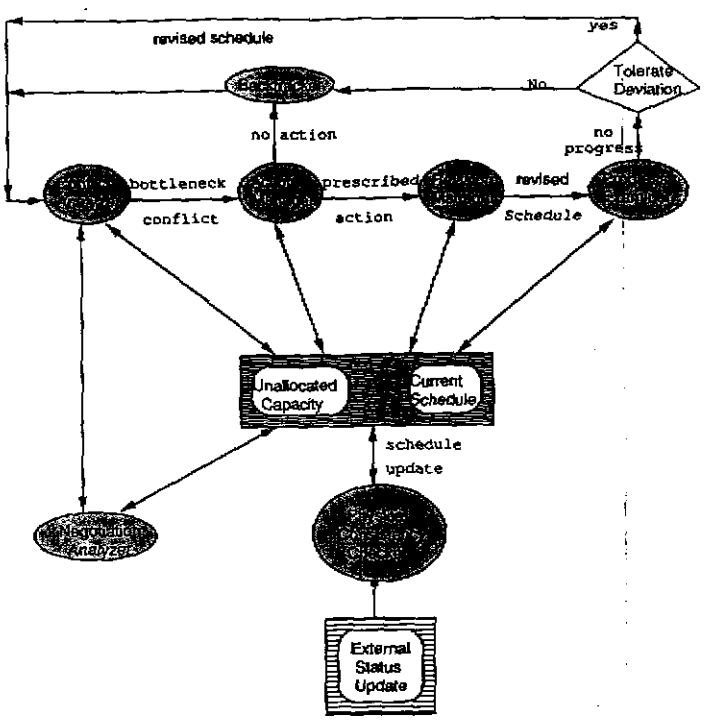


Figure 1-4: Architecture of NEGOPRO

The database in NEGOPRO holds the *current schedule* and the information about *unallocated capacities* of all resources. Each scheduling session begins by entering the specification of the schedule assumptions that have to be updated or by entering a new seed schedule[5] to NEGOPRO. If the schedule assumption update violates a constraint that is related to the physics of the problem (e.g. committing to a reservation which is larger than the available capacity of a resource) then the *physical-consistency checker* will restore the consistency of the physical constraints that have been violated (e.g. by selectively undoing the reservations that violate the physical

[5]A seed schedule can also be constructed manually or through other techniques such as linear programming. In appendix E, a linear programming design has been described for building a seed schedule.

constraints[6]). Once the *physical-consistency checker* completes, NEGOPRO begins heuristic search through incremental refinement of the existing schedule. During each incremental step, first *conflict analyzer* is invoked to find the conflict that needs to be resolved next. *Action manager* includes a set of operators and a set of heuristics to reason about the circumstances that each operator is applicable. Action manager prescribes one or more operators that will be applied to revise the schedule by *revision manager*. Once the schedule is revised, it is dispatched to *progress examiner* to measure the degree of progress in resolving the intended conflict. If the measurement indicates that sufficient progress has been made (or if the temporary digress is tolerable), then the next incremental step will begin. Otherwise, NEGOPRO backtracks and tries alternative commitments that have not yet been examined. The scheduling process will stop when there is no more conflict left to be resolved or when the user specified bound on scheduling time has been reached.

1.4. Contributions

The main contribution of this thesis is that it represents the first major effort in building a *problem solving model* that addresses the major issues of SPPS. Other artificial intelligence approaches to software project management have focused primarily on the development of high-level specification languages for specifying scheduling knowledge and frame-based knowledge representation schemes [9, 64, 119]. Our problem solving model is based on the previous results in social analysis of computing, operations research in manufacturing, artificial intelligence in manufacturing planning and scheduling, and software project management.

We found the social analysis works to be a good starting point to understand the complexity and issues of SPPS. *Social analysis* approaches [121, 48] identify the recurring patterns of social actions in software projects that influence the planning and scheduling and formulate a set of strategies to pursue in dealing with them. Since the social analysis studies rely on empirical data, they provide a significant insight into why software projects fail, what the nature of conflicts in software projects are, and how negotiation is used to resolve those conflicts.

As in many recent operations research approaches [3, 66, 98], we use cost/benefit analysis to compare alternative scheduling commitments during manufacturing scheduling. In our heuristic search model, cost/benefit analysis has been used to construct an evaluation function which measures the change in the consistency of a given schedule before and after making a scheduling commitment. The problem with cost/benefit analysis approaches in operations research, when applied to SPPS, is that they assume all cost and benefit data are provided by a single agent. However, in SPPS there is a possibility of discrepancy in the perceived costs and benefits of product requirements since they are specified by different decisions-makers with conflicting interests. We have extended cost/benefit analysis to distributed decision-making environments in which decision makers have conflicting interests, and have provided a framework for reconciling conflicting positions (these positions are expressed as resource requests) that can not be directly compared. In addition, our framework (1) can model the delegation of authority, (2) is flexible enough to allow the impact of an alternative decision to be measured both locally and globally, and (3) can attend to a rich set of static and dynamic preferences. The framework that we have developed to build an evaluation function for SPPS through cost/benefit analysis can be applied to other manufacturing scheduling domains as well.

---

[6]Of course, undoing these reservations is likely to create new constraints in the form of resource requests that are not met by the current schedule. However, the new conflicts will not be of physical nature any longer.

Our approach to designing of a set of schedule revision operators and a control strategy for applying these operators during the scheduling process extends the previous artificial intelligence-based scheduling approaches [104, 120, 128] in two respects. First, a search space is defined that includes selection among alternative process plans, and a problem solving model that integrates the search for a process plan with the search for a schedule that implements that process plan. This enables preferential concerns relating to process plan selection to be appropriately balanced against those relating to resource allocation and time interval selection.

A second contribution of the present work is that it formally defines a criterion, *navigational minimality*, for improving the design of search operators for manufacturing scheduling problems which have to optimize the search relative to the time allowed for scheduling. If we consider that the search space in these problems (e.g. SPPS) is extended along many dimensions, it is important to allow selective disbandment of search or reprioritization of search operators along some dimensions during scheduling. A navigationally minimal set of primitive operators (which constitute the building blocks of more complex operators) insures that (during scheduling) this goal can be achieved without a major reorganization of the underlying heuristic theory of schedule revision - a task which is both difficult and costly.

Our approach in the area of incremental schedule revision also differs from previous approaches in the heuristic strategies used for opportunistic scheduling. Specifically, the present work focuses on minimization of disruption (or change) to the schedule as the primary criterion for operator selection. Disruption of (lack of stability in) the schedule over time is a particularly important concern in SPPS as (1) a project schedule serves to coordinate the interdependent activities of a large number of project teams, and (2) attempting to change the schedules of several other teams in order to fix the problems that has arisen in the schedule of one frequently has undesirable political ramifications. Although the importance of non-disruptive incremental revision is noted in OPIS [104], it is considered as secondary to reoptimization concerns during operator selection. We believe that minimization of disruption as the primary goal also could be helpful because disruptive revisions might undo the commitments made during previous iterations of opportunistic scheduling [128] by creating new conflicts at the points where previously resolved conflicts used to reside, thereby slowing down the convergence of scheduling. Although this strategy has been implemented, its utility has not been compared with the utility of the previous selection strategy that considers the importance of non-disruptiveness as secondary to reoptimization.

Traditionally, interactive scheduling has been used to refer to the ability of receiving and processing scheduling input interactively. In the present work, we use this term to refer to the ability of a human scheduler to control the heuristic search interactively. Domain independent planning systems [146, 15] have proposed and implemented schemes to allow a human expert to control the expansion of goal nodes, and also to post constraints on plan variables thereby guiding the planning along intended paths. Although these systems increase the involvement of a human scheduler in the scheduling process, they still fail to allow him to become involved in search-related decision making (e.g. make or undo a scheduling commitment while the heuristic search is in progress) to improve the efficiency of search[7]. We have developed a multi-mode scheduler which conducts heuristic search in two modes: automatic and manual. A human scheduler can enter the manual mode any time, where he is allowed to make (or undo) scheduling commitments (e.g. apply an operator, disband a search sub-tree), and switch back to automatic mode later.

For long, operations research has been involved in developing methods that can account for the uncertainty in

[7]the ability to reach results that are as good (schedules that are not less consistent) faster, by conducting search more intelligently.

the duration of project activities [79]. PERT (D.G. Malcolm) provides a probabilistic treatment of activity duration by relying on interval duration estimates, and Monte Carlo [127] uses simulation to estimate the duration of a schedule within a range. The weighted interval estimate approach that we have developed for calculating the value of evaluation function in conducting heuristic search extends the existing operations research approaches (which account for the uncertainty data only *after the schedule has been built* to analyze the potential risk) by taking into account the uncertainty data *during* the scheduling. This allows us to optimize scheduling decision-making relative to uncertainty data. Moreover, we do not limit the consideration of uncertainty data to activity duration by generalizing it to all resource requirements.

We provide a realistic analysis of negotiation and its role in SPPS, and investigate that how our heuristic search model can be used to support negotiation without assuming that

1. *project agents are aware of (or willing to) specify their compromises in advance*. This is captured in the reactive view of SPPS in which each reactive cycle schedules a fixed set of resource requests (and their associated compromises) by revising the pre-existing schedule with the objective of minimizing its inconsistency.
2. *face-to-face negotiation can be entirely eliminated from the scheduling process*. Face-to-face negotiation is supported by determining and prioritizing the conflicts and resources that need to be negotiated, and also by identifying the agents that have to be involved in the negotiation.

In contrast, the presence of these assumptions in [120, 134, 81] prevents them from modeling the negotiation process and its support mechanisms in SPPS in a realistic way.

We also extend previous artificial intelligence-based manufacturing planning and scheduling works in the area of constraint knowledge representation by allowing: (1) the specification of a reservation that includes variable quantities of a resource over time (2) a higher-level representation of temporal relations among activities[8]. Our representation provides a useful abstraction device, and more importantly, allows a more efficient interpretation for scheduling purposes.

1.5. Verification

We demonstrate the sufficiency of the basic heuristic search model that has been developed on specific test cases that reflect actual SPPS circumstances, using NEGOPRO, in chapter 8. The test cases do not include experiments that deal with uncertainty or support for negotiation because the present implementation of NEGOPRO does not support these features. We argue the validity of the extension of the basic heuristic search model to deal with uncertainty on the basis that it is consistent with the process behaviour descriptions that are abstracted from experimental studies of software project risk assessment and risk control literature [13]. We also argue the validity of our approach to support human negotiation through heuristic search on the basis that it is consistent with the empirical studies of social analysis of computing [47, 121].

[8]We allow the start (or completion) time of an activity to be specified relative *to any* point between the start time and completion time of another activity.

1.6. Overview of the Thesis Chapters

The remaining chapters of this thesis are organized as follows. The first step in describing our problem solving model is to develop a formal definition of SPPS that can be modeled by heuristic search. This is provided in chapter 2. In chapter 2, we also describe a constraint representation language for specifying SPPS resource constraints for the case that uncertainty information need not be modeled. Last but not the least, in chapter 2, we provide an overview of the components of our basic heuristic search model. These components are later described in detail in chapters 3 and 4. The evaluation function component of our basic heuristic search mode is discussed in chapter 3, while the scheduling strategy and search operators are discussed in chapter 4. The issues which are related to the control of heuristic search, and the collaborative role of human schedulers in making scheduling decisions are described in chapter 5. The design of experiments for validating the basic heuristic search model and the experimental results are provided in chapter 6. The extension of the basic heuristic search model to deal with uncertainty is discussed in chapter 7, and the use of heuristic search to support negotiation is studied in chapter 8. The final chapter of this thesis, chapter 9, is devoted to exploring the avenues for future research.

Chapter 2

Formal Definition of SPPS and Our Problem Solving Approach

SPPS is a distributed and reactive scheduling problem involving a set of agents with conflicting interests [48] involving revising the existing project schedule in order to maximize the overall satisfaction of all constraints and preferences. SPPS is subject to four types of constraints:

1. *feature requirement constraints* denoting the constraints on each feature requirement of a product,
2. *available capacity constraints* governing the available capacity of each resource over time,
3. *resource request constraints* denoting the required capacity of each resource over time by each project agent (these capacities are essential to him in meeting his responsibilities), and
4. *authority constraints* used to limit the agents who can specify their preferences with respect satisfying various constraints.

These constraints often interact (e.g. might compete over the same set of resources) and can be satisfiable to varying degrees. Moreover, constraints of different types can not often be directly compared.

The SPPS problem is defined in two stages: first it is defined under a centralized authority framework (CAF) and then the definition is extended to accommodate a distributed authority framework (DAF). In the centralized authority framework, the specification of all preferences is made by a single agent. In contrast, in a distributed authority framework, alternatives are selected according to the preferences of multiple decision making centers that are organized in an organization hierarchically. We begin by defining the notation that we use throughout this work.

In the first section of this chapter, we provide a formal definition of SPPS problem. This definition provides the basis to viewing SPPS as heuristic search under a centralized authority framework. In the first section, we also describe a constraint representation language for specifying SPPS resource constraints. In section 2, we analyze and compare the constraint representation language in section 1 with the constraint representation languages that have been used in related research. In the third section, we generalize the formal definition of SPPS to a distributed authority framework, and in the fourth section we discuss an overview of our problem solving approach.

2.1. Notation

The notation used is described in two parts: symbols and constructs. The symbols are merely introduced in this section, and will be formally defined as they appear during the formal definition of the problem in the next section. The symbols uniquely identify SPPS objects while constructs are used to manipulate the symbols. Let

- r, p, and q be resources,
- G denote a software developing organization,
- ρ denote a set of available capacity (resource availabilities) constraints and ρ_r denote the available capacity constraint of resource r over time,
- Π denote a project,
- R_{pri} denote a set of primitive resources and R_{pro} denote a set of products,
- Φ denote a set of product feature requirements,
- θ be a set of user defined preferential concerns,
- Γ be a process plan,
- Λ be a schedule,
- ζ be a temporal capacity resource constraint, and
- Λ^* and Γ^* denote the set of all schedules and all process plans of a product respectively.

Furthermore, let

- $\{A_i\}$ denote a set of objects A_i,
- $[A_i]$ denote an ordered list of objects A_i (if A is declared as an ordered list, then A_i is used to refer to its i-th element),
- $Card(A)$ denote the cardinality of an ordered list or set A,
- f,g,h be functions,
- $n\text{-}th(A,i)$ be a function which returns the i-th element of an ordered list A.
- membership test for ordered lists be defined, and be referred to by the symbol $\in$,
- and $[a\ b]$ $s.t.$ $a,b \in N$ represent a time interval that denotes the inclusive set of time points between a and b.

Last but not the least, all sets, ordered lists, and domains will be finite unless stated otherwise.

2.2. Formal Definition of SPPS Under Centralized Authority Framework

The set of resources that need to be allocated to support or be produced during the execution of a project Π can be broken to two subsets: R_{pro} and R_{pri}. R_{pri} denotes the set of *primitive resources* and includes those resources that are not produced inhouse. In contrast, R_{pro} denotes the set of *products* and includes those resources that are produced inhouse[9]. A file server, an office, any type of software, hardware, documentation, staff, or time are typical examples of primitive resources. Furthermore, all products in a software project are either software or documentation and are considered to be "infinite capacity" resources (therefore they need to be produced only once). R_{pri} itself includes two types of resources: unshared primitive resources which is denoted by R_{Upri} and

[9]In software projects, these resources will remain available indefinitely once they are produced.

represents all primitive resources that their capacity is divided between their reservations, and shared primitive resources which is denoted by R_{Spri} and represents all "infinite capacity" primitive resources. A file server, an office, and any type of software or documentation are typical examples of shared resources while a system analyst, a coder or a workstation are typical examples of unshared resources. A system designer is an unshared resource despite the fact that it can be allocated to different projects at the same time. This is because if the size of time window is chosen small enough (e.g. manhour) a system designer can be working only on one project at any given time. Time has a special status that no other resource has, namely it is not treated as a separate resource and instead is implicit in the specification of every temporal resource constraint.

According to our definition, if each product that shares a resource reserves a portion of its capacity, then the resource should be specified *unshared* and the scheduler should handle its reservation as if it is an unshared resource that can be allocated to several products (projects) at the same time. However, if we are certain that the available capacity of an unshared resource always exceeds its reservations, it is recommended that it be specified as a shared resource. This is because the procedures that are used to handle a shared resource are more efficient than the procedures that are needed to handle an unshared resource. The librarian of a library is a typical unshared resource that can not service an arbitrary number of library users at the same time because each library user reserves a portion of his capacity. However, it can be specified as a shared resource if the aggregate required capacity for the librarian is less than one.

To allow an agent to work on several products (or projects) at the same time (i.e. 1/3 time on product *X* and 2/3 time on product *Y*), the available, requested, and reserved capacities of an unshared resource can be multiplied by the common denominator of all requests for that resource. For instance, in the above case, they all will be multiplied by three.

Both available capacity constraints and required capacity constraints (resource requests) are types of temporal resource constraints surrounding resource usage. Moreover, they share a common representational syntax despite their semantic differences (this representational syntax has been described in definition 2.1) because they both represent variable capacities of a resource over a period of time. Resource reservation (allocation) of a given resource to a schedule of a product also shares this syntactical representation since it represents the capacity of the resource that has been allocated to the product over a period of time.

The relaxation of an available capacity constraint represents an increase in the available capacity of the resource, and can be specified as capacities over a period of time (i.e. the syntactical representation of a temporal capacity constraint). When a required capacity constraint is relaxed, a new reactive scheduling phase is triggered. This is because we assume the resource requirements that are specified during a reactive phase are not artificially inflated by the agents who specify them. The relaxation of a required capacity constraint marks a drop in the required capacity of the resource, and is represented as a temporal capacity constraint as well.

We use the symbol ζ to refer to a prototypical temporal resource constraint. When multiple temporal resource constraints are needed, we use subscripts to distinguish between different temporal resource constraints.

Definition 2.1: if ζ is a temporal resource constraint for a resource r which is required by a product p, then ζ is of the form:

if $r \in R_{Upri}$ then $[(t_{2i}\ t_{2(i+1)}\ q_i)\ \ \forall i \in 0..n]$ $\quad q_i$ is a rational number
else if $r \in R_{Spri}$ then $[(t_{2i}\ t_{2(i+1)})\ \ \forall i \in 0..n]$
else if $r \in R_{pro}$ then t

such that t is a negative offset from the date that p is expected to be completed (initially mapped to

zero) and denotes how early r has to become available in order to complete p on schedule, and for all i, $(t_{2i}\ t_{2(i+1)})$ denotes the period during which r needs to be reserved (like t, t_{2i}[10] and $t_{2(i+1)}$ are negative offsets from the date that p is expected to be completed) and q_i is the quantity of r that is requested over $(t_{2i}\ t_{2(i+1)})$. Since t_{2i} denotes the beginning of a period and $t_{2(i+1)}$ its end, then $t_{2i}<t_{2(i+1)}\ \ \forall i \in 0..n$.

The above definition implies that when r is a primitive resource, then ζ specifies variable capacities of a resource over different intervals (e.g. required capacities of a primitive resource r for a product p). Otherwise (i.e. when r is a product), ζ specifies a time point that indicates how early r has to be available in order to complete p on time. If r is a primitive resource, then ζ can be plotted as a step function, whereas if r is a product, then ζ will represent a single point on the coordinate system.

The following two examples illustrate the use of our constraint representation language in specifying resource requirement constraints:

Example: Consider the resource requirement specification *[(-5 -3 2) (-1 0 1)]* for a senior programmer. The specification requires 2 senior programmers for the first three months, no senior programmer for the fourth month (therefore the specification of this month is left out) and 1 senior programmer for the last month of development.

Example: Consider that to develop a debugger (a product) we need a simulator (also a product). Furthermore, suppose that the simulator has to be available at least three months before the debugger can be completed. This resource requirement can be specified as *-3*. The specification of the upper bound of the interval is redundant because the required resources which are products will remain available once they are produced for the first time.

Given the specification of ζ, we are now in a position to introduce a set of symbols that are defined for temporal resource constraints. For a product p, we use r_ζ to refer to the resource requested by ζ, $\zeta(p)$ to refer to the set of resource requests of a given schedule of p, and $r_\zeta(p)$ to refer to the set of all resources requested by a given schedule p[11].

In section 2.1, we defined ρ as a set of available capacity constraints. This implies that ρ, for a given project, can be formally defined as a set of temporal resource constraints:

Definition 2.2: Resource Availability Constraints

$\rho = \{\zeta_i\}$ s.t. $\forall \zeta \in \rho$ r_ζ is unique, and in the specification of p, t is the date from which r_ζ becomes available, q_i is the quantity of r_ζ that will be available over $(t_{2i}\ t_{2(i+1)})$, $(t_{2i}\ t_{2(i+1)})$ is the period during which r_ζ will be available, and $\forall i \in 0..n\ \ t_{2i}<t_{2(i+1)}$.

On the basis of the definition of ζ, we can also formalize the overall software project planning and scheduling problem. A software development organization is an organization that can carry out orders from several clients concurrently:

Definition 2.3: Software Development Organization

$G = <\{\Pi_i\},\rho>$

---

[10] *the value of subscript 2i is 2 times i*

[11] Since our notation allows r_ζ to be used both as a function and as an object, we use the context of usage to determine whether it is a function or an object. More specifically, if it is followed by an argument which is enclosed in a parenthesis, then it is a function (the function value is a set). Otherwise, it is an object.

This definition implies that all projects within G compete for resources that are globally available (i.e. they can be shared by all projects).

For every order, a project is created to undertake that order. However, our definition of a software project is recursive in that every inhouse order generated to meet a client order can be considered a project in its own rights.

Definition 2.4: Software Project

$\Pi = <p,\Phi,\alpha,\beta,\theta>$ s.t. p is the product of Π, Φ is the ordered list of feature requirements Φ_i that p has to meet, $\alpha=[\alpha_i]$ s.t. α_i is the ordered list of levels at which feature requirement Φ_i of p can be met, $\beta=[\beta_i]$ s.t. β_i is the ordered list of desired levels of meeting each feature requirement[12], and θ is a set of user defined preferential concerns.

The above definition implies that each feature requirement Φ_i of a product can be satisfied at multiple levels l ($1 \le l \le Card(\alpha_i)$), from which, one, β_i, constitutes the desired level, and all others constitute different degrees of relaxation. Of course, a schedule can be constructed to satisfy a feature requirement above its desired level but this might be undesirable or incur additional cost without satisfying any new objectives. Modeling the satisfaction of feature requirements at multiple levels allows us to study the consequences of relaxing the desired level of meeting a feature requirement, or alternatively the consequences of reserving additional resources to assure that a feature requirement will be met at a level which is closer to what is desired.

A software project considers a wide range of preferences including the preferences among

- resource or resource mixes,
- process plans of a product,
- different feature requirements of a product,
- degrees of meeting each feature requirement of a product,
- degrees of relaxing available capacity constraints (if in fact these constraints can be relaxed), and
- degrees of relaxing required capacity constraints (if in fact these constraints can be relaxed).

Formally, a preferential concern $\theta_i \in \theta$ is a function $\theta_i: A \rightarrow N$ s.t. A is a finite set of alternatives (e.g. substitutable resources) and N is the set of possible ratings of A (e.g. preference of each substitutable resource).

Consider a hypothetical software project to develop a runtime environment for a TMS3020 chip[13]. Furthermore, consider that the environment has to provide interactive response time and integrate the debugging and assembly functions. Once the project is awarded, the developing organization produces *production dependency graph* of the final and intermediate products that need to be developed along with the feature requirements of each product (figure 2-1). In the production dependency graph of figure 2-1, products are represented by nodes while activities (productions) are represented by directed arcs. For instance the activity of producing the debugger is denoted by the directed arc that is incident from the simulator on the debugger.

According to definition 2.4, the project to develop a run-time environment in figure 2-1 only includes the activity to produce the run-time environment once the assembler and the debugger are completed. In general, the production of every single product (e.g. the simulator in figure 2-1) is formalized as a project. A *complete*

---

[12]Therefore, $\exists f$ 1-1 onto Φ $f:\alpha \rightarrow \beta$ s.t. $\forall \alpha_i \ f(\alpha_i) \in \alpha_i$.

[13]TMS3020 is a trademark of Texas Instrument Inc.

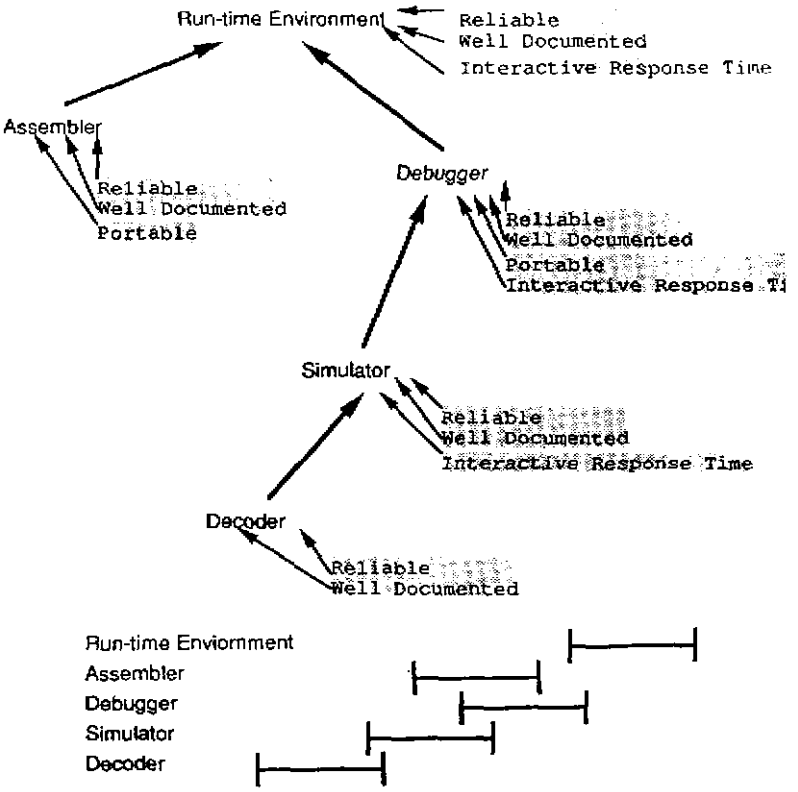


Figure 2-1: Production Dependency Graph of the TMS3020 Runtime Environment

project of producing a run-time environment is constructed by recursively replacing each intermediate product by its complete project.

To illustrate the use of our representation language to specify the constraint knowledge of a problem, consider again the example of developing a runtime environment for a TMS3020 chip (figure 2-1). Figure 2-2 depicts the basic topology of the resource requests of each product that has been specified in our representation language under a fixed process plan. Hardware description *(HD)* is the only shared primitive resource that is included in the resource request graph while the unshared resources consist of junior programmer *(JP)*, senior programmer *(SP)*, graphic generator *(GC)*, and lexical analyzer *(LA)*. The labels of the arcs that connect two nodes reflect the resource requirement constraints of a product end of the arc for the resource end of the arc. For instance, *HD* is required one month into the production of assembler while to develop the run-time environment 2 senior programmers will be required for the first two months and only one senior programmer for the third month.

Each product can typically be developed through many process plans. A process plan specifies how a product can be produced from its required resources but stops short of allocating resources and temporally instantiating that plan. However, the process plan of a product does not specify how its required resources are acquired or produced. A *complete* process plan of a product p can be constructed by recursively merging p with the *complete process plans* of all products that are required by p.

Process plans of a product are different in the type of resources that they use to produce the product. For instance, a business data base application software can be developed by acquiring a business application generator and hiring an application generator expert to develop the application, or by designing and then developing the entire application inhouse using database designers and data base coders. Project agents

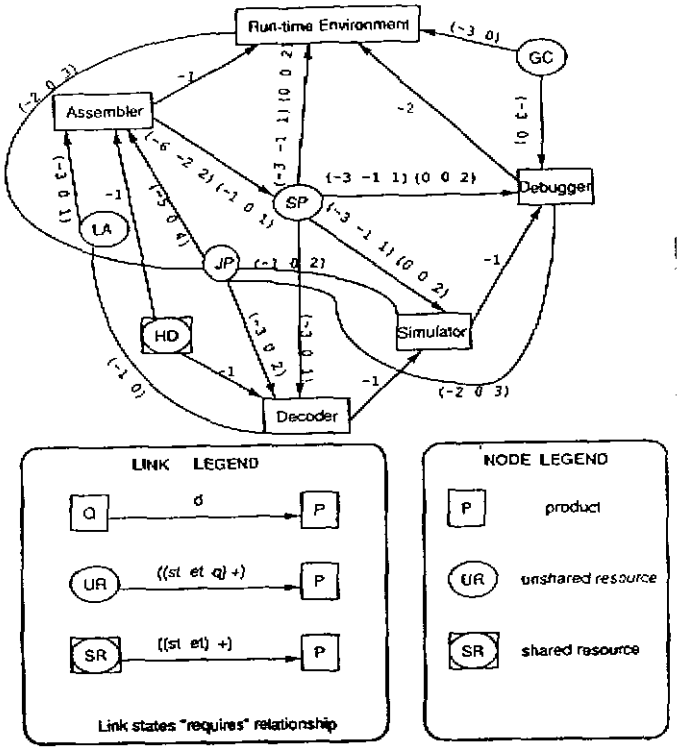


Figure 2-2: Resource Request Graph of the TMS3020 Runtime Environment

normally indicate their preference with respect to different process plans. The preference of each process plan of a product p can be specified by associating it with the degree that it meets different feature requirements of p. For instance, if a process plan p achieves more reliability than another at the cost of additional development time, then that process plan meets the reliability requirement of developing p to a higher degree while it meets the deadline requirement of developing p to a lesser degree.

Among the set of resources required to produce a product under a process plan, some resources can be replaced by others. For instance, it might be possible to implement a process plan of a product by either 2 senior programmers and 3 junior programmers or 1 senior programmers and 6 junior programmers. The set of resources required under a process plan can be broken down to disjoint partitions υ such that each resource is in the same partition as all other resources that it is replacable with. For each partition, the user can specify multiple mixes of resource allocation requirements (e.g. two senior programmers and 3 junior programmers vs. one senior programmers and 6 junior programmers). We denote the set of these mixes for all partitions by τ. In the following, we have illustrated υ and τ through an example.

Example: Suppose that a possible plan for a product p is to use 4 Spark workstation and either two senior programmers (Sprog) and 3 junior programmers (Jprog) or one senior programmer and 6 junior programmers for 8 months. Then the set of all resources used in producing p, R_p = {*Spark*, *Sprog*, *Jprog*}, the set of partitions of R_p, υ = {*Spark*} $\cup$ {*Sprog*,*Jprog*} and the set of resource mixes of *Sprog* and *Jprog*, τ={((*Sprog*(–802))(*Jprog*(–803)))((*Sprog*(–801))(*Jprog*(–806)))}.

The specification of a process plan Γ of a product p is formalized in the following definition:

Definition 2.5: Process Plan

Let Γ be a process plan of project Π which produces p. Then $\Gamma = <R_p,\upsilon,\tau>$ s.t. R_p is the set of resources required to implement Π, $p \notin R_p$, $\upsilon \subseteq 2^{R_p}$ (power set of R_p) is a set of disjoint partitions of R_p denoting substitutable resources, and $\tau=[\tau_i]$ s.t. τ_i is the ordered list of resource mixes of each partition in υ[14].

Each process plan can be implemented through many schedules. A schedule is constructed by

1. determining which feature requirements (if any) have to be compromised (to avoid violating more important requirements such as missing a delivery deadline). For instance, the testing of a software module might require conducting formal reviews periodically, however, the development organization might decide to abandon this requirement if a major cost overrun would occur otherwise. We use ψ to denote the actual level at which each feature requirement of a product is met in a schedule.
2. committing to specific resource mixes among all alternative sets of resource requirements for each set of substitutable resources. We let χ be the ordered list of indices of the selected resource requirement mixes.
3. committing to a set of resource allocations π to budget the process plan which the schedule implements. In section 2.1, we described that each available capacity constraint possesses the syntactical format of a temporal resource constraint. Then π can be formally defined as a set of temporal resource constraints:

Definition 2.6: Resource Reservations of Λ

$\pi = [\zeta_i]$ s.t. $\forall\zeta \in \pi$ r_ζ is unique, and in the specification of p, t is the date from which r_ζ becomes available, q_i is the quantity of r_ζ that is allocated over $(t_{2i}\, t_{2(i+1)})$, $(t_{2i}\, t_{2(i+1)})$ is the period during which r_ζ is allocated, $\forall i \in 0..n$ $t_{2i}<t_{2(i+1)}$.

A schedule can be formally defined in terms of π, χ, and ψ, the elements that are used to construct it:

Definition 2.7: Schedule

Let Λ be a schedule of Γ and p be the product of the project for which Γ is a process plan. Then $\Lambda = <\pi,\chi,\psi>$ s.t. π denotes the reservations of members of R_p[15], χ is an ordered list denoting the selected resource requirement mixes[16], and ψ is an ordered list that denotes the actual level of meeting each feature requirement[17].

The duration and start time of a schedule can be derived from the resource requirement specification of that schedule. The duration of a schedule can be calculated by finding the start time of requiring the resource that is needed the earliest in implementing the schedule. This can be achieved by scanning all resource requirement specifications of the form ζ and recording the earliest required date of each resource, then taking the smallest of the numbers that have been recorded, and finally taking the absolute value of that number. This is depicted in the following definition:

Definition 2.8: Duration of a Schedule

Let $\forall\zeta \in \pi$ t_ζ = *if* $r_\zeta \in R_{pro}$ *then* t *else* t_0 *endif* where t and t_0 are as defined in definition 2.1. Then d_Λ = $max(|t_\zeta|\ \forall\zeta \in \pi)$.

---

[14]This can be formally stated as $\exists$ 1-1 onto $f:\upsilon \rightarrow \tau$ where $\forall\tau_i$ $f(\tau_i)$ denotes the alternative resource requirement mixes of τ_i and $Card(\tau_i)=Card(\zeta)$ $\forall\zeta \in f(\tau_i)$.

[15]This can be formally stated as $\exists$ 1-1 onto $h:R_p \rightarrow \pi$.

[16]This can be formally stated as $\exists g$ 1-1 onto $g:\tau \rightarrow \chi$ s.t. $\forall\tau_i$ $1 \le g(\tau_i) \le Card(\tau_i)$ (a position within τ_i).

[17]This can be formally stated as $\exists f$ 1-1 onto $f:\alpha \rightarrow \psi$ where $\forall\alpha_i$ $f(\alpha_i) \in \alpha_i$.

The start time of a schedule can be calculated by deducting the duration of that schedule from its deadline. This is formally illustrated in the following definition:

Definition 2.9: Start Time of a Schedule Λ

Let e_Λ be the value of the level at which the deadline feature requirement of p has to be met. Then $s_\Lambda = e_\Lambda - d_\Lambda$.

Therefore these parameters need not be specified as independent variables.

Normally we are interested not only in the schedule that describes how p is produced from its required resources but also the schedule of all *intermediate* products that need to be produced in order to develop the required resources of p. The complete schedule of p, Λ_{p^*}, can be defined as the union of the schedule of p and the *complete schedules* of all required products of p. This can be formally written as follows:

Definition 2.10: Complete Schedule

Let Λ_p and Λ_{p^*} denote the schedule and the complete schedule of producing p respectively. Then $\Lambda_{p^*} = \Lambda_p \cup \Lambda_{q^*}$ s.t.
$\forall q \quad p \in product\text{–}transitive\text{–}closure(q)$

Λ_{p^*} can be constructed by starting from p working back recursively and including the schedules of all resource requirements of p that are of the type product. The *product-transitive-closure* of a resource r is the set of all products $\{p_i\}$ that require r either directly (because r is on the requirement list of each) or indirectly because r is on the requirement list of a product q which is required by every member of $\{p_i\}$.

A schedule Λ for a software project is *consistent* if and only if it meets the resource requirements that are necessary to satisfy the feature requirements of the product p of that project without violating the temporal and capacity restrictions ρ of available resources (recall that ρ is a set of element of the form ζ)[18]. Since a consistent schedule might be non-existent (i.e. the problem might be over-constrained) or exist but take a very long time to find, the task of scheduling during each reactive phase can be formulated as finding the assignment of values to variables R_p, ψ, χ, *and* π that maximizes the overall satisfaction of feature requirements of the project product and capacity restrictions of primitive project resources under a variety of preferences. Later in this chapter, we discuss the development of an evaluation function which measures the overall satisfaction of all constraints and preferences for a given assignment of values to all the variables. Variables p, Φ, β, υ, τ, and ρ_r $\forall r \in R_{pri}$ are assumed to be fixed since their values are set by project agents.

In our formalism, the consistency of a schedule Λ for a product p can be verified by comparing the aggregate demand[19] and the available capacity for r, for every resource $r \in R_p$. The aggregate demand for a resource by a schedule Λ reflects the capacity of r that is essential to satisfy all feature requirements of p. Λ is *consistent* if and only if the aggregate demand for every $r \in R_p$ never exceeds the available capacity of r. This can be formally stated as follows:

Definition 2.11: Schedule Consistency

Let Λ be a schedule to produce p, $\exists$ 1-1 *onto* $h{:}R_p \rightarrow \pi$, and $\exists g$ 1-1 onto $g{:}\tau \rightarrow \chi$ s.t.
$\forall \tau_i \quad 1 \le g(\tau_i) \le Card(\tau_i)$ (a position within τ_i). Then Λ is *consistent* iff
$\forall \tau_i \, \forall q \quad p \in product\text{–}transitive\text{–}closure(q) \;\; n\text{–}th(\tau_i \, , \, g(\tau_i)) \;\subseteq\; h(q)$

---

[18]Since meeting the *feature requirements of p* is tied to meeting the feature requirements of the resources that p requires, then *consistency* needs to be measured across the *complete schedule* of p.

[19]Aggregate demand for a resource can be constructed by aggregating the requests for that resource by all products produced under Λ.

2.3. Analysis of Constraint Representation Language

In chapter 1, we stated that our constraint representation language (which we defined in the previous section) allows the specification of a reservation that includes variable quantities of a resource over time, and also provides a higher-level representation of temporal relations among activities by allowing the start (or completion) time of an activity to be specified relative to any point between the start time and completion time of another activity. The need to specify a reservation that includes variable quantities of resource over time frequently arises in SPPS. For instance, many development tasks require more senior programmers during the starting and completion stages of the task, while they require fewer ones in the middle stages. This is because senior programmers are used to construct a detailed design of different components of a software module before more junior programmers are assigned to undertake the development. Senior programmers are also used to integrate the module components after they have been separately developed.

The need for a higher-level representation of temporal relations among activities during scheduling arises when the traditional specification of activity precedence relations (i.e. linking activities through *before* relation) is not sufficient to specify more sophisticated temporal relations among activities [1]. For instance, suppose that the execution of an activity is allowed to overlap partly with the execution of another activity which succeeds it. Then, a limited concurrent scheduling of the two activities could lower the combined duration of both.

Previous research [41, 104] require that the activities which require complex reservations or the activities that the relation *before* is not sufficient to express their temporal relation with other activities be exploded to a larger number of more primitive activities which can be represented using previous schemes. Our representation provides a useful abstraction device in these cases by allowing the user to specify reservations and temporal relationships among activities at a higher level. More importantly, our representation allows a more efficient interpretation of specification for scheduling purposes. In the remainder of this section, we illustrate that why our representation allows a more efficient interpretation of constraint specification through an example.

Example: Suppose that O is an activity with a linear sequence of 5 resource reservations each for a different resource r_1 through r_5 as depicted in figure 2-3; the available capacity of r_1 through r_5 is shown in figure 2-4. Furthermore, suppose that O has to be completed in 12 months and that the objective is to optimize the allocation of resources to O. The priority of each resource in the descending order is r_4, r_3, r_2, r_5, r_1, and time is assumed to be the least important of all.

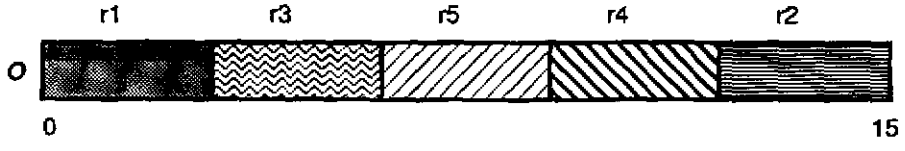


Figure 2-3: Resource Requirements of Activity O

In order to schedule O using the previous approaches, O needs to be exploded to 5 primitive activities A_1 through A_5 (see figure 2-5) reflecting the reservations of r_1 through r_5. The relation between these primitive activities is that the completion time of A_i maps to the start time of A_{i+1} for all i between 1 and 4.

If a scheduler which uses previous approaches does not use opportunistic scheduling (i.e. an order scheduler),

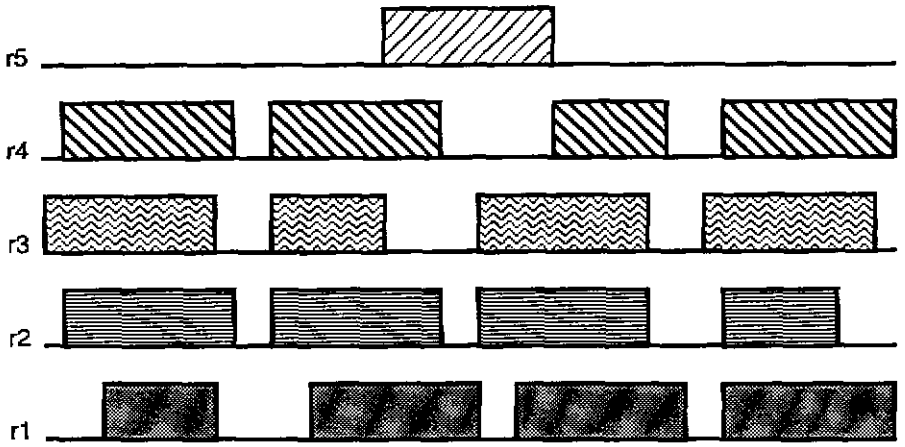


Figure 2-4: Resource Availabilities

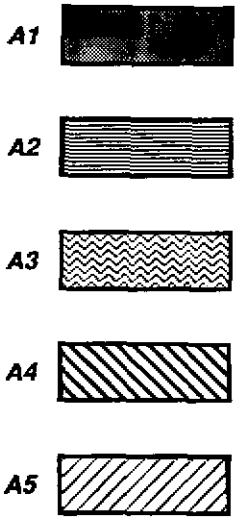


Figure 2-5: Resource Requirements of A_1 through A_5

then it will not be able to meet the optimization objective of the scheduling problem because it will ignore the resource-based perspective in scheduling A_1 through A_5. Otherwise, the activities are scheduled in the order A_5, A_4, ..., and A_1. The scheduling of A_5 is straight forward and requires only one iteration of opportunistic scheduling since there exists only one choice of time that satisfies the resource requirements of A_5, namely to start A_5 when r_5 becomes available first. During the scheduling of A_4, the three periods during which r_4 is available and can best satisfy the demand for r_4 are examined first. Once each of the three periods is examined, opportunistic scheduling concludes that scheduling of A_4 to map to either one is inconsistent with the previously made scheduling commitment to start A_5 when r_5 becomes available. Therefore the scheduler will schedule A_4 to start when A_5 completes, and to do that it executes 3×4 (i.e. 12) iterations. Since the same process will repeat for scheduling A_3, A_2, and A_1, the scheduler will require $(4 \times 12 + 1)$ (i.e. 49, where 1 is for the iteration that scheduled A_5) iterations in order to build a schedule for O.

Now consider the case that a scheduler can schedule O without exploding it to more primitive activities. Then, once the reservation for r_5 is temporally instantiated to start at the point that r_5 becomes available, the start time and completion time of O (including the reservations for r_4 through r_1) will be temporally instantiated

automatically. This results in a performance ratio improvement of roughly 50 to 1 over the previous case (where each reservation that resulted from exploding O was scheduled independently). In general, the performance improvement ratio is a multiple of the number of complex reservations that are required by an activity.

Moreover, in the above example we chose O such that it can be exploded to a set of non-overlapping primitive activities. However, if we had chosen O to explode to overlapping primitive activities, then the ability to represent O at a higher level enhances the performance ratio even to a larger degree. For instance, in the example that we described earlier, if the reservations for r_1 through r_5 overlap temporally then the explosion of O might produce many more primitive activities than when r_1 through r_5 did not overlap. This will in turn increase the computational cost of scheduling the exploded pieces of O further.

2.4. SPPS Under A Distributed Authority Framework

Scheduling under distributed authority assumes a framework in which the agent who is responsible for a product can delegate part of his authorities and responsibilities to other agents in the form of assigning them to produce more primitive products and allowing them to decide the weight (preference) that should be given to each feature requirement of the primitive product. Distribution of authority is inspired by the fact that global decision making by a central authority is very costly and has to be avoided as much as possible. This is because global decision making requires the measurement of the effect of a commitment which is local to a part of project on the entire project before making that commitment.

The authority and responsibility structure of a typical software development organization is illustrated in figure 2-6. *Rlinks* denote the assignment of responsibilities by one agent to other agents. A responsibility consists of a project Π that produces a product p. Each agent who receives a responsibility makes a set of resource requests that are essential to carrying out that responsibility. The resource requests are denoted by *Qlinks*. *Alinks* denote the authority to make preference rulings (relative preference of each feature requirement). If this authority is not delegated with the responsibilities, then the decision to make preference rulings can not be made locally.

Normally, in a software project, the authority to weigh different feature requirements of an intermediate product locally is delegated conditionally (if it is delegated at all): if the revised schedule insures that the overall requirements of the parent products of that intermediate product can be met to a certain degree, then the authority might be delegated. The reason for this heuristic is that the agent who delegates the authority tends not to consider the decision making crucial enough to directly step in if most requirements are guaranteed to be satisfied under the new schedule.

An agent can delegate the authority to make a decision only if (1) he himself possesses the authority to make that decision in the first place, and (2) he has been given the authority to delegate the responsibility. Furthermore, the set of agents who he can delegate to is always constrained by the authority space of the agent[20].

While we allow the distribution of responsibility among various agents, the responsibility to develop a product is assigned to exactly one agent. Moreover, the weight (preference) that should be given to each feature

[20]The authority space of an agent who is responsible for developing a product p includes the agents who are assigned to develop the more primitive products that are used in developing p.

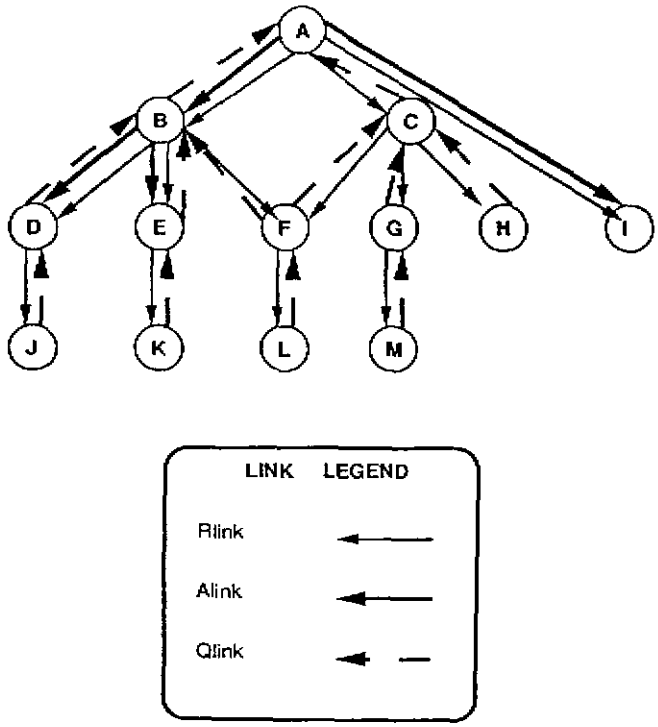


Figure 2-6: A Distributed Authority and Responsibility Structure

requirement of a product is determined by exactly one agent[21]. Our assumption is based on the realization that the agents who are directly responsible for a product normally belong to the same team (in SPPS a team consists of 3-4 agents) and it would be far more efficient if they develop their positions and reconcile their differences through face-to-face negotiation.

Since responsibilities and authorities are specified at the product level (in contrast to feature requirement level), extending the formalism that we presented in the previous section to a distributed authority framework involves refining the product level definitions. The following definition is a distributed authority version of the definition of software project which is the main product level definition in our formalism. We use $a_{manager}$ to refer to the agent who originally possesses the authority to delegate his authorities and $a_{assigned}$ to refer to the agent who $a_{manager}$ delegates this the authority to.

Definition 2.12: Software Project (Distributed Version)

Let Π be $<p,\Phi,\alpha,\beta,\theta,a_{manager},a_{assigned},\varepsilon>$ s.t. p,Φ,α, and β are as in definition 2.4,
$a_{manager}$ is the agent who is responsible for the development of p,
$a_{assigned}$ is the agent who $a_{manager}$ assigns to develop Π, and ε is an
authority activation threshold specifying the minimum overall level at which Φ has to be
met before $a_{assigned}$ can exercise (activate) his preferences. Otherwise,
the preferences of $a_{manager}$ will be considered instead. θ is the set of
user defined preferences that will be considered.

In the above definition, $\Phi,\alpha,\beta,\varepsilon$ are decided by $a_{manager}$. The authority to weigh different resource requests

[21]The agent who decides the weight that should be given to each feature requirement of a product is either the agent who is responsible for the product or the agent who requires that product to meets his own responsibilities.

locally is delegated to $a_{assigned}$ only if the present schedule insures that at least a prespecified percentage of the requirements of p (i.e. ε) can be met. The reason for this heuristic is that $a_{manager}$ does not consider the decision making crucial enough to directly step in if most requirements are guaranteed to be satisfied under the present schedule. An advantage of the above definition is that it can model an array of software project organizations by avoiding commitment to a predetermined policy of how agents $a_{assigned}$ and $a_{manager}$ are picked as well as who is responsible for picking them.

2.5. Overview of Our Problem Solving Approach

SPPS is a *reactive* process which involves continuous revision of project schedule during the course of project. In our approach, each reactive cycle in SPPS establishes R_p , χ , ψ, π by heuristic search during which p , Φ , β_i $1 \le i \le Card(\Phi)$, υ , τ , ρ_r $\forall r \in R_{pri}$ are fixed. During each reactive cycle, heuristic search is initiated with respect to an input schedule that contains one or more conflicts (i.e. constraints or preferences that can not be satisfied), and the goal is to transform this schedule into one in which these conflicts (schedule inconsistencies) are eliminated (or reduced) within the bound on scheduling time. In other words, the goal is to revise the existing project schedule in order to maximize the overall satisfaction of all constraints and preferences.

The heuristic search model that has been developed is composed of three principle components: a set of *search (revision) operators*, each of which modifies a subset of the commitments comprising the current schedule to produce a new schedule, an *evaluation function* (i.e. a metric on the potential solution space), which provides a basis for comparing the transformations produced by the application of alternative operators to a given schedule, and a *scheduling strategy* , which specifies knowledge relating to use of the search operators and the evaluation function within the search (e.g. conflict prioritization heuristics, operator selection heuristics, termination criteria, etc).

To guide the heuristic search process, we use an evaluation function to compare alternative scheduling commitments[22] at any point during the search. By comparing the utility calculated for each alternative commitment, the commitment that yields the highest utility can be chosen. The evaluation function measures the distance *cd* between a given schedule and a schedule in which all constraints are fully satisfied (consistency-distance of the given schedule). *cd* of a schedule can be calculated by inverting a measure of the overall satisfaction of all constraints and preferences under that schedule. Although *cd* only reflects the distance of the schedule being evaluated and should not be perceived as a look-ahead (predictive) measure, it can be exploited for look-ahead purposes if it is used to evaluate potential alternative revisions of the schedule: if *cd* is lowered after applying a revision operator, then the application of that operator should improve the schedule; otherwise, the application of that operator, at least on an immediate basis should worsen the schedule or have no effect. Potential alternative revisions can be compared on the basis of the *cd* of the revised schedule which each generates. The use of an evaluation function to guide the heuristic search enables the search to focus immediately on those decisions most critical to schedule revision objectives as opposed to encountering them only after other restricting commitments have been made.

---

[22]A schedule is a collection of scheduling commitments and a schedule revision involves changing some of the commitments that were made in the schedule that is being revised.

The basic heuristic search model uses two types of operators: primitive and complex. Primitive revision operators are responsible for implementing an elementary change in the schedule (e.g. moving the start time of an activity). More complex revision operators, which can be constructed from primitive revision operators, combine many elementary changes and implement them as an atomic revision (e.g. relaxing the quality assurance requirements of all activities that complete behind schedule).

Chapter 3
Evaluation Function

In this chapter, we describe the design of an evaluation function for incremental heuristic search in SPPS. Our framework can be used to design evaluation functions for heuristic search in other manufacturing scheduling domains as well. In the last section of this chapter, we specifically provide a taxonomy of the preferences that our evaluation function accounts for.

In section 2.5, we described that an evaluation function for SPPS should measure the overall satisfaction of all constraints and preferences of all scheduling agents under a given schedule. The impact of a scheduling decision on a schedule can be evaluated by comparing the overall satisfaction of these constraints and preferences before and after revising the schedule to incorporate the decision. In the following, we review the recent research in operations research and artificial intelligence in evaluating decisions during scheduling, and compare it to our own approach.

The design of evaluation functions for evaluating scheduling decisions based on utility theory [144, 68] has been under investigation by investigators in operations research and more recently by artificial intelligence researchers independently. Although the focus of these approaches has been on the shop and factory scheduling, the techniques that they have developed can be refined for use in SPPS as well.

Recent operations research approaches in scheduling [66, 3, 98] describe that utility of a scheduling alternative is driven by cost and benefit. The cost associated with a scheduling decision refers to the cost of such items as material, machine, and labor which are needed in order to implement that decision, while the benefit associated with a decision refers to the effect of implementing that decision on such items as revenue. The reason for breaking the utility of a scheduling decision to cost and benefit is that

1. labor and raw material cost of budgeting alternative decisions are not the same, and
2. client requirements that can be satisfied under alternative decisions are not the same.

Once the cost and benefit of a scheduling decision are calculated, they are traded off to measure the utility of that decision. The following example explains that how alternative scheduling decisions can be compared on the basis of this trade off: if the decision is related to choosing one among two substitutable resources r_1 and r_2 to allocate to a product, and r_1 is significantly more expensive to acquire than r_2 but satisfies a set of feature requirements that are only *marginally more important* than those satisfied by a_2, then a tradeoff between the cost and benefit for each resource, r_1 and r_2 determines that r_2 is a more appropriate choice.

Artificial intelligence approaches to job-shop and factory scheduling [42, 104, 120] also have relied on utility functions as a means of evaluating a decision, however, these approaches assume that the tradeoff between the cost and benefit of a decision has been already made by the user, and the result is encoded in a function that

reflects the utility of the decision[23]. For instance, job-shop and factory scheduling approaches use utility function to specify the preference to reserve one resource (or start time) over another [42] in an activity, or to prioritize orders. This differs from cost/benefit analysis in operations research which calculates this utility from primitive cost and benefit data.

Cost/benefit analysis has been widely used by operations research for making scheduling decisions and optimizing shop parameters by working with externally given cost functions and with internally derived prices for work center capacity. Zimmerman and Sovereign [147] have developed a "dual" pricing scheme for machines in simple production contexts. Prices are positive if the machine is fully utilized, and zero otherwise.

Cost/benefit analysis also has been studied in economics literature. Turvey [139] has developed "peak load" pricing for resources in production contexts with nonstationary demands. Essentially, demand can be increased or decreased to capacity in each period by decreasing or increasing the price in that period. Thus, in this type of analysis, prices are explicit instead of implicit and vary in a less extreme fashion than for the operations research models.

There is also a fair amount of literature on developing costs and decisions when the demand is stochastic. Karmarkar [66] has developed the idea that congestion (demand exceeding the supply) develops from the inability to fully adjust in the planning process to dynamic changes in the lead times. He also argues that capacity (availability) constraints are effective (in making scheduling decisions) at any level of effective utilization, and not just at full utilization. His queuing model gives an important link between lead time, capacity, and a determination of the optimal batch size. The basic model of Karmarkar has been extended to multi-item and multi-workcenter models [66]. Banker [3] uses the multi-item model of Karmarkar as a foundation for determining relevant costs (resource prices) in a stochastic single machine shop with a mixed Poisson arrival stream and stationary demand. Morton [98] generalizes this scheme to also work when the demand is not stationary. In his model, the objective of a firm is to maximize the net present value of cash flows, even if some of the flows are implicit and therefore difficult to deal with, while others are explicit and therefore easy to deal with. Figure 3-1 shows the stream of cash flow due to scheduling a single job on a single machine.

There are two major problems with this approach when applied to SPPS. First, the investigation of cost/benefit analysis in operations research and economics literature is limited to the case where all cost and benefit data are provided by a single agent. In SPPS there is a possibility of discrepancy in the perceived costs and benefits of product requirements since they are specified by different decisions-makers with conflicting interests. Sathi [120] and Sycara [134] have proposed distributed frameworks to combine the utilities of different agents. The problem with these frameworks is that they assume the preferences of agents are on the same scale and therefore can be compared and used directly to prioritize them. In chapter 8, we argue why this assumption is not valid in the general case.

We have extended cost/benefit analysis to distributed decision-making environments, and have provided a framework for reconciling conflicting preferences of a group of agents by relying on primitive cost/benefit data which includes the specification of resource requirements of each project product, and acquisition cost of each resource. In our framework, the agents who carry conflicting preferences are only required to specify the

---

[23]A utility function is normally a 1-1 continuous function $utility: A \rightarrow I$ where A is the set of alternative decisions and $I = [0\ 1]$ represents the normalized preference of each.

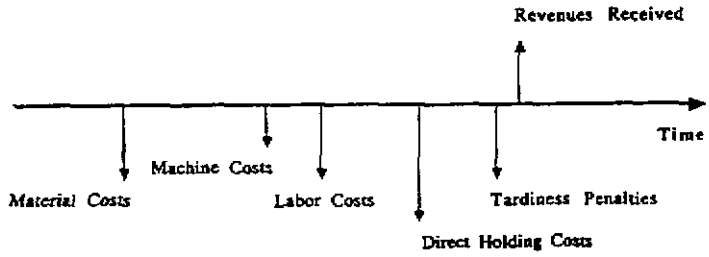


Figure 3-1: *Cash flows due to scheduling a job*

resource requirements of the products that they are responsible for[24]. These requirements reflect the preferences of each agent since he decides the capacities required to meet the feature requirements of the product that he is responsible for, to varying degrees. The acquisition cost of a resource is independent from agent preferences and is used to discriminate alternative resource requirements by comparing the acquisition cost that each incurs on the project (i.e. comparing resource requirement alternatives on the basis of project preferences). In the next section, we discuss the representational issues of acquisition cost. To evaluate a scheduling decision, such as reallocating a capacity from one product to another, first the new set of resource allocations that result from making the decision are calculated, and then the new resource allocations are propagated through the production dependency network (see section 2.2) until they reach the final product of the project. At that point, the effect of those allocations on meeting different feature requirements of the final product will be measured, and the decision will be evaluated on the basis of how important each feature is to the customer and also on the basis of how the cost of budgeting the project will be affected by the decision.

The second problem with cost/benefit analysis in operations research and economics literature is that they evaluate the cost and benefit of a scheduling decision locally relative to the conflict that the decision is trying to resolve. Suppose that a local evaluation of a decision (i.e. the impact of the decision on an intermediate product) concludes that the decision is important to meeting the feature requirements of an intermediate product, whereas a global evaluation of the decision concludes that the decision is of little importance to meeting the feature requirements of the final product of the project. Then a local evaluation of the decision would depict an inaccurate picture of how important it is to make the decision. We evaluate a decision (i.e. assess the global impact of a decision on different project feature requirements) locally when local impact is a good estimator of global impact. Otherwise, the impact of the decision is measured globally. The scheme that we described in the previous paragraph evaluates a decision globally. To measure the local impact of a decision, we rely on the authority structure of the project organization and the delegation of decision making privilages to local agents. In the next section, we describe a model of the distributed authority structure of software project organizations, and the rules and regulations which govern the delegation of authority within them.

---

[24]In chapter 2, we defined the structure of resource requirement data and provided a language to specify these requirements.

We also have identified two problems with artificial intelligence approaches to decision evaluation when applied to SPPS. First, it is difficult to develop a utility function that accounts for the preference to reserve one resource (or start time or product feature requirement) over another when preferences have a dynamic nature (i.e. their value depends on the scheduling state). Our evaluation function derives the preference of resources, start times, and product feature requirements by measuring their impact on meeting the requirements of (the final product of) the project.

A second problem with artificial intelligence approaches is that they are not appropriate for a distributed framework because the utilities of different agents relative to the same decision (scheduling commitment) can not be compared accurately. A utility function for a scheduling commitment (to resolve a conflict involving a product p) can be modeled by $\frac{C}{B} \times w$ where C denotes cost, B denotes benefit, and w specifies the weight that has to be given to their ratio. In the above equation, B is the only variable that reflects the agent preference. Benefit of a scheduling commitment represents the local assessment of a manager about the importance of feature requirements of p that will be met if the commitment is made, but this local view may not be shared by other managers. C on the other hand, is independent of agent preferences (and instead depends on the market forces), reflects the cost of making the commitment to the project, and needs to be measured only once. If the calculation of $\frac{C}{B}$ is left to the agent (as required by related artificial intelligence approaches), then different agents might calculate different costs for the same commitment which in turn will result in a bias comparison of preferences. Our evaluation function solves this problem by measuring the cost of scheduling commitments while requiring each agent to only specify the benefit. The idea of measuring the cost of a commitment is not advocated by ISIS [41] and OPIS [104] on the ground that the cost of certain primitive resources is not usually available.

More recent work (e.g. [43]) has used constraint propagation and analysis to measure the global impact of a commitment, but this impact is measured only relative to the project deadline (while a project has other feature requirements such as reliability and portability that might be affected by propagation as well).

3.1. SPPS Evaluation Function Design

The first step in developing an evaluation function for a scheduling problem domain consists in identifying and classifying the problem constraints according to their types. We provided a classification of all SPPS constraints in chapter 2. There, we divided all problem constraints to four classes: feature requirement constraints, available capacity constraints, resource request constraints, and authority constraints.

The second step in developing an evaluation function for a scheduling problem involves separating the hard constraints (constraints that can not be relaxed) from the constraints that can be relaxed, and establishing a representation of relaxation level for each relaxable constraints. Generally speaking, each SPPS constraint that has a potential of being relaxed is either a product feature requirement constraint or an available capacity constraint. These two sets of constraints are conflicting because normally by relaxing some of the available capacity constraints, feature requirement constraints can be satisfied to a higher degree, and vice versa. Moreover, some feature requirements are in conflict with each other since they compete for scarce resources that are essential to meeting them. Authority constraints can change in an organization but they are not relaxable since they change over time scales that are longer than scheduling horizon. Resource request constraints simply translate the feature requirement constraints to a form that can be traded off with available capacity constraints but they too can not be relaxed. This is because changing the resource requirements of a product feature marks the beginning of a reactive cycle (i.e. it triggers a new reactive cycle).

The third step in the development of an evaluation function involves developing a metric for measuring the penalty of relaxing each type of constraint as a function of its relaxation levels. This metric serves as a reverse indicator of the degree that a constraint has been satisfied. In the case of SPPS, this penalty has to be measured for feature requirement and available capacity constraints since they constitute the only types of constraints that are relaxable. In the following, we discuss how the penalty of relaxing these types of constraints can be measured.

The penalty of not meeting the feature requirements of a product by a given schedule can be calculated by summing the penalties of the feature requirements that can not be met by the schedule. For instance if meeting the deadline and meeting the reliability goals are the only two feature requirements of a product and one is twice as preferred as the other, then the benefit of a schedule that meets only the more preferred one is 2/3. The benefit of a schedule can be calculated by subtracting the penalty of relaxing the feature requirements that can not be satisfied (within the relaxation limits allowed by the user) from one. We normalize the relative preference of feature requirements of a product by mapping their sum to one: (1) "one" quantifies the case that all feature requirements of the project product are satisfied at the desired level, and (2) the relative preference of each feature requirement is normalized to a number between zero and one.

In the following definition, we have shown that how the benefit of a schedule can be calculated formally for the case that each feature requirement can be met at only one level. The benefit has been calculated in terms of two intermediate functions: Ω which returns the penalty of not meeting a feature requirement of the project product p, and σ which indicates whether schedule Λ of p meets each feature requirement.

Definition 3.1: benefit of a schedule Λ

Let Λ be a schedule of Γ (a process plan for a product p) and $\phi = \Phi_i$. Furthermore, let

- $\Omega: \Phi \rightarrow [0\ 1]$ be the penalty of not meeting each Φ_i, and
- $\sigma: \Phi \rightarrow \{0,1\}$ be such that $f(\sigma)=1$ iff Λ does not meet ϕ.

Then $B(p,\Lambda) = 1 - \sum^{\phi \in \Phi} \Omega(\phi) \times \sigma(\phi)$.

Later in this section, we discuss that how this scheme can be generalized to the case that each feature requirement can be relaxed to different degrees.

Example: Consider the specification of penalties and the ability of two alternative schedules of a process plan for developing the simulator in figure 2-1 to meet its feature requirements as depicted in table 3-1. The column headings of table 3-1 denote the feature requirements of the simulator and the row headings denote alternative schedules as well as the penalty of not meeting each feature requirement.

To be able to fill out the rows that correspond to schedules *1* and *2*, first the reservations needed to meet each feature requirement of the simulator have to be identified, and then it has to be determined whether those reservations are satisfied under the schedules that are being compared. In table 3-1, *X* is used to mark if a feature requirement is met under a given schedule. Once table 3-1 has been filled, the formula in definition 3.1 can be used to calculate the benefit of a given schedule of the simulator: $B(p,\Lambda_1) = 1 - .1 = .9$ and $B(p,\Lambda_2) = 1 - .3 = .7$. We therefore prefer the first schedule over the second.

Relaxation of capacity constraints refers to increasing the supply of resources by purchasing them off-the-shelf or by renting them. Therefore, the degree of satisfying a capacity constraint in a schedule can be measured by calculating the cost associated with relaxing the constraint. If a capacity constraint is not relaxed, then it will incur no cost (zero cost). The combined satisfaction of all capacity constraints under a schedule can be measured by summing the benefit associated with each capacity constraint.

Simulator	meet-the deadline	reliable	well documented	portable
production plan 1	X	X	X	
production plan 2		X	X	X
penalty	.3	1	.1	.1

Table 3-1: Input for the Measurement of Local Benefit

The fourth step in developing an evaluation function for a scheduling problem domain consists in investigating the comparison of the metrics developed for measuring the satisfaction of each type of relaxable constraints by bringing them on a common scale. This can be used as a basis to measure the overall satisfaction of all problem constraints. Since feature requirements and capacity constraints are the only types of relaxable constraints in SPPS, the evaluation function in SPPS measures the combined satisfaction of both feature requirement and capacity constraints under a given schedule. The value of this satisfaction is maximized if the benefit of that schedule is maximized and its cost is minimized (i.e. a schedule is preferred over another if it balances the satisfaction of a more important subset of feature requirements or incurs a smaller cost overrun). Since the cost is normally stated in dollars while the benefit is typeless, we rely on a user specified weight factor to bring the two on a common scale and preferring one over the other. The evaluation function to measure the overall desirability of a schedule in SPPS can be built by taking the ratio of cost and benefit of the schedule and then multiplying the result by the weight factor. The evaluation function measures the distance *cd* between the schedule being evaluated, and the ideal schedule:

Definition 3.2: Consistency-Distance *cd*

Let Λ be a schedule of Γ (a process plan of Π which produces p). Then

$cd(p,\Lambda) = \frac{C(\Lambda)}{B(p,\Lambda)} \times w$ s.t. $0 < w \leq 1$ is a weight.

The smaller the value of the evaluation function for a given schedule, the more desirable the schedule. Therefore, the goal of search should be to minimize the value of evaluation function. The weight w can be specified as a function (e.g. constant, linear, quadratic) depending on the requirements to tradeoff cost and benefit in a given SPPS problem. However, it is usually suitable to specify w as a step function of cost and benefit. The reason for using a step function (as opposed to a constant function) is that the weight should reflect the magnitude of both cost and benefit. For instance, if the benefit is already low but the cost is very high, the value of w should favor cutting both the cost and benefit in half by multiplying the benefit (denominator) by a positive integer which is greater than 1 (how large this number should be depends again on how important it is to cut both the cost and benefit by similar ratios). On the other hand, if the benefit is large relative to the cost, then a weight that discourages cutting both the cost and benefit by similar ratios might be more appropriate. By defining w as a step function, we will have the freedom of associating a different weight with different ranges for the cost and benefit. Moreover, it is relatively convenient to develop a step function than high order polynomial or non-polynomial functions that reflect the actual tradeoff needs between the cost and the benefit.

The value of evaluation function in definition 3.3 is always non-negative since the cost function always returns a non-negative value and the benefit function always returns a positive value which ranges between *0* and *1* (excluding *0*). *0* is excluded from the range of benefit function since the search algorithm is prohibited from compromising any feature requirement beyond the maximum level of relaxation allowed (a level that yields a minimal yet positive benefit). The fact that the benefit of a schedule is always a positive number prevents the emergence of the situation in which both cost and benefit are zero at the same time (i.e. evaluation function is ambiguous). A schedule is consistent if its cost is zero and its benefit is one. The value of the evaluation function can be interpreted as the balance between the degree of satisfaction of feature requirements that are met and the cost overrun that is incurred in order to meet those requirements in a given schedule.

Moreover, according to definition 3.3, two schedules are equally optimal if the cost of each is zero but the benefit of one is greater than the other. In this case, the scheduler insures stability (i.e. it will always stick with the schedule that has the highest benefit), since it avoids a compromise (of a feature requirement) when the cost of implementing the schedule is already zero.

To analyze the opportunities (possible commitments) at a given point during search, the evaluation function can be used to measure the progress due to making a commitment. Earlier in this section, we stated that this progress can be measured by deducting the inconsistency of Λ_2 (the new schedule) from the inconsistency of Λ_1 (the old schedule). This is formalized in the following definition:

Definition 3.3: progress due to a commitment

Let Λ_1 denote the previous schedule of p and Λ_2 denote the schedule of p after making a commitment s. Then $cd(p,\Lambda_1) - cd(p,\Lambda_2)$ measures the scheduling progress due to s.

For an indication of the utility of making a commitment s during search, let C_1 and C_2 denote the costs of implementing Λ_1 and Λ_2, and B_1 and B_2 denote the benefits of Λ_1 and Λ_2 respectively. Then, according to definition 3.3, s is desirable if and only if $C_1=C_2$ but $B_2>B_1$, or $C_1>C_2$ but $B_1=B_2$, or $C_2>C_1$ but $B_2>B_1$ at a greater proportion, or $C_2<C_1$ but $B_2<B_1$ at a smaller proportion; otherwise, s is undesirable.

Our methodology for measuring the cost and benefit of a schedule can be extended to evaluate process plans as well. Since each process plan can have many schedules, then we define the cost of a process plan Γ as $min[C(\Lambda \ \forall\Lambda \in \Lambda(\Gamma)]$, where $\Lambda(\Gamma)$ denotes the set of all schedules of Γ, and the benefit of a process plan Γ of a product p as $min[B(p,\Lambda \ \forall\Lambda \in \Lambda(\Gamma)]$.

In the following two sections, we discuss the low level issues related to the measurement of cost and benefit. These include: (1) how to measure the cost associated with relaxing a capacity constraint, and (2) how to figure out which feature requirements will be met given a set of reservations.

3.1.1. Cost Associated With Relaxing A Capacity Constraint

The cost of purchasing (or renting) a primitive resource to increase the capacity of that resource to its relaxed level[25] depends on both the time window over which the capacity is requested and the capacity size. The relaxed capacity can be viewed as a *resource request* and is of the form ζ. The cost of a resource request ζ can be calculated from the marginal cost of purchasing or renting the resource, a function that is available.

---

[25]We call the difference between the before and after relaxation capacity of a resource the *relaxed capacity* of that resource.

Marginal cost of an unshared primitive resource (e.g. manpower, off-the-shelf hardware) returns the cost of acquiring each incremental unit of that resource over time intervals of different length. This cost is allowed to vary in relation to the capacity size as well as in relation to the length of the interval over which the capacity is requested. For instance, the average cost of hiring a programmer can be specified larger if he is hired on a month-to-month consulting basis than when he is hired as a regular employee, as is the case in the real world. We use $MC(r,[t_b\ t_e],q_r,q_a)$ to denote the marginal cost of acquiring q_r new unshared primitive resource r over a period $[t_b\ t_e]$ when q_a of r has been already acquired. Marginal cost of a shared resource is only a function of the duration that the resource is requested because shared resources are infinite capacity. We use $MC(r,[t_b\ t_e])$ to denote the marginal cost of acquiring q_r shared primitive resource r over a period $[t_b\ t_e]$.

The cost of satisfying a request for a resource r can be calculated using the marginal cost of that resource. First the request is broken to more primitive requests that their cost can be directly returned by the function, and then the component costs are summed up (aggregated). If r is a shared primitive resource, then the aggregation takes place only along time as the capacity of a shared primitive resource is infinity. This is to insure that the cost of a shared primitive resource will not be counted more than once during any period. The cost of satisfying a resource request ζ (for a resource r) is formally defined in definition 3.4. If r is an unshared primitive resource, however, aggregation takes place along both time and capacity of the resource. Definition 3.5 calculates the cost of satisfying the relaxed capacity of r by summing the costs of satisfying each unsatisfied request for r. Since definition 3.5 does not optimize the cost of the relaxed capacity of r with respect to the marginal cost of r, we have developed a mechanism to preprocess the unsatisfied requests for r for optimization purposes before attempting to measure their combined cost. This mechanism is discussed in the next section.

Definition 3.4: cost of satisfying a resource request ζ

$$\texttt{if } r_\zeta \in R_{Spri} \texttt{ then } \sum^{(t_b\ t_e)\in\zeta} C(r_\zeta,[t_b\ t_e])$$
$$\texttt{else if } r_\zeta \in R_{Upri} \texttt{ then } \sum^{(t_b\ t_e\ q)\in\zeta} C(r_\zeta,[t_b\ t_e],q)$$

Definition 3.5: cost of q_r unshared primitive resource r over $[t_b\ t_e]$

let Ξ be the set of largest adjacent subintervals over $[t_b\ t_e]$ s.t. the present capacity of r over each subinterval is constant. Then

$C(r,[t_b\ t_e],q_r) = \sum^{I\in\Xi} MC(r,I,q_r,q_a)$.

of satisfying a request of the size equal to the difference of the relaxed capacity and the original capacity of r.

3.1.2. Unification

Unification is a form of preprocessing the unsatisfied resource requests in a schedule before attempting to measure their combined cost. The purpose of unification is to translate the requests for a resource to utilize its marginal cost. The translation will result in reduction in the number of requests and also in minimization of the cost to satisfy them. These features of unification are formally stated in definition 3.6. In definition 3.6, unification is defined as a translation function f:

Definition 3.6: Unification

$f:\{\zeta_i\} \rightarrow \{\zeta_j\}$ is an onto function s.t. $Card(\{\zeta_j\}) \le Card(\{\zeta_i\}) \ \wedge\ C(\{\zeta_j\}) \le C(\{\zeta_i\})$.

Therefore, the requests that emerge from unification are at least as desirable as the initial resource requests.

The advantage of reducing the number of requests during the scheduling is that the fewer the number of unsatisfied resource requests (conflicts), the fewer iterations of the algorithm will be needed (each incremental iteration of the algorithm is to resolve one conflict). Minimization of cost in unification is different from the cost reduction that results from discovering the correlation between requests although the end result will be the same in both cases.

The reason for unification is to optimize the requests for a resource with respect to the marginal cost function of that resource which is often not a linear; see figure 3-2 which illustrates the marginal cost of acquiring a programmer as a concave curve. According to figure 3-2, the average cost of acquiring a programmer drops with the duration that he is required. This is because hiring of most human resources on a month-to-month consulting basis is more costly on the average than hiring them over a longer period as regular employees. On the other hand, the average cost of purchasing hardware or the right to use a software decreases with the quantity purchased or installed. For instance, the average cost of a workstation to a company drops with the number of workstations being bought.

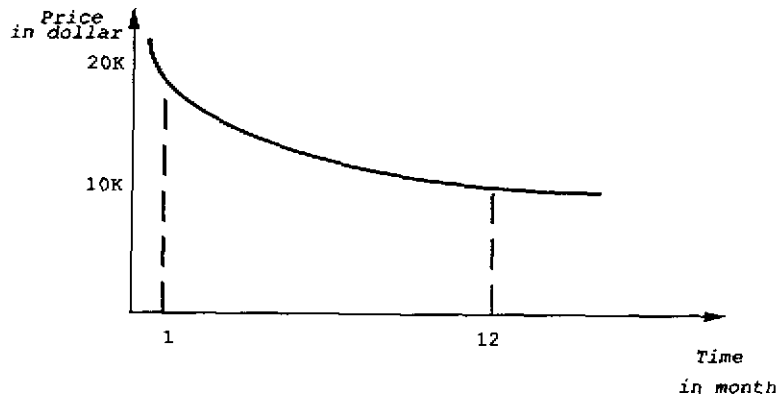


Figure 3-2: Marginal Cost Curve for Programmer

Unification is implemented through leveling temporally adjacent resource requests of the same resource. The leveling effectively coalesces any two adjacent resource requests that are at the same level until the smaller request is completely consumed. The resource requests that are leveled need not be fully adjacent as long as the "no request" whole between them is sufficiently narrow in comparison to them. Unification is different from existing resource leveling techniques in that it tries to optimize the marginal cost function of the resource. This implies that the shape of the leveled curve will depend strictly on the marginal cost function of the resource while existing resource leveling algorithms [53] implement cost independent leveling heuristics (i.e. the aggregate demand curve[26] of a senior programmer will be leveled exactly the same way that the aggregate demand curve for a workstation is).

Consider the aggregate demand curve (aggregation of 15 requests) for programmer in figure 3-3. The cost of satisfying these demands as they are stated amounts to $3,074,000. By breaking down these requests vertically at the points that the request level changes (figure 3-4), the number of requests is reduced to 7 and the cost of satisfying them is lowered to $2,380,000. By breaking down the requests horizontally (figure 3-5), the number

[26]Aggregate demand curve for a resource can be constructed by aggregating all resource request curves of that resource

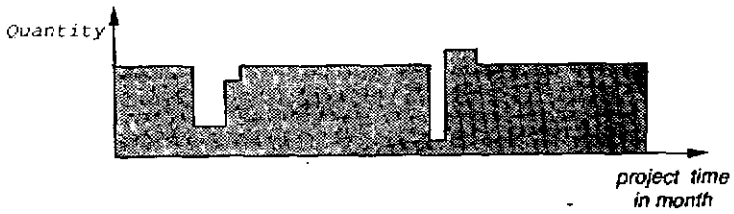


Figure 3-3: Aggregate Demand Curve for Programmer

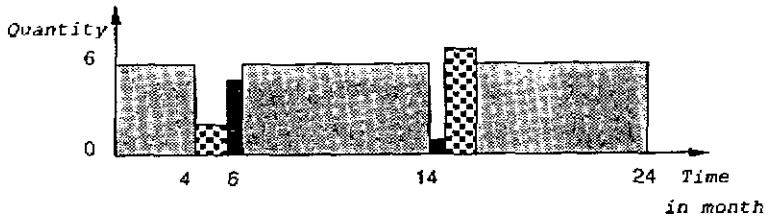


Figure 3-4: Vertical Breakdown

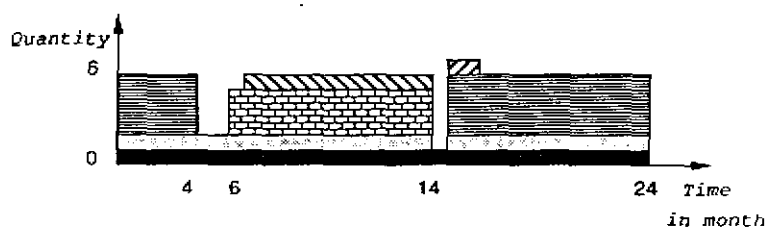


Figure 3-5: Horizontal Breakdown

of requests will increase to 8 but the cost of satisfying them is lowered further to $2,080,000. However, if we unify the requests (figure 3-6) then the number of requests will drop to 2 and the cost of satisfying them drops to $1,768,000.

Below the kernel of the algorithm for unification is presented. This algorithm should be executed for each primitive resource (member of R_{pri}) once. Suppose Λ is a schedule of p, d_Λ is the duration of Λ, and r is the resource such that the resource requests of r are to be unified. Furthermore, suppose s_Λ, the start time of Λ, is mapped to 0. This mapping can be obtained by the translation of absolute times. The input of the algorithm

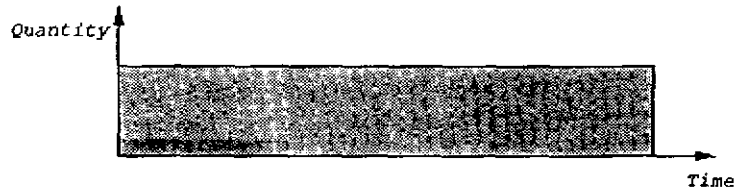


Figure 3-6: Unified Demand Curve for Programmer

consists of the unsatisfied request list of a resource *r*. This list is stored in the array *ulist* that has a dimension d_Λ. Each index of the array stores the size of unsatisfied request at the point in time that is designated by the array index. *ulist* can be calculated by summing the requests for *r* and subtracting the result from the available capacity of *r*. Since unsatisfied resource requests denote a deficit of capacity, *ulist* is stored as an array of negative capacities. The output of the algorithm, the new set of unsatisfied resource requests, will be stored at *unified-requests* at the end of execution. The new set of requests will consist of a list of triples of the form *<s d ulevel>* where *s* is the start time of the request, *d* is the duration of the request, and *ulevel* is the quantity requested. The algorithm includes two loops: and inner loop and an outer loop. The outer loop iterates over all intervals that are shorter than d_Λ. The inner loop iterates within the d_Λ interval with steps equal to the size of the interval fixed by the outer loop combining adjacent resource requests when the cost of the combined request is less than the sum of the cost of individual requests. Then the algorithm is as follows:

```
unified-requests=∅
ulist = ρ_r − Σ^{q∈P} h_q(r)
    {where h denotes requests of q for r that remain unsatisfied, and
    P={x|x∈R_p∩R_pro}∪p}
for d = d_Λ down-to 1
  j = 1
  while (j + d) ≤ d_Λ
    ulevel = 0
    demand = Σ_{i=j}^{j+d_Λ−1} if ulist(i)<0 then 1 else 0
    if (demand × C(r,1,1,q_a)) > C(r,d,1,q_a)
    then
      ulevel = ulevel + 1
      for k = j to j + d
        if ulist(k) < 0 then ulist(k)=ulist(k)+1
      endfor
    else
      if ulevel > 0
      then
        push(<j d ulevel> , unified-requests)
        j = j + d
      else
        j = j + 1
        while (ulist(j) = 0)
          j = j + 1
      endif
    endif
  endwhile
endfor.
```

In the worst case, the above algorithm requires $1+2+...+d_\Lambda$ or $\frac{Card(d_\Lambda)\times(Card(d_\Lambda)-1)}{2}$ steps to measure the unified cost of r. This occurs when the height of aggregate demand curve for r is very uneven over a period. The best case occurs when the height of resource requirement curve for r is constant. In that case, the algorithm will require only l steps where l is the height of the constant resource requirement curve. The quadratic complexity of the algorithm can be eliminated if the "while loop" is always incremented by cd. This is because the algorithm will need to execute only $\frac{d_\Lambda}{1}+\frac{d_\Lambda}{2}+...+\frac{d_\Lambda}{d_\Lambda}$ steps. To achieve this however, one would risk the chance of computing a slightly inaccurate cost.

Without unification, the computational cost of measuring the dollar cost of satisfying the relaxed capacity of r is $O(1)$. However, calculating the relaxed capacity of r requires $O(n+1)$ where $n+1$ is the number of intermediate products produced by the project.

3.1.3. Propagation of Reservations to Feature Requirements

At the beginning of this chapter, we stated that we have developed a scheme to evaluate a schedule (or a sequence of scheduling decisions) on the local basis when local measurement is a good estimator of the global evaluation, and on a global basis otherwise. Moreover, we stated that this scheme involves constraint propagation. In this section, we describe the scheme that we have developed, for evaluating the benefit of a schedule.

To measure the benefit of a schedule (for a product p), a scheme is needed to find the feature requirements of p

that will be met given the reservations available to the schedule. This scheme can be extended to measure the benefit of a scheduling commitment by fixing the reservations available to the schedule and finding the change in meeting each feature requirement if the new reservations tied to making the scheduling commitment are taken into account.

If the local benefit of a set of reservations (i.e. the benefit of the feature requirements of the products that have directly required those reservations) is a good estimator of the global benefit of those reservations, then the effect of reservations need not be propagated to all other project products that are indirectly influenced by the reservations. Otherwise, the effect of reservations have to be propagated globally to all other products produced by the project.

The benefit of a set of reservations is measured locally when

1. there is a fair idea about the usefulness of a scheduling commitment to the entire schedule.
2. there is not enough time to measure the global usefulness of a scheduling commitment.

The drawback of local measurement of benefit is that global consequences of unmet feature requirements are speculated not measured. On the other hand, it has the advantage of being computationally faster. In contrast, this benefit is measured globally when

1. there is a great deal of uncertainty about the global usefulness of a local scheduling commitment.
2. there is enough time to measure the global benefit.

The advantage of measuring the global benefit of a scheduling commitment is that global consequences of a local decision will be taken into account. If the global benefit is measured, then the need for specifying the penalties will be eliminated in all but the level that ultimately makes the ruling. The main drawback of global measurement of benefit is that it is computationally slow.

In the global measurement of benefit, to measure the degree that Λ meets each feature requirement of p, the reservations of Λ (i.e. π) have to be propagated to reach the feature requirements of p. The reason for propagation is that the resource requirements of p are often expressed recursively (by the agents who work on the project) in terms of more primitive products than directly in terms of primitive resource requests. Moreover, it is not possible to measure the satisfiability of feature requirements on the basis of available capacity of each primitive resource because this capacity can be divided between the resource requests in different ways each causing the feature requirements to be satisfied differently.

Since the computational cost of propagating a reservation ζ increases rapidly (with the number of levels that separates ζ from p), it is desirable to measure the effect of that reservation locally when the local impact of satisfying a reservation is believed to be a good estimator of the global impact. Under a centralized authority scheduling, the propagation stops when the effect of all feature requirements of intermediate products which have been affected by propagating ζ have been neutralized (e.g. the delay in the production of a product is offset by the slack of that product) or when p has been reached. In a distributed authority framework, the propagation can also stop when it reaches an agent who has the authority to make preference ruling on the feature requirements of the product that the propagation has reached (under a centralized authority framework, this agent is always positioned at the final product of the project). If the authority has been delegated conditionally, then the condition for delegation (see section 3) has to be evaluated first to determine whether the authority can be delegated. A major limitation in applicability of stopping the propagation after reaching a local agent with appropriate authority is that it is allowed only when the propagation being stopped is the only path along which propagation is taking place.

If there are multiple paths along which propagation is taking place, then propagation should continue even if the authority to rule on the preferences of feature requirements of every product that a propagation path has reached has been delegated to local agents. This is because the interests of the local agents clashes and we have no scale to weigh (balance) the preferences of each local manager against the others. In general, the propagation to measure the global benefit under distributed scheduling should continue along each path until all propagations reach a *single product*. When all propagation paths have reached a single product, then the evaluation function only needs to consider the preferences of the agent who possesses the authority to rule on that product. Since a reservation might be required by multiple products, the propagation path at every product point could spread to many more product points (spawn multiple propagation paths). In other words, every step of propagation could yield a *list of* products, and also for each product the feature requirements of that product which can be met.

The computational cost of propagation is proportional to the number of products that propagation spreads to. At each propagation step this is equal to the number of products that require the resource from which propagation is being spawned. In the worst case, the cost of propagation is equal to the number of intermediate products produced by the project. The best case occurs when no propagation is needed (one of the propagation stoppage criterias is met before any propagation has started). In appendix F, we have described an algorithm to measure *global benefit* of a scheduling commitment with respect to a product p if all other scheduling commitments are fixed:

A major point in our propagation scheme is that it is the *feature requirements* not the *penalties* that are propagated. Propagation of feature requirements amounts to propagating the data on the satisfaction of product features until a stoppage criteria is satisfied. In contrast, if the penalties were propagated then the measured benefit would have been the combined preference of all agents while we are interested in the preference of the customer.

3.1.4. Multiple Level Relaxation

In this section we discuss that how the scheme for measuring the benefit of a schedule can be generalized to the case that each feature requirement is relaxable to different degrees. If the feature requirements of a product can be met only at one level, then it will not be possible to specify that a feature requirement is only partially satisfied under a fixed schedule. If the binary restriction on meeting a feature requirement is relaxed, then each feature can be satisfied at multiple levels.

The main advantage of allowing a feature requirement to be relaxed at multiple levels is that the satisfaction of the feature requirements of a product and the balance between the cost and the benefit can be specified at a finer level of detail. The disadvantages of allowing a feature requirement to be relaxed at multiple levels are:

1. the resources required to meet a feature requirement at multiple levels should be specified independently.
2. the scheduler should search a larger space for finding a solution.

The following definition extends definition 3.1 to the case that each feature requirement can be met at multiple levels. As in definition 3.1, the benefit has been calculated in terms of two intermediate functions: Ω which returns the penalty of not meeting a feature requirement of the project product p, and σ which indicates whether schedule Λ of p meets each feature requirement.

Definition 3.7: benefit of a schedule Λ

Let Λ be a schedule of Γ (a process plan for a product p) and $\phi = \Phi_i$. Furthermore, let

- $\Omega:(\Phi,\alpha) \rightarrow [0\ 1]$ be the penalty of not meeting each Φ_i at each level α_{ij}, and
- $\sigma:(\Phi,\alpha) \rightarrow [0\ 1]$ be such that $\sigma(\Phi_i,\alpha_{ij})=0$ iff Λ meets Φ_i at or above the desired level that generates a benefit.

Then $B(p,\Lambda) = 1 - \sum^{\phi \in \Phi} \Omega(\phi,\alpha_{ij}) \times \sigma(\phi,\alpha_{ij})$ where α_{ij} is the level at which Λ meets Φ_i.

Since each requirement of a product is allowed to be met at different levels and each level can be met with different mixes of resources, the amount of specification needed to propagate a set of reservations would grow rapidly with the number of products being produced by a project. By observing that the specification of different resource mixes and also the specification of resources needed to meet a feature requirement at different levels are only slightly different (their difference is in the quantity of some required resources), we use an alternative form of specification which only requires the resources that will be saved if that feature is relaxed. We require the specification of a 1-1 onto function $h:(\Phi,\alpha) \rightarrow \eta$ that returns the possible resource savings from relaxing a feature requirement Φ_i of each project product to a degree α_{ij}.

The advantage of the new form of specification is that (1) agents need to specify the resources that will be saved only for those feature requirements that can be relaxed, and (2) there is no redundancy in the specification (in contrast to the previous form that repeated the part of specification common between different resource mixes of each relaxed level of a feature requirement). The disadvantage of this scheme is that the interaction between feature requirements can not be captured. This, however, is offset by the fact that these interactions are infrequent in the software project scheduling domain.

Interaction between feature requirements is manifested in a set of resource requirements that once met result in the simultaneous satisfaction of multiple feature requirements. To capture the interaction between feature requirements in the feature-to-cost traceability scheme, the scheduling system needs to be extended to support the specification of a new class of resources that has limited sharability. This means allowing a resource to be sharable among a selected set of products while remaining unsharable to others. By specifying an unshared resource 'shared' among the feature requirements that interact then a resource satisfaction for that resource can be tied to the simultaneous satisfaction of multiple feature requirements thereby capturing the interaction among them.

Example: In this example, we examine that how a scheduler can decide the relaxation of a feature requirement in a product when there exist multiple feature requirements, some of which can be relaxed at multiple levels. The product being considered is a software module, the feature requirements being considered are quality assurance (QA) and documentation (Doc), and senior programmer (Sprog) is the only resource being considered for relaxation:

Φ	Ω	Resources Saved
QA	.1	.5 fewer Sprog Needed
Doc	.1	1 fewer Sprog Needed
	.2	1.5 fewer Sprog Needed

If the weight w (of cost with respect to benefit) is *1*, then the relaxation of documentation at the first level is the most appropriate choice. This is because $\frac{.1}{1}$ is smaller than both $\frac{.2}{1.5}$ and $\frac{.1}{.5}$.

3.2. Consideration of Preferences

It is important to account for preferences in evaluating a schedule because frequently problem constraints can not be met at full. When a set of schedules can not meet the problem constraints at full, then they be compared on the basis of the degree that each meets constraint related preferences and also other types preferences.

In section 2.2, we described that a software project needs to consider a wide range of preferences including the preferences among resource or resource mixes, process plans of a product, different feature requirements of a product, degrees of meeting each feature requirement of a product, degrees of relaxing available capacity constraints, and degrees of relaxing required capacity constraints. We divide these preferences to two groups based on whether they represent the utility (in actuality the penalty) of relaxing a constraint to varying degrees. The preferences that represent the utility of relaxing a problem constraint include feature requirement preferences (which represent the utility of relaxing a feature requirement constraint) and available capacity preferences (which represent the utility of relaxing an available capacity constraint). Much of the discussion in this chapter up to this point concerned how the value of evaluation function is calculated on the basis of these preferences. More specifically, we described how the dissatisfaction with having to commit to the relaxed capacity of a resource or the penalty associated with relaxing the feature requirements of a product can be measured and traded off.

The second group of preferences include the preferences that do not represent the utility of relaxing a constraint. These preferences include resource or resource mix preferences, process plan preferences, start time and completion time preferences, and feature requirement preferences (between feature requirements). In contrast to the first group, this group of preferences are dynamic in the sense that they depend on the scheduling state[27]. For instance, the preference of a resource over another, which is substitutable with the first, depends on the available capacity and the acquisition cost of each, both of which depend on the scheduling state. The available capacity of a resource depends on the scheduling state since it is calculated by subtracting the reserved capacity of the resource under the schedule that represents the scheduling state from the initial available capacity. The acquisition cost of a resource also depends on the scheduling state because it is a function of the capacity which is required but is not available under the schedule that represents the scheduling state. The first group of preferences are static because they are solely dependent on the agent preferences and are independent of the scheduling state.

The specification of dynamic preferences is redundant in evaluation approaches that are based on cost/benefit analysis since they themselves are derived from cost/benefit analysis results. For instance, the preference of using a resource over using another resource in an activity is the result of comparing the cost involved in acquiring each when they make no difference to the activity, and the result of comparing both the cost and benefit involved in choosing each when they make a difference to the activity.

Our evaluation function can also be extended to model other preferences that do not belong directly to either of the main categories which we defined. For instance, it is preferred to assign the same programmer to two activities (e.g. A and B) that involve developing similar software (if each activity requires only a portion of the time of a programmer) than assigning each activity to a different programmer (because the programmer productivity will he higher under the first arrangement). This can be viewed as preferring a resource mix that

[27]We define the scheduling state as the schedule that is being revised at any given point during scheduling.

involves assigning the same instance of a resource (programmer in this case) to a group of activities (two in this case) over another (which involves assigning different instances of the resource).

To model these type of preferences, we need to support the specification of alternative resource mixes and alternative process plans for a *group of activities* in addition to supporting it only for a single activity, and also use a set of operators (built from the primitive operators in our heuristic search model) that can manipulate a group of activities. Then, the above preference can be indicated by specifying two alternative resource mixes for a group of two activities, *A* and *B*. Suppose that the first resource mix, which involves using two different programmers for *A* and *B*, requires a total of 3/2 of programmer. Moreover, suppose that the second resource mix, which involves using the same programmer for both *A* and *B*, requires less programmer time (e.g. 1 programmer) because of the jump in programmer productivity. The evaluation function realizes the advantage of the mix that involves assigning the same programmer to both activities *A* and *B* because it incurs a smaller cost while achieving the same benefit (because both mixes meet product feature requirements equally).

Chapter 4
Scheduling Strategy And Search Operators

In chapter 2 we described that our basic heuristic search model is composed of three principle components: a set of *search (revision) operators*, each of which modifies a subset of the commitments comprising the current schedule to produce a new schedule, an *evaluation function* (i.e. a metric on the potential solution space), which provides a basis for comparing the transformations produced by the application of alternative operators to a given schedule, and a *scheduling strategy*, which specifies knowledge relating to use of the search operators and the evaluation function within the search (e.g. inconsistency prioritization heuristics, operator selection heuristics, termination criteria, etc).

In this chapter, we present the design of a set of schedule revision operators and a scheduling strategy for applying these operators during the scheduling process. Several previous efforts in manufacturing scheduling have considered the problem of incremental schedule revision[28]. The ISIS job shop scheduling system [41, 42] provides the capability to reschedule an order in response to the unexpected loss of required resources. This is accomplished by transforming the commitments pertaining to the problematic order into scheduling preferences, and generating a new schedule for the order. In situations where multiple orders are found to have schedule conflicts, the priorities of orders are used to determine the sequence of rescheduling them. Thus, in terms of the above heuristic search model, ISIS can be seen as utilizing a single search operator (i.e. the order rescheduling procedure). The overall revision strategy (highest priority order first) dictates a single trajectory through the space and thus there is no use for a global evaluation function[29]. The OPIS factory scheduling system [104, 128] implements a more sophisticated approach to reactive schedule revision. It emphasizes the use of several schedule revision operators, each with selective advantages in resolving certain types of scheduling conflicts, and operates according to an *opportunistic* scheduling strategy. More specifically, a heuristic theory relating the implications of current solution constraints (e.g. important reoptimization needs and opportunities) to the strengths and weaknesses of various revision operators is used as a basis for conflict prioritization and operator selection [115]. This enables the scheduler to focus immediately on those decisions most critical to overall schedule revision objectives as opposed to encountering them only after other restricting commitments have been made. This heuristic theory (scheduling strategy) is used in lieu of a global heuristic search[30]. A final approach to incremental schedule revision is implemented in the RESOURCE REALLOCATOR system [120], although in this case the problem addressed is quite different in that it is strictly a resource reallocation problem

---

[28]Both the ISIS and OPIS scheduling systems mentioned below also address the problem of schedule generation. We limit our attention here to issues relating to schedule revision.

[29]It should be noted that the "order rescheduling" procedure itself employed a heuristic beam search to locally explore alternative sets of commitments for the order being scheduled, and this search was focused by an evaluation function that reflected scheduling preferences relevant to the order (e.g. meeting the due date, utilizing preferred resources, etc)

[30]Although, as in ISIS, OPIS operators do exploit local heuristic search.

which is void of any temporal constraints. Nonetheless, the approach constitutes a heuristic search model that includes each of the principal components that can be found in our model [116].

Our overall incremental revision methodology is similar to that of OPIS [104] and includes choosing a conflict (focal point around which revision should be centered), selecting a revision operator (action) to resolve the conflict, applying the operator to the schedule, and repeating the process until all conflicts have been resolved. Detection of conflicts in the current schedule is the means by which the need for reaction is recognized. Constraint propagation in response to schedule changes can lead to detection of two types of conflicts in OPIS: time conflicts and capacity conflicts. Moreover, sometimes the aggregation of currently posted conflicts may be necessary. Conflict aggregation is intended to group together those conflicts that should be simultaneously addressed (e.g. conflicts that are caused by lack of the same resource). In this chapter, We provide a novel way of transforming conflicts through *unification* (a special form of aggregation that involves coalescing adjacent conflicts). The advantages of unification are that it aggregates conflicts only if the cost of satisfying the aggregate conflict is less than the combined cost of conflicts which are aggregated, whereas the benefit of the aggregate conflict is equal to the combined benefit of the conflicts that are aggregated.

The design of our search operators and operator selection heuristics differ from other approaches in two major ways. First, we define a scheduling search space that includes alternative process plans, and a problem solving model that integrates the search for a process plan with the search for a schedule that implements that process plan. This enables preferential concerns relating to process plan selection to be appropriately balanced against those relating to resource allocation and time interval selection.

Secondly, we formally define a criterion of *navigational minimality* for measuring the utility of design of a set of search operators. A set of operators is navigationally minimal if and only if it is the smallest set of operators that insure any given schedule is reachable from any other schedule. A navigationally minimal set of operators insures that it is feasible to start the heuristic search from any point in the search space in order to reach the desired schedule.

If we consider that the search space in SPPS is extended along many dimensions, it is important to allow selective disbandment of search or reprioritization of search operators along some dimensions. A navigationally minimal set of primitive operators (which constitute the building blocks of more complex operators) insure that this goal can be achieved without a major reorganization of the underlying heuristic theory of schedule revision - a task which is both difficult and costly. With respect to our formulation of the software planning and scheduling problem, we define a navigationally minimal set of primitive search operators. More complex search operators can be constructed by sequencing the primitive ones.

Our opportunistic scheduling strategy also differs from other related research in that it does not exclude global heuristic search. While we have developed a set of heuristics to relate the implications of current solution constraints to the strengths and weaknesses of various revision (search) operators, we use a global evaluation function to compare revision operators if the heuristics fail to subscribe to one.

In addition to the above differences, we also have defined an alternative heuristic strategy for opportunistic scheduling which focuses on minimization of disruption (or change) to the schedule as the primary criterion for operator selection. The purpose of this heuristic is to minimize the disruption of the project schedule which has undesirable political ramifications and complicates the coordination of interdependent activities of a large number of project teams as well. Within any opportunistic scheduling scheme, the revisions prescribed by a

selected operator to solve a particular conflict can lead to considerable disruption of the original schedule (i.e., create new conflicts and necessitate a large number of additional schedule revisions). Given the existence of a complete schedule, the search space is often highly constrained and thus provides little flexibility for revision. Disruption of (lack of stability in) the schedule over time is a particularly important concern in SPPS as (1) a project schedule serves to coordinate the interdependent activities of a large number of project teams, and (2) attempting to change the schedules of several other teams in order to fix the problems that has arisen in the schedule of one frequently has undesirable political ramifications. Although the importance of non-disruptive incremental revision is noted in OPIS [104], it is considered as secondary to reoptimization concern during operator selection. We believe that minimization of disruption as the primary goal also could be helpful in other distributed scheduling domains because disruptive revision might undo the commitments made during previous iterations of opportunistic scheduling by creating new conflicts at the points where previously resolved conflicts used to reside, thereby slowing down the convergence of scheduling. To minimize the disruption, we have studied the amount of disruption that is caused by each search operator, and have developed a set of heuristics to control the application of each operator on the basis of the disruption that it causes.

In the remaining of this chapter, first we discuss our conflict prioritization scheme, and then describe the design of primitive operators. In section 3, we explain the operator selection heuristics that are used in our basic heuristic search model. In the fourth section, we describe the advantages of designing complex operators from sequencing the primitive operators that are discussed in section 1, and show how these operators can be constructed.

4.1. Conflict Prioritization

Our goal in prioritizing the scheduling conflicts is to allow the scheduler to focus immediately on those decisions most critical to overall schedule revision objectives as opposed to encountering them only after other restricting conflicts have been resolved. In the previous chapter, we used the cost and benefit functions to measure the consistency distance of a schedule from the cost and benefit of individual conflicts (constraints that can not be satisfied). However, these functions can be used to measure the criticality of a conflict (urgency of resolving a conflict in relation to the urgency of resolving other conflicts). Since the criticality of a conflict is directly proportional to both the cost and benefit of that conflict (because the higher the cost and benefit of resolving a conflict, the smaller the chance that it can be resolved merely through a compromise), we measure it in terms of their product.

The advantage of using cost as a criticality factor is that it provides a qualitative measure of the expense of resolving a conflict (recall that a conflict is represented as variable capacities of a resource required over different time intervals). The advantage of using benefit as a criticality factor is that it provides a qualitative measure of the desirability of resolving a conflict from the stand point of project requirements.

Although our formulation of criticality accounts fully for both the cost and benefit of a conflict, in reality it is not desirable to compute the benefit of each conflict during every reactive scheduling cycle, from a computational cost stand point. In the present implementation of NEGOPRO, we measure the criticality of a conflict by relying only on the cost.

4.2. Primitive Operators

In this section, we address the issue of search operators. We first define the concept of *navigational minimality* as an operator design objective, and then describe a set of primitive operators for software project schedule revision that satisfy this property.

A search (revision) operator is an operator that transforms an input schedule to an output (revised) schedule:

Definition 4.1: Search (Revision) Operator

Let Λ^* be the set of possible schedules of Π and δ_i. Then

$\delta_i:(\Lambda^*,A)\rightarrow\Lambda^*$ where A is the set of possible arguments with which δ_i can be applied.

In the definition of each revision operator in the remaining of this section, the formalism will focus on the part of schedule that each operator transforms. This is provide a clearer understanding of the revision operator.

In order to define *navigational minimality*, first we need to state three other definition: composition of operators, orthogonal set of operators, and navigational completeness. Composition of a set of operators is the successive application of a sequence of operators to a schedule. The end result of this application will be a new schedule. A set of revision operators is orthogonal if and only if no member of that set can be written by a composition of other members of that set. A set of revision operators is navigationally complete with respect to a set of schedules if and only if for every two schedules in that set, there exists a composition of revision operators (that belong to the set of revision operators) that transforms one schedule to the other. In the following, the above definitions are restated formally.

Let S be the search space of the problem (which includes the set of all possible schedules Λ that can be constructed) and Δ be the set of all possible revision operators that is defined on that space. Then

Definition 4.2: Composition of Operators

$C: (\delta_1,...,\delta_n,S)\rightarrow S$ s.t. $C(\delta_1,...,\delta_n,\Lambda_1)=\delta_1(\delta_2(...\delta_n(\Lambda_1)))=\Lambda_2$
where $\delta_1,...,\delta_n\in\Delta \ \wedge \ \Lambda_1,\Lambda_2\in S$

Definition 4.3: Orthogonal Set of Operators

Δ is orthogonal iff $\forall\delta_j\in\Delta \ \forall\Lambda\in S \ \neg\exists \ \delta_1,...,\delta_n\in\Delta-\{\delta_j\}$ s.t $C(\delta_1,...,\delta_n,\Lambda)=C(\delta_j,\Lambda)$

Definition 4.4: Navigational Completeness

A set of operators Δ to navigate through a search space S is
navigationally complete iff $\forall\Lambda_1,\Lambda_2\in S \ \exists w=\delta_1...\delta_n \ \delta_1,...,\delta_n\in\Delta$ s.t. $C(w,\Lambda_1)=\Lambda_2$.

According to this definition, Δ is navigationally complete if and only if a machine scheduler that uses it can navigate from any points within the space of possible schedules of a project to another point within that space. A navigationally minimal set of operators would assure that it is feasible to start the heuristic search from any point in the search space (i.e. any $\Lambda\in S$) in order to converge to a solution.

Definition 4.5: Navigational Minimality

Δ is navigationally minimal for S iff Δ is orthogonal and navigationally complete.

Our principal goal in designing Δ for software project scheduling is navigational minimality. We divide the design process into two steps: (1) formal declaration of the space to be searched, and (2) development of a set of operators that are orthogonal and navigationally complete with respect to this space. In chapter 2, we formulated a search space S for software project scheduling that involves search along the following dimensions (which refer to as D):

1. the amount of resources that are available for allocation to a project can vary over time.

2. the same production (activity) can start at different dates.
3. the same production can be carried out with different mixes of resource capacities.
4. different productions might require the same resource. Then, if there is not enough of the resource to satisfy all requirements, a scheduler has to decide which production the resource should be allocated to or whether a product should be preempted from its resource so that the resource can be reallocated to another product.
5. product feature requirements can be compromised thus multiplying the number of ways that the requirements of a product can be satisfied.
6. The same production can be carried out with different process plans.

We now define a set of search operators that span this search space.

Definition 4.6: Operator Supply

$\delta_{supply}(\rho_x,\zeta) = \rho_y$ s.t. ρ_x and ρ_y denote the available capacity of r_ζ before and after adding the capacity ζ.

Supply involves resolving a conflict by providing the disputed resource. A project is an *open-ended* system that communicates with the outside world by receiving budgets and delivering products. The commitments that a parent organization has made about the budget of a project could change if the major requirements of the project can not be met under the current budget. An operator, *supply*, is defined to increase the supply of that resource when major requirements of the project can not be met under the current level of supply. Similarly another operator, *take-away*, can be defined to decrease the supply of a resource dynamically.

Definition 4.7: Operator Move

$\delta_{move}(\Lambda_x,l) = \Lambda_y$ s.t. Λ_x and Λ_y denote the schedule of a product p before and after the move, and $l \in [n\ m]$ denotes the amount of move (n represents the maximum possible left shift and m denotes the maximum possible right shift of Λ_x).

Move is to move the start date of a schedule by delaying or expediting it: the start date is expedited if the schedule is moved left and delayed if the schedule is moved right. Let p be a product and χ_j and τ_j denote the

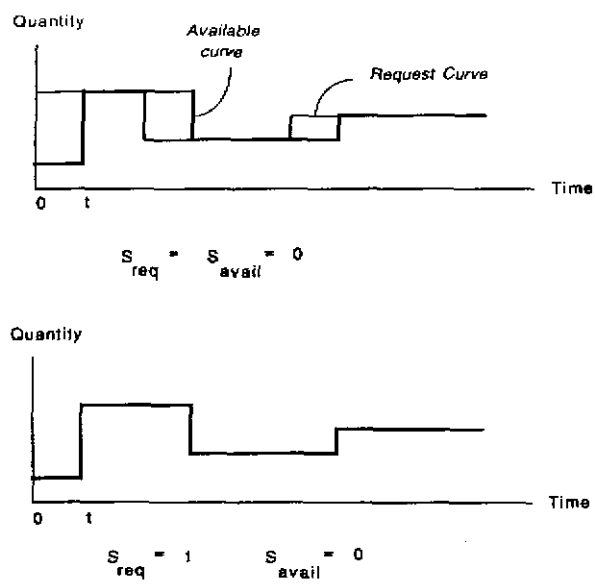


Figure 4-1: A Request Curve (a) prior to the Move (b) after the Move

j-th element of χ and τ. Then, to expedite the completion of p by l days, the following steps need to be executed in the order presented:

1. $\forall\zeta$ *s.t.* $r_\zeta \in (R_{pri} \cap R_p)$ first remove r_ζ from R_p and then return ζ to the list of available resources i.e. let $\rho_{r_\zeta}=\rho_{r_\zeta} \cup \zeta$. This would preempt p from the primitive resources that are allocated to it.
2. Let Φ_i denote the deadline feature requirement of p. Then let $\psi_i = \psi_i + l$ (this moves the completion deadline of p forward by l days).
3. $\forall\chi_j \in \chi$ and ζ the χ_j element of τ_j, if $r_\zeta \in R_{pro}$ then translate ζ by l on the time line.
4. Rebudget p. This would rebudget the primitive resource requests of p after they have been temporally moved.
5. $\forall q \in R_p$ move the schedule of q by $k \le l$ such that q is completed before p requires it. If q already becomes available prior to the shifted completion time (i.e. q already enjoys a *left slack* which is larger than l) then the schedule of q need not be moved. However if q holds no left slack w.r.t. p or holds a left slack that is smaller than l, then a left shift in the schedule of q is essential. The left shift has to be *propagated* to any q that holds an insufficient left slack w.r.t. to their product.

If Λ_p is moved left, then not only the schedule of s *s.t.* $s \in$ *product–transitive–closure+*(p) remains unaffected, but also the slack of s w.r.t. p will grow.

If Λ_p is moved right by l, then the move needs to be propagated to q *s.t.* $q \in$ *product–transitive–closure+*(p). If the *right slack* of p w.r.t. q is larger than l then q need not be moved. The right shift should be propagated recursively until the final products of the project are reached.

If the propagation reaches the final products of a project, then there is a chance that the duration of that project will increase. Duration of the project could increase even before the project has reached its final products. The recursive application of move-right along the path that starts with p should stop if the propagation causes the duration of the project to become the dominant cost factor.

Definition 4.8: Operator Substitute (Switch-Mix)

$\delta_{substitute}(\chi_x,y) = \chi_y$ where x and y are the indices of the set of selected mixes of resources before and after one of the selected mixes is changed.

Substitute involves switching from one mix of levels of resources that produces a product to another mix of levels of the same group of resources producing that product. For instance, by allocating more manpower to a project that includes many parallel tasks, the duration of that project could be reduced; figure 4-2.

Definition 4.9: Operator Reallocate

$\delta_{reallocate}(\pi_w,\pi_x,r) = (\pi_y,\pi_z)$ s.t. w and x are the present indices of the schedules of p and q, r is the resource to be reallocated from p to q, and y and z are the new indices of the schedules of p and q.

Reallocate is to reallocate an unshared primitive resource from one product to another. The application of *reallocate* to a schedule does not affect the overall cost of resolving the conflicts in that schedule; figure 4-3.

Definition 4.10: Operator Compromise

$\delta_{compromise}(\Phi_x,y) = \Phi_y$ where x is the index of the set of levels of meeting each feature requirement before the compromise and y is the index of the set of compromised levels of meeting each feature requirement.

Compromise is to lower the desired level at which the feature requirements of p should be met. Lowering the desired level at which the feature requirements of a product should be met in turn might affect the amount of resources that will be needed to meet the feature requirements of the product; this is illustrated in figure 4-4. Although it appears that a lower amount of resource requests will always lower the cost that is incurred for

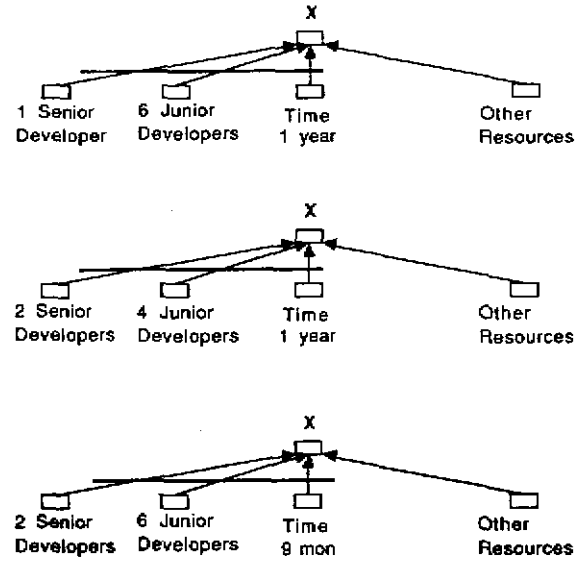


Figure 4-2: Different Mixes of Levels of Resources Produce the Same Product

	Module X	Module Y
Before Reallocation	10 Senior Prog 40 Junior Prog	2 Senior Prog 20 Junior Prog
After Reallocation	8 Senior Prog 40 Junior Prog	4 Senior Prog 20 Junior Prog

Figure 4-3: Reallocation of Senior Programmers From Module X to Module Y

satisfying those requests, if the request that is lowered belongs to a shared resource r and the lowering refers to shortening the period I that a product p requires r, then the relaxation might save no new cost. This is because although the request of p for r is lowered, other products could continue to require r over I. The demand for r over a designated period I can be eliminated only if all requesting products drop or compromise their requests for r together.

Definition 4.11: Operator Switch-Plan

$\delta_{switch-plan}(\Gamma_x, y) = \Gamma_y$ where x is the index of the present process plan among all process plans of the present schedule and y is the index of the new process plan to which the switch is to be made.

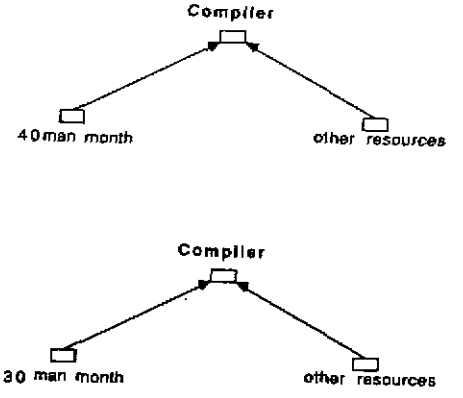


Figure 4-4: Compromise: compiler (a)has (b)does not have high reliability requirement

Switch-Plan is to switch from the present process plan to another. Suppose Z is an application software. Moreover, suppose that Z can be produced in three different ways:

1. Acquiring an application generator off the shelf to expedite the development.
2. Subcontracting the entire production to another firm.
3. Producing *Z* entirely inhouse.

In each case an entirely different process plan will emerge (see figure 4-5).

The procedures that implement each operator that we have described invoke two primitive operators: *release* and *allocate*. However, the procedures that implement each operator can not be expressed as a sequence of calls to *release* and *allocate* because they also include other decision constructs and heuristics.

Definition 4.12: Operator Release

$allocate(\Lambda_x,\zeta) = \Lambda_y$ where $\zeta = \pi_x - \pi_y$.

The capacity of r_ζ that is released from p will be added to ρ_{r_ζ}.

Definition 4.13: Operator Allocate

$allocate(\Lambda_x,\zeta) = \Lambda_y$ where $\zeta = \pi_y - \pi_x$.

The capacity of r_ζ that is allocated to p will be subtracted from ρ_{r_ζ}.

For instance, *reallocate* can be constructed from *release* and *allocate* as follows:

$\delta_{reallocate}(\Lambda_w,\Lambda_x,r) = (\Lambda_y,\Lambda_z)$ s.t. $(release(\Lambda_w,\zeta) = \Lambda_y) \wedge (allocate(\Lambda_x,\zeta) = \Lambda_z)$

Let $S^* = \{supply, take\text{-}away, reallocate, move, substitute, compromise, switch\text{-}plan\}$. Then

Claim1: S^* is Navigationally Minimal

Proof: We show that S^* is navigationally minimal by proving that it is navigationally complete and also it is orthogonal. Let $f:D \rightarrow S^*$ be a function that returns the name of the operator that spans the search space dimension denoted by the domain value such that *f(1)=supply f(2)=take-away f(3)=move f(4)=substitute f(5)=reallocate f(6)=compromise f(7)=switch-plan.* Clearly f is 1-1 onto. Therefore, S^* is navigationally complete. Furthermore, since the dimensions of the search space are orthogonal (by the definition of each dimension), the cardinality of S^* is minimal.

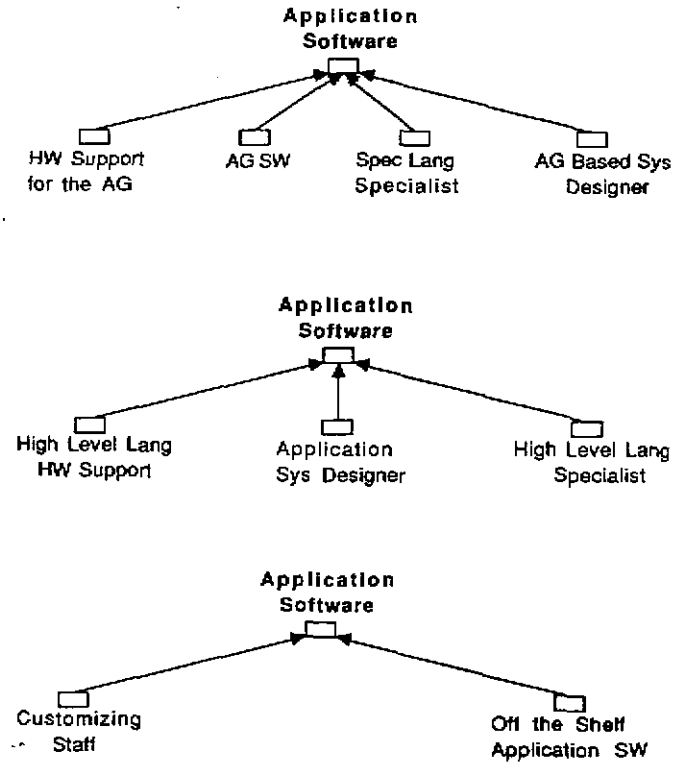


Figure 4-5: Three alternative process plans for developing an Application Generator

Given the generality of our formulation of the problem, we argue that S^* can be used to realize the same results as the "larger-grained" operators employed for incremental schedule revision in previous approaches[31]. In the following, we demonstrate how two operators employed by RESOURCE REALLOCATOR [120], a *transaction* and the more complex *cascade* of transactions, can be constructed by a sequence w of operators in S^*.

Suppose that G is a multiproject software developing organization. G begins a new project Π in an environment that several projects are already in progress each controlling a set of resources. Π usually has to deal with the problem that there are less resources available than it has requested. One way to resolve this problem is to reallocate the resources that are demanded by Π from other in-progress projects. This alternative is preferred if other projects can replace the reallocated resource with other resources that are available. RESOURCE REALLOCATOR refers to this sequence of *reallocation* and *replacement (substitution)* as a *transaction*. A chain of transactions is called a *cascade* [120]. For instance, group G_a might be in possession of *r1* but have no skilled personnel, *r2*, to operate it. In contrast, G_b might be in possession of skilled personnel *r2* but lack resource *r1* to operate on. As a result of a transaction, both G_a and G_b could possess *r1* and *r2*.

The semantics of a transaction in Sathi [120] is slightly different from the one we defined above because of the different rules that govern the organization that they assume. According to Sathi, the set of available resources consists of the resources that are under the discretionary control of some organizational unit. G_a can obtain a new

[31] Of course the advantage of doing so from the standpoint of scheduling efficiency is another issue.

resource (e.g. r) only from other units within G (e.g. G_b) that control r. The only method to obtain r in G is to trade a resource that G_b requires and G_a controls with r, a resource that G_b controls and G_a requires.

The main difference between the setting that Sathi defines and the one used here is that here an organizational unit is not allowed to maintain its control over a resource unless that unit keeps the resource at work. This however does not prevent G_a from bidding for r and at the same time offering s which it currently uses.

Claim2: $\exists w=\delta_1...\delta_n \ \delta_1,...,\delta_n \in \Delta \ s.t. \ C(w,\Lambda)=\delta_{transaction}(\Lambda)$.

Proof: The main point in the proof is that a conditional release such as "G_a will give up s only if it can obtain r" can be broken down to "G_a will give up s" and "G_a will obtain r." Consider the sequence "*substitute reallocate substitute*" of operators in S^*. First *substitute* substitutes r for s in G_a, then *reallocate* reallocates r from G_b to G_a, and finally *substitute* substitutes s for r in G_b. Since the initial substitution has released s from G_a to ρ, s will be allocated to G_b from ρ.

Claim3: $\exists w=\delta_1...\delta_n \ \delta_1,...,\delta_n \in \Delta \ s.t. \ C(w,\Lambda)=\delta_{cascade}(\Lambda)$.

Proof: Recall that a cascade is a sequence of transactions and consider all sequences of operators in S^* that is represented by the regular expression e of the form "*substitute (reallocate substitute)+*." Then the same argument that we used to prove claim2 can be used to show a cascade of length n (a sequence of n transactions) is the sequence e such that the sequence *(reallocate substitute)* in it is repeated exactly n times.

The only difficulty that we faced in realizing the function of "larger-grained" operators used in previous approaches through our primitive operators was related to scheduling of setup operations. In the formalism which we presented in chapter 2, setup operations can be viewed as activities that produce a dummy product. While the semantics of setup operations in a job-shop allow that operation to be repeated many times, the definition of a resource in chapter 2 required that all products have infinite capacity (i.e. the operations which produce them are carried at most once). To remedy this problem, our formalism can be extended to allow mutually exclusive products. A set of products are mutually exclusive if at most one of them can be available at any given time, and the production of one implies the exclusion of all other members of the set. Once a product which belongs to this set is produced, it will remain available (with infinite capacity) until another member of the set is produced (to exclude it).

4.3. Operator Selection

In this section, we describe an operator selection strategy which attempts to keep schedule disruption at a minimum while making the most progress toward resolving a given conflict. To reduce disruption during schedule revision, we have studied the amount of disruption that is caused by each primitive search operator, and have developed a set of heuristics to control the application of each operator on this basis. To break ties among operators that are likely to cause roughly the same degree of disruption, we consider the progress that each will make toward solving the target conflict. This implies that operators will first be sorted in the decreasing order of disruption and then in the decreasing order of progress made toward resolving the conflict. The advantage this heuristic strategy for opportunistic scheduling (which is a goal oriented process) is that it helps to maintain the solution structure that earlier opportunistic operators have shaped unless it is found not to be converging to a solution. For instance a disruptive revision might undo the commitments made during previous iterations of opportunistic scheduling by creating new conflicts at the points where previously resolved conflicts used to reside.

Although the minimization of disruption is not a major concern during the initial construction of a schedule, it becomes critical during the reactive refinement of an existing schedule (which characterizes the software project scheduling process). This is because decision making is much more constrained in the presence of a pre-existing set of commitments. Although the importance of non-disruptive (local) operators is noted in OPIS [104], it is considered as secondary to reoptimization concerns during operator selection.

An analysis of the operators in S^* illustrated that *reallocate*, *supply*, *take-away*, and *compromise* are the only operators that always have strictly local effects. For instance, *reallocate* only affects the allocation of resources in the product that the resource is reallocated from and the product that the resource is allocated to. Since *reallocate*, *supply*, *take-away*, and *compromise* can cause no disruption, then they are considered first during the operator selection. Discrimination between these operators is made by comparing the degree of progress that each makes toward resolving the intended conflict. The remaining operators namely *substitute*, *move*, and *switch-plan*, have the potential of causing disruption. Substitution of a mix of required resources for another in the schedule of a product p will be disruptive only if

1. a resource r in that mix will be required earlier by p after the substitution is made. In this case disruption will occur if the new required date of r is earlier than the available date of r.
2. the duration of p increases after the substitution is made. In this case disruption will occur if the new available date of p is later than when other products require p.

The disruptiveness of *move* and *switch-plan* can be studied in a similar way.

Intuitively, we anticipate that *move* will be more disruptive than *substitute* since it requires that the entire schedule of p be shifted. We also anticipate that *switch-plan* be more disruptive than either *move* or *substitute* since not only the schedule but also the process plan of p (consequently the required resources of p) is altered. In our approach, we first determine which of the operators *move*, *substitute*, and *switch-plan* will actually disrupt the schedule if they are applied to resolve the current conflict. This is accomplished by simulating the effect of applying each operator on the schedule without the need to study the type of global changes that it will cause. The simulation is not computationally expensive because it will stop as soon as it learns that other parts of the schedule need to be changed. If none are found to be disruptive, then the operator that makes the most progress toward resolving the intended conflict is chosen. Otherwise, if all three operators are found to be disruptive then we choose them in the following order: *substitute*, *move*, *switch-plan*. This ordering is based on the least-anticipated-disruption-first heuristic which we briefly described above.

The policy of prioritizing the operators on the basis of how local their consequences will be could also be generalized to operator arguments by considering the arguments that cause a smaller disruption in the schedule first. Figure 4-6 shows the implementation of this policy in the action manager.

In the remaining of this section, we describe two alternative methods to approximate the degree of progress that an operator makes toward resolving a conflict. The first method involves explicit measurement of the *cd* of both the current schedule and the schedule that results from applying the operator, and then calculating their difference. Although the *cd* of a schedule can be measured for the entire schedule, it is often preferred to restrict the measurement to a subset of the schedule in order to avoid the computational complexity of repeated measurements. In chapter 3, we provided a detailed description of how the *cd* of a schedule (or part of a schedule) can be measured.

An alternative method to approximate the degree of progress that an operator makes toward resolving a conflict is to use a set of heuristics that has been developed to selectively choose an operator by assessing the parameters

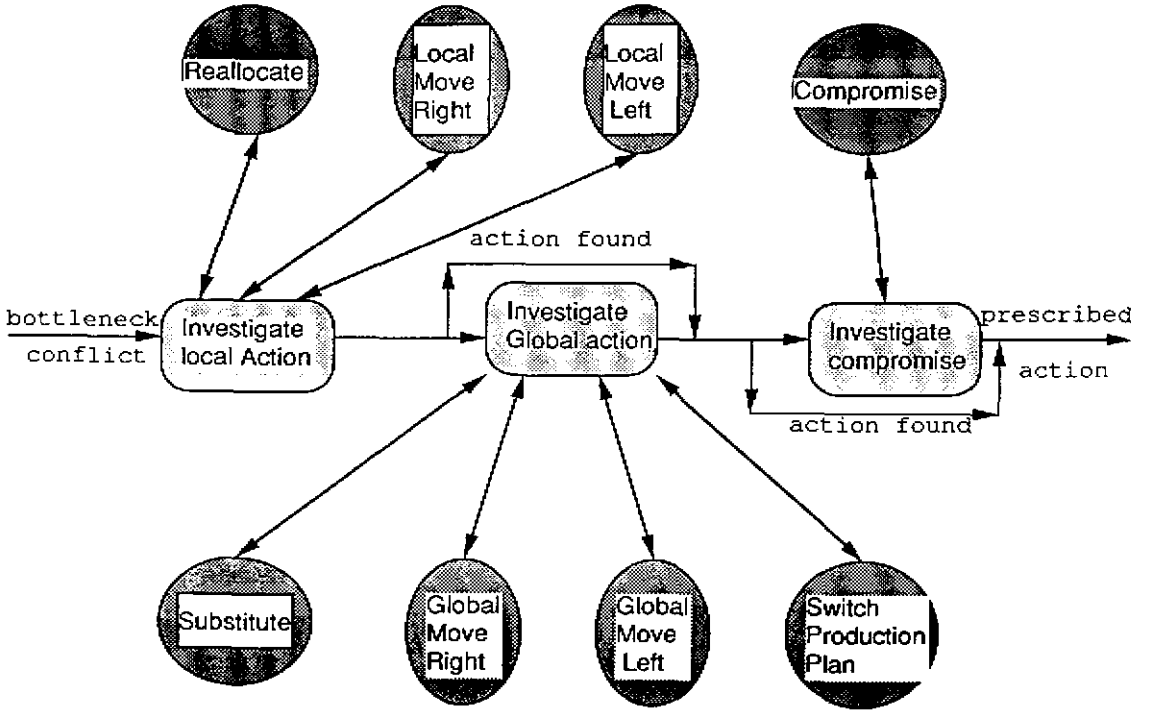


Figure 4-6: A Diagram of the Action Manager

of the conflict or the schedule. These heuristics eliminate the need to measure the *cd* altogether when they are successful in selecting an operator. If they fail, however, we need to fall back to the first alternative. These heuristics have been developed by studying the effectiveness of each operator in resolving different conflict scenarios and also the degree of disruption that the operator might cause in each case, and are outlined as follows for the case that the conflict is over a resource *r*:

1. If the aggregate demand for *r* (combined demand of all products for *r* which is not satisfied) is constant throughout most of the project, then increase the *supply* of *r* uniformly across the entire project schedule.
2. If the available capacity of *r* increases in the neigbourhood of the disputed period, then *move* one of the products that requires *r* over the disputed period.
3. If the request for *r* can be substituted in one of the products that requires *r* over the disputed period and one of the resources that *r* can be substituted with is largely unutilized over the disputed period, then *substitute* is recommended.
4. If *r* can be reallocated from another product which has the choice to substitute or compromise *r*, then a *reallocate* is recommended.
5. If the conflict can be entirely resolved by agreeing to a marginal loss in the feature requirements of the project product, then a *compromise* is recommended.

A heuristic for selecting "switch-plan" is not recommended because the switch from a process plan to another might affect many schedule parameters which in turn significantly reduces the accuracy that can be achieved by providing a heuristic which relies on the value of only a small subset of those parameters. These heuristics represent the scenarios (patterns) that make the application of an operator attractive, and are unrelated to the heuristics that measure the global disruption of each operator.

4.4. Design and Selection of Complex Operators (Primitive Operator Sequences)

In addition to improving the efficiency of search by developing a set of heuristics that approximate the effect of an operator on a schedule without measuring the *cd* of that schedule, we have also recognized the utility of heuristics which prescribe a *sequence* of operators (i.e. more complex operator) once the schedule is known to satisfy a set of properties. The advantage of these heuristics is that they require only a single iteration of the algorithm to decide a *sequence* of operators that should be applied while normally each iteration is able to prescribe only one operator. Although the present prototype of NEGOPRO does not include these heuristics, we expect that they improve the efficiency of search significantly.

A typical situation where a whole *sequence* of operators can be prescribed is when we are certain that many conflicts in the schedule can be resolved by the repeated application of the same operator/argument pair to each conflict. For instance, when there is a manpower shortage across the board and formal review can be eliminated from the quality assurance requirements, the machine scheduler could enforce a *policy* to relax (compromise) the need for formal reviews altogether. This can be implemented by coding a heuristic that is triggered once a pattern of shortage of manpower across the board is detected during the reactive scheduling.

Some of the heuristics that we described in section 4.3 also provide a crude way of prescribing a *sequence* of operators. For instance the heuristic that approximates the applicability of "reallocate" examines if the resource to be reallocated can be later "substituted" or "compromised". In contrast, the direct measurement of *cd* can provide information on only one operator (*reallocate* in this case). It is important to be able to approximate the *cd* due to a sequence of operators since occasionally a sequence of operators should be applied to a schedule before the effect of them on the schedule can become apparent. For instance, a reallocation of r from p to q might have no affect on the consistency of the schedule. However, if p can substitute the resource r with another resource which is unallocated, then one could anticipate that the schedule become more consistent after a "*reallocate substitute*" sequence.

In the same way that a sequence of operators could provide a uniquely efficient way to converge to a solution, there exist sequences of operators that should be prohibited because they might cause redundancy or cause the formation of cycles. We use additional heuristics to prevent the formation of these sequences. For instance, consecutive *moves* can be simulated by a single move in order to prevent redundancy. In this case, we check that if the previous operator was "move", then "move" is prevented from being the next operator because any number of consecutive moves could have been achieved by a single move.

Chapter 5
Interactive Revision of Schedules

In the previous two chapters, we provided a detailed description of a heuristic search reasoning framework to reactively revise schedules. In the present chapter, we deal with the issues that are related to the control and triggering of heuristic search from the standpoint of a human user. The main issue in the control of heuristic search is defining the involvement of human schedulers in conducting heuristic search. The main issue in triggering heuristic search is to determine the types of reactive changes that are allowed and the way that these changes can be introduced. These changes trigger new reactive cycles. We study these two issues in the order which we have stated them above.

5.1. Interactive Human-Computer Collaboration During Search

The architecture of our heuristic search control system allows human schedulers to control the heuristic search interactively. The involvement of human schedulers in conducting heuristic search goes beyond defining the problem and changing its parameters interactively, and is designed to engage the human schedulers in search-related decision making to improve the efficiency of search. Although good heuristics can increase the accuracy of search, experimental studies suggest that most heuristics are problem specific. A solution to this problem is to allow the automatic search algorithm to use a partner who masters a broader knowledge base and can propose new scheduling commitments or screen out the unpromising commitments that are proposed by the automatic search algorithm, namely a scheduling expert.

On the basis of the interviews that we conducted with software project managers, human schedulers can make qualitative assessment of schedules and propose new commitments (to revise a schedule) without working out the details effectively, but they usually have difficulty in checking whether the schedules that have been constructed independently are consistent. Machine schedulers, however, are more suitable for

1. checking whether different parts of a schedule that have been developed independently are consistent and can be merged together.
2. reacting to resolve the discrepancies that arise between the scheduled and actual courses of action during the project execution.

Traditionally, manufacturing scheduling systems have been either completely human centered or completely machine centered. However, the goal of our design is to allow human and the computer to collaborate as partners. In a collaboration, each partner acts according to each one's competence. This is a more generous view of the computer than that of viewing it as a submissive server where the scheduler is the principle actor. In this view of human-computer interaction, the computer should behave as the extension of the human scheduler's skills. The computer should let the human scheduler act freely and take control arbitrarily. The difficulty of designing an interactive interface for this collaboration lies in identifying the transition points where control shifts from human to the computer and back.

The state of the art commercially available SPPS tool capabilities and features vary a great deal among the many tools available but the variation is more in the depth and sophistication of the features such as its storage, display, interoperability, and user friendliness, than in the role of the human and the machine during the problem solving. Most project planning and scheduling support software systems provide an interactive interface but this interactive feature is only within a machine centered framework.

In both cases, whether the computer is a tool or a collaborator, users should not be modeled as finite state machines [7]. States involved in human problem solving are rather unknown and their relations are mostly unpredictable. Human problem solving is basically opportunistic, mixing various problem solving approaches. As a result, it must not be constrained by an inflexible model of interactions. To summarize, human should be given the illusion of driving the system. To this end, he should be allowed to stop the search process at a desired state, change the value of a variable in that state, contemplate to determine how the search should proceed, command the program to apply a set of operators in a sequence to implement his choice, and resume the automatic search or backtrack to any previous search state.

Domain independent planning systems [15, 146] have proposed and implemented schemes to allow a human to control the expansion of goal nodes, and also to post constraints on plan variables thereby guiding the planning along intended paths. Although these systems increase the role of human in getting involved in the problem solving, they still do not allow him to control the search.

In our implementation, collaborative problem solving between human schedulers and our search algorithm is supported by providing a set of functions to control the search process. Human schedulers can interrupt or resume the search through a key stroke. Two functions, *push* and *pop*, are provided to allow human schedulers browse through the search-history tree for this purpose. A search-history tree is a directed n-ary tree $T=(V,E)$ such that each node v in V represents a schedule S and includes a package of data that is essential to restoring the schedule which belongs to *parent(v)* in T[32] from S, and each edge in E is a transition tuple of the form (O,A) where O points to an operator name, and A is a list of arguments that O is being applied with. By revising the schedule that a node represents, through a transition (O,A), we obtain the schedule that belongs to the node that (O,A) is incident on. The root node represents the original schedule before any revision has taken place. At each point, there is exactly one current node, and the schedule which belongs to that node represents the current schedule.

The semantics of E implies that each node in the search-history tree represents a schedule which is an incremental revision of the schedule that belongs to its parent node. Since an incremental revision usually affects a small part of a schedule, to curb the memory consumption of T, the restoration data which is kept in each node includes only the incremental changes that are essential to restore (or backtrack) to the schedule of the parent node. The schedule of the parent node can be restored by starting from the current schedule and orderly undoing the changes that have taken place until all incremental changes have been undone. If the operator which links the parent schedule to the current schedule is invertible, then the content of the current node could be empty since the changes can be undone uniquely by inverting the operator.

Pop instructs the algorithm to backtrack to the schedule that preceded the current schedule (see figure 5-1) and to reinstate it as the current schedule. This implies that required capacity, available capacity, and allocated capacity are all reinstated.

---

[32]Each node (except the root node) has exactly one parent because the data structure which is being examined is a tree.

Push is used to advance to a child node and update the current schedule to the schedule of that node. This update is achieved by revising the current schedule through a transition (which is represented by an edge). Since the current schedule can usually be revised through multiple transitions, a call for *push* should be accompanied by an operator-argument pair to uniquely distinguish a transition (see Figure 5-1). If the target node already exists as a child of the present node in T, then the tree itself need not be updated. In this case, the scheduler revises the current schedule through the transition data which accompanies *push* and advances the current tree node pointer to point to a child node (the one reached through the transition). Otherwise, the algorithm also has to record the data which is affected by the transition, create a new tree node, store the data which is affected by the transition in that node, and attach the node under the current tree node.

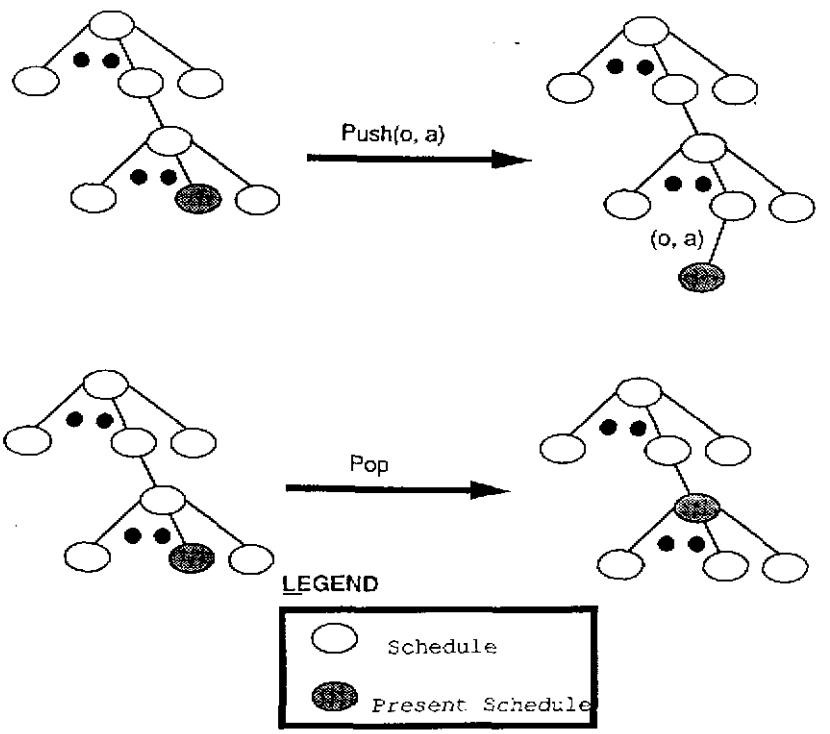


Figure 5-1: Functions to Control the Search Process

The algorithm works in two modes: *automatic* and *manual*. In the manual mode, a human scheduler can control the search by a series of pushes and pops. For instance, he can opportunistically terminate the search along the present path, backtrack, and then manually guide the search along another path. Once the search is guided to a point from which automatic search can take over, automatic search can be resumed by a key stroke. The search can be switched from manual mode to automatic mode and vice versa by a single key stroke.

A human scheduler can

1. set one of several parameters that the algorithm provides for monitoring the automatic search. For instance, a human scheduler can establish the maximum number of operators that the algorithm can apply, or establish a cost or benefit threshold that has to be met before transferring the control of program to manual mode.
2. control the level of detail at which scheduling (henceforth search) has to be carried out by only encoding the resource requirements that are more important, and also by avoiding to break down a production to more primitive productions that in turn need to be scheduled.

To collaborate with the computer, a human scheduler needs to be familiar with the interactive heuristic search framework interface that has been developed. This includes familiarity with the search control functions (i.e. push and pop), and also familiarity with the structure of the search-history tree. Moreover, he should be able to understand the representation that we use for a schedule. It is also useful if the human scheduler is familiar with the primitive search operators and building more complex operators from them, since the scheduling problem that he is trying to solve might require building more effective operators.

Although the interactive heuristic search framework has been implemented, the impact of collaborative human-computer heuristic search on increasing the efficiency of search has not been measured. The evaluation of this framework requires designing sophisticated experiments using human subjects and the development of special evaluation measures.

5.2. Interactive Specification of Reactive Changes

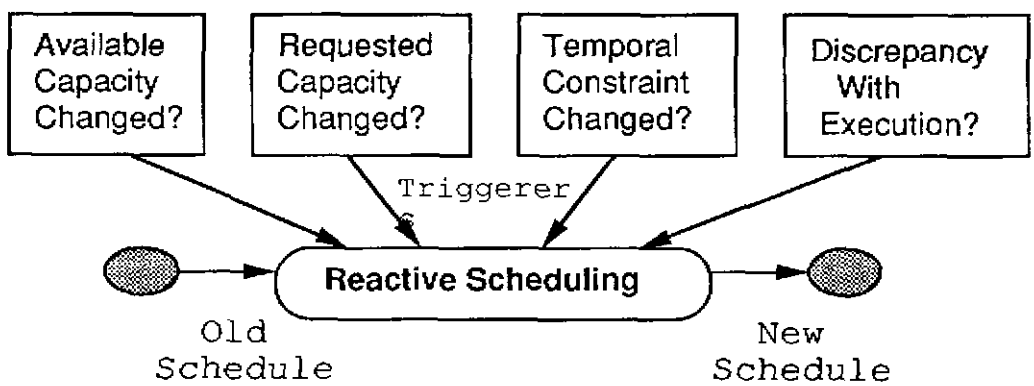


Figure 5-2: Reactive Scheduling

The control of reactive revision of schedules in our implementation is highly interactive. During each reactive schedule revision cycle, a human scheduler is allowed to:

1. alter the available capacity constraints (associated with each resource) and resume automatic search to react to them. The available capacity of a resource could change during the execution of a schedule in the real world because the commitments that were made earlier might fail to materialize. Moreover, human schedulers tune the available capacity of different resources by adjusting them to the schedule needs (after the schedule has been constructed). This allows a project manager to modify the resources that were initially committed to a project to reach a higher resource utilization by allocating more capacity to high contention intervals and deallocating capacity from highly unutilized intervals.
2. alter the capacities of the resources that are required for developing a product. Initial resource requirement estimates for developing a product usually changes during the implementation of the schedule, as those estimates are refined to reflect the actual needs of those products[33].
3. alter the feature requirement constraints of a product. These constraints have to be altered because

[33]Basically, a clock is provided to monitor the execution of a project. If the actual execution data which is entered by a human operator differs from the planned behaviour, then heuristic search is triggered to to repair the schedule. Otherwise (i.e. when the actual execution data is identical to the planned behaviour), the history clock will simply be advanced.

the customer requirements might change during the execution of the project. Moreover, human schedulers could change these requirements to balance the cost and benefit of satisfying them.

The changes that a human scheduler can make to the problem interactively is depicted in figure 5-2. Alteration of the available capacity constraints, required capacity constraints, or feature requirement constraints all trigger a new reactive cycle that resumes heuristic search to revise the current schedule (to accommodate the change).

5.3. An Example of the Role that A Human Can Play During Search

In this section, we provide an example of the role that a human scheduler can play during collaborative heuristic search in increasing the efficiency of reactive scheduling. Figure 5-3 illustrates the aggregate demand curve for a programmer before the arrival of a new request for programmer (in figure 5-3, the new request is attached to a question mark).

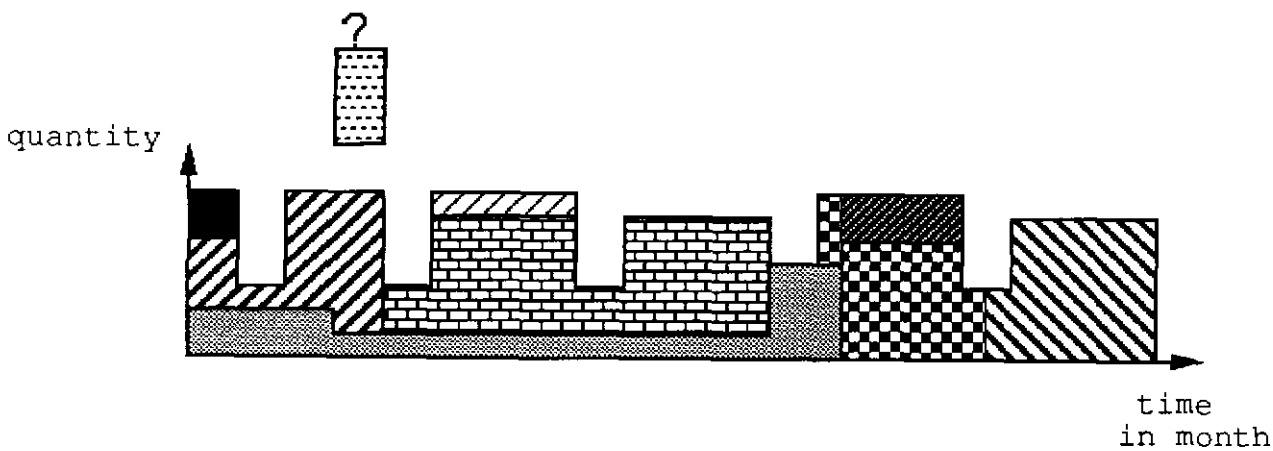


Figure 5-3: Aggregate Demand Curve And A New Request For Programmer

The new request reactivates NEGOPRO to schedule that request. The most logical course of action for NEGOPRO is to schedule the request at first possible available slot (unless there exists a scheduling requirement that prefers to schedule the request at a later time); this is illustrated in figure 5-4.

Now consider a new request for programmers that arrives after the previous request has been scheduled. This request has been depicted in figure 5-5. As figure 5-5 shows, this request has a hard deadline (all parts of the request have to be scheduled before the deadline) as well. If the rescheduling is carried out in the automatic mode, then NEGOPRO first tries to schedule the request at time zero, but since the cost of such allocation is very high (because it results in a highly unbalanced curve), the next candidate time slot (i.e. the next unit of time) is examined. Since all candidates fail, the iteration continues until the request curve reaches the deadline. At that point, the scheduler begins to consider other alternatives such as compromise or reallocation of a previously scheduled request. The examining of each of these alternatives can be very time consuming since it might require further investigation to examine the consequences of committing to that alternative. For instance, in the case of reallocation, NEGOPRO has to decide the request that has to be reallocated, the new time to reschedule that request, and so forth.

Now, the same rescheduling problem can be solved through simple visual analysis of a human scheduler. By

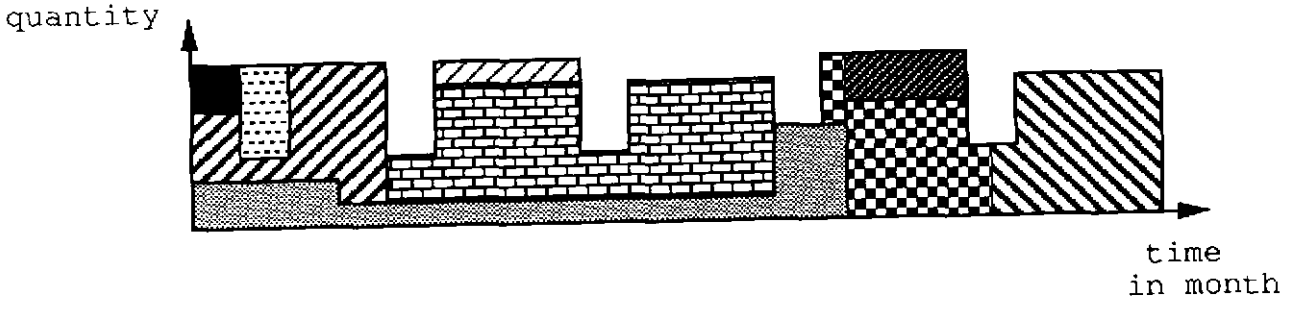


Figure 5-4: Aggregate Demand Curve For Programmer After the Request Has Been Satisfied

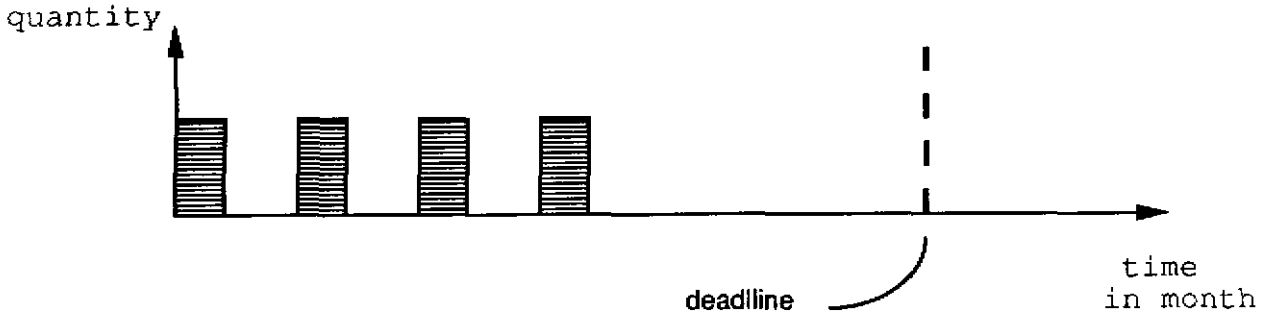


Figure 5-5: Another Request for Programmer

looking at the aggregate demand curve for a programmer, the attention of a human scheduler will be immediately focused on the four holes that marks a lower demand for a programmer. At the same time, the human scheduler would notice that the new request curve can be projected over the empty holes, and that the outcome will be an exact fit, but the deadline of the request would prevent him from instructing NEGOPRO to schedule the request to start where the first (leftmost) empty hole is situated. Next, the human scheduler would spot the request that was last scheduled and the hole that it is occupying. As a result of a simple inference, the human scheduler would figure out that by reallocating the previous request to the last (rightmost hole), a sequence of four holes is generated that constitutes an exact fit for the new request. Moreover, this fit does not violate the deadline of the request either. The resulting solution is depicted in figure 5-6.

Once a human scheduler finds a solution, he would interrupt the machine scheduling, instruct NEGOPRO to make the necessary commitments, and resume the rescheduling activity.

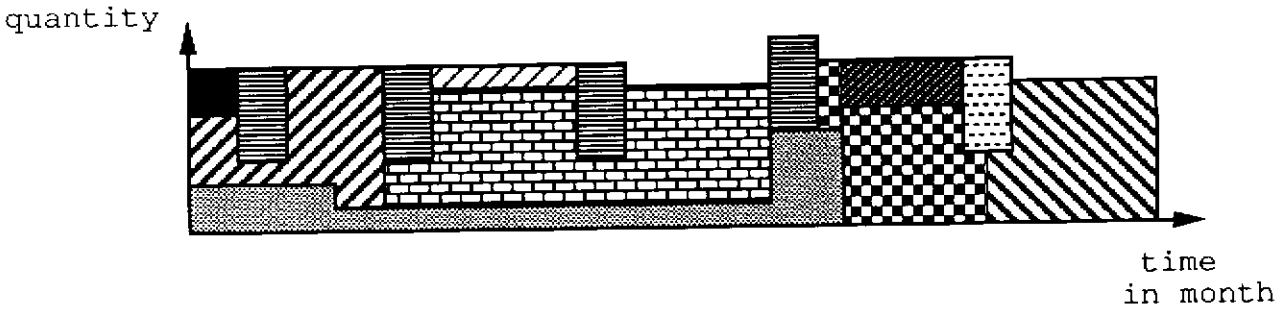


Figure 5-6: Aggregate Demand Curve After Scheduling Both Requests

Chapter 6
Verification of the Basic Heuristic Search Model

In the present chapter, we describe the design of a set of experiments to verify the basic heuristic search model and analyze the results of running those experiments on NEGOPRO. NEGOPRO is the name of a program that has implemented the basic heuristic search model which was described in chapters 1 through 5.

Although the collaborative human-computer problem solving architecture that was presented in chapter 5 for increasing the efficiency of search has been partly implemented in NEGOPRO (we have mainly implemented the *push* and *pop* functions), the experiments are not suitable for evaluating this architecture. Empirical verification of the collaborative problem solving architecture requires more sophisticated experiments that include human subjects and also new criterias to evaluate the impact of collaboration on the time and quality of schedule revision. Instead, in this thesis, we have confined ourselves to providing an example in which the use of typical human intuition to complement the search reduces the search space significantly (this example was provided earlier at the end of chapter 5).

In the first section, the verification measures that have been used are defined, In the second section, a simple experiment to develop a schedule using NEGOPRO is worked through. Design of experiments is discussed in section 3. In the fourth section, first we describe the experiments and then provide an analysis of those results.

6.1. Verification Measures

We rely on two measures for verifying the basic heuristic search model:

1. *computational cost of rescheduling a project.* In most rescheduling problems, there is always some kind of restriction on the "time" that is allowed for rescheduling. Therefore, computational cost of a scheduling algorithm is almost always a concern in evaluating the usefulness of that algorithm.
2. *improvement in the quality of the revised schedule.* We rely on the "quality" of revision as a criteria for verification since it is important to be able to distinguish between two algorithms that spend the same amount of time on rescheduling a schedule, but one generates a far better revised schedule than the other.

Computational cost and quality of search can be generally viewed as opposite measures in heuristic search that cancel out each other. This is because in order to construct a better revised schedule, an algorithm might have to search a larger space and therefore it is likely to spend more time. Conversely, in order to lower the time that an algorithm would use to revise a schedule, the quality of the revised scheduled might have to be compromised. Due to the opposing nature of these measures, in our analysis, we

calculate both the "computational cost" and "quality" of a schedule revision, and tradeoff the value of one against the value of the other to determine the overall performance of the algorithm.

Although it is trivial to measure the "time" that takes to reschedule a project in NEGOPRO (by inserting a time

parameter in the body of the program code), "execution time" of the algorithm constitutes only one component of the computational cost in NEGOPRO. The other component of this cost is the number of operators that the algorithm applies before reaching the desired solution (i.e. completing the schedule revision). The "time" that takes to reschedule a project (through heuristic search) alone is not a good indicator of computational cost of the search since it does not take into account size of the scheduling problem. This implies that slowness in rescheduling a large problem would be interpreted as a weakness of the algorithm, despite the general fact that the larger the size of a problem, the longer it takes to reschedule that problem. On the other hand, the number of operators that are applied during search alone is not a good indicator of computational cost of the algorithm either, since one algorithm could spend a long time to look ahead before applying a new operator, whereas another algorithm could spend much less time on looking ahead but instead apply more operators. To achieve a more accurate measurement of the computational cost of NEGOPRO, we measured and based our analysis on the value of both "execution time" and "operator count" parameters.

It is important to base the verification on the "improvement" in the quality of a schedule (rather than the "quality" of a schedule itself), since by starting from a bad schedule we reduce the chance of constructing a revised schedule that is as qualified in absolute terms as another revised schedule that is created from a near optimal seed. By calculating the improvement in the quality of a schedule, we effectively account for not only the quality of the revised schedule, but also the quality of initial schedule.

The problem in using the improvement in the quality of a schedule as a criteria to verify our model is that it is difficult to quantify the degree of this improvement. Although a scheduling problem might have many objective functions (e.g. maximize the resource utilization, JIT, minimize the tardiness), there is no single function that calculates the overall satisfaction of all of them. The construction of this function is specially complicated by the fact that objective functions are interacting and their interaction varies from one problem to another. Other approaches (e.g. RESOURCE REALLOCATOR [120]) have used such measures as "the number of conflicts solved minus the number of conflicts generated" to estimate the quality of a schedule, but these measures do not account for different objectives functions and the interaction that exists between them. One of the main contributions of the present work has been to develop a scheme to measure the overall satisfaction of all objective functions. For instance, a user can easily raise the importance of meeting the deadline by assigning a higher penalty to violating that objective, thereby insuring that the tardiness will be minimized. We measure the "improvement" in the quality of a revised schedule by calculating the cost and benefit of both the initial and revised schedules and trade off their values. Since the preference of many objective functions including JIT, minimizing the tardiness, and maximizing the resource utilization can be expressed in terms of the cost and benefit of a schedule (see chapter 3 for more details), our analysis provides an overall measure of satisfaction of a variety of objective functions.

6.2. Solving A SPPS Problem with NEGOPRO

The purpose of this section is to discuss the behaviour of NEGOPRO program and show how a software manufacturing problem is rescheduled using NEGOPRO. During each incremental step, first conflict analyzer is invoked to find the conflict that needs to be resolved next. Once conflict analyzer has found a bottleneck conflict, action manager prescribes the most suitable operator that can be applied to revise the schedule. Action manager includes a set of operators and a set of heuristics to reason about the circumstances that each operator is applicable. Progress examiner measures the degree of progress in resolving the intended conflict by examining the revised schedule that has resulted from applying the candidate operator.

In contrast to linear programming techniques, NEGOPRO relies on heuristic search to cut down the search space intelligently at the risk of not finding the best solution. More specifically, NEGOPRO always tries to find a good schedule, not necessarily an optimal one. Moreover, in contrast to many schedulers that rely on heuristic search, NEGOPRO does not extend the deadline of a schedule by default when all other choices fail, thereby increasing the tardiness. This is because NEGOPRO treats time like any other resource that beyond its available capacity incurs a cost. Once the deadline of deliverables for a project is set, the cost of completing the project on or before the deadline is automatically set to zero (because software products require no inventory cost). The user of NEGOPRO, however, is allowed to set the penalty of a late delivery. A different penalty can be attached to each day/week/month of late delivery. The above scheme enables NEGOPRO to tradeoff time and other resources, and to look at the size of unavailable capacity of each important resource when the scheduling is completed. Thus in NEGOPRO the violation of a capacity constraint is not exclusive to "time".

Since NEGOPRO continues the search only along the paths that make constant progress, it was important to figure out how many scheduling revisions has to be tolerated along a path before that path is abandoned. The determination of this parameter is important because frequently a schedule worsens before it significantly gets better. If the tolerance is chosen to be too high, then the size of search space could increase significantly. If the tolerance is chosen to be too low, then some converging solution paths could be overlooked and missed. For the experiments which we ran, we found that a limit of 4 revisions before abandoning a path is sufficient to enable NEGOPRO reach the same results that by increasing the limit of revisions to 5, 6, or 7 can be achieved in most cases, and therefore provides the best tradeoff between the limit on the computational cost of the algorithm and quality of the revised schedules. We were unable to compare the decision to limit the number of revisions to 4 with the decision to increase this limit to 8 or more since the search space resulting from this increase became too large for NEGOPRO to explore.

To demonstrate the ability of NEGOPRO in successfully revising a schedule, in the remainder of this section, we explain that how NEGOPRO revises a seed schedule that incurs a high cost for implementation. First, the project and the seed schedule are described. Then, the execution of NEGOPRO on the seed schedule to revise that schedule is discussed. Finally, the schedule that has emerged from executing the NEGOPRO on the seed schedule is evaluated and analyzed.

Suppose that ChipTest is a software development organization involved in the development of software for testing of integrated circuit boards. Moreover, suppose that ChipTest has received two orders: one to develop SoftTest, a software that tests manufactured Intel chips, and another to develop SIM, a simulator to test the design of a circuit board. ChipTest also has to complete an inhouse project, MAIN, which involves enhancing and tuning a test generating software. There are two kinds of resources: programmer and time.

The process plan of SoftTest is described in terms of two modules: X and Y, and is illustrated in figure 6-1. While the development of X and Y can overlap with the development of SoftTest, SoftTest needs to use module X at least five months and module Y at least three months before it can be completed. Both modules X and Y are carried out in two phases. Activities of the first phase of either module are parallel in nature and hence by assigning more manpower they can be completed in a shorter time. Activities of the second phase are sequential in nature and hence the addition of more manpower would not affect their duration.

The process plans of SIM and MAIN are described directly in terms of programmer and time (figure 6-2). Since the design of circuit board is done in three phases, the development of SIM is also carried in three phases such that each development phase succeeds a design phase. Since the customer of MAIN is the firm itself, the

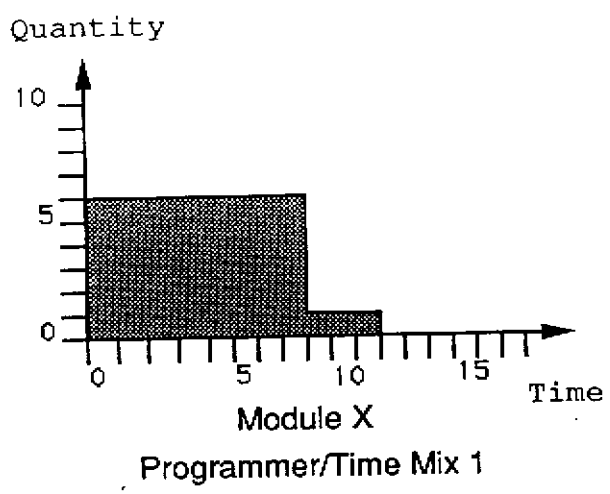
management has agreed to compromise its quality assurance requirements by relaxing the need to conduct a formal review. The management decides not to associate a requirement loss with relaxing the quality assurance requirements of MAIN because the sacrifice will be insignificant. Should the quality assurance requirements of MAIN relax, the need of MAIN for programmers will drop to half the level before the compromise.

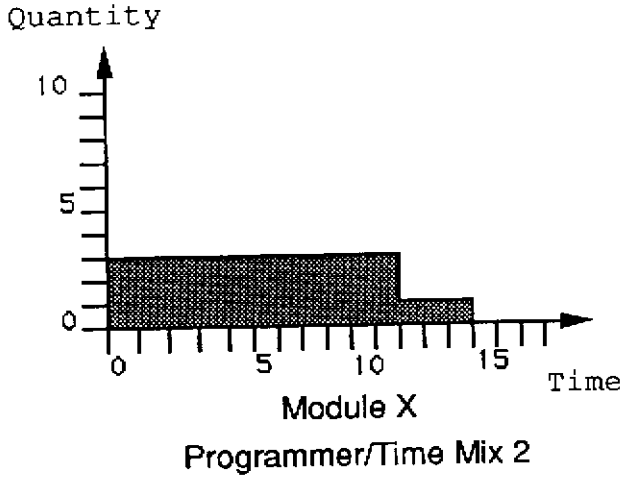
The level of manpower available to the organization to assign to various projects is assumed to vary over time. The curve of this availability is shown in Figure 6-3. The marginal cost of allocating new programmers in addition to the existing level is given in figure 6-4. The penalty of delivering each of the projects later than their specified deadline is also given in the form of cost in the table of figure 6-4; An asterik in a table entry denotes that the cost is unbearable if the project is not completed within the period specified in months specified by row label. Our goal in designing a seed schedule was to minimize the duration of each project at the cost of other resources. To achieve this goal, we built two sets of seed schedules where the first set was characterized by scheduling the development of module *X* (of SoftTest) first, and the second set was characterized by first scheduling the development of module *Y* (of SoftTest). We ruled out the scheduling of either *SIM* or *MAIN* first because we believed that since the development of *X* and *Y* always has to be succeeded by the development of *SoftTest* scheduling of *X* or *Y* is more urgent. Furthermore, we ruled out simultaneous scheduling of *X* and *Y* because it did not utilize the availability of programmers which is depicted in figure 6-3. Among the seed alternatives that we generated, the seed that its resource requirements is depicted in figure 6-5 was chosen based on human intuition with the goal of minimizing the cost.

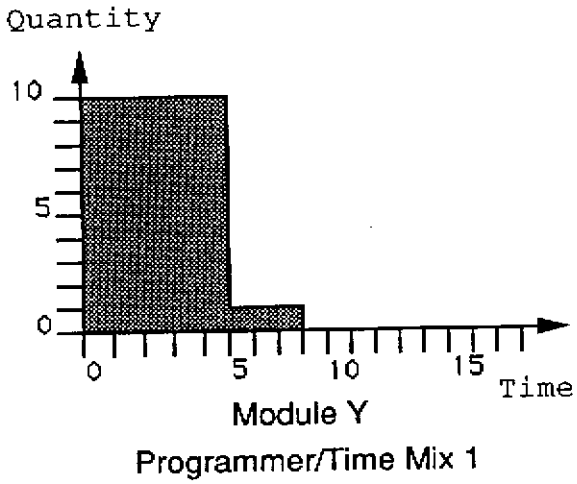
The seed schedule that is used in this example requires eighteen months to complete and schedules X to be the product with which the development begins. If the start of X is taken as the reference, then the seed schedule requires that Y and SIM start a month later, MAIN start seven months later, and SoftTest start 11 month later. Furthermore, the seed schedule uses the first personnel/manpower combination for module X and the second personnel/manpower in the case of module Y. The aggregate manpower requirements of this schedule is depicted in Figure 6-5. The requirements are colored to denote the product or project that has requested them. As Figure 6-5 illustrates, aggregate manpower requirements of the seed schedule does not form a level curve. A level curve is ideal from the standpoint of resource allocation since project staff cannot be hired on a month-to-month basis, and hiring consultants that work on a month-to-month basis turns out very expensive.

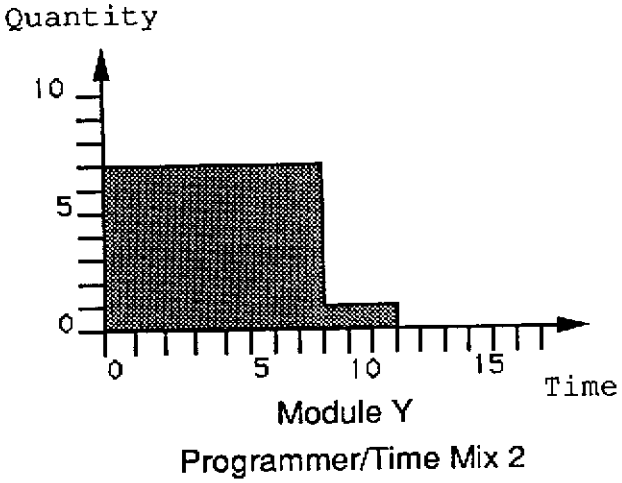
Figure 6-5 depicts the manpower that is no longer available (it is allocated to the seed schedule). This Figure also shows the manpower that would remain unutilized if the seed schedule is implemented. The unutilized area constitutes nearly half of the available capacity during the second half of the development period. In contrast, there is a significant need for additional manpower in the first 7 months of development that can not be fulfilled. The cost of acquiring new manpower to cover these needs during the the first 7 months of the development exceeds $300,000[34].

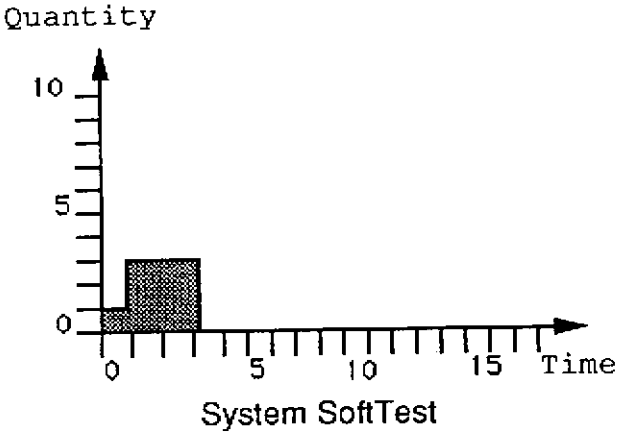
[34]this figure can be calculated on the basis of the manpower required and the manpower cost curve.











Demand for Programmer

Figure 6-1: Primitive Resource Requirements of SoftTest

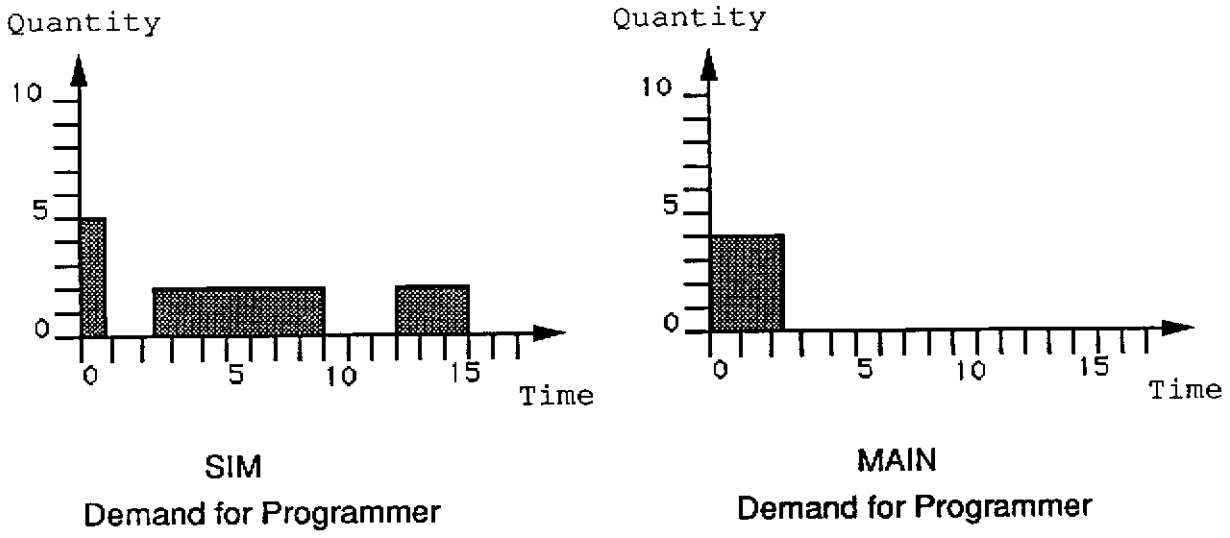


Figure 6-2: Primitive Resource Requirements of (a) SIM (b) MAIN

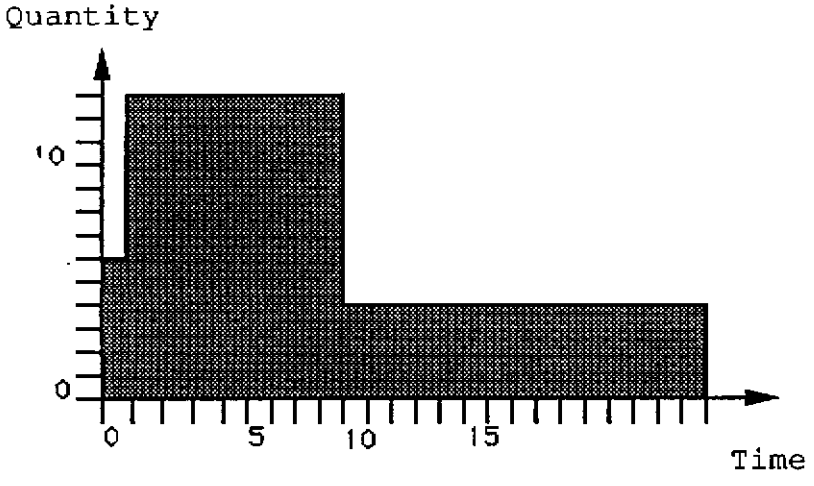
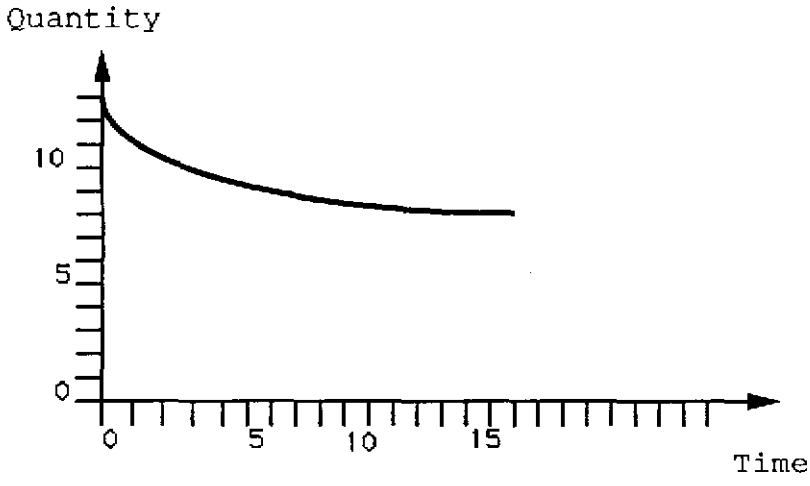


Figure 6-3: Programmer Availability Curve (time in months)



Marginal Cost Curve for Programmer

SoftTest	SIM	MAIN
0	0	0
0	10	0
100	*	*
*	*	*

Penalty Cost Curve for Late Delivery

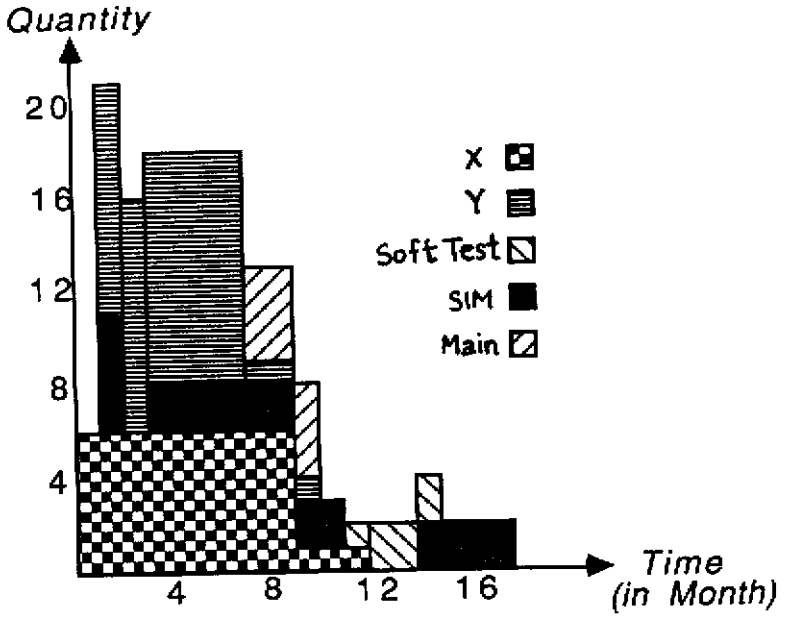
Figure 6-4: Marginal cost curve for programmer and tardiness penalties for the project

6.2.1. NEGOPRO in Operation

NEGOPRO starts with passing the control to the conflict analyzer to discover whether the seed schedule needs to be revised at all. The transcript of the actions that NEGOPRO has prescribed along with the progress that has been made after exercising each action is shown in figure 6-6. The progress is measured through the cost of additional resources that need to be allocated to the activities within the schedule. After examining the unmet resource requests, the conflict analyzer finds out that the seed schedule contains two conflicts: one due to the unmet programmer requests of X and the other due to the unmet requests of Y for programmer. Furthermore, the cost/benefit scheme computes that the cost of resolving the conflict due to Y is $285,000, far more than the $12,000 that is needed to resolve the conflict due to X. This suggests that the conflict due to Y is currently the bottleneck.

Once the action manager is invoked, it first examines whether any of the local actions including *substitute*, *reallocate*, *local-move-right*, and *local-move-left* can resolve the bottleneck conflict. At the end of the examination, *substitute* emerges as the best candidate since a substitution in Y would relax the conflict to a large extent without generating any new conflict. This is because the sizable right slack of Y can be traded to lower the per unit of time requirement of Y for programmer. On the basis of above reasoning, action manager chooses the operator *substitute* and passes the control to the revision manager which in turn applies the action.

A substitution from the first manpower/time combination to the second in Y nearly relaxes the conflict due to Y but at the same time it generates a new conflict. This conflict involves acquiring new manpower for the extended tail of Y. To determine the choice of action to be taken next, the set of local actions have to be examined again. Since a second substitution in Y undoes the previous substitution, it will be rejected. A local move left or move right do not produce any progress either. NEGOPRO chooses *reallocate* to reallocate programmers from X to Y because it figures out that programmers can be traded for time in X thus relaxing the conflict that would shift to



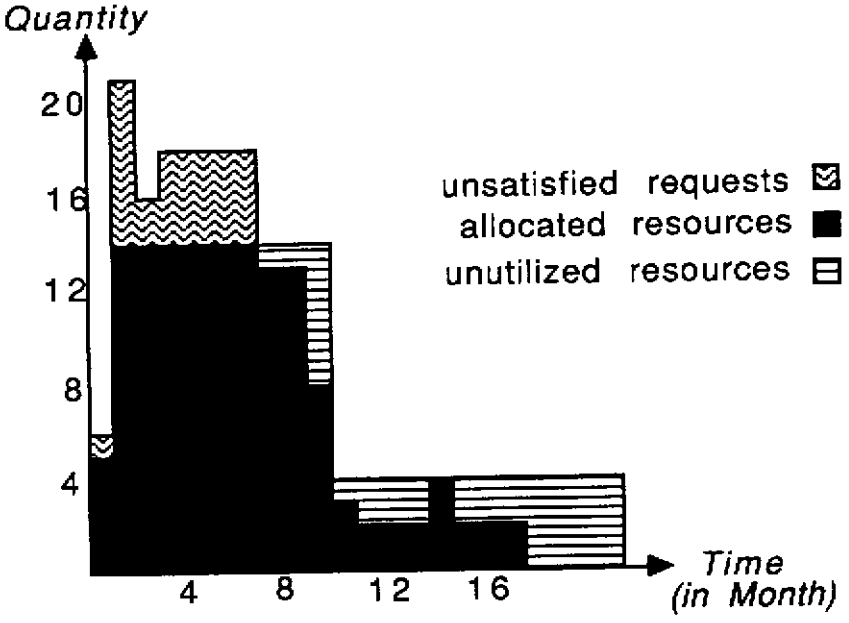
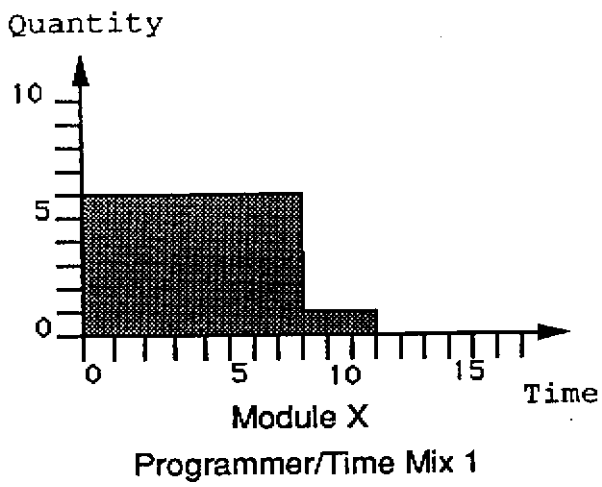
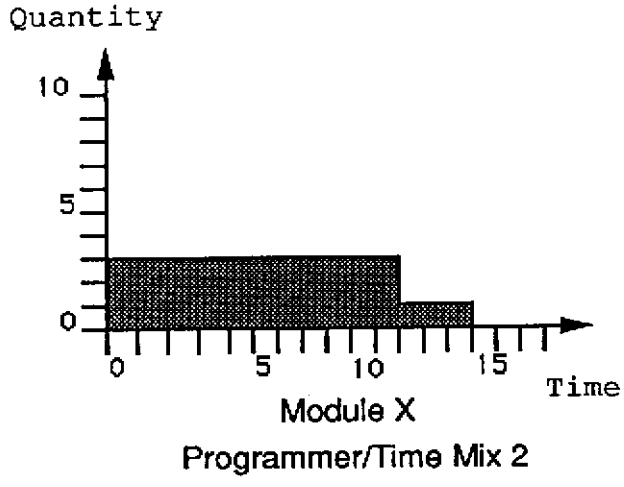
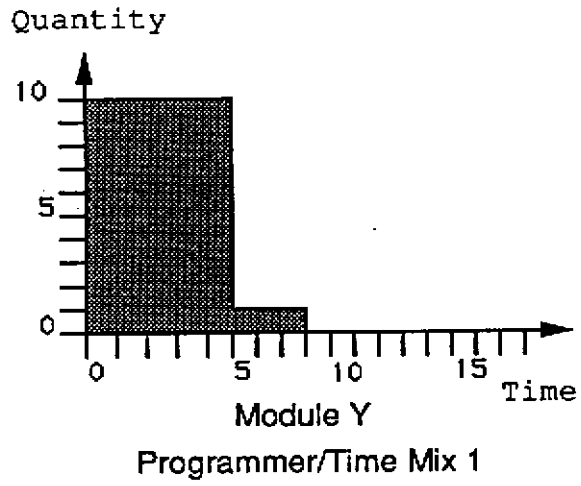
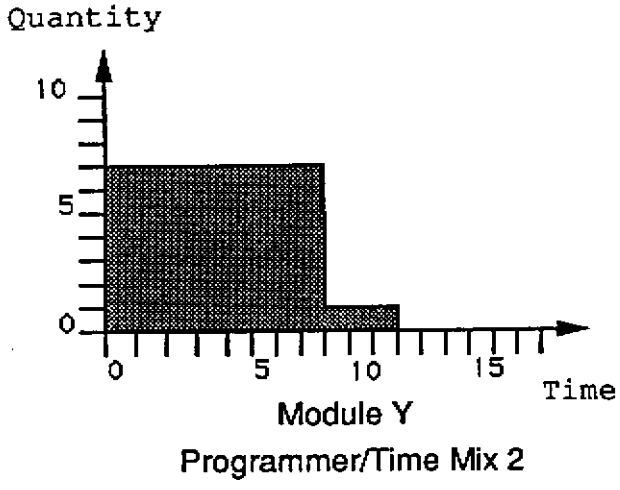


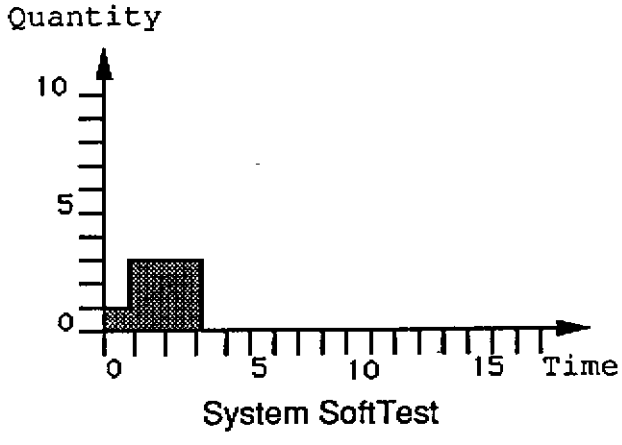
Figure 6-5: Seed Schedule (a) Resource Requirements (b) Statistical Properties











Demand for Programmer

Figure 6-1: Primitive Resource Requirements of SoftTest

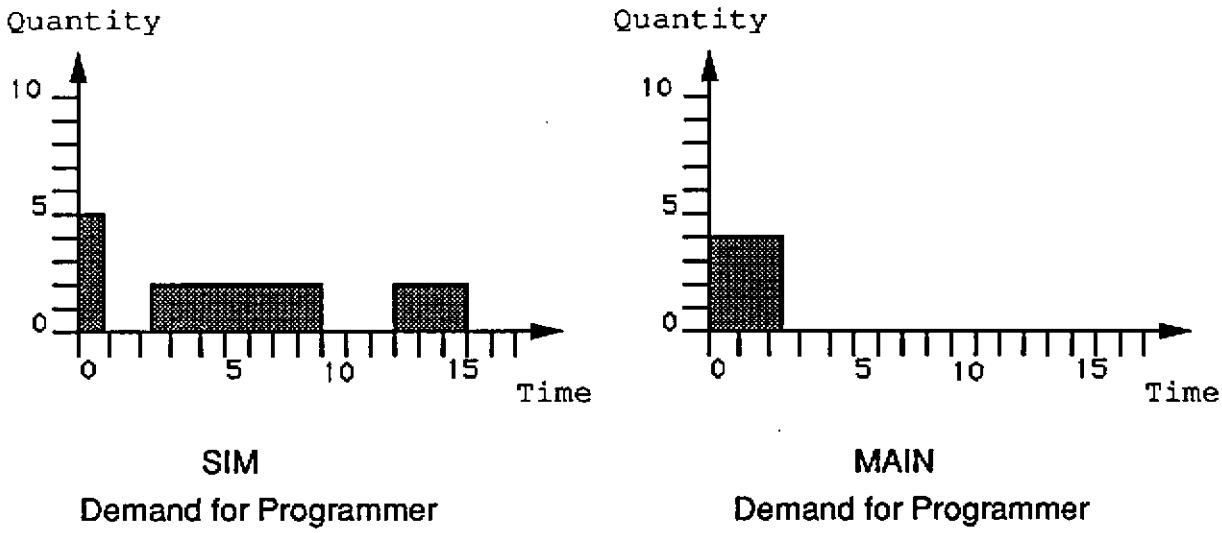


Figure 6-2: Primitive Resource Requirements of (a) SIM (b) MAIN

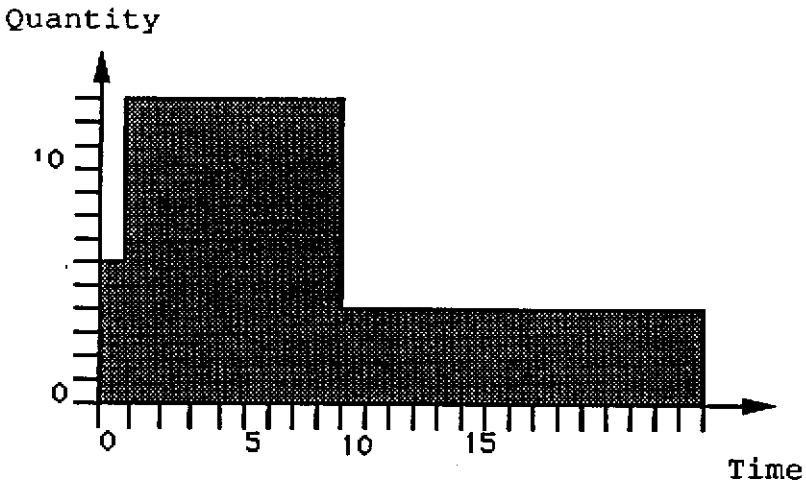
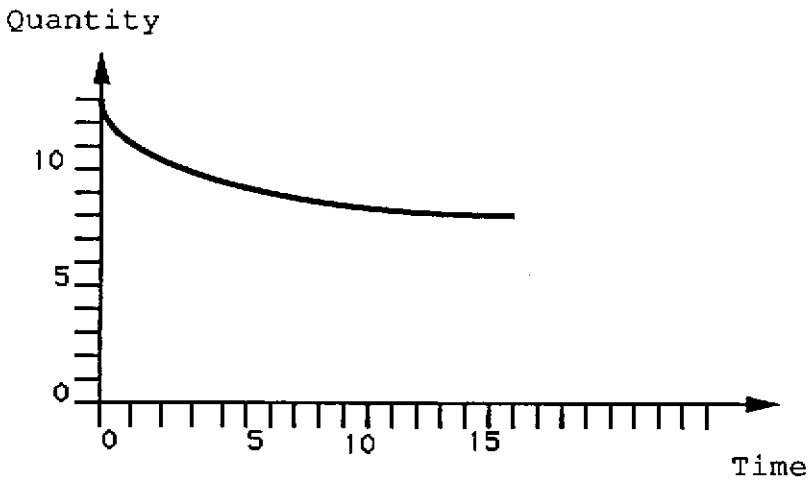


Figure 6-3: Programmer Availability Curve (time in months)



Marginal Cost Curve for Programmer

SoftTest	SIM	MAIN
0	0	0
0	10	0
100	*	*
*	*	*

Penalty Cost Curve for Late Delivery

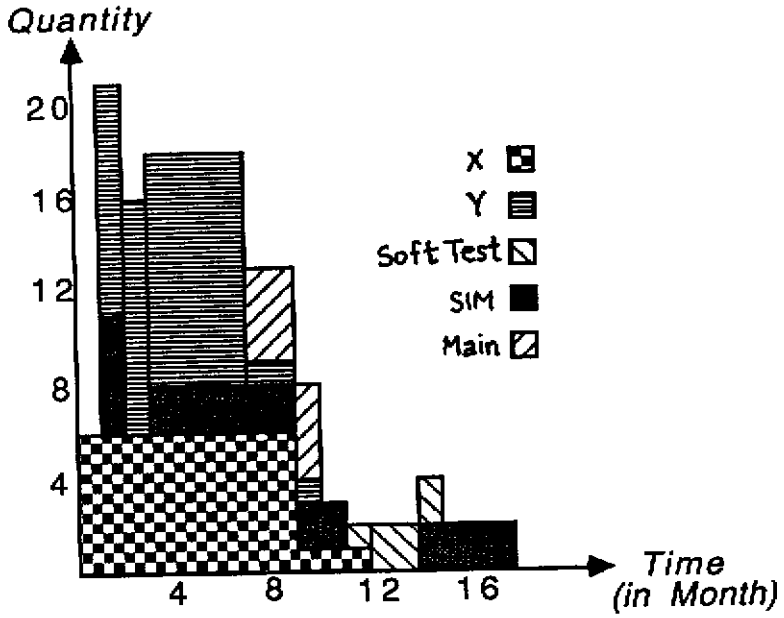
Figure 6-4: Marginal cost curve for programmer and tardiness penalties for the project

6.2.1. NEGOPRO in Operation

NEGOPRO starts with passing the control to the conflict analyzer to discover whether the seed schedule needs to be revised at all. The transcript of the actions that NEGOPRO has prescribed along with the progress that has been made after exercising each action is shown in figure 6-6. The progress is measured through the cost of additional resources that need to be allocated to the activities within the schedule. After examining the unmet resource requests, the conflict analyzer finds out that the seed schedule contains two conflicts: one due to the unmet programmer requests of X and the other due to the unmet requests of Y for programmer. Furthermore, the cost/benefit scheme computes that the cost of resolving the conflict due to Y is $285,000, far more than the $12,000 that is needed to resolve the conflict due to X. This suggests that the conflict due to Y is currently the bottleneck.

Once the action manager is invoked, it first examines whether any of the local actions including *substitute*, *reallocate*, *local-move-right*, and *local-move-left* can resolve the bottleneck conflict. At the end of the examination, *substitute* emerges as the best candidate since a substitution in Y would relax the conflict to a large extent without generating any new conflict. This is because the sizable right slack of Y can be traded to lower the per unit of time requirement of Y for programmer. On the basis of above reasoning, action manager chooses the operator *substitute* and passes the control to the revision manager which in turn applies the action.

A substitution from the first manpower/time combination to the second in Y nearly relaxes the conflict due to Y but at the same time it generates a new conflict. This conflict involves acquiring new manpower for the extended tail of Y. To determine the choice of action to be taken next, the set of local actions have to be examined again. Since a second substitution in Y undoes the previous substitution, it will be rejected. A local move left or move right do not produce any progress either. NEGOPRO chooses *reallocate* to reallocate programmers from X to Y because it figures out that programmers can be traded for time in X thus relaxing the conflict that would shift to



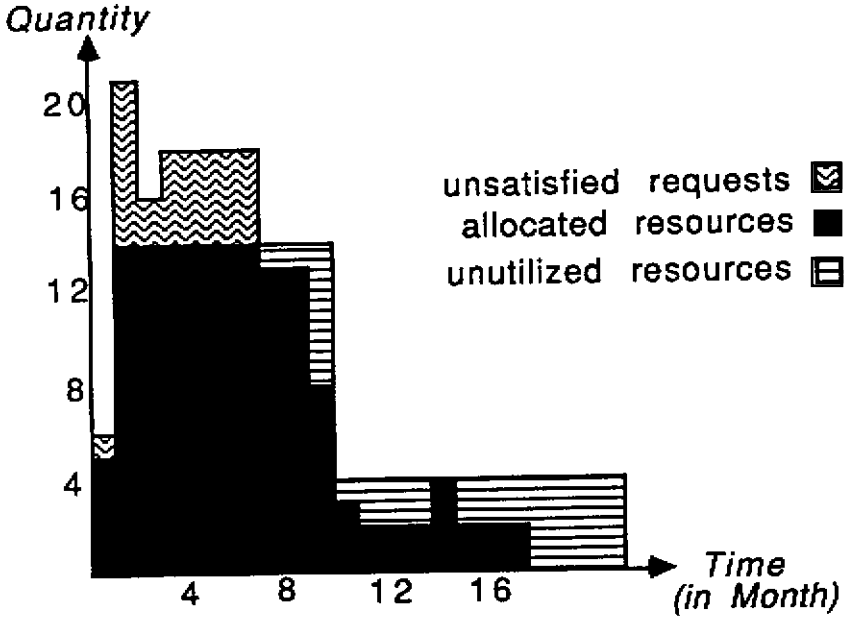


Figure 6-5: Seed Schedule (a) Resource Requirements (b) Statistical Properties

cost	bottleneck conflict	action	argument
534	(Y prog)	SUBSTITUTE	0
236	(Y prog)	REALLOCATE	*from X capacity:* ((0 1 5)(3 6 2)(7 8 6))
236	(X prog)	SUBSTITUTE	0
236	(X prog)	REALLOCATE	from MAIN capacity: ((7 8 3))
176	(MAIN prog)	LOCAL-MOVE-R	MAIN leap size: 1
116	(MAIN prog)	COMPROMISE	MAIN 2 programmers
98	(X prog)	GLOBAL-MOVE-R	X leap size: 2
84	(X prog)	GLOBAL-MOVE-R	X leap size: 1
236	(MAIN prog)	LOCAL-MOVE-R	MAIN leap size: 3

Figure 6-6: Sequence of Actions Prescribed by NEGOPRO

X. The application of *reallocate* and *substitute* in sequence moves the conflict from the tail end of Y to the tail end of X.

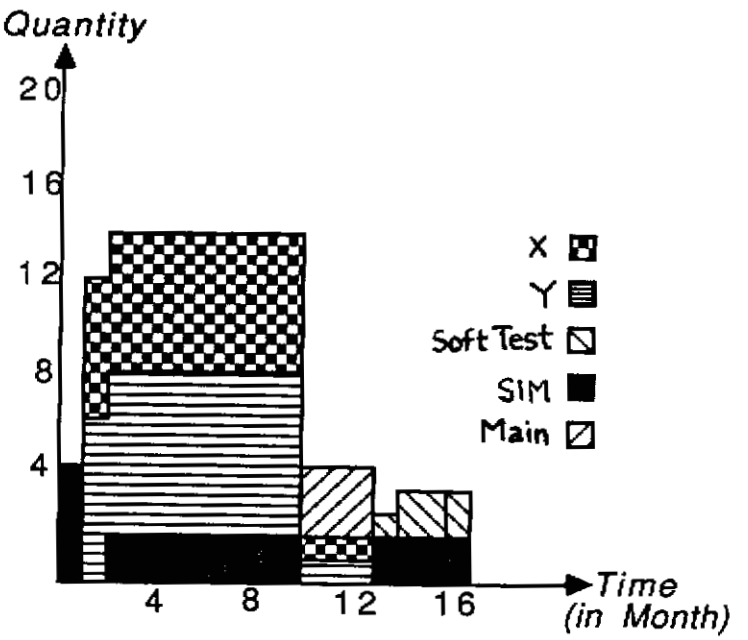
At this stage of the search, all substitution choices have been exhausted and no choice of reallocation appears to be promising. In contrast, a local right move of the MAIN could provide the much needed programmers to X because MAIN and the tail of X share the same time intervals in the revised schedule. Once the schedule of MAIN is moved to the right where there is not enough programmers to be allocated to MAIN, the bottleneck conflict is carried over to MAIN. Figure 6-6 shows that *compromise* is the next action that is prescribed by NEGOPRO[35]. This is because neither local nor global actions produce any promising results. Once a compromise relaxes the conflict due to MAIN, the conflict due to X becomes the bottleneck conflict. The next action by NEGOPRO, a global move right, generates the best result that NEGOPRO produces within the sequence of the first twenty five actions prescribed by NEGOPRO.

6.2.2. Analysis

The total cost of hiring more programmers to satisfy all demands for programmers in the revised schedule is $84,000 while it amounted to $534,000 in the seed schedule. This shows an improvement of about 250 percent. The results that are tabulated in figure 6-8 suggest that the schedule at this stage is very close to optimal because:

1. the available manpower is nearly fully utilized.
2. the unsatisfied requests for manpower constitute a thin band (i.e. one with a short height). This is ideal, since according to the marginal cost function, the total cost for satisfying an aggregate demand is minimized when the demand curve is flat and continuous.
3. the only compromise that has been made does not sacrifice the requirements in any significant way.
4. there is no tardiness cost as all three projects can be completed on time.

[35]The specification of *compromise* in the problem suggests a compromise in the quality assurance of MAIN will save 2 programmers



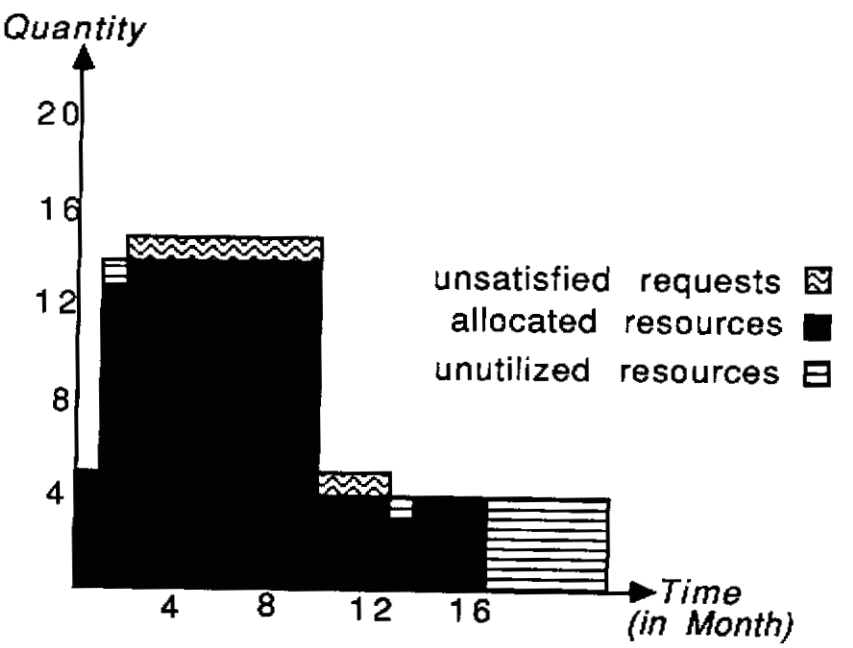


Figure 6-7: Revised Schedule: (a) Resource Requirements (b) Statistical Properties

	Resource Utilization	Resource Leveling	Benefit Sacrificed	Tardiness Penalty	Cost
Seed Schedule	Low	Low	None	$ 0	$534,000
Revised Schedule	High	High	None	$ 0	$84,000

Figure 6-8: Comparison of the Seed Schedule and the Revised Schedule

6.3. Experiment Design

We designed a total of 46 experiments that varied in the number of operations (or alternatively the number of products), resources required, resource mixes allowed, plans available, compromises allowed, and objective functions specified. All experiments sketched a multi-project organization that is engaged in the concurrent execution of three projects. A summary of the characteristics of all 46 experiments is shown in table 6-9. Each row of the table corresponds to two experiments (i.e. although the two experiments are different, they share the same statistics in case of the number of products, compromises, and so forth). The "number of exploded activities" specifies the total number of activities (operations) that have to be scheduled if each activity that requires different capacities of a resource over different intervals is exploded to a sequence of activities each requiring a constant capacity of a resource during the entire duration of that activity. The "duration" of each experiment, which specifies the maximum duration of the project that it represents without any penalty of late delivery, is approximately 16 months (or 480 days).

In accordance with the requirements of our verification strategy, we divided the experiments (all 46 of them) to subgroups along two different dimensions: problem size and quality of the seed. The experiments could be divided along problem size to two general groups on the basis of the number of their products. The number of products in each experiment in the first and second groups are 5 and 12 respectively. This implies that the second group of experiments had to deal with a considerably larger search space. The experiments within each group shared the same production dependency graph (see chapter 2), however, they varied in the number of resources being scheduled, substitution opportunities among those resources, number of process plans, and the number of possible compromises.

On the basis of the quality of seed, the experiments were divided to three groups: problems with bad seeds, problems with average quality seeds, and problems with good seeds. The quality of a seed has been determined by the cost of allocating more capacity to meet all unsatisfied resource requirements of that seed: good seeds represent the seeds that are least costly whereas bad seeds represent the most costly ones. The degree that each seed meets the feature requirements is not a criteria for discrimination since each seed meets all feature requirements.

Exp No.	No. of Products	No. of Compromises	No. of Process Plans	No. of Resource Mixes	No. of Primitive Resources	No. of exploded Activities
1	5	0	1	4	2	16
2	5	0	1	4	2	16
3	5	0	1	4	2	16
4	5	0	4	1	3	23
5	5	2	1	4	2	16
6	5	2	1	4	2	16
7	5	2	1	4	3	23
8	5	2	4	1	2	16
9	5	4	1	4	2	23
10	5	4	1	4	3	16
11	5	4	4	1	2	16
12	5	6	1	4	2	16
13	5	6	4	1	3	23
14	12	0	1	4	3	27
15	12	0	1	4	3	27
16	12	0	2	1	3	27
17	12	0	4	1	7	35
18	12	2	1	4	3	27
19	12	2	2	1	3	22
20	12	2	4	1	7	22
21	12	4	1	4	3	35
22	12	4	2	1	3	22
23	12	4	4	1	7	35

Figure 6-9: Summary of the Characteristics of the Experiments

In assigning the relative weight of the feature requirements of each experiment, we assumed that all feature requirements are roughly equally important. As a result of this assumption, the choice of "which compromise to make next" at each decision point during running each experiment was not transparent any longer.

By dividing the experiments along the size and quality of the seed, we were able to measure the effect of the choice of seed schedule (which needs to be revised) and size of the scheduling problem on (1) the quality of the revised schedule, and (2) the computational cost of revision.

Although it has been common among previous artificial intelligence based factory scheduling programs [42] to test the system on randomly generated experiments, our experiments were not randomly generated because we had no strong reason to believe that there is no difference between a set of randomly generated experiments and a set of experiments that reflects realistic micro multi-project cases.

6.4. Analysis of Experimental Results

We conducted all 46 experiments that we had designed and in each case we measured the improvement in the quality of the revised schedule in comparison to the quality of the seed schedule, the time that took NEGOPRO to produce the revised schedule, and the number of revision operators applied. We were unable to compare our results with those of other approaches [9, 64, 96] since they do not provide the type of problem solving capabilities that NEGOPRO offers.

Once we conducted all 46 experiments, we analyzed the results by measuring the correlation between the input and output parameters. In each case one or two input parameters were allowed to vary while all others were fixed. Among the output parameters, those that were relevant to our verification goal, were recorded and then the results were tabulated into a number of tables such that each cell of a table averaged all experiments that satisfied its rows and column attributes.

Table 6-10 compares the effect of the two different groups of input on the output. This table shows that by increasing the overall size of the problem, the time that is spent on conflict analysis, operator selection and schedule revision, and schedule evaluation all increase. However, the rate by which the computational cost of the latter two increase is smaller than the rate by which the computational cost of conflict analyzer increases. This we learned is because the number of conflicts that have to be analyzed is directly affected by both the number of products and the number of resources, while the number of potential operators that have to be compared is less strongly affected by these factors. Another important conclusion which can be drawn from the results in table 6-10 is that schedule evaluation does not constitute a bottleneck from the computational standpoint and therefore may be coupled with other scheduling strategies as well.

Group No	AVG # of Operators	Time in Conflict Analyzer	Time in Action Manager	Time in Progress Analyzer	Quality Improvement
Group1	6	1.2	0.7	0.1	490%
Group2	19	3.1	1.1	0.2	260%

Figure 6-10: The effect of the size of the problem on the speed and quality of the revised schedule (time is in minutes).

Table 6-10 also shows that with increasing the size of the problem, the number of operators that need to be applied to achieve the same degree of improvement in the schedule increases, but that this increase appears to be linear. Last but not the least, table 6-10 shows that NEGOPRO has improved the quality of the seed schedule in the first group of experiments more than it has improved the quality of the seed schedule in the second group. However, this result would be offset by the fact that the available capacity for the second group of experiments was chosen to be significantly tighter than the available capacity for the first group of experiments.

Table 6-11 shows that the choice of seed schedule affects the quality of the revised schedule as well as the speed by which that schedule is produced (measured in terms of the number of operators applied), but that the quality

of the revised schedule is affected most. Although our goal in designing seed schedules was to minimize the duration of each project at the cost of other resources, the resulting seed sometimes proved close to the schedule that NEGOPRO generated after revising it and far from it in other cases. Rather than averaging the quality improvement of all NEGOPRO generated schedules we decided to break the seeds to three groups according to their cost where, *bad seeds* represented the seeds that were the costliest to revise, while *good seeds* represented the seeds that were the least costly to revise. design enabled us to study the variance in quality improvements between groups that are different in the quality of their seeds.

Table 6-11 also shows the conclusion that bad seed schedules lead to lower quality revised schedules could be misleading if it is not studied in conjunction with the results shown in the second column of the table. This column illustrates that the quality improvement of the final schedule w.r.t. to the seed schedule (improvement ratio) is not affected by the quality of the seed schedule in a significant way. Although the experiments confirmed that the seed schedule made a difference, they also confirmed that NEGOPRO improved the quality of the seed schedule significantly in all cases. The fact that the rate of convergence is not very sensitive to the seed schedule is encouraging since it implies that the response time of NEGOPRO remains reasonably low over different choices of seed.

Seed Quality	AVG # of Operators	Quality Improvement	Quality of Final Schedule
Bad Seed	9	380%	.2
AVG. Seed	13	430%	1
Good Seed	21	340%	2.6

Figure 6-11: The effect of the seed schedule on the quality of the final schedule

Table 6-12 shows the level of success that NEGOPRO had in revising each schedule. A major obstacle was that we only knew what the optimal schedule was in less than 25 percent of the experiments while in the remaining 75 percent of the experiments we were not sure what the optimal schedule is. In the latter case, we compared the NEGOPRO generated schedule with those generated by hand (with the help of a human expert), and found out that over 70 percent of times, NEGOPRO generated schedules were better that the hand generated ones, while 16% of the time they were worse.

The reason that NEGOPRO does not produce the optimal solution within a finite amount of time is that it always relies on a small set of heuristics that it has been programmed to use. Human experts on the other hand employ a large set of heuristics to analyze a scheduling problem. For instance, in one of the experiments we found out that when NEGOPRO had to choose between scheduling a final product and an intermediate product, it chose to schedule the final product first despite the fact that the intermediate product was linked to many other products that also needed to be scheduled once the intermediate product was. NEGOPRO made this opportunistic decision because the constraint to produce the final product was a bottleneck constraint. The right decision in this case, however, was to schedule the intermediate product first because it affected the schedules of many other products as well.

Within X% of the optimal solution

X	Percentage of Times off by at most X%
5	87
20	100

Figure 6-12: Comparing quality of NEGOPRO generated schedule against the optimal schedule

The last issue that we dealt with in the analysis was whether there exists a shorter sequence of revisions that achieves the same results that NEGOPRO achieved in the case of each experiment. The examination of all runs showed that in no case there existed a shorter sequence of revisions that could have reached the same point as NEGOPRO did (because we found no repetition of a sequence of operators within the entire list). However, the results of our sample does not imply that NEGOPRO always finds the shortest possible sequence of revisions to revise a schedule.

Chapter 7
Extending Heuristic Search To Deal With Uncertainty

A major problem with scheduling of software development projects is that there exists a great deal of uncertainty about their budget estimates (resource requirement predictions). As a result, the schedule which is originally constructed is at best a rough approximation of the actual execution and fails (to predict the actual resource requirements) frequently. Experimental studies [121] show that schedule failures could also be avoided or strongly reduced if there is an explicit early concern with identifying and accounting for the high-risk elements (uncertainty, in other words) [13] in them. The uncertainty in budget estimates can be attributed to the following reasons:

1. Software projects are different thus the data from previous projects can not be freely used to predict the duration and resource requirements of a new project [13].
2. The process of developing large software systems is not understood well enough to provide sufficient guide to predict resource requirement needs accurately [69].
3. Unexpected events which cause schedule breakdown are not uncommon [75]. For instance, a major bug may go undetected and surface very late in the development. Unexpected events are not just caused by improper implementation of project procedures (e.g. formal reviews), and can be attributed to politics inside the organization, change in customer requirements, failure of a supplier to deliver a resource by a deadline, sudden departure of a senior staff member, and so forth as well.

Human errors only add to the inaccuracy in resource requirement predictions by inflating or deflating them to serve their own personal objectives.

A major problem with the basic heuristic search approach that we presented in chapters 3 and 4 is that it assumes all resource requirement predictions are accurate. In this chapter, we discuss that how the basic heuristic search model can be extended to deal with the uncertainty in resource requirement predictions. The main issues that are related to extending the basic heuristic search model to deal with uncertainty are:

1. how to represent uncertainty.
2. how to quantify and measure uncertainty.
3. how to account for the uncertainty during heuristic search.

To manage the uncertainty in schedules, they have to be *continuously revised* over the course of project. To minimize the need for schedule revision, a number of *revision prevention* measures are used in practice including building a detailed schedule only for short time horizons and avoiding commitment to detailed schedules for long horizons, enforcing higher standards for quality assurance, and reserving additional resource upfront for high risk activities. While being popular methods for managing uncertainty, each of these measures raises a new set of problem. For instance, building detailed schedules only for short time horizons limits the schedule predictability, or the reservation of additional resource upfront for high risk activities [53] has the disadvantage that the reserves (which are kept as a safety net) might never be used. The main problem with preventive measures is that they do not modify the underlying scheduling process to account for the uncertainty in resource requirement estimates.

A second set of measures have been developed to manage the uncertainty by building contingency plans to be used if the present schedule fails. A contingency plan modifies the requirements of the original plan (e.g. if an activity takes more time than specified in the basic plan, then the documentation requirements of the remaining activities could be relaxed to reduce the time needed to complete the activity thus compensating for the time lost), or changes part of the original plan (e.g. if several members of a development team quit and other development teams are busy working on other projects, then the assignments of the development group could be subcontracted to an outside consulting firm). The main problem with managing the uncertainty through contingency planning is that it does nothing to reduce the chance of a schedule breakdown. Moreover, integration of a contingency plan with the rest of the project plan might require other parts of the plan (and subsequently the schedule) to be fixed as well.

A third set of measures have been developed to analyze schedules which have been constructed without any regard to uncertainty. In particular, these measures calculate the duration of a project according to its schedule when there exists uncertainty in the duration of various activities. PERT [79], developed by D.G. Malcolm, requires the attachment of three duration estimates to each activity: *most likely duration*, *most optimistic duration*, and *most pessimistic duration*. These three estimates are used to produce a probability distribution for estimating the duration of an activity. The resulting probability distribution is set aside until after the schedule has been constructed when it is used to develop a minimum and maximum estimate for the duration of project. The Monte Carlo simulation approach to computing project duration [127] uses the average of a series of independent simulation runs to estimate the project duration. During each run, a random number generator is used to choose a duration for each task based on beta distribution. The Monte Carlo method eliminates the major problem with the PERT technique known as *merge event bias*. Merge event bias is caused because the PERT technique implicitly assumes that there is only one critical path. By simulating the network multiple times, Monte Carlo eliminates the problem. The Monte Carlo method also generates an important value for each task known as the *criticality index* [79]. The criticality index, computed by dividing the number of times a task is on the critical path by the total number of simulation runs, represents the probability that a task will be on the critical path. This variable is a better estimator of how much attention a task deserves than slack is [96].

The main problem with existing measures for managing uncertainty (which we divided to three groups and discussed each separately above) is that they try to account for the uncertainty either before or after schedule revision, and fail to model it during the scheduling, where it is needed most. Our approach is to extend the basic heuristic search model by modifying the evaluation function to consider the uncertainty in resource requirement predictions. By implementing a more intelligent search (i.e. one that considers the uncertain nature of a resource requirement estimate while making a scheduling commitment), we reduce the likelihood of schedule failure. Like PERT, we use weighted interval estimates to represent the uncertainty in activity duration predictions of each project activity. If a user can predict a resource requirement accurate enough, then he can specify as a point estimate. Otherwise, he could predict a range within which the actual resource requirement is likely to fall. It is important to provide this freedom because it is not always possible to make accurate point estimates of a resource requirement. In fact, by committing to a single point estimate, the interval estimate information which reflects the state of affairs more accurately, will be lost. Unlike PERT, we do not limit the use of weighted interval estimates to activity duration predictions, and instead extend this concept to all types of resource requirements. For instance, an activity might be specified to need 2 to 4 programmers while it is more likely that it requires exactly three programmers.

As opposed to other approaches account for the uncertainty *only before* or *after* the revised schedule has been

constructed, our approach takes into account the uncertainty knowledge *during* the making of each scheduling decision. The advantage of our approach over contingency planning is that it reduces the chance of a schedule failure by taking into account the uncertainty knowledge in revising project schedules. Furthermore, our approach differs from preventive measures for managing the uncertainty in that it does not achieve more reliable schedules at the cost of slashing the requirements. The key to our approach is the ability to represent and model uncertainty during the scheduling. In the next section, we discuss how uncertainty knowledge can be represented and modeled, and also how the heuristic search can be modified to account for this knowledge.

7.1. Heuristic Search in the Presence of Uncertainty

The purpose of this section is to describe how uncertainty can be represented, measured, and taken into account in our basic heuristic search model. The representation of uncertainty can be achieved by modifying the definition of temporal resource constraints (definition 2.1). The following definition illustrates how definition 2.1 can be modified to allow weighted interval estimates:

Definition 7.1: if ζ is a resource requirement for a resource r by a product p, then ζ is of the form:

if $r \in R_{Upri}$ **then** $[(t_{2i}\ t_{2(i+1)}\ q_{ia}\ q_{ib}\ q_{im})\ \ \forall i \in 0..n]$
else if $r \in R_{Spri}$ **then** $[(t_{2i}\ t_{2(i+1)a}\ t_{2(i+1)b}\ t_{2(i+1)m})\ \ \forall i \in 0..n]$
else if $r \in R_{pro}$ **then** $[t_a\ t_b\ t_m]$

such that t, q_i, t_{2i}, and $t_{2(i+1)}$ are as in definition 2.1 and subscripts a, b and m distinguish the most optimistic, most pessimistic, and most likely approximations of the requirement for r respectively.

Example: Consider the resource requirement specification *[(-5 -3 1 4 2) (-1 0 1 1 2)]* for a senior programmer. The specification requires at least 1, at most 4, and most likely 2 senior programmers for the first three months, no senior programmer for the fourth month (therefore the specification of this month is left out) and at least 1, at most 2, and most likely 1 senior programmer for the last month of development.

Example: Consider the resource requirement specification *[-2 -4 -3]* for a debugger, that is being developed inhouse (i.e. the debugger is a product within the project), by a simulator. The specification requires the debugger at least 2 months, at most 4 months, and most likely 3 months before completing the simulator.

We can measure the probability of meeting Φ, the feature requirements of a product p, under an exact resource allocation specification ζ_{rx} of a resource r by calculating the probability that the request for r by p is a subset of ζ_{rx} (i.e. $P(\Phi,\zeta_r \subseteq \zeta_{rx})$).

Using our weighted interval representation, we can measure $P(\Phi,\zeta \subseteq \zeta_{rx})$ for all ζ by modeling it through a probability distribution function. By default, we construct a β distribution from the three point estimates most likely, most optimistic, and most pessimistic. However, a user might provide a probability distribution function based on the three point estimates as well.

If a user has more accurate information about the probability that Φ can be met under a given resource allocation, then he might choose an alternative representation such as breaking down the range between the most optimistic and most pessimistic approximations to a set of subintervals and associating a probability with each.

We can measure the probability of meeting Φ, the feature requirements of a product p, under the combined exact resource allocations specifications ζ_{rx} for all resources $r \in R_p$ by multiplying the probabilities that the request for each $r \in R_p$ is a subset of ζ_{rx} (i.e. $P(\Phi,\zeta_r \subseteq \zeta_{rx})$). This measurement can be generalized to a given schedule of p which can be stated formally as follows:

Definition 7.2: Probability of Meeting Φ under A Schedule Λ of *p*

$P(p,\Lambda) = \prod_{r \in R_p} P(\Phi,\zeta_r \subseteq \zeta_{rx})$.

This definition assumes that the contribution of each resource *r* to satisfying Φ of a product *p* is independent of the contribution of all other resources that are also required by *p* (i.e. are in R_p).

We define the reliability of a schedule Λ of a product *p* as the probability of meeting Φ of *p* under Λ and denote it by $\Theta(\Lambda,p)$. The definition of reliability implies that to increase the reliability of a schedule effectively, it is not enough to increase the allocation of only one required resource. Furthermore, the definition implies that the reliability is maximized when the level of allocations is balanced across all required resources of *p*.

Definition 5.3 is recursive since R_p, the set of required resources of *p*, usually includes both primitive and product resources. The recursion occurs only at the required resource nodes that are product since their probability has to be calculated in terms of their resource requirements as well. The reliability of a primitive resource is *1* since it denotes an allocation commitment. The computational complexity of measuring the probability of meeting the feature requirements of *p*, the final product of a project, is $O(n^2)$ where *n* is the number of products that produced by the project. If the probabilities of meeting the feature requirements of all products which are required by *p* are given, then the computational complexity measuring the probability of meeting the feature requirements of *p* will drop to $O(1)$.

To account for the uncertainty during the search, the heuristic evaluation function of our basic heuristic search model needs to be modified to consider the uncertainty in resource requirement predictions. In chapter 3, we described that the value of heuristic evaluation function for SPPS depends on two measures: cost and benefit. Although the exact cost of completing a project can not be predicted unless its resource requirements are exactly known, cost of implementing a given schedule is independent of any uncertainty can be calculated accurately (its value is the same as if there was no uncertainty). This is because the resource allocations of a given schedule are fixed.

In contrast, the benefit of a given schedule of *p* (i.e. the degree that the resources allocated to a schedule meet the requirements *p*) is affected by the uncertainty of resource requirement predictions. This is because the less certain the predictions, the smaller the probability that the feature requirements would be met and the schedule would not fall apart. We have stated this relationship formally in the following definition:

Definition 7.3: Benefit Under Uncertainty

Let UB(Λ,p) be the benefit of a schedule Λ of *p* under uncertainty.
Then $UB(\Lambda,p)=B(\Lambda,p)\times\Theta(\Lambda,p)$ where *B* denotes the benefit of Λ
if there was no uncertainty.

The calculation of benefit according to definition 5.3 allows the consideration of uncertainty during the calculation of heuristic search evaluation function. Since the value of the heuristic that approximates the degree of progress that an operator makes toward resolving a conflict (section 4.3) is dependent on the *cd* of the revised schedule and the *cd* of the revised schedule is dependent on the benefit of the revised schedule, our approach allow the uncertainty knowledge to be taken into account during each scheduling decision.

Although we have not implemented the approach which we described in this chapter to manage uncertainty, we can argue for the validity of our design on the basis that it is consistent with the process behaviour descriptions that are abstracted from experimental studies of software project risk assessment and risk control literature [13]. These studies stress that while project schedules need to be continuously revised in order to account for the

uncertainty in their resource requirement estimates, the need to for revisement can be reduced by accounting for the uncertainty knowledge during the making of each scheduling decision. This is because such consideration increases the reliability of the schedule that has been developed.

In our approach, uncertainty knowledge is accounted for during the making of each scheduling decision because (1) the value which is returned by the evaluation function constitutes the basis for evaluating a decision, and (2) the value of the evaluation function is computed on the basis of uncertainty knowledge. The former point has been discussed thoroughly in chapters 3 and 4. The latter point is evident from definition 5.3, where an increase in the allocation of resources to Λ will increase the value that the evaluation function returns for Λ, in accordance with the uncertainty knowledge. By increasing the level of allocations in Λ, first the reliability of Λ, Θ, will increase in accordance with the probability distribution of the resource requirements of each project product (definition 5.2). This, in turn, will affect the benefit of Λ since the benefit of Λ is directly proportional with Θ (definition 5.3). Lastly, an increase in the benefit of Λ (which has resulted from an increase in the allocation of resources to that Λ), while the cost remains fixed, will cause the evaluation function to return a higher value for Λ. The same argument can be repeated to show that the evaluation function would return a smaller value for Λ in accordance with the probability distribution function of the resource requirements of each project product, if the allocation of resources to Λ is lowered.

Chapter 8
Using Heuristic Search to Support Negotiation

Throughout this thesis, up to this point, we discussed a heuristic search model for scheduling which is consistent with human negotiation. However, we never discussed that how negotiation itself could be supported. In chapter 1, we characterized SPPS as a reactive process which involves continuous revision of project schedule. During each reactive cycle, first heuristic search is used to reduce the overall inconsistency of the schedule. Once such schedule (a schedule with minimal inconsistency) is found, it can be used to advise the users on the constraints that still need to be satisfied in order to make the schedule consistent. The users could then use this advice in conducting negotiation aimed at making the schedule consistent. For instance, they could reconcile the requirements of the constraints that could not be satisfied (to varying degrees), or they could propose alternative process plans under which those constraints are more likely to be satisfied. These changes in turn trigger a new reactive cycle which tries to exploit those changes to develop a less inconsistent schedule (by revising the existing schedule). A new reactive cycle will begin to be executed until a consistent assignment (a schedule without a conflict) has been found, or until the users stop introducing changes that trigger reactive response. In the present chapter, we provide a realistic analysis of the nature of negotiation and its role in each reactive cycle of SPPS, and investigate that how our heuristic search model can be used to support the negotiation process by advising the project agents.

Negotiation during SPPS is a process of multi-agent problem solving between a group of heterogenous agents with conflicting goals and interests to (resolve their conflicts and to) produce agreements among them. Negotiation during SPPS has both a cooperative and an antagonistic nature. The cooperative nature of negotiation is due to the fact that it is often essential for an agent to reach an agreement with other agents in order to protect his individual goals (e.g. continued employment or promotion) because they happen to coincide with the overall goals of the organization (meeting the requirements of its clients). The antagonistic nature of negotiation is due to the fact that each individual competes with other agents to insure that his own demands will receive top priority even if it means that the goals of other agents will not. Negotiation during SPPS occurs at different levels of organization, continues during the lifecycle of a project, and carries a mix of technical and organizational agendas.

A conflict in a schedule is a constraint that can not be satisfied by that schedule, and is caused by an array of technical and organizational reasons such as informational deficiency and goal incompatibilities. *Informational deficiency* is exemplified by the failure to execute an assigned task (which constitutes the constraints that can not be satisfied in this case) because the instructions are ambiguous or lack sufficient detail such that they become subject to multiple interpretations among the task carriers. In other cases, a decision maker may arrive at different conclusions because he might have used different sources of information. Conflicts based on misinformation or misinterpretation of information tend to be easier to resolve in the sense that clarifying the previous messages or obtaining additional information can often resolve the dispute. *Incompatibility of goals*

refers to the incompatibility between the goals that are pursued by different project agents. This type of conflict is exemplified by the classic goal conflicts between line and staff, production and sales, or production and R&D. Each organizational unit has different responsibilities in the organization, and as a result each places different priorities on the organizational goals (customer satisfaction, product quality, production efficiency, compliance with government regulations, etc).

Although negotiation is the principal way of resolving conflicts during the scheduling process, it is not the only one. By scrutinizing the nature of each conflict, human experts reason about the conflict resolution methods that need to be employed for resolving the conflict. Other popular ways of resolving a conflict (excluding negotiation) include

1. *resorting to authority:* when this method is used to resolve a conflict, the agent who forces his way out will feel vindicated, but other agents will feel defeated and possibly humiliated.
2. *avoid dealing with the conflict:* the agents who are involved in the conflict choose to remain neutral on the issue because they expect the conflict to relax as a result of the passage of time.
3. *quick fix:* agents become conditioned to seek expedient rather than effective solution.
4. *accommodating:* agents compromise because they believe that their positions are not so important that is worth risking bad feelings between each other.

It is interesting to note that not all types of conflicts are entirely negative to always need to be resolved. As a matter of fact, sometimes they are intentionally generated. Occasionally progress is achieved by engaging uninvolved individuals in a cause, and the creation of tension and conflict may be a desirable organizing strategy.

If negotiation is chosen as the method of resolving a conflict, then further scrutiny of the characteristics of the conflict is needed to identify the type of negotiation needed to resolve it. There are many characteristics of a conflict that affect negotiation type including:

1. *principle causes of the conflict*: For instance, a conflict which is motivated by political rather than purely technical reasons can not usually be resolved by a technical solution such as changing of product requirements.
2. *the number of parties (agents) involved*: For instance, if a negotiation involves more than two parties, then the geographical location and area of expertise of the parties could affect the negotiation plan.
3. *monoliticity of each party*: If the parties are not monolitic (i.e. include smaller groups that carry conflicting interests), then their internal conflicts might affect the process of formulating their positions.
4. *the number of issues involved*: For instance, if a negotiation involves many issues, then the existence of linkage effects between the issues affects the order of considering those issues for negotiation.
5. *requirement for an agreement*: If reaching an agreement is mandatory, then the negotiators can not risk the failure to reach an agreement, thus will be more willing to compromise.
6. *existence of time-related costs*: If one or more parties are constrained by a time-related cost, then they are likely to lower their requests when the negotiation appears to drag.
7. *private or public nature of negotiation*: If a negotiation should be conducted in private, then a set of measures have to be taken to insure this requirement, etc.
8. *group norms*: This refers to whether each party should expect that the other parties tell what they truely feel, disclose all the relevant information, honor their word, and so forth.
9. *knowledge of each party about the reservation prices* of other parties]

10. [36]: If a party has this information, then he can begin with a more intelligent bid and bargain better than the others.

11. *the need to resolve the conflict permanently*: If a conflict needs to be resolved permanently, then the parties might have to scrutinize the conflict in more depth than when it only needs to resolved temporarily.

Previous research in negotiation in artificial intelligence [21, 81, 120, 134]has not considered different methods of resolving conflicts (excluding negotiation) which are common in SPPS. Moreover, previous research has been limited to studying negotiation under idealized circumstances by assuming that there is no politics, all parties tell the truth, and so forth. Based on these assumptions, previous research in negotiation has developed computational models for negotiation process in a number of domains such as labor management contracts, air-traffic control, and resource reallocation. Among these works, we study RESOURCE REALLOCATOR (which we abbreviate as RR) [120], the one which is most relevant to this thesis from the stand point of commonality in the domain, in more detail. We also study PERSUADER [134] to show that how the difference in domain affects the underlying negotiation model.

RR [120]studies the reallocation of resource and identification of changes to resource allocations for a given set of changes to resource requirements in engineering projects via negotiation. Negotiation occurs when project agents who "own" a set of resources *sell* their allocation in order to buy new allocations that meet their requirements; see figure 8-1. The decision to reallocate a resource from one owner to another depends on the utility of the resources that an owner is prepared to sell and also on the utility of the resources that he is prepared to buy. The utility of a resource to a user is stored in a payoff (position) matrix.[37]. Scheduling constraints constitute the columns of the matrix and scheduling alternatives constitute its rows. Each project agent specifies his own payoff (position) matrix.

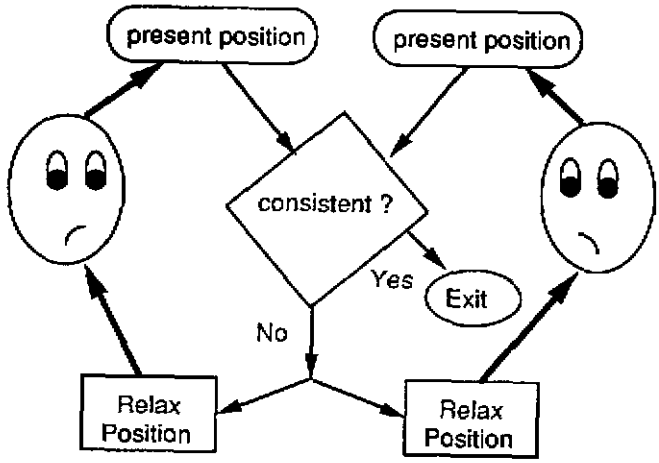


Figure 8-1: Negotiation Model in RESOURCE REALLOCATOR

Agents can propose a compromise using a variety of negotiation operators, such as slightly relaxing interacting

[36]The reservation price of a party as defined in negotiation literature is the degree that it will be prepared to compromise if the alternative is the failure to reach an agreement.

[37]Payoff matrix was first used by game theoretic approaches [112].

constraints (log-rolling), agreeing to relax the solution criteria, substituting one resource with another, bridging the different viewpoints by proposing a completely new solution, overlooking weak interactions among constraints, or appealing to third parties to settle the disagreement. The problem with RR is that it assumes project agents have fully determined their positions and also the compromises that they are willing to make, well in advance[38]. Consequently, negotiation is reduced to iteratively relaxing constraints until a compromise (feasible) reallocation of resources is reached[39]. This assumption oversimplifies the negotiation process in SPPS since

1. the space of feasible agreements and compromises that are acceptable to a negotiating agent is defined and refined (evolved) during the negotiation and not in advance.
2. a negotiating agent usually specifies enough information to only generate those solutions which are most desirable to him unless he is convinced that his demands as they are stated can not be satisfied.

Moreover, even if the positions (payoff matrices) were known in advance, for large negotiations involving many agents and outcomes, the matrix may quickly become intractable.

Compromise-negotiation to iteratively exchange offers until a compromise is found also has been advocated in the design of COEX [81] and PERSUADER [134]. However, the difference between compromise-negotiation in PERSUADER and compromise-negotiation in RR is that in PERSUADER relaxation is not constrained by the payoff matrix and continues until a solution is found. The idea is to continue to relax the less important constraints to generate a solution that can be proposed if no solution with positive payoff can be generated.

PERSUADER [134] implements a model of negotiation in which it acts as a mediator in union/management labor contract negotiation. PERSUARDER performs negotiation through proposal modification and goal relaxations. PERSUADER integrates case-based reasoning with the use of multi-attribute utilities to portray tradeoffs and propose novel goal relaxations and compromises. PERSUADER also generates persuasive arguments to compel an agent to change his position. The difference between the negotiation during SPPS and labor management contract negotiation is that:

1. In the software project domain, project agents tend to resolve their conflicts locally, within the organizational unit they reside, and will propagate them to a more global level (their parent organizational unit) only if the local problem solving effort has failed. In contrast to this, in the labor management contract negotiation domain, the entire negotiation is carried at a global level by two parties each representing one side of the dispute.
2. In contrast to the labor management contract negotiation domain, where conflicts always arise as a result of organizational politics, scheduling conflicts in the software project domain at times arise because there has been a technically inaccurate projection of the level of resources required.

Furthermore, negotiation in SPPS is a harder problem than labor management contract negotiation in two respects:

1. software project scheduling is analogous to a contract negotiation case that every individual negotiates for a contract separately. If the agents negotiate for separate contracts and the sum of salaries and benefits that can be granted is limited, then the employees will have to compete for meeting conflicting goals.
2. contracts (agreements) that result from negotiation in the software project domain have to meet not only the requests of the negotiating agents but also the objectives of the project. These objectives include meeting the delivery deadlines without violating the available capacity constraints.

---

[38]*This is true generally of payoff matrix schemes (e.g. Rosenshein).*

[39]This method of negotiation is known as *compromise-negotiation*.

Although some of the ideas used in PERSUADER (e.g. persuasive argumentation) can be applied to negotiation in SPPS, the domain differences prevent us from using the negotiation process model in PERSUADER to reason about the negotiation in SPPS for the most part. For instance, PERSUADER formulates *novel solutions* to resolve a conflict by focusing on only satisfying the constraints that are considered to be very important to reach a solution[40] and considering drastic relaxation of other constraints, while generation of novel solutions during SPPS negotiation involves redefining parts of the problem space such that we no longer deal with the same set of constraints.

Our approach is to try to support the negotiation during SPPS (making it more efficient) by providing the information which is essential to organizing and conducting negotiation to project agents. In contrast, previous approaches have concentrated on *automating* the negotiation processes by replacing the human negotiators with automated reasoning systems. Our approach relaxes the need to be concerned with modeling restrictions which oversimplify the actual negotiation process during SPPS.

In fact, the need for human negotiation is *imperative* in many cases including when

1. the way an agent presents his positions does make a difference in resolving the conflict. A program cannot present a position with emotion, with sincerity, or any other presentation and speech technique that are specific to humans. Sometimes presentation and speech acts turn to an effective tool in resolving a scheduling conflict.
2. human-centered attributes of a negotiation such as whether the negotiation is public or private influence the negotiation.
3. negotiators are not logic bounded. Previous work (e.g. [135]) assume that all negotiators are logic bounded beings and are strictly motivated by economic interests. These assumption do not agree with what goes on in the real world. Frequently, a political or personal motivation or even illogical rationality might be behind a stand of a project agent. Suppose that a testing team is scheduled to carry integration testing on a set of module that are to be delivered by a group of development teams. Furthermore, suppose that one of the groups is way behind the schedule and requests an extension of the delivery date. After being convinced that the module can not be completed by the deadline, the project manager contacts the testing team to find out if a delay in the delivery of the module will delay the delivery of the integration testing. The testing team might choose to complain but agree in principle to make up for the delay in obtaining the software modules to complete the integration testing on time. The testing team might take this stand despite being aware of the unlikelihood to complete the testing on time because it can not only strengthen its position by creating the impression that it is very responsible, but also will provide an excuse if it fails to complete the testing on time. The testing team also might use the agreement to make up for the delay in obtaining the modules as a bargaining chip to get concessions later.

Even if human negotiation is not imperative, automating the negotiation should be avoided when the cost of encoding a conflict and the data needed to resolve it exceeds the time that takes to resolve the conflict on a face-to-face basis.

Unlike previous approaches, in our approach we do not keep relaxing the feature requirement constraints until a conflict free schedule has been reached, unless the resources which will be saved by each relaxation have been specified in advance. On the other hand, the cost of resolving a conflict through increasing the available capacity of the required resources (i.e. relaxing the available capacity constraints) could be determined easily. This can be used to tradeoff the possibility of relaxing a feature requirement constraint of a product with other feature requirement constraints of that product, with the feature requirement constraints of other products, or with the available capacity constraints of the project.

---

[40]This approach is along the line of selective relaxation but implements it to a greater extent.

In the remainder of this chapter, we discuss that how heuristic search can be used to support negotiation during SPPS. The interviews that we conducted as well as the software project management literature that have been cited at the end of this thesis showed that the first step in a negotiation during SPPS is to identify the conflict that has to be resolved; this the conflict which is most critical to the overall improvement of the schedule. The task of finding the bottleneck conflict is not as easy as it might look. Frequently, it is not possible to speak about a conflict without including other conflicts that it relates to.

The second step in a negotiation is to find the conflict that has to be negotiated. Frequently, further examination of the bottleneck conflict might suggest that it can be translated to one or several other conflicts that are easier to resolve. Suppose that a team leader needs a senior programmer for a task when the available capacity of senior programmers is zero (conflict *A*). Moreover, suppose that the same task can also be carried out by two junior programmers, and that it is possible to acquire 2 junior programmers from a different division. Then it might be best to translate the first conflict to the second, and resolve the second conflict instead.

In NEGOPRO, first finding the bottleneck conflict and then finding the conflict that has to be negotiated is achieved through heuristic search. Basically, NEGOPRO finds the bottleneck conflict during each incremental revision step and (if possible, translates it to one or more conflicts that appear to be easier to resolve), until what it considers as the conflicts that can be solved through the least costly set of commitments, are reached. The conflict which are returned by NEGOPRO represent those that need to be negotiated.

NEGOPRO could provide several kinds of assistance in resolving the conflicts that are found. These include

1. *examining the circumstances that led to the emergence of the conflict;* since NEGOPRO keeps a history of the scheduling states, a user can find when and what led to the emergence of the conflict which has been selected for negotiation (or aggravation of the conflict that previously existed). This is achieve by undoing the operators that have been applied to resolve each conflict chronologically, and by stepping back through the conflicts that have been examined. Although this reasoning can not provide any clue about the source of a conflict in the seed schedule (before being revised by the heuristic search), it can reason about these sources since the heuristic search was first applied to revise the schedule.
2. *finding the agents who should be involved in resolving it;* this involves not only the agent who has made the resource request that can not be satisfied, but also other agents whose decision can affect the conflict, and benefit or lose from resolving the conflict.
3. *impact analysis:* impact analysis refers to finding the impact of making a new scheduling commitment on the schedule as a whole. Impact analysis allows negotiators to gain more insight into possible settlements routes by investigating the consequences of different compromise scenarios not only on the conflict under negotiation but also on all other parts of the project schedule. These statistics show that how important it is to resolve a given conflict, not just from the local perspective, but also from the perspective of the project as a whole. In NEGOPRO, impact analysis is achieved through constraint propagation.

Persuasive argumentation [134] and due process conflict resolution [48] might be integrated with our model to further assist the negotiation process. For instance, persuasive argumentation could be provided to convince the negotiators of the importance of a conflict by showing the impact of leaving it unresolved.

A diagram of negotiation support architecture is depicted in figure 8-2.

Although we have not implemented the approach which we described to support human negotiation during SPPS through the results that can be obtained from heuristic search in this chapter, we can argue the validity of our design on the basis that it is consistent with the empirical studies of social analysis of computing [47, 121]. These studies provide a realistic analysis of the complexity of human negotiation process during SPPS. We

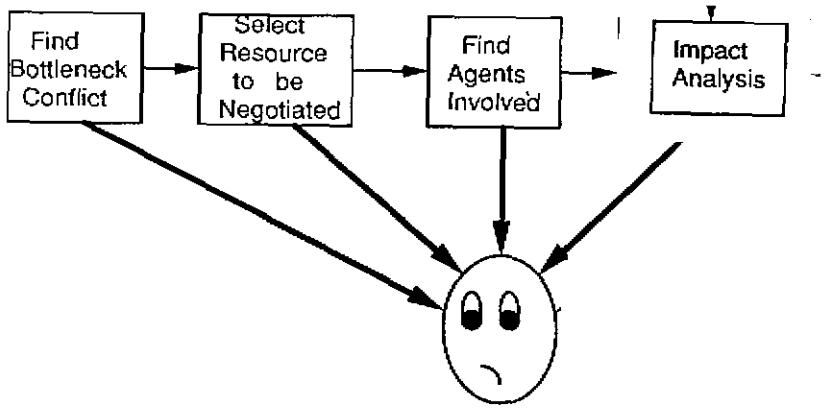


Figure 8-2: *Negotiation Support Architecture*

explored the obstacles that exist in replicating human negotiation through automated reasoning, and recognized the need to focus on supporting human negotiation by providing analysis capabilities, as opposed to making simplifying assumptions that depict an unrealistic sketch of the negotiation process.

Although the analysis capabilities that can be provided by implementing our approach satisfy analysis needs that are commonly raised during human negotiation in SPPS, it would be interesting to measure their impact on increasing the productivity and effectiveness of negotiation. To measure these impacts, in addition to implementing the analysis capabilities, a suitable user interface to communicate the analysis results to the user needs to be designed. This is because demonstration of the effectiveness of our approach also depends on how the analysis results are presented to the user.

Chapter 9
Conclusions and Future Directions

In today's highly competitive market for software products, the *on-time* and *within budget* delivery of software products has become the key to the success of a software development organization. Furthermore, the ever growing demand for software products has led to the establishment of large software development organizations that work on multiple software projects concurrently. Manual SPPS in these organizations is not only time consuming and costly, but also produces limited success as the size of organization grows.

Software projects are planned and scheduled under an array of conflicting technical and organizational constraints including budget constraints, temporal and resource capacity constraints, tool and staff productivity limitations, organizational rules and regulations, and so forth. The task of planning and scheduling is further complicated by the fact that some of these constraints are negotiable (i.e. can be satisfied to varying degrees) and are subject to a diverse set of preferences. SPPS is distributed among multiple agents each with its own set of resource requirements and involves face-to-face human negotiation between them to resolve the scheduling conflicts. Uncertainty in the budget estimates of a project activity is another source of problem in SPPS. Human errors only add to the inaccuracy in resource requirement estimates by inflating or deflating them to serve their own personal objectives. Last but not the least, the occurrence of frequent unexpected events (e.g. discovery of a major bug in the product, high rate of staff turnover) can make some aspects of the plan or schedule that has been developed obsolete frequently.

The main contribution of this thesis is that it represents the first effort in building a *problem solving model* that accounts for the major issues of SPPS. We identified the full range of issues that make SPPS a difficult problem and described the reason that previous approaches have been unsuccessful in addressing them in chapter 1. Our approach was to

1. define SPPS as a reactive process that involves human negotiation, and
2. develop a heuristic search model, that is consistent with the negotiation process, to improve an existing schedule by incrementally revising it.

Our approach was the result of recognizing that (1) software project schedules evolve through human negotiation, which continues through all phases of SPPS, and (2) SPPS is more of a schedule revision problem than that of schedule generation. In the contrary, previous research in developing a problem solving model for planning and scheduling [64, 9, 85, 96]considers that negotiation during SPPS is limited to a set of compromises that are known in advance, and also assumes that SPPS is a purely schedule generation problem. Our approach was also the result of recognizing that the problem solving process in SPPS can be modeled by a synthesis of heuristic search and human negotiation. This recognition enabled us to formulate a problem solving model that is consistent with the process of human negotiation during SPPS.

We divided the reactive revision of a software project schedule to a set of reactive cycles such that each cycle involves successive revisement of the schedule until the penalty of resolving the conflicts in that schedule(i.e.

making the schedule consistent) becomes minimal. While heuristic search is carried out within each cycle to minimize the inconsistency, human negotiation takes place between two reactive cycles to revise the scheduling assumptions. In chapter 2, first we provided a formal definition of SPPS and on the basis of that showed each reactive cycle can be achieved by heuristic search through the space of possible schedules to improve (lower the inconsistency of) a given schedule by incrementally revising it. Once a schedule with minimal inconsistency is found, it can be used to advise the users on the constraints that remain unsatisfied. The users, we showed in chapter 8, could then use this advice in conducting negotiation aimed at making the schedule consistent. The result of negotiation can be recorded as changes in the assumptions of the problem. We described the set of these possible changes in chapter 5. The changes trigger a new reactive cycle. Heuristic search is used to exploit these changes to develop a less inconsistent schedule (by revising the existing schedule). A new reactive cycle will begin to be executed until a consistent assignment (a schedule without a conflict) has been found, or until the users stop introducing changes that trigger reactive response.

At the end of chapter 2, we provided an overview of the components of our basic heuristic search model. The evaluation function component of our basic heuristic search model was discussed in chapter 3, while the scheduling strategy and search operators were discussed in chapter 4. In chapter 3, we described that how our evaluation function takes into account an array of conflicting technical and organizational constraints including budget constraints, temporal and resource capacity constraints, tool and staff productivity limitations, and so forth, which are negotiable to varying degrees and are subject to a diverse set of preferences. In previous SPPS approaches, these factors were largely not modeled. In chapter 4, we discussed a design of operators that allows the span of the entire search space for SPPS and building of more complex operators from a set of primitive operators. In this chapter, we also discussed a heuristic theory to control the application of operators. Our heuristic theory was different from that of previous approaches (e.g. [104] in that it emphasized the minimization of disruption as the primary criteria for optimization.

We explored how the assistance of a human scheduler might increase the efficiency of search by providing an interactive collaborative scheduling framework in chapter 5. In previous SPPS approaches, a human scheduling expert has mainly played a passive role in problem solving. In chapter 6, we verified our basic heuristic search model through a set of experiments which we ran using NEGOPRO, a program which we developed to implement our problem solving approach. In summary, these experiments showed that

1. by increasing the overall size of the problem, the time that is spent on conflict analysis, operator selection and schedule revision, and schedule evaluation all increase. However, the rate by which the computational cost of the latter two increase is smaller than the rate by which the computational cost of conflict analyzer increases.
2. schedule evaluation does not constitute a bottleneck from the computational standpoint and therefore our scheme based on cost/benefit analysis may be coupled with other scheduling strategies as well.
3. the choice of seed schedule affects the quality of the final schedule as well as the speed by which that schedule is produced (measured in terms of the number of operators applied), but that the quality of the final schedule is affected most.

In chapter 7, we extended the basic heuristic search model by designing a more intelligent search (i.e. one that considers the uncertain nature of resource requirement predictions) to reduce the likelihood of schedule failure.

9.1. Future Work

While our goal from conducting experiments throughout this thesis was to study the effectiveness of the basic heuristic search model as a whole, it will be useful to conduct additional experiments to test each component of this model separately. This is because the present work has developed several novel ideas in modeling scheduling problems through heuristic search that each is worthy of being investigated on its own. For instance, it would be interesting to see how the heuristic strategy which we have proposed (i.e. keep the schedule disruption at a minimum while making the most progress toward resolving a given conflict) actually compares with other heuristic strategies that have been proposed in the literature [43, 98, 104] when other components of the heuristic search model are fixed.

Furthermore, it would be useful to

1. implement our design to extend the basic heuristic search model to account for uncertainty in resource requirement predictions and measure its effectiveness in building more robust schedules,
2. evaluate the impact of supporting negotiation on the efficiency of negotiation through the results that can be obtained from conducting heuristic search, and
3. evaluate the impact of collaborative human-computer heuristic search on increasing the efficiency of search

through empirical results. Although the reliance on empirical studies in addressing these issues [13, 47, 121] in the present work was sufficient to verify the correctness of our approach, the degree of its effectiveness can be measured only by conducting sophisticated experiments at real world software development organization sites in a timely fashion.

Last but not the least, we have recognized the importance of developing a heuristic to optimize the search within the available time for scheduling. This heuristic should allow users to balance the quality of a solution with the time that they can afford for finding it. Since we have adopted a schedule revision (in contrast to a schedule generation) approach to scheduling, we always keep a copy of the most recently revised schedule. Moreover, since this schedule represents the best schedule that has been developed so far, we can always return the best complete schedule that the heuristic search has produced as soon as the bound on the scheduling time is reached. However, the heuristic search is not optimized within the available time.

Our preliminary analysis suggests that we could exploit the conflicting nature of problem constraints in order to optimize the search within the available time for scheduling. In planning and scheduling problems such as SPPS that constraints are frequently of conflicting nature, sometimes it is useful to relax a less important constraint at the cost of satisfying more important constraints that it is in conflict with, before exhaustively considering all alternative ways of satisfying all conflicting constraints together. This is because if the search space is highly constrained, then we can relax some of the less important constraints without searching for an assignment that satisfies all conflicting constraints together.

Exploitation of the conflicting nature of problem constraints as a basis to optimize the search has the potential of allowing a user to trade off search time with the probability of finding a consistent schedule. The search space is reduced according to the bound on scheduling time by discontinuing the search for finding a consistent schedule along the paths that the probability of finding a consistent schedule in them is low (i.e. are highly constrained) during scheduling (i.e. switching to solving a weaker version of the problem that is in line with the bound on scheduling time). In trading off search time with the probability of finding a consistent solution, the saving in search time should be even greater when a schedule exists that satisfies every disputed constraint but at the same time violates new constraints elsewhere within the network of problem constraints (i.e. will disrupt the network).

In the present work, we distinguished SPPS from traditional AI planning approaches through its unique emphasis on *efficiency* during the revision of schedules, a goal that is apparent in the design of our heuristic search approach. Unfortunately, the emphasis on efficiency did not become a priority in the implementation of NEGOPRO and remains a potential that needs to be exploited, because the the development of NEGOPRO itself took a very long time. Presently, NEGOPRO schedules projects that contain 15 or fewer activities very efficiently; *this is true even if most* activities have alternative process plans and allow variable mixes of resources, some of the feature requirements of project products can be compromised, and an array of constraints and preferences have been specified. However, *real world software projects* include a larger number of activities. In order to scale up NEGOPRO for scheduling larger experiments, its present implementation needs to be optimized considerably.

There are two other areas which deserve further exploration in the framework of the SPPS problem solving model that we developed in this thesis: types of resource related preferences and organizational modeling constraints that can be posted, and *learning new heuristics. In the remaining* of this chapter, we investigate these two area in the order that have been stated above.

The constraint and preference representation language which we formally defined in chapter 2 lacks the *expressiveness needed to discriminate among* different instances of the same resource. For instance, a project manager might advise against assigning John to the same task that Dave is assigned to because they might have a history of personal *differences. The discrimination among* different instances of the same resource can be provided if the present representation language is extended to support hierarchical (and recursive) representation of resource classes and instances (in order to be able to reason *at different levels of* abstraction). Our representation also lacks the expressiveness needed to specify such organizational policies and regulations as "each junior programmer should be assigned to work with one senior programmer and at most three *junior* programmers can be assigned to each senior programmer." Although the problem specification language that has been developed for PMA [64] supports the specification of a wide variety of constraints and preferences (including those that are stated above), there exists no underlying problem solving model to generate a schedule that satisfies them.

Although we have modeled the authority structure (constraints) of a software development organization *during* planning and scheduling, the organizational structure of a firm including the policies and regulations that govern it impact the planning and scheduling decisions as well. For instance, in a matrix organization functional units are organized on the basis of their functions (thus a functional unit can be working on several projects concurrently), while in a different type of organization, they might be divided along the project boundaries. *Furthermore*, a software development organization might include interdisciplinary units, special committees, and so forth that allocating them to different projects might require following special rules. We advocate the development of a *representation language that allows the specification* of the structure of an organization from a set of primitives organizational elements (e.g. employee), and extending our heuristic search model to take the organizational rules and regulations into account during SPPS. *Previous research in this area* [24, 40, 47] *can* provide a starting point for this effort.

The need for diversity in the type of resource preferences and organizational modeling constraints which have to be modeled has been *manifested in an experiment involving the scheduling* of a real world software development project. The data which is used in this experiment is a combination of real world and hypothetical data. The real world data belongs to Travelers' Guide (TG), a real world software project that started *in the summer of 1988 in*

Los Angeles under the initiation of Negofirm. The hypothetical data belongs to NegoTech, a medium size software manufacturing firm which is engaged in software development for business applications and will play the role of parent organization of TG. In contrast to TG which describes a single project, NegoTech serves as a multi-project case. A study of multi-project software organizations was conducted to assure that NegoTech is representative.

The organization that began to develop TG was a startup software company that did not have a well defined organizational structure. The data about the development of TG that is used in the experiment is collected from reviewing the project planning and scheduling reports that we obtained in the hardcopy form and conducting a set of lengthy interviews with the manager of the project.

We believe that the data which has been collected is valid because:

1. the manager was highly informed on the undocumented planning and scheduling data. The main reason for being informed was that the project was very young at the time of interview and the manager remembered most undocumented details.
2. the manager did not have any interest to provide biased data.
3. the data which had been collected was reviewed to assure that it was consistent.

Information on the present state of the project and the organization is withheld on the request of the management.

The hypothetical data covers the organization on which TG is being executed. The design of this organization was influenced by three factors: the actual organization that carried out the TG project, demands of the experiment, and the documented description of the structure of software organizations in the sources that we studied. Since as it presently stands NEGOPRO is insensitive to project communication, it fails to account for the organizational structure that has been designed. Furthermore, NEGOPRO is not able to model various staffing and resource allocations regulations and policies that are described in the experiment. A detailed description of the experiment is provided in appendix A. This experiment can serve as a benchmark for evaluating and comparing the problem solving models that will be developed for SPPS in the future.

While in this thesis we emphasized the problem solving aspect of planning and scheduling that is based on precompiled knowledge, much remains to be done in providing NEGOPRO with a capability to learn from the previous projects that it has scheduled. Since it is not feasible to encode every possible heuristic in NEGOPRO, the need to learn new heuristics increases with the diversity of the problems that have to be solved by NEGOPRO. Existing general purpose planning systems that learn, largely operate on problems with small search spaces and limited subgoal interaction [95, 113]. Furthermore, the interaction of learning and scheduling, for the most part, has been in the arena of genetic algorithms. Application of these techniques to software manufacturing scheduling constitutes an area of future research. More Recently, Merl-Soar [58] has casted the job-shop scheduling problem as a set of problem spaces within the Soar architecture [102] and has investigated the use of chunking [102] for learning purposes. The merits of the capability to learn is by no means limited to learning new heuristics in NEGOPRO. For instance, one area that learning can be very useful is in screening inflated resource requests (also called validating resource requests). Inflated resource requests (demands) are caused by inexperience, or are motivated by personal interests and politics. By screening out inflated resource requests, a scheduler can

1. eliminate the need for negotiation by lowering the resource requirements to a level that can be met.
2. help the negotiation process by preventing the negotiators from making inflated demands that could endanger the success of negotiation.

Appendix A
Travelers' Guide

Travelers' Guide is to be a widely available electronic system which provides a community (in particular travelers) information in the following three areas:

1. Local entertainment alternatives
2. Local points of interest for Tourists
3. Navigational information to help a driver travel from one part of the city to another

We use A1, A2, and A3 to cross reference these.

TG is functionally divided into three logical components, each with the purpose of fulfilling the above system objectives. Below is a description of the three components of the system.

The entertainment component of the TG allows the user to query and receive information about the following items:

1. Local Restaurants
2. Local Movies
3. Local Stage Performances
4. Sporting Events
5. Shopping Centers

The user will select one of these categories to receive a hardcopy printout listing of pertinent information about the category that he chose.

The tourist information component of the TG allows a tourist to query about the:

1. Local Tourist Attractions
2. Local Hotels

The navigational (path finding) component of TG allows the user to enter a source and a destination address within a city, finds a path that can be followed to reach the destination address from the source address, and outputs the path to the user. If TG is accessed through a touchtone phone, then the output will be in form of voice. If TG is accessed a teller machine, then the output will be a hardcopy map including a graphical representation of the path and written directions which describe the path.

A decision has to be made as to whether TG should be widely accessible through touchtone phones or only accessible through geographically distributed teller machines with local computing power throughout the city. In either case, a powerful central host computer will be needed. If tellers are used, then user input is entered at a teller and then transmitted to the host computer. The processed information is later transmitted back to the teller where it is displayed on the teller monitor. Information is passed between the host and the tellers via phone lines. Users will be able to pay for the service that they receive in cash or charge.

Customer Requirements

The customer has defined the main requirements of TG as follows:

1. TG has to be a user friendly system.
2. The product must work on a network of hosts and teller machines.
3. Multiple users have to be supported simultaneously.
4. The response time of TG once the user completes typing a string of input or selects a menu option has to be below 10 seconds regardless of the number of users.
5. A prototype of the system has to available for demo within 4 months.
6. TG has to complete in 8 months.
7. The maximum budget ceiling is $900,000.

The implementation of TG has to validated against these requirements.

Initial Deals

The description of the process of reaching an agreement between NegoTech and Negofirm, the firm that has ordered TG, has been greatly simplified since the process is not the focus of attention of the experiment. The policy of NegoTech is to first conduct a technical feasibility study to learn if a project is technically feasible. If the answer is positive, then a cost estimate for each component or requirement of the system that can be measured independently is produced. The cost sheet is then passed on to Negofirm. Once the cost and the requirements are negotiated between NegoTech and Negofirm, a new budget and a new set of requirements is mapped out and an agreement is signed between the parties. For instance, after a negotiation, NegoFirm agreed to drop its requirements to see a demo of the prototype of the system in 4 months after NegoTech argued that the time which is needed to go into the preparation of a demo might result in missing the completion deadline.

Project Initiation

A project is initiated in NegoTech by selecting a project manager. In the case of TG, the name of the project manager is NegoMan. NegoMan brings together a small group of team leaders and consults with them to construct a project plan. The plan divides the development process into four major phases: specification, design, coding, and testing and validation.

Basic Setting for Inter-Project Resource Sharing

NegoTech starts in an environment where two other projects are already under development in NegoTech. The addition of TG implies that organizational resources have to be shared between all three projects.

Activities

This section is to describe the project activities that belong to various software lifecycle phases. Although the choice of touchtone or teller machine for accessing the system was made prior to system specification, the activities that are related to making this decision are included as part of the specification. The reason for this decision was to capture the dependency of major planning decisions even prior to system specification.

It is important to note that the specification of project activities that are enlisted in this section was itself a result of preliminary project work that has not been included in the present experiment.

The activities that belong to the specification phase include:

1. specify the exact queries that an end user can ask the system.
2. specify the maximum time that can be afforded to log on an end user and respond to a query of an end user. To this end, the time that takes to initially hook up through a touchtone phone or a teller, and the time that takes to respond to queries of different types needs to be estimated. The time that takes to respond to queries of different types depends on the portion of query processing that can be done by the teller or the front-end computer.
3. estimate the minimum, maximum, and average load (number of users logged on at any given time) and the kind of queries that they ask. Furthermore, determine the number of phone lines or teller machines that are required to handle this load.
4. determine whether TG should be accessible through touchtone phones or through geographically distributed teller machines with local computing power.
5. specify the choice of hardware for use. If information is entered on teller machines, then it has to be decided how to divide the processing power between a teller and the host.
6. specify the choice of hardware and software for development. This includes the choice of computer language, operating system, graphic tools, and so forth.
7. specify the exact end user input and machine output in A1, A2, and A3. As part of this, it needs to be determined if the secondary streets are to be included in the representation. Although the task of specifying the input and output of A1, A2, and A3 can be executed in parallel, the present task design provides a more interrelated specification process.
8. specify the format in which an end user has to provide his input, and also the format in which the machine has to provide the output to him in A1, A2, and A3.
9. specify the information that needs to be updated in A1, A2, and A3 by an engineer user, and furthermore specify the frequency of updating.
10. specify the exact input and machine output for the engineer user menu interface.
11. specify the format in which an engineer user has to provide his input, and also the format in which the machine has to provide the output for the engineer user menu interface.
12. specify the criteria that can be used to compare the paths that exist between a source address and a destination address. Although distance is one of these criteria, other criteria such as the time to travel, and the number of traffic light (which contributes to the chance of getting lost) also have to be considered.

These activities are cross referenced as SP1 through SP12.

The activities that belong to the design and implementation of A1 include:

1. design a file system for A1 that specifies the data files that have to be constructed, the schema of each file, and the format of information in each schema. The design should consider many factors such as optimizing the retrieval (search) time at the cost of update time. Such information as an estimate of the number of entries in the entertainment directory files could effect the choice of structure of a file in the file system.
2. design a set of file access and file update operations for the file system to allow the access that is essential to respond to end user queries and allow the update that is carried out by the engineer user. Such questions as "what part of the entertainment database has to be resident?" has to be answered here.
3. design and attach an access right to each file system access function.
4. prototype the file system and the operations that access the file systems to evaluate the file system. The prototype of the operations has to enforce the access rights. Use the prototype to revise the original design of the file systems to fix the problems that are identified in the prototype.
5. implement the file system for A1.

According to the cross referencing scheme, these activities are A1.DC.1 through A1.DC.5. The same set of activities need to be executed for the design and implementation of A2. According to the cross referencing scheme, these activities will be A2.DC.1 through A2.DC.5.

To implement A1 and A2, the organization can alternatively use a data base generator such as RBASE, Clipper, or DBASE. The use of a DB generator will significantly reduce the time to execute A1.DC.4, A1.DC.5, A2.DC.4, and A2.DC.5.

A data base generator also can be used to reduce the time to prototype and implement the menu interfaces. This is because DB generators such as Lisa or RBASE provide high-level screen layout specification languages.

There are three problems with using a DB generator. The first problem with using a DB generator is that it usually requires an inflexible interface to a programming language. This poses a problem to smooth integration of A1, A2, A3, and the menu interfaces thus increasing the integration time. The second problem with using a DB generator is that it could degrade the response time of the system. The third problem with using a DB generator is that NegoTech does not employ the necessary skills to develop a sophisticated application in RBASE. This implies that the skill has to be acquired from the marketplace.

The activities that belong to the design of A3 include:

1. define the logical structure of the city map. This requires the identification of the elements of a city map that are important in processing end user queries, and building an abstraction of the city map that includes those elements in a logical structure.
2. design a search algorithm to search through the logical structure of the city map and find the most qualified path that connects a source to a destination according to the criterias that are defined in SP12. The algorithm is also required to take into account the
    a. congestion due to traffic at different periods during a day.
    b. on line traffic information about the accidents, freeway closing, etc.
3. design and declare the data structure of the city map and the functions necessary to access it. The design should distinguish between one way streets and two way streets, 3-way, 4-way, and 5-way intersections, streets with only one exits, and so forth.
    a. congestion due to traffic at different periods during a day.
    b. on line traffic information about the accidents, freeway closing, etc.
4. design a module that takes input from a scanner and encodes a map of the city in the designated data structure.
5. design an algorithm to sort multiple destinations that are to be reached from a source address. The goal is to minimize the overall travel distance, travel time (traffic lights and traffic can increase the travel time), and the number of turns.

These activities are cross referenced as A3.D.1 through A3.D.5 where A3 is a component name, D is an abbreviation for design, and the number is the index of the activity.

The activities that belong to the coding phase of A3 include:

1. develop the functions that have been designed to access the data structure of the city map.
2. construct a toy map that is representative of the city but encodes only a small subset of the map of the city.
3. build a prototype of the search algorithm and test the algorithm to find the path between a source and a destination address on the toy map. The implementation needs to account for
    a. the congestion due to traffic,

b. on line traffic information, and

c. multi-destination search.

4. implement the search algorithm that has been designed. The implementation needs to account for
   a. the congestion due to traffic,

   b. on line traffic information, and

   c. multi-destination search.

5. construct a complete map of the city. In order to construct a city map, questions such as "what part of the city map should be kept resident?" should be answered first.

According to the cross referencing scheme, these activities are A3.C.1 through A3.C.5 .

The design and implementation of a menu determines what should be displayed on that menu and also when and where that information has to be displayed. The activities that belong to design and implementation of the end user menu interface (DMI) include:

1. develop a set of standards for screen design, warning and error message display, and on line help for the end user menu interface system.
2. design credit card and cash taking menus. The design is required to display appropriate warnings and error messages on erroneous input and also provide multiple levels of help.
3. build a prototype of the design of the credit card and cash checking menus to evaluate the design and use that to revise the design to fix the problems that are detected in the prototype.
4. implement the revised design.
5. design the menu system for A1. The design is required to display appropriate warnings and error messages on erroneous input and also provide multiple levels of help.
6. build a prototype of the design of A1 to evaluate the design and use that to revise the design to fix the problems that are detected in the prototype.
7. implement the revised design.
8. design the menu system for A2. The design is required to display appropriate warnings and error messages on erroneous input and also provide multiple levels of help.
9. build a prototype of the design of A2 and use that to revise the design to fix the problems that are detected in the prototype.
10. implement the revised design.
11. design the menu system for A3. The design is required to display appropriate warnings and error messages on erroneous input and also provide multiple levels of help.
12. build a prototype of the design of A3 and use it to revise the design to fix the problems that have been detected during the prototyping.
13. implement the revised design.
14. revise the design of menu systems for A1, A2, A3, and credit card and cash taking menus to meet the requirements of integration and then integrate the designs of menu systems for A1, A2, A3, and credit card and cash taking menus into an integrated end user menu interface.
15. prototype the integrated menu interface design and use that to revise the integrated menu interface design to fix the problems that have been detected during the prototyping.
16. implement the integrated menu interface design.

According to the cross referencing scheme, these activities are DMI.DC.1 through DMI.DC.12 where DC indicates the fact that coding does not sequentially follow the design thereby design and coding are accomplished in one phase.

The design of a menu determines what should be displayed on that menu and also when and where that information has to be displayed. The activities that belong to design and implementation of the engineer user menu interface (GMI) include:

1. design the update and maintenance menus. The design is required to display appropriate warnings and error messages on erroneous input and also provide multiple levels of help.
2. build a prototype of the design of the update and maintenance menus and then revise the design to fix the problems that are detected in the prototype.
3. implement the new menu design.

According to the cross referencing these activities are GMI.DC.1 through GMI.DC.3 where DC is as in the DMI.

The activities that belong to specifying, designing and implementing a hardcopy map generator (HCPY) include:

1. specify the information that has to be drawn on the map and specify the format and layout of that information.
2. conduct a study to choose a hardcopy device that is suitable for the rapid generation of the specified format.
3. design a program to generate a hardcopy map using the hardcopy device.
4. implement the hardcopy generating program.

According to the cross referencing scheme, these activities are HCPY.SDC.1 through HCPY.SDC.4.

During a congestion (i.e. multiple users asking for service simultaneously), end users can not be serviced on a FIFO basis. A FIFO policy inevitably causes long delays for the end users that have been among the last to request a service. In general, any major variance in the response time could be very annoying. This suggests that a special design is needed to service a set of requests in parallel. The activities that belong to designing and coding a control system that can service multiple end user requests in parallel (PCS) include:

1. define the elements of a state of an end user query process and design a process management model that contains process swapping strategies, process state queues, process data structures.
2. prototype a multi-tasking demo kit that simulates the process swapping on a single monitor.
3. prototype the process management design on multiple processors to evaluate the design and revise the design to account for the problems that are detected in the prototype.
4. implement the new process management design.

According to the cross referencing scheme, these activities are PCS.DC.1 through PCS.DC.4.

The activities that belong to designing the communication between the subsystems of TG (COM) include:

1. design the protocol for communication and format of the information that is passed between the end user menu interface and the multi-user process manager.
2. design the protocol for communication and format of the information that is passed between the engineer user menu interface and A1, A2, and A3.
3. design the protocol for communication and format of the information that is passed between multi-user process manager and A1, A2, and A3.

These activities are COM.D.1 through COM.D.3. For each design activity, a coding activity is needed to encode and decode these protocols and communication formats. These activities are COM.C.1 through COM.C.3.

The activities that belong to unit testing include:

1. develop a set of test cases for each component of TG that verifies the requirements and specifications of that component and a set of test procedures to carry out those tests.

2. test each component of TG and tabulate the results for analysis and analyze the test results.

3. conduct a formal review.

These activities need to be executed for each component of the system independently. If the activities are executed for a component Z, then they are cross referenced as Z.UT.1 through Z.UT.3. For instance if Z is A1 then the cross referencing codes will be A1.UT.1 through A1.UT.3.

The activities that belong to integration and integration testing of the system (IT) include

1. develop a set of test cases for integration testing of TG that verifies the requirements and specifications of TG and a test procedure to carry out those tests.

2. integrate the components of TG to build an integrated system.

3. test the integrated system and tabulate the results for analysis and analyze the test results.

4. conduct a formal review.

The cross referencing codes for these will be IT.1 through IT.4.

The unit and integration testing are parts of the alpha-test. If the test of a unit fails, then the unit has to be sent back to the developers and in some cases to the unit designers in order to fix the bug. This implies that a new set of activities have to be scheduled and further integration be delayed until the unit is successfully tested again. To reduce the work delay in the case of finding a major bug, it is preferred that the development of each unit be scheduled with a slack.

Partial Ordering of the Activities

A major limitation of the activity ordering model of many production scheduling systems is that all activity ordering constraints should be expressed through the binary operator "before". This limitation prohibits the user from specifying the situations in which two activities can be executed partly concurrent. For instance, it is not possible to use the binary relation "before" to specify that a coding activity can begin as soon as two thirds of the design is completed.

In contrast to the production scheduling systems in which all activity ordering constraints should be expressed through the binary operator "before", NEGOPRO provides a ternary version of "after" to allow the specification of concurrency. In NEGOPRO, *after(A1,A2,i)* specifies that *A1* should complete *i* days after *A2* starts at the latest. From a production dependency perspective, this means that the products of *A1* are used in *A2* but that they are needed only after the execution of *A2* has already begun. The partial ordering of the activities in TG is provided in appendix B.

Staff Classification

In NegoTech, a staff member is one of the following: *team member, team leader, project manager.* A staff member can assume only one of these roles at any given time. These roles are cross referenced by TM, TL, and PM. The staff is also divided along the line of experience to senior and junior such that *senior* can be cross referenced by S and junior can be cross referenced by J. Last but not the least, the staff is divided along the line of expertise. A division based on life cycle phases produces four categories of expertise: *specification specialist, system analyst/ system designer, coder, quality assurance specialist.* These categories are cross referenced as R1, R2, R3, and R4. A senior coding team leader is referenced as TL.R3.J. A junior member of a design team is referenced as R2.J.

Only a senior staff can assume the role leading a team. The leadership of a team involves controlling,

replanning, and monitoring of the team work. Carrying the leadership activities of a team takes up to half of the time of a senior staff. Thus a senior staff can spend only half of his time on the development.

Those lines of expertise that are not based on a lifecycle phase include database specialist, database generator specialist, language specialist, graphic and menu design specialist, and multi-tasking specialist. These expertise are referenced by *DB*, *DBG*, *LG*, *GD*, and *MT*. The cross referencing codes imply that support staff, secretaries, office space, and so forth will not be accounted for in the plan as was the case in the real world TG project plan.

It is possible to assign two roles to a staff member if he possesses the required expertise for both roles and the interests of the project dictates so. It is also possible to have a staff member to work on multiple projects or multiple activities; In fact, it is preferred that a staff member work on concurrently executable activities at the same time because their works often overlaps.

According to the organizational regulations, a rule of thumb to compare the productivity of the junior and senior staff. The rule of thumb says that the productivity of a senior staff is three times the productivity of a junior staff but one third of the time of a senior staff has to be spent on speaking to a junior staff if he is to work with a junior staff.

Initially, the manager of the project had requested that the manpower requirements of each activity be expressed only in terms of senior staff but a more closer look showed that junior and senior staff should not be considered replacable under all circumstances. For instance, the activities that are critical during each phase should be strictly assigned to senior staff and it should be required that each junior staff should always work along and be supervised by a senior staff. As a result of this observation, the project manager decided that manpower requirement specification of each activity should list all possible combinations of junior/senior staff that can carry out that activity.

Activity Resource Requirements

Activities are referenced by their cross reference name. The resource requirements of an activity is specified through a sequence of request pairs such that each request pair consists of the cross referenced name of a requested resource and a pair of numbers. The first element of the pair of numbers denotes the requested quantity and the second element denotes the requested duration. If two request pairs are sequenced then the second request pair is to start after the first request is completed. If two sets of pairs are separated by an *or*, then those two sets are substitutable. A request pair can be augmented by a compromise pair. A compromise pair specifies the reduction in the cost if a requirement that is referenced in the compromise pair is relaxed. The only type of *switch plan* is specified in the request lists is when RBASE (a data base application generator) is available. In general, the introduction of RBASE greatly reduces the development of a user interface or data base application but slows down the integration of those applications with modules that have been developed in a programming language. If RBASE is not used, then the alternative is to develop the entire system inhouse. A glance through the resource requirement lists shows that whenever the execution of subactivities of an activity can be carried out concurrently, the assignment of more heads will reduce the development time. The effect of assigning more resource to an activity on the development time of that activity will be marginal if the subactivities of an activity are largely sequential. All requests are assumed to have been expressed in terms of senior staff. The time to develop the coding document is always included in the coding activity. A rule of thumb is that 10% of the time to code a module goes into documenting that code. Thus if the project plan decides to relax the requirement on documenting the code, the duration of development will be reduced by 10%. Activity

resource requirements of TG is outlined in appendix C. Since all resource requests are stated in terms of senior staff, a possible area of tradeoff is related to balancing the number of senior and junior staff.

Resource Configuration

We describe the resource configuration by first specifying the resources that are configured throughout the organization and then specifying the allocation plan of these resources to other projects that are already undertaken by NegoTech.

After negotiating with other project managers and the senior management, NegoTech agrees to assign 5 teams to work on TG on a full time basis. Among these 5, two are coding teams, two are design teams, and the remaining team can handle both specification and quality assurance. In addition, one other experienced coder will be available on a full time basis, two more experienced coders will be available on a half time basis, and a experienced designer who is presently assigned to another project will become available all in 3 months. One senior staff who is experienced in both specification and testing but is currently working on a low priority assignment can be asked for help on a half time basis at any time as well.

Each team has a team leader. The first coding team has only two members: a team leader and a regular member. The second team has three members: a team leader and two regular team members. One member of each coding team is a junior programmer while all other coders are experienced coders. Both design teams have three members and only one member of each team is a junior designer. The specification/quality assurance team has four members. The first two members of the specification/quality assurance team are experienced in quality assurance but unexperienced in testing. The other two members of the specification/quality assurance team are experienced in testing but unexperienced in specification/quality assurance. The high percentage of junior staff in the organization is the result of the fact that NegoTech has recently expanded and hired a number of junior staff. Organizational policies require that all members of a team work on the same activity if such choice exists during the resource allocation.

Organizational policies require that each junior staff has to be assigned to and work with exactly one senior staff. Furthermore, organizational policies require that a senior staff can work with at most three junior staff.

Neither of the coding teams has had any prior experience with RBASE but the strong background of the second coding team in data base applications makes its experienced members a good potential for becoming junior RBASE application developers. To use the RBASE, NegoTech basically needs to hire at least one experienced RBASE application developer.

The management of NegoTech has declared the staff budget of TG as fixed and is unlikely to change this commitment unless it would be in the benefit of the organization to temporarily reallocate one or two more coders or designers from another project to NEGOPRO. The management also has required that the budget for hiring new staff, paying consulting fees, and paying for new software licenses to help the development of TG should not exceed $100,000.

The fact that other resources such as workstations, time-share computers, other software beside RBASE, and offices are not encoded implies that either they are less important or there is an excess supply of them.

Other attributes of a staff member such as his personal interests, his experience with the tools and so forth were taken into account during the real world planning of TG but these attributes played a secondary role in the analysis and we decided not to include them in the experiment.

Resource Acquisition Cost

The cost of acquiring each incremental unit of a resource for different spans of time beyond the level that is available within the organization is specified in the cost table of that resource. The rows of the table denote incremental units and columns denote duration in month. The cost tables of some resources are depicted in appendix D. Cost of the resources in appendix D does not vary with multiple units because all resources are of the type human. However, the cost would have changed with multiple units if we were considering a hardware resource such as a workstation. The initial cost of purchasing sufficient number of RBASE licenses for use in the organization will be $10,000. Once RBASE is purchased, the cost of maintaining it will be less than $500/month.

Appendix B

Ordering of Activities in TG

In this appendix, a partial ordering of all project activities within each unit of TG that has a unique name for cross referencing is described through the ternary relation "after":

1. partial ordering of the specification activities:

   after(SP1,SP3,0), after(SP1,SP7,0), after(SP1,SP6,0),
   after(SP1,SP9,0), after(SP1,SP12,0), after(SP2,SP3,0),
   after(SP3,SP4,0), after(SP4,SP5,0), after(SP7,SP8,0),
   after(SP9,SP10,0), and *after(SP10,SP11,0)*.

2. partial ordering of the activities within A3:

   after(A3.D.1,A3.D.2,0), after(A3.D.1,A3.D.3,0),
   after(A3.D.3,A3.D.2,0), after(A3.D.3,A3.D.4,0),
   after(A3.D.3,A3.D.5,0), after(A3.C.1,A3.C.5,0),
   after(A3.C.2,A3.C.5,0), after(A3.C.3,A3.C.4,0).

3. partial ordering of the activities within A2:

   $\forall i \in [1,3]$ *after(A2.DC.i,A2.DC.(i+1),0),*
   after(A2.DC.4,A2.DC.5.,8).

4. partial ordering of the activities within A1:

   $\forall i \in [1,3]$ *after(A1.DC.i,A1.DC.(i+1),0),*
   after(A1.DC.4,A1.DC.5.,6).

5. partial ordering of the activities within HCPY:

   $\forall i \in [1,2]$ *after(HCPY.SDC.i,HCPY.SDC.(i+1),0),*
   after(HCPY.DC.4,HCPY.DC.5.,6).

6. partial ordering of the activities within PCS:

   $\forall i \in [1,3]$ *after(PCS.DC.i,PCS.DC.(i+1),0)* .

7. partial ordering of the activities within DMI:

   after(DMI.DC.1,DMI.DC.2,0), after(DMI.DC.1,DMI.DC.5,0),
   after(DMI.DC.1,DMI.DC.8,0), after(DMI.DC.1,DMI.DC.11,0),
   after(DMI.DC.2,DMI.DC.3,0), after(DMI.DC.3,DMI.DC.4,0),
   after(DMI.DC.5,DMI.DC.6,0), after(DMI.DC.6,DMI.DC.7,0),
   after(DMI.DC.8,DMI.DC.9,0), after(DMI.DC.9,DMI.DC.10,0),
   after(DMI.DC.11,DMI.DC.12,0), after(DMI.DC.12,DMI.DC.13,0),
   after(DMI.DC.4,DMI.DC.14,0), after(DMI.DC.7,DMI.DC.14,0),
   after(DMI.DC.10,DMI.DC.14,0), after(DMI.DC.13,DMI.DC.14,0),
   after(DMI.DC.14,DMI.DC.15,3), and
   after(DMI.DC.15,DMI.DC.16,6).

8. partial ordering of the activities within GMI:

   $\forall i \in [1,2]$ *after(GMI.DC.i,GMI.DC.(i+1),0)* .

9. partial ordering of the activities within COM:

after(COM.D.1,COM.C.1,0), after(COM.D.2,COM.C.2,0),
and *after(COM.D.3,COM.C.3,0).*

10. partial ordering of the unit testing activities:

$\forall i \in [1,2]$ *after(A1.UT.i,A1.UT.(i+1),0),*
$\forall i \in [1,2]$ *after(A2.UT.i,A2.UT.(i+1),0),*
$\forall i \in [1,2]$ *after(A3.UT.i,A3.UT.(i+1),0),*
$\forall i \in [1,2]$ *after(DMI.UT.i,DMI.UT.(i+1),0),*
$\forall i \in [1,2]$ *after(GMI.UT.i,GMI.UT.(i+1),0),*
$\forall i \in [1,2]$ *after(PCS.UT.i,PCS.UT.(i+1),0),*
$\forall i \in [1,2]$ *after(CPY.UT.i,CPY.UT.(i+1),0),*
$\forall i \in [1,2]$ *after(HCPY.UT.i,HCPY.UT.(i+1),0)* .

11. partial ordering of the integration testing activities:

$\forall i \in [1,3]$ *after(IT.i,IT.(i+1),0)* .

A partial ordering of all project activities between the units of TG that have a unique name for cross referencing is also described through the ternary relation "after":

after(SP5,COM.D.1,0), after(SP5,DMI.DC.1,0),
after(SP5,CPY.SDC.1,0),
after(SP6,DMI.DC.1,0), after(SP6,CPY.SDC.1,0),
after(SP7,DMI.DC.1,0), after(SP7,CPY.SDC.1,0),
after(SP7,A1.DC.1,0), after(SP7,A2.DC.1,0),
after(SP8,COM.D.2,0), after(SP8,COM.D.3,0),
after(SP10,GMI.DC.1,0),
after(SP11,GMI.DC.1,0),
after(SP12,A3.D.1,0),

after(COM.D.1,PCS.DC.1,5), after(COM.D.2,A1.DC.1,3),
after(COM.D.3,PCS.DC.1,3), after(A3.D.3,A3.C.1,0),
after(A3.D.5,A3.C.4,3), after(A3.D.5,A3.C.3,5),
after(A3.D.2,A3.C.3,5), after(A3.D.4,A3.C.5,5),
after(A3.C.4,A3.UT.1,0), after(A3.C.5,A3.UT.1,0),
$\forall X \in$ {*A1,A2,PCS,CPY,GMI,DMI*} `let i be the max index`
`in indices(X)`
`then` *after(X.DC.i,X.UT.*1,1), and

$\forall X \in$ {*A1,A2,A3,PCS,CPY,GMI,DMI*} *after(X.UT.*3,*IT.*1,2).

Appendix C

Activity Resource Requirements of TG

SP1: (R1.S (1 4))
SP2: (R1.S (1 4))
SP3: (R1.S (1 6))
SP4: (R1.S (1 4))
SP5: (R1.J (1 7)) (R1.S (1 10))
SP6: (R1.J (1 6)) (R1.S (1 7))
SP7: (R1.S (1 6))
SP8: (R1.S (1 4))
SP9: (R1.S (1 6))
SP10:(R1.S (1 5))
SP11:(R1.J (1 4)) (R1.S (1 5))
SP12:(R1.J (1 6)) (R1.S (1 7))
A3.D.1: (R2.S (1 8)) or (R2.S (2 5))
A3.D.2: (R2.S (1 20)) (f1 ((.2 (R2.S -2))) (.4 (R2.S -4)))) or
((R2.J (1 10) (R2.S (2 12))) (f1 ((.2 (R2.S -1))) (.4 (R2.S -2))))
A3.D.3: (R2.S (1 20)) (f1 ((.2 (R2.S -2))) (.4 (R2.S -4)))) or
((R2.J (1 10) (R2.S (2 12))) (f1 ((.2 (R2.S -1))) (.4 (R2.S -2))))
A3.D.4: (R2.J (1 8)) (R2.S (1 10)) or (R2.J (1 7)) (R2.S (2 7))
A3.D.5: (R2.J (1 8)) (R2.S (1 10)) or (R2.J (1 7)) (R2.S (2 7))

A3.C.1: (R3.J (1 8)) (R3.S (1 8)) or (R3.J (2 5)) (R3.S (2 5))
A3.C.2: (R3.S (1 10)) or (R3.S (2 7))
A3.C.3: (R3.J (1 20)) (R3.S (1 25)) (f1 ((.2 (R2.S -3))) (.4 (R2.S -5)))) or
(R3.J (1 15)) (R3.S (2 15)) (f1 ((.2 (R2.S -2))))
A3.C.4: (R3.J (1 15)) (R3.S (1 15)) (f1 ((.2 (R2.S -2))) (.4 (R2.S -3))))
(f2 ((.2 (R2.S -1)))) or (R3.J (2 10)) (R3.S (2 10))
A3.C.5: (R3.J (1 25)) (R3.S (1 25)) or (R3.J (2 12)) (R3.S (2 14))

DMI.DC.1: (R2.J (1 8)) (R2.S (1 10))
DMI.DC.2: (R2.S (1 5))
DMI.DC.3: (R3.S (2 4)) (R2.S (1 2))
if RBASE then (RB (1 3))
DMI.DC.4: (R3.S (1 5))
if RBASE then (RB (1 2))
DMI.DC.5: (R2.J (1 5)) (R2.S (1 5))
DMI.DC.6: (R3.S (2 6)) (R2.S (1 3))
if RBASE then (R3.S (1 4)) (RB (1 4))
DMI.DC.7: (R3.J (1 5)) (R3.S (1 6))
if RBASE then (RB (1 3))
DMI.DC.8: (R2.S (1 5))
DMI.DC.9: (R3.S (2 6)) (R2.S (1 3))
if RBASE then (R3.S (1 4)) (RB (1 4))
DMI.DC.10: (R3.S (1 6))
if RBASE then (R3.S (1 4)) (RB (1 3))

DMI.DC.11: (R2.S (1 5))
DMI.DC.12: (R3.S (2 8)) (R2.S (1 3))
if RBASE then (R3.S (1 4)) (RB (1 5))
DMI.DC.13: (R3.J (1 5)) (R3.S (1 6))
if RBASE then (R3.S (1 4)) (RB (1 4))
DMI.DC.14.(R2.S (1 5))
DMI.DC.15:(R3.J (1 8)) (R3.S (2 10)) (R2.S (1 3))
DMI.DC.16:((R3.S (1 40)) or (R3.S (2 25)))
if RBASE then (R3.S (1 10)) (RB (1 10))

GMI.DC.1: (R2.S (1 10))
GMI.DC.2: (R3.S (1 10)) (R2.S (1 2))
GMI.DC.3: ((R3.S (1 40)) or ((R3.S (1 5)) (R3.S (2 20)) (R3.S (1 2))))
if RBASE then (R3.S (2 10)) (RB (1 10))

HCPY.SDC.1: (R2.S (1 5))
HCPY.SDC.2: (R2.S (1 5))
HCPY.SDC.3: (R2.S (1 5))
HCPY.SDC.4: ((R3.S (1 20)) or (R3.J (1 10)) (R3.S (2 10)))

A2.DC.1: (R2.J (1 15)) (R2.S (1 15))
A2.DC.2: (R2.S (1 10))
A2.DC.3: (R2.S (1 5))
A2.DC.4: (R3.S (1 15)) (R2.S (1 3))
A2.DC.5: (R3.J (1 30)) ((R3.S (1 30)) or (R3.J (2 15)) ((R3.S (2 15)) (R3.S (1 4))))
if RBASE then (R3.S (1 10)) (RB (1 10))

A1.DC.1: (R2.S (1 10))
A1.DC.2: (R2.S (1 7))
A1.DC.3: (R2.S (1 5))
A1.DC.4: (R3.S (1 12)) (R2.S (1 2))
A1.DC.5: ((R3.S (1 20)) or (R3.S (2 12))) (R3.S (1 3)))
if RBASE then (R3.S (1 10)) (RB (1 10))

PCS.DC.1:(R2.J (2 15)) (R2.S (1 20)) or (R2.S (2 12))
PCS.DC.2:(R2.J (2 20)) (R3.S (1 25)) or (R3.S (2 15))
PCS.DC.3:(R3.J (1 20)) (R3.S (1 20)) (R2.S (1 3)) or
(R3.J (1 15)) (R3.S (2 15)) (R2.S (1 3))
PCS.DC.4:(R3.J (1 25)) ((R3.S (1 30)) or (R3.J (2 18))) (R3.S (2 20))

COM.D.1: (R2.S (1 4))
COM.D.2: (R2.S (1 4))
COM.D.3: (R2.S (1 4))

COM.C.1: (R3.S (1 4))
COM.C.2: (R3.S (1 4))
COM.C.3: (R3.S (1 4))

A1.UT.1: (R4.J (1 3)) (R4.S (1 3))
A1.UT.2: (R4.J (1 3)) (R4.S (1 4))
A1.UT.3: (R4.J (1 3)) (R4.S (1 4))

A2.UT.1: (R4.J (1 3)) (R4.S (1 4))
A2.UT.2: (R4.J (1 3)) (R4.S (1 5))
A2.UT.3: (R4.J (1 3)) (R4.S (1 5))

A3.UT.1: (R4.J (1 3)) (R4.S (1 4))
A3.UT.2: (R4.J (1 3)) (R4.S (1 5))
A3.UT.3: (R4.J (1 3)) (R4.S (1 5))

DMI.UT.1:(R4.J (1 3)) (R4.S (1 4))
DMI.UT.2:(R4.J (1 3)) (R4.S (1 5))
DMI.UT.3:(R4.J (1 3)) (R4.S (1 5))

GMI.UT.1:(R4.J (1 3)) (R4.S (1 3))
GMI.UT.2:(R4.J (1 3)) (R4.S (1 4))
GMI.UT.3:(R4.J (1 3)) (R4.S (1 4))

PCS.UT.1:(R4.J (1 3)) (R4.S (1 4))
PCS.UT.2:(R4.J (1 3)) (R4.S (1 5))
PCS.UT.3:(R4.J (1 3)) (R4.S (1 5))

HCPY.UT.1:(R4.J (1 2)) (R4.S (1 2))
HCPY.UT.2:(R4.J (1 2)) (R4.S (1 2))
HCPY.UT.3:(R4.J (1 2)) (R4.S (1 2))

IT.1: (R4.J (1 10)) (R4.S (1 10))
IT.2: (R3.J (1 25)) (R3.S (1 30)) or (R3.J (1 15)) ((R3.S (2 15)) (R3.S (1 5)))
IT.3: (R4.J (1 10)) (R4 (1 10))
IT.4: (R4.S (1 20)) or ((R4.S (2 10)) (R4.S (1 5)))

Appendix D

Resource Acquisition Cost

RB:

	1	2	3	4	5	6	7	8
1	*18*	*36*	*54*	*72*	*90*	*108*	*120*	*130*
2	*18*	*36*	*54*	*72*	*90*	*108*	*120*	*130*

R1.S:

	1	2	3	4	5	6	7	8
1	*12*	*24*	*36*	*45*	*54*	*63*	*72*	*80*
2	*12*	*24*	*36*	*45*	*54*	*63*	*72*	*80*

R1.J:

	1	2	3	4	5	6	7	8
1	*8*	*16*	*24*	*32*	*38*	*44*	*50*	*55*
2	*8*	*16*	*24*	*32*	*38*	*44*	*50*	*55*

R2.S:

	1	2	3	4	5	6	7	8
1	*12*	*24*	*36*	*45*	*54*	*63*	*72*	*80*
2	*12*	*24*	*36*	*45*	*54*	*63*	*72*	*80*

R2.J:

	1	2	3	4	5	6	7	8
1	*8*	*16*	*24*	*32*	*38*	*44*	*50*	*55*
2	*8*	*16*	*24*	*32*	*38*	*44*	*50*	*55*

R3.S:

	1	2	3	4	5	6	7	8
1	*10*	*20*	*30*	*38*	*46*	*54*	*62*	*70*
2	*10*	*20*	*30*	*38*	*46*	*54*	*62*	*70*

R3.J:

	1	2	3	4	5	6	7	8
1	*7*	*14*	*20*	*26*	*30*	*34*	*38*	*42*
2	*7*	*14*	*20*	*26*	*30*	*34*	*38*	*42*

R4.S:

	1	2	3	4	5	6	7	8
1	*12*	*24*	*36*	*45*	*54*	*63*	*72*	*80*
2	*12*	*24*	*36*	*45*	*54*	*63*	*72*	*80*

R4.J:

	1	2	3	4	5	6	7	8
1	*8*	*16*	*24*	*32*	*38*	*44*	*50*	*55*
2	*8*	*16*	*24*	*32*	*38*	*44*	*50*	*55*

Appendix E
Construction of A Seed Schedule

This appendix shows how a seed schedule is constructed through incremental application of linear programming to initialize the start time and resource allocations of every project activity:

let $\{A_i\}$ be an enumeration of all activities to be executed.
assume that no activity can be suspended once it starts.

let A_0 be a *starting dummy activity* and A_N a final dummy activity.
assume that the length of a dummy activity is zero.

let $PR=\{(A_k\ A_l)\}$ be the set of all pairs s.t.
A_l is allowed to start only after A_k has completed.

$\forall$ activities A_j
let s_j denote the start time of activity j,
d_j denote the duration of activity j - this duration is
broken down to a sequence of m intervals of unit length
numbered from 1 to m.

let $h_{jr}:N \rightarrow N$ denote the quantity of resource r that A_j requires
during its n-th interval $1 \leq n \leq d_j$

$$h_{jr}(n)=\begin{cases} 0 & \forall n \leq 0 \\ q & \forall 1 \leq n \leq d_j \\ 0 & \forall n > d_j \end{cases}$$

let $f_r:N \rightarrow N$ specify the quantity of r available during an interval,
D(f) indexes the interval and R(f) specifies
the available quantity s.t.

$$f_r(n)=\begin{cases} 0 & \forall n \leq 0 \\ q & \forall 1 \leq n \leq max\text{–}allowable\text{–}duration \\ 0 & \forall n > max\text{–}allowable\text{–}duration \end{cases}$$

let $g_r:N \rightarrow N$ specify the quantity of r required by the entire
project during an interval. the domain indexes the interval and
the range specifies the quantity. g can be computed through the
following formula:

$$g_r(i) = \sum_{\forall j \in activity\text{–}indices} h_j(i\text{-}s_j)$$

The problem of scheduling for the above set of definitions is to compute the set $\{s_i\}$ for all activities A_i s.t. the *following sets of constraints are satisfied:*

$s_i+d_i < s_j \; \forall \; (A_i \; A_j) \in PR$
$g_r(i) \leq f_r(i) \; \forall$ intervals $i + x$ s.t.
x is the level by which $f_r(i)$ is incremented

and the goal will be to minimize the following function:
$Min(v(s_N-(s_0+d_0)) + w(x))$
where v denotes the penalty of a long duration and
w denotes the penalty of a large increase in availability

The linear programming subroutine will return a value u_i for every constraint i. u_i would denote the importance of constraint i in forming the solution. For instance, if u_i is 27 for the second type of constraints, then by adding 1 to x, i will contribute to a 27 unit drop in the quantity we are trying to minimize. If a constraint does not contribute to the solution, i.e. has a small u_i, then we could rewrite it using h_r of another resource. If there is path from the beginning to end that none of the activities on it use r and that path is not critical, then the u value of every activity on that path will be very small. This would allow us to rewrite those constraints *in terms of the* next most important resource.

If a linear programming search fails to produce a $\{s_i\}$ which satisfies every specified constraint, then

$\exists$ i s.t. $f_r(i) < g_r(i)$

The number of such i's is a function of f_r. If f_r is increased evenly to a certain level, all such i's will be eliminated. By increasing f_r evenly, we also encourage the linear programming function to smooth the function

$e_r(i) = g_r(i) - f_r(i)$

The second type of constraints play a dominant factor in complicating the linear programming problem. For instance, if the duration of project is 1 year and the unit interval length is 15 days, then 104 inequalities will result. Although theoretically each inequality could incorporate all of the project activities, in practice the number of activities that participate in the inequality is much smaller. This is due to three factors:

1. few activities last during the entire project
2. few activities need r during their entire duration
3. few activities can execute in parallel i.e. be on the same inequality

The following algorithm (P is the final product) will show how the optimal level by which the availability of r (f_r) has to be increased or decreased:

function availability-fix(P)
repeat
let *c1 = p-supply-cost(P)*
let $f_r(i)=f_r(i)+1 \;\; \forall i \leq \frac{allowed\text{-}duration(P)}{unitinterval}$
let *c2=p-supply-cost(P)*
let *flat-increase=supply-cost(next-unit-interval)*
until *c1 < c2+flat*

Appendix F
Global Benefit of A Scheduling Commitment

Let A (active set) denote the set of products that the propagation reaches after each propagation, D (deleted set) denote the set of products that are deleted from A but maintained as inactive, *marked(p)* be a predicate defined on the elements of A which is *TRUE* if p is the product of another element in A and *FALSE* otherwise, T_p denote all possible alternative sets of product feature requirements of a product p that can be met by a set of reservations, *responsible-for(p)* be a function which returns the name of the agent who is responsible for p, *authorized(a,p)* be a predicate which is *TRUE* if and only if a is responsible for p, R_p be the set of all resources required by p, and $q \in$ *product–transitive–closure+*(r) iff $(q \in$ *product–transitive–closure*$(r) \wedge q \neq r)$. The following algorithm measures *global benefit* of a scheduling commitment with respect to a product p if all other scheduling commitments are fixed:

$D=\emptyset$
$A=\{p\}$
$marked(p) = FALSE$
L1:
{add the immediate products that are not already in A to A}
$\forall r \in A \;\; s.t. \;\; \neg marked(r)$
$\forall q \;\; r \in R_q \;\; s.t. \;\; q \notin (A \cup D)$
$A=A \cup \{q\}$
{mark all non-primitive products}
$\forall q \in A \;\; s.t. \;\; \neg marked(q)$
if $\exists r \in A \;\; s.t. \;\; q \in product\text{–}transitive\text{–}closure(r)$ then $marked(q)=TRUE$
{if R_x contains a resource other than q}
$\forall q \in A \;\; s.t. \;\; \neg marked(q) \;\; \forall x \;\; q \in R_x$
if $\neg \exists y \;\; (y \in A \;\wedge\; x \in product\text{–}transitive\text{–}closure^{+}(y))$
{x is only an immediate product of the resources in A}
then
if $(\sum B(x,\zeta_z) \;\; \forall z \;\; s.t. \;\; z \in R_x \;\wedge\; z \in A)=1$ then
{there is no benefit lost}
$A=A-\{x\}$
$D = D \cup \{x\}$
if $\forall w \;\; (q \in R_w \;\wedge\; w \notin D) \;\rightarrow\; w=x$ then $A=A-\{q\}$
else if
$(\forall z \in A \;\; (z \in R_x \;\vee\; z=x) \;\wedge\; authorized(responsible\text{–}for(x),x)) \;\vee$
$((1 - \sum B(x,\zeta_z) \;\; \forall z \;\; s.t. \;\; z \in R_x \;\wedge\; z \in A) \;<\; \varepsilon)$ then
{agent responsible for x has ruling authority or the lost benefit is insignificant}
$return(B(x,\zeta_a) \;\; a \in T_x \;\; s.t. \;\; \forall b \in T_x \;\; B(x,\zeta_b) < B(x,\zeta_a))$
else
{propagate the feature requirements}
$construct \; T_x$
$\forall z \in R_x$
if $z \in A \;\wedge\; \neg marked(z) \;\wedge\; \forall w \;\; (z \in R_w \;\wedge\; w \notin D) \;\rightarrow\; w=x$
then
$A=A-\{z\}$ {drop z from A}
$D=D \cup \{z\}$
endif
$marked(x) = FALSE$
endif
endif
goto *L1*

References

1. J.F. Allen. "Towards A General Theory of action and Time". *Artificial Intellignece 23*, 2 (1984).

2. Ardayfio. Expert Systems in Manufacturing Automation. In S.C.Y. Lu and R. Komanduri, Ed., *Knowledge-Based Expert Systems for Manufacturing*, American Society of Mechanical Engineers, 1986, pp. 25-39.

3. R. Banker, S. Datar, S. Kekre. "Relevant Costs, Congestion and Stochasticity in Production Environments". *Journal of Accounting and Economics* (1987).

4. K.M. Bartol. "Professionalism as a predictor of Org'l Commitment, Role, Stress, and Turnover: A Multidimensional Approach". *Academy of Mgmt Journal* , 4 (Dec 1979), 815-821.

5. K.M. Bartol and C.D. Martin. "Managing Information Systems Personnel: A Review of Mangeerial Implications". *MIS Quarterly* (Dec 1982), 49-70.

6. V. Basili, R. Selby, D. Hutchens. "Experimentation in Software Engineering". *IEEE Transactions on Software Engineering SE-12*, 7 (July 1986), 733-743.

7. Len Bass and Joelle Coutaz. Human-Machine Interacton Considerations for Interactive Software. Software Engineering Institute, Feb., 1989.

8. Robert Balzer, Thomas E. Cheatham, and Cordell Green. "Software Technology in the 1990's: Using a New Paradigm". *Computer Magazine* (1983).

9. Kent Bimson and Linda Burris. "A Knowledge Based Approach to Software Project Management". *IEEE Expert* , 2 (1989).

10. Robert W. Blanning. Expert Systems As an Organizational Paradigm. Proceedings of the Eight International Conference on Information Systems, Dec, 1987, pp. 232-240.

11. W.E. Boebert. Software Quality Through Software Management. In Cooper and Fisher, Ed., *Software Quality management*, Petrocelli, 1979.

12. B.W. Boehm. *Software Engineering Economics.* Prentice Hall, Englewood Cliffs, N.J., 1981.

13. B.W. Boehm. *Software Risk Management Tutorial.* TRW, 1988.

14. A. Borning, R. Duisberg, B. Freeman, A. Kramer, M. Woolf. A Planning/Scheduling Methodology for the Constrained Resource Problem. Proceedings of OOPSLA 87, 1987.

15. J. Bresina. An Interactive Planner that creates a Structured, Annotated Trace of its Operation. Computer Science Dept, Rutgers University, Dec., 1981.

16. F.P. Brooks. *The Mythical Man-Month.* Addison Wesley Publishing Co, 1978.

17. G. Bruno, A. Elia. "A ruled Based System To Schedule Production". *IEEE Computer* (1986), 32-40.

18. Jean-Lou Chameau and Juan Charlos Santamarina. "Membership Functions I: Comparing Methods of Measurement". *Artificial Intelligence* , 3 (July 1987), 287-302.

19. Philip Agre, David Chapman. Pengi: An Implementation of a Theory of Activity. In Proceedings of American Association of Artificial Intelligence, Seattle, Washington, 1987, pp. 268-272.

20. Cheatham, T.E. Supporting the Software Process. Proceedings of Ninetheen Annual Hawaii International Coference On System Science, 1986.

21. S. Conry, R. Meyer, and V. R. Lesser. Multistage Negotiation in Distributed Planning. In A.H. Bond and L. Gasser, Ed., *Readings in Distributed Artificial Intelligence*, Morgan Kaufman, 1988.

22. S. Cook. The Complexity of Theorem Proving Procedures. In Proeceedings of the 3rd Annual ACM Symposium on Theory of Computing, ACM, 1971.

23. L. Cooper. *Managing Program Risk: One Way to Reduce Cost Growth.* Proceedings of 1983 Federal Acquisition Research Symposium, U.S. Naval Academy, December, 1983.

24. E.B. Daly. Organizational Philosophies used in Software Development. In R. Goldberg and H. Lorin, Ed., *The Economics of Information Processing*, John Wiley & Sons, 1982.

25. R. Davis and R. Smith. Negotiation as a Metaphor for Distributed Problem Solving. MIT Artificial Intelligence Lab, May, 1981.

26. T. Dean. Decision Support for Coordinated Multi-Agent Planning. Proceeding of the Third International ACM Conference on Office Information Systems, New York, 1986.

27. R. Dechter and J. Pearl. "Tree Clustering for Constraint Networks". *Journal of Artificial Intelligence* , 38 (1989), 353-365.

28. Y. Descotte, J.C. Latombe. "Making Compromises Among Antagonistic Constraints in a Planner". *Journal of Artificial Intelligence* , 27 (1985), 183-217.

29. Rina Detcher and Judea Pearl. Tree-Clustering Schemes for Constraint-Processing. Proceedings of American Associate of Artificial Intelligence, 1988, pp. 150-155.

30. Vasant Dhar and Nicky Ranganathan. "Integer Programming vs. Expert Systems: An Experimental Comparison". *Journal of ACM* (1990), 323-337.

31. J. Distaso. "Software Management - A survey of Practice". *Proceedings of IEEE 68* (1980), 1103-1119.

32. J. Doyle. "A Truth Maintenance System". *Journal of Artificial Intelligence 3*, 12 (1979), 231-271.

33. S. E. Drefus. *Formal Models vs. Human Situational Understanding: Inherent limitations on the Modelling of Expertise* . University of California Berkeley, rep ORC 81, 1981.

34. M. Drummond. A Represenation of Action and Belief for Autmoatic Planning Systems. In Proceedings of American Association of Artificial Intelligence Workshop on Planning and Action, 1986.

35. E. Durfee, V. Lesser, and D. Corkill. Coherent Cooperation Among Communicating Problems Solvers. Computer Science Dept, University of Massachusetts at Amherst , sept, 1985.

36. Lee D. Erman, Fredrick Hayes-Roth, Victor R. Lesser and D. Raj Reddy. "The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty". *Computing Surveys 12*, 2 (June 1980), 213-253.

37. W.F. Fabrycky, Ghare, and Torgersen. *Applied Operations Research and Management Science.* Prentice Hall, NJ, 1984.

38. William S. Faught, IntelliCorp. "Applications of AI in Engineering". *IEEE Computer* , 7 (July 1986), 17-27.

39. R. Fikes, P. Hart, and N. Nilsson. *Learning and Executing Generalized Robot Plans.* Tiago Publishing, Palo Alto, California, 1981.

40. Mark Fox. Organization Structuring: Designing Large Complex Software. Tech. Rept. CMU-CS-79-155, Department of Computer Science, Carnegie Mellon University, 1979.

41. Mark Fox. *Constraint-Directed Search: A case study of job-shop scheduling.* Ph.D. Th., Carnegie-Mellon University, Pittsburgh, 1983.

42. Mark Fox and Stephen F. Smith. "ISIS - A Knowledge Based System for Factory Scheduling". *Expert Systems 1*, 1 (July 1984).

43. M. Fox, N. Sadeh, and C. Baykan. Constrained Heuristic Search. International Joint Conference on Artificial Intelligence, Detroit, 1989.

44. M. Fox and E.K. Sycara. An Overivew of CORTES. Center For Integrated Manufacturing, Carnegie Mellon University, 1990.

45. E. C. Freuder. Partial Constraint Satisfaction. International Joint Conference on Artificial Intelligence, Detroit, 1989, pp. 28-285.

46. G.D. Frewin. SPMS A Support for the Management Process. Proceedings of Software Process Workshop, 1984.

47. Les Gasser. *The Social Dynamics of Routine Computer Use in Complex Organizations.* Ph.D. Th., University of Southern California, Feb 1984.

48. Les Gasser. "The Integration of Computing and Routine Work". *ACM Transactions on Office Information Systems* (July 1986), 205-225.

49. Gehring. *A Quantitative Analysis of Estimation Accuracy in Software Development.* Ph.D. Th., Texas A&M, 1976.

50. M. Georgeff and A. Lansky. Reactive Reasoning and Planning. Proceedings of American Association of Artificial Intelligence, Seattle, 1987, pp. 677-682.

51. Ira Goldstein. Bargaining Between Goals. Proceedings of International Joint Conference in Artificial Intelligence, 1985.

52. L.H. Green. Organizing for Project management. In J. Hannan, Ed., *Systems Development Management*, Auerbach Publishers, New Jersey, 1982.

53. M. Grool, G. Wijnen, C. Visser, and W. Vriethoff. *Project Management In Progress: Tools & Strategies for the Nineties.* North Holland, 1986.

54. Abdel-Hamid Tarek. *Project Management Modeling.* Ph.D. Th., MIT, School of Business Management, 1984.

55. Abdel-Hamid Tarek, and Stuart Madnick. "Impact of Schedule Estimation on Software Project Behaviour". *IEEE Software* (1986), 70-75.

56. R. Haralick and G. Elliot. "Increasing Tree Search Efficiency for Constraint Satisfaction Problems". *Journal of Artificial Intelligence* , 14 (1980), 263-313.

57. Robert Hink. "How Humans Process Uncertain Knowledge: An Introduction". *AI Magazine* , 3 (Fall 1987), 41-54.

58. W. Hsu, M. Preitula, D. Steier. Merl-Soar: Scheduling within a general architecture for intelligence. Proceedings of the Third International Conference on Expert Systems and the Leading Edge in Production and Operations Management, South Carolina, May, 1989.

59. W. Humphrey. The Software Engineering Process, Definition and Scope . In Proceedings of 4th International Process Workshop, England , 1988.

60. Ibaraki and Katoh. *Resource Allocation Problems.* MIT Press, 1988.

61. R.H. Irving, C.A. Higgins, and F.R. Safayeni. "Computerized Performance Monitoring Systems: Use and Abuse". *Communications of ACM 28*, 8 (August 1986), 794-801.

62. M. Garey and D. Johnson. *Computers and Interactibility.* Freeman Publisher, 1979.

63. David Johnson. "The NP-Completeness Column: An Ongoing Guide". *Journal of Algorithms* , 4 (1983), 189-205.

64. Richard Jullig, Wolfgang Polak, Peter Ladkin and Li-Mei Gilham. KBSA Project Management Assistant. Tech. Rept. RADC-TR-87-78, Kestrel Institute, 1987.

65. P. Kalhr and W.S. Fought. Knowledge Based Simulation. Proceedings of First American Association of Artificial Intelligence Conference, Stanford, Calif, 1980, pp. 181-183.

66. U. Karmakar, S. Kekre, S. Freeman. "Lot Sizing in Multi-machine Job Shops". *I.I.E. Transactions* (1985).

67. Karnaugh, Maurice. *Computer Aided Planning: Interactive Control of FAME*. IBM Watson Research Center, Yorktown Heights, 1988.

68. R.L. Keeney, H. Raiffa. *Decisions with Multiple Objectives*. John Wiley and Sons, 1975.

69. Marc Kellner. Representation Formalisms for Software Process Modeling. In Proceedings of the 4th International Software Process Workshop, Hawaii, Mar, 1988.

70. J. Kelly. "Critical Path Planning and Scheduling". *Operations Research 9*, 3 (May-June 1961), 296-320.

71. N. Keng and D.Y. Yun. A Planning/Scheduling Methodology for the Constrained Resource Problem. International Joint Conference on Artificial Intelligence, Detroit, Mar, 1989.

72. H. Kerzner. *Project Managment: A systems Approach to Planning, Scheduling and Executing*. Van Nostrand Reinhold Company, 1984.

73. Kiyoshi Niwa. "A Knowledge-Based Human Computer Cooperative System for Ill-Strucutred Management Domains". *IEEE Transactions on Systems, Man and Cybernetics SMC-16*, 3 (June 1986).

74. G. A. Klein. Automated Aids For the Proficient Decision Maker. Proceedings of IEEE Conference on Cybernetics, 1980, pp. 301-304.

75. Rob Kling and Walt Scacchi. "Computing as Social Action: The social dynamics of computing in complex organizations". *Advancements in Computers* , 19 (July 1980), 249-327.

76. Knutson, J. Developing the Project plan. In *Advances in Computer Programming Management*, Heyden & Sons, Philadelphia, 1980.

77. C.C. Koo and P. Cashman. A commitment-Based Communication Language for Distributed Manufacturing. In S.C.Y. Lu and R. Komanduri, Ed., *Knowledge-Based Expert Systems for Manufacturing*, American Society of Mechanical Engineers, 1986, pp. 155-166.

78. Richard Korf. "Planning Viewed as a Search Problem". *Journal of Artificial Intelligence* (1987).

79. D. Kostetsky. A Simulation Approach to Managing Engineering Projects. Proceedings of IEEE International Conference on Robotics and Automation, New York, 1986.

80. D. Kumar. "A Novel Approach to Sequential Simulation". *IEEE Transactions on Software* (September 1986).

81. Susan Lander and Victor Lesser. A Framework for the Integration of Cooperative Knowledge-Based Systems. International Joint Conference in Artificial Intelligence Workshop on Integrated Architecture for Manufacturing, 1989.

82. Lehman, J. H. "How Software Projects are Really Managed". *Datamation* (Jan 1979), 119-129.

83. D. Lenat. "EURISKO: A Program that learns new heuristics and domain concepts. The nature of heuristics: Program design and results". *Journal of Artificial Intelligence* , 21 (1983), 61-98.

84. Mathew Liberatore, George Titus. "Management Science Practice in R&D Project Management". *Management Science 29* (August 1983).

85. Micheal K. Lillard. Towards Intelligent Tuning of Resource Allocation Plans. Master Th., Dept. of Computer Science, MIT,Sept 1987.

86. Lubbes, H.O, Naval Electronic Systems Command. "The Project Management Task Area". *IEEE Computer* (1983), 56-62.

87. A. K. Mackworth. "Consistency in Networks of Relations". *Journal of Artificial Intelligence* , 8 (1977), 99-118.

88. A. K. Mackworth and E. C. Freuder. "The Complexity of some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems". *Journal of Artificial Intelligence* , 25 (1984), 65-74.

89. Charles Marshall. Uncertainty Resolution Through Learning by Asking Questions: An Anslysis of the Engineering-to-Manufacturing Inerface. Digital Equipment Corporation, 1990.

90. J. McDermott. "A Temporal Logic for Reasoning About Processes and Plans". *Cognitive Science* , 6 (1982).

91. K. McKay, J. Buzacott, F. Safayeni. The Scheduler's Knowledge of Uncertainty: The Missing Link. Proceedings of IFIP Conference on Knowledge Based Production Management Systems, August, 1988.

92. J.D. McKeen. *An Empirical Investigation of the Process and Product of Application System Development.* Ph.D. Th., University of Minnesota, 1981.

93. Metzger, P. W. *Managing a Programming Project.* Englewood Cliffs, Prentice-Hall, 2nd Ed., 1981.

94. Milton D. Rosenau, Marsha D. Lewin. *Software Project Management: Step by Step.* Van Nos Reinhold, 1984.

95. S. Minton. *Learning Effective Search Control Knowledge: An Explanation Based Approach.* Ph.D. Th., Carnegie Mellon University, 1988.

96. Moder, Philips, and Davis. *Project Management with CPM, PERT & Precedence Diagramming.* Van Nostrand Reinhold Company, New York, 1983.

97. U. Montanari. "Networks of Constraints: Fundamental Properties and Applications to Picture Processing". *Information Sciences* (1974), 95-132.

98. T. Morton, S. Lawrence, S. Rajagopolan, K. Kekre. "SCHED-STAR: A Price based Shop Scheduling Module". *Journal of Manufactiring and Operations Management* , 1 (1988), 131-181.

99. N. Muscettola and S. F. Smith. A Probabilistic Framework for Resource-Constrainted Multi-Agent Planning. In Proceedings of 10th International Joint Conference on Artificial Intelligence, 1987.

100. B. Nadel. "Constraint Labeling Problems and Their Algorithms: Expected Complexities and Theory Based Heuristics". *Journal of Artificial Intelligence* , 21 (1983), 135-178.

101. Arch W. Naylor and Mark C. Maletz. "The Manufacturing Game: A Formal Approach To Manufacturing Software". *IEEE Transactions on Systems, Man and Cybernetics SMC-16*, 3 (June 1986).

102. J. Liard, A. Newell, and P. Rosenbloom. "Soar: An Architecture for General Intelligence". *Journal of Artificial Intelligence* , 33 (1987), 1-64.

103. Kiyoshi Niwa. "A New Project Management System Approach: The Know-how Based Project Management System". *Project Management Quarterly 14* (March 1983), 65-72.

104. P. S. Ow, S. F. Smith, and A. Thiriez. Reactive Plan Revision. Proceedings of American Association of Artificial Intelligence, 1988.

105. P.M. Pardalos, J.B. Rosen. *Lecture Notes in Computer Science - Constrained Global Optimization: Algorithms and Applications.* Springer-Verlag, 1987.

106. R. Dechter and J. Pearl. "Network Based Heuristics for Constraint Satisfaction Problems". *Journal of Artificial Intelligence* , 34 (1987), 1-38.

107. A. Peschel. Implementing Risk Management. Proceedings of NSIA Software Risk Management Conference, NSIA, September, 1987.

108. D.G. Pruitt. "Methods of Resolving Differences of interest: A theoretical analysis". *Journal of Social Issues* , 28 (1972), 133-154.

109. D.G. Pruitt. *Negotiation Behaviour.* Academic Press, 1981.

110. Paul Purdom. "Search Rearrangement Backtracking and Polynomial Average Time". *Journal of Artificial Intelligence* , 22 (1983), 117-133.

111. R. Reddy, M. Fox, M. McRoberts. "The Knowledge Based Simulation System". *IEEE Software* (March 1986), 26-40.

112. Genesereth, M.R. and Rosenschein J.S. Deals among Rational Agents. International Joint Conference in Artificial Intelligence Los Angeles, California, 1985, pp. 91-99.

113. B. Ruby and D. Kibler. Learning to Plan in Complex Domains. Proceedings of the Sixth International Workshop on Machine Intelligence, Itacha, 1989.

114. Sacerdoti. "Planning in a Hierarchy of Abstraction Spaces". *Artificial Intelligence 5*, 2 (July 1974).

115. A. Safavi and S.F. Smith. A Heuristic Search Approach to Scheduling Software Manufacturing Projects. In M. Golumbic, Ed., *Advances in Artificial Intelligence, Natural Language, and Knowledge-Based Systems*, Springer-Verlag, 1990.

116. A. Safavi and S.F. Smith. An Evaluation Function to Compare Alternative Commitments During Manufacturing Planning and Scheduling. International Conference on Expert Planning Systems, June, 1990.

117. A. Safavi. Optimization of Relaxable Constraints Subject to Dynamic Preferences. Working Paper, .

118. A. Sathi, M. Fox and Greenberg. "Representation of activity knowledge for project management". *IEEE Transactions on pattern analysis and machine intelligence 5* (1985), 531-552.

119. A. Sathi, T. Morton, and S. Roth. "Callisto: An Intelligent Project Management System". *AI Magazine* , Winter (1986).

120. A. Sathi. *Cooperation Through Constraint Directed Negotiation: Study of Resource Reallocation Problems*. Ph.D. Th., Carnegie Mellon University, 1988.

121. Walt Scacchi. "Software Engineering: A Social Analysis Study". *IEEE Transacations on Software Engineering* (Jan. 1984), 45-60.

122. E.H. Schein. *Organizational Psychology*. Englewood Cliffs, Prentice-Hall, New Jersey, 1980.

123. Joobin Choobineh and Arun Sen. A Framework Deductive Database Design in Decision Support Systems. Proceedings of the Eight International Conference on Information Systems, Dec, 1987, pp. 241-250.

124. Shneiderman. *Software Psychology - Human Factors in Computer and Information Systems*. Winthrop Inc, Cambridge, Mass, 1980.

125. Y. Shoham. *Reasoning About Change: Time and Causation from the Standpoint of Artificial Intelligence*. MIT Press, Cambridge, MA, 1988.

126. H.A. Simon. "Search and Reasoning in Problem Solving". *Journal of Artificial Intelligence* , 21 (1983), 7-29.

127. J.C. Van Slyke. "Monte Carlo Methods and the PERT Problem". *Operations Research 11*, 5 (Sept-Oct 1963), 839-860.

128. S. F. Smith, P. S. OW, J. Potvin, N. Muscettola, and D. Matthys. "An Integrated Framework for Generating and Revising Factory Schedules". *Journal of Operations Research 41*, 6 (1990), 539-552.

129. G. Spur. "Growth, Crisis, and Future of the Factory". *Robotics and Computer Integrated Manufacturing 1*, 1 (1984), 21-37.

130. Steele. "The Definition and Implementation of a Computer Programming Language Based on Constraints". *MIT tech rep. AI-TR-595, Cambridge MA* (1980).

131. M. Stefik. "Planning and Constraints (MOLGEN: Part 1)". *Artificial Intelligence 16* (1981), 111-140.

132. A. Strauss. *Negotiations: Varieties, Contexts, Processes, and Social Order*. Jossey-Bass, San Fransisco, 1978.

133. Nam P. Suh. "The Future of the Factory". *Robotics and Computer Integrated Manufacturing 1*, 1 (1984), 39-49.

134. E. P. Sycara. *Resolving Adversarial Conflicts: An Approach Integrating Case-Based and Analytical Method.* Ph.D. Th., Georgia Institute of Technology, 1987.

135. E. K. Sycara. Resolving Goal Conflicts via Negotiation. Proceedings of American Association of Artificial Intelligence, 1988, pp. 245-250.

136. Austin Tate and Ken Currie. O-Plan: The Open Planning Architecture. Computer Science Dept, University of Edinburgh, 1988.

137. R.H. Thayer. *Modeling A Software Engineering Project Mangement System.* Ph.D. Th., University of California at Santa Barbara, 1979.

138. Christopher Tong. "Toward an Engineering Science of Knowledge-Based Design". *International Journal of Artificial Intelligence* , 3 (July 1987), 133-165.

139. R. Turvey. "Peak Load Pricing". *Journal of Political Economy* (1968).

140. Steven Vere. "Planning in Time: Windows and Durations for Activities and Goals". *IEEE Transactions on Pattern Recognition and Machine Intelligence PAMI-5*, 3 (May 1983), 246-278.

141. Ajay Vinze, Mari Heltne, Minder Chen, Jay Nunamaker Jr., and Benn Konsynski. A Knowledge Based Approach For Resource Management. Proceedings of the Eight International Conference on Information Systems, Dec, 1987, pp. 351-367.

142. H. J. Warnecke. "Conflicting Objectives in Designing Market-Oriented Production Structures". *Robotics and Computer Integrated Manufacturing 1*, 1 (1984), 51-60.

143. R.L. Weil. Industrial Dynamics & MIS. In E.B. Roberts, Ed., *Managerial Applications of Systems Dynamics*, M.I.T. Press, Cambridge, Mass., 1981.

144. G.A. Whitmore, G.S. Cavadias. "Experimental Determination of Community Preferences for Water Quality Alternatives". *Decisions Sciences* (1974), 614-631.

145. David Wile. "Program Developments: Formal Explanations of Implementations". *Communications of the ACM* (Nov. 1983).

146. David Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann, 1988.

147. J. Zimmerman and M. Sovereign. *Quantitative Models for Production Management.* Englewood Cliffs, NJ, Prentice-Hall, 1974.

148. R.W. Zmud. "Management of Large Software Development Efforts". *MIS Quarterly* , 2 (1980), 45-56.