SPACE COMMUNICATIONS SCHEDULER: A RULE-BASED APPROACH TO ADAPTIVE DEADLINE SCHEDULING

Nicholas Straguzzi
GE Advanced Technology Laboratories
Moorestown, NJ 08057

ABSTRACT

Job scheduling is a deceptively complex subfield of computer science. The highly combinatorial nature of the problem, which is NP-complete in nearly all cases, requires a scheduling program to intelligently traverse an immense search tree to create the best possible schedule in a minimal amount of time. In addition, the program must continually make adjustments to the initial schedule when faced with last-minute user requests, cancellations, unexpected device failures, etc. A good scheduler must be quick, flexible, and efficient, even at the expense of generating slightly less-than-optimal schedules.

The Space Communications Scheduler (SCS) is an intelligent rule-based scheduling system developed at GE's Advanced Technology Laboratories. SCS is an adaptive deadline scheduler which allocates modular communications resources to meet an ordered set of user-specified job requests on board the NASA Space Station. SCS uses pattern-matching techniques to detect potential conflicts within a schedule, then resolves these conflicts through algorithmic and heuristic means. As a result, the system generates and maintains high-density schedules without relying heavily on backtracking or blind search techniques. SCS was designed to allocate communication devices on board the Space Station, but its general-purpose scheduling strategy is suitable for many common real-world applications.

1.0 INTRODUCTION

"Scheduling" is a term very familiar to most people. Personal schedules are routinely made and revised; it is a task so common that few people think about the cognitive actions involved in its performance. Yet, the seemingly simple act of scheduling, which can be loosely defined as allocating the resources necessary to perform a set of jobs over a specific time interval, is one of the more complex problem areas of computer science.

Two general classes of scheduling problems exist: *precedence constrained* and *deadline*. Precedence constrained scheduling (sometimes called simply constraint scheduling) is very closely related to classical computer planning problems. In its most basic form, constraint scheduling generates an agenda for performing subtasks of a specified job, given a partial ordering of the subtasks and a deadline for completing the job. Garey and Johnson (Garey and Johnson, 1979) illustrate constraint scheduling with the example of a college freshman building a four-year course plan. Because certain courses are required for graduation, and because most of these courses have prerequisites of their own, developing an appropriate schedule is, as every college student knows, a non-trivial task.

Deadline scheduling (sometimes called interval, appointment, or timetable scheduling) is a somewhat more familiar problem class. Here, the goal is to create an optimal timetable for the execution of a set of jobs over a specific interval of time, given a finite set of available resources and a set of acceptable release times (earliest start times), deadlines (latest completion times), and priorities for each job. Common real-world examples include a doctor's receptionist scheduling patient appointments and a computer's operating system scheduling the execution of batch programs. What actually constitutes an "optimal" schedule varies from application to application. In the first example, an optimal schedule would be one that allows the maximum number of appointments, while in the second, it might maximize the sum of the priorities of the executed programs.[1]

1. Many application areas fall into both scheduling classes. Engineers at a car manufacturing plant, for example, must make use of both constraint scheduling (deciding the optimal order for assembling the parts of a car) and deadline scheduling (allocating the personnel and resources to perform each task at the appropriate time).

Both constraint and deadline scheduling are NP-complete in virtually all non-trivial cases (Garey and Johnson, 1979), which forces all automated scheduling systems to rely heavily on heuristic search techniques. Unfortunately, certain real-world scheduling considerations make good schedules extremely difficult to generate, even heuristically. The resources available to perform the jobs may be very limited or they may not be shareable between jobs. Jobs may have varying durations or variable release times, they may be uninterruptible, or they may not be permitted to run concurrently with other jobs. In addition, a scheduler is not necessarily finished after the initial schedule is made. Unforseen circumstances often arise during job execution time, such as a last-minute emergency request or an unanticipated equipment failure, that require the scheduler to make "on-the-fly" adjustments.

1.1 Related Research

Because of the very diverse nature of scheduling problems, as well as their inherent intractability, the goal of computer science researchers is not to create one general-purpose program that can handle every conceivable scheduling problem, nor to create a program that guarantees optimal schedules instantaneously. Rather, the goal is to create programs that generate near-optimal schedules, in a reasonable amount of time, for one specific subclass of scheduling problems.

Most recent research has concentrated on Job-Shop Scheduling (JSS), a general subclass of problems within the domain of Operations Research (Martin and Pling, 1978; Marcus, 1984). The goal of most JSS systems is to develop near-optimal schedules for manufacturing or industrial facilities where slight improvements in scheduling efficiency may translate into huge amounts of savings to a company. Another popular research area is the development of schedulers for computer operating systems (OS) (Deitel 1984; Tanenbaum, 1987). Many OS textbooks include a chapter on scheduling; Deitel's *An Introduction to Operating Systems* (Deitel 1984) contains a good set of goals for an OS scheduler.

Within the field of Artificial Intelligence (AI), scheduling is part of the Planning Systems domain (Nillson, 1980). Much of this research is concerned with modeling the real-world planning environment and representing the effects that certain actions have on the model. Deadline scheduling is often represented as the lowest level on the planning tree, performed only after goals are identified and task sequences are determined. Sophisticated planning systems will consider deadline scheduling restrictions as part of the overall plan generation process (Hayes-Roth et al., 1979).

Fox and Kempf (1985) have studied the dual problems of computational complexity and executional uncertainty on job-shop scheduling domains. From this, they have defined two basic principles for building an efficient scheduler. The "Principle of Least Commitment" states that a scheduler should never commit a job to a specific time interval or resource set until there is a good reason to do so. The "Principle of Opportunism" states that a scheduler should take advantage of all available opportunities to reduce its search space.

1.2 Terminology

A *job* (also called a *service*) is a single, indivisible, real-world task to be performed within a specified time interval. The actual nature of a job varies from application to application. To a doctor, a job may be one consultation session with a patient. To a factory line worker, a job may be assembling ten electric motors by a certain deadline. Associated with each job are a *priority*, a set of *preconditions* that must be met before the job may begin (also expressible as *set-up time*), the time *constraints* for scheduling the job, and a set of *resources* needed to perform the job.

A *resource* is anyone or anything available for use in the execution of a job, such as a person, a work area, a tool, or a raw material. Like jobs, resources are assumed to be indivisible units for scheduling purposes. The maximum number of jobs that a resource can support at one time is known as its *capacity*. A *dedicated resource* has a capacity of one, while a *shareable resource* has a capacity greater than one.2

2. Note that *shareability* and *indivisibility* are not mutually exclusive terms. A mainframe computer is *shareable* in the sense that more than one user may be logged in at any given time. It is *indivisible* in that a user does not request the "left half" or the "bottom one-third" of the computer when reserving CPU time. Similarly, a box of ten identical screwdrivers may be thought of as ONE shareable, indivisible resource with a capacity of TEN jobs.

The number of jobs actively being supported by a resource at any given time is known as the resource's *load*. An *idle* resource is one with a load equal to zero. A *free* (or *available*) resource has a load less than its capacity, while a *busy* resource has a load equal to its capacity. An *overloaded* resource has a load greater than its capacity and indicates that an error condition is present in a schedule. The jobs competing for an overloaded resource are said to be in *conflict*.

A *scheduler* (whether human or machine) takes a description of the set of jobs to be performed and the resources available to perform them, and produces a schedule which maps resources to jobs. An *allocation* is when one resource is reserved for one job over one interval of time, and a *supported* job is one which has reserved all of the resources necessary for its execution. Finally, a *schedule* is any mapping of resources to jobs.

2.0 SPACE STATION COMMUNICATIONS

GE's Government Communications System Division (GCSD) is a member of the McDonnell-Douglas team awarded NASA's Space Station Work Package II. GCSD's task is to develop the Space Station's Communications and Tracking System (C&TS).

C&TS will be comprised of a number of subsystems, each handling a specific class of communications (see Figure 2-1). The Space-To-Space Communications (STSC) Subsystem, for example, supports links between the Space Station and non-terrestrial sources (satellites, the Space Shuttle, etc.). All subsystems consist of a set of modular communications devices that can be configured in a variety of ways, depending on the Space Station's current needs. A set of devices that supports a single communication link is called a *string*; at any given time, a subsystem may have several (or zero) active strings.

All C&TS subsystems are managed by the Control and Monitoring (C&M) Subsystem, which is responsible for allocating communications resources, monitoring the performance of on-line devices, diagnosing equipment failures, and taking whatever actions are necessary to maintain error-free communication links. GE engineers, as part of an ongoing IR&D project, have been eval-

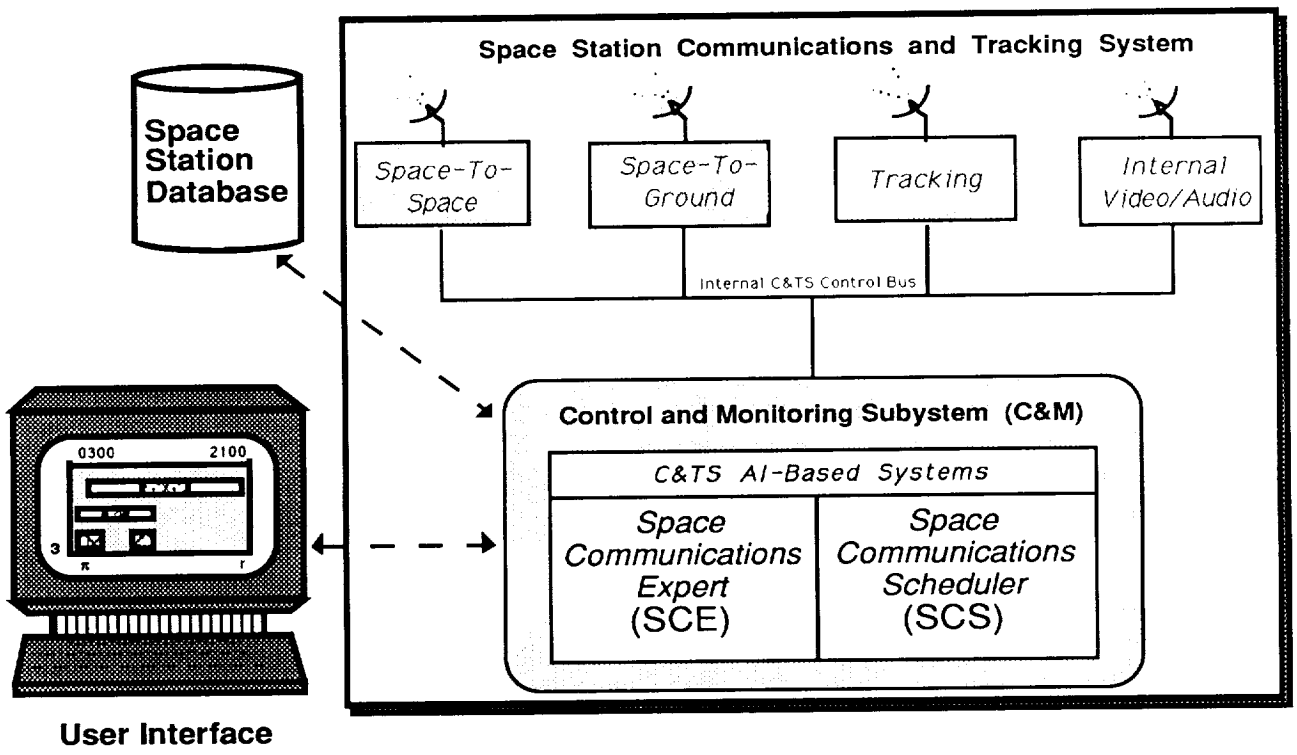


Figure 2-1. Architecture of Communications and Tracking System (C&TS).

uating the feasibility of integrating two knowledge-based systems within C&M: the Space Communications Expert (SCE) and the Space Communications Scheduler (SCS).

GE's Advanced Technology Laboratories (ATL) developed SCE in 1987 as an embedded expert system designed to monitor and maintain strings within the STSC Subsystem. SCE allocates new strings on command, then evaluates data from various sources, such as external test procedures, device status reports, and the global Space Station database, to ensure that the string is operating normally. When anomalies are detected in a communication link (comlink), SCE isolates and replaces the device causing the problem. In 1988, GCSD developed an expanded version of SCE, the Prototype Intelligent Space Communications Expert System (PISCES). PISCES extends SCE to include both strings and partial device failures in the Space-To-Ground Communications (STGC) Subsystem.

SCS and PISCES were conceived as cooperating expert systems that form the "brain" of C&M. SCS's role is to allocate and schedule C&TS devices, and then transfer control to PISCES which assembles and maintains the resulting strings. When PISCES recognizes a device failure, it notifies SCS, which in turn adjusts its schedule accordingly and specifies an available replacement device to PISCES.

2.1 SCS Scheduling Domain

C&TS is a relatively standard deadline scheduling domain in which "jobs" correspond to individual communication links (called "services"), and "resources" are the modular communication devices used to create strings (transceiver-modems, switches, fiber-optic links, antennas, etc.).

As with SCE, the first-year development effort of SCS concentrated solely on the STSC Subsystem. A string within STSC typically consists of five interconnected devices (see Figure 2-2). Transmitted signals first travel through a Baseband Signal Processor (BSP) which connects STSC with the many data busses on board the Space Station. The Transceiver-Modem (XMODEM) modulates this signal and sends it through an

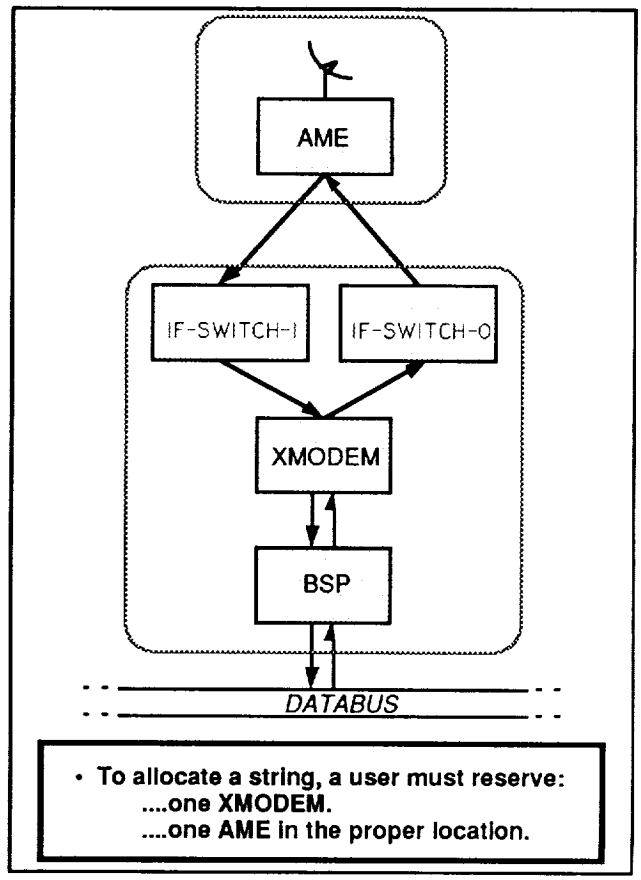


Figure 2-2. Standard Space-To-Space Communications String.

outgoing Intermediate-Frequency Switch (IF-SWITCH) to a specific Antenna Mounted Equipment (AME) from which it is transmitted. Received signals traverse the same path in the opposite direction, except that an incoming IF-SWITCH is used.

The two critical devices on an STSC string are the Transceiver-Modem and the Antenna. Selecting an XMODEM forces the selection of the BSP and IF-SWITCHes because they are hardwired together. The four types of AMEs are OMNI, AIR-LOCK, SERVICE-BAY, and PARABOLIC. They are located at various fixed spots on the outside of the Space Station. To allocate an STSC string, one must allocate an XMODEM (any operational one will do) and an AME of the appropriate type in the appropriate location.

Figure 2-3 shows the XMODEMs and AMEs of STSC represented as SCS tables.

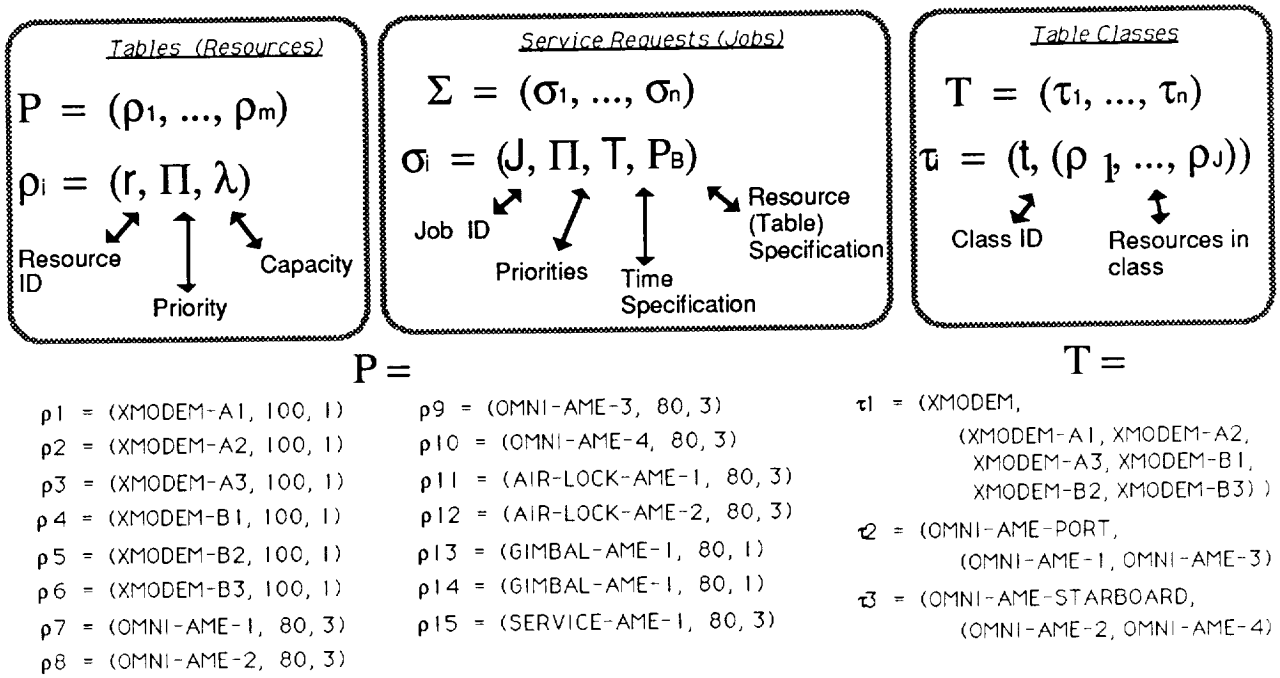


Figure 2-3. Resource tables and table classes for C&TS.

3.0 SPACE COMMUNICATIONS SCHEDULER

SCS is a rule-based scheduling system developed by GE's Artificial Intelligence Laboratory in Moorestown, NJ. SCS was designed to allocate and schedule modular communications equipment on board the NASA Space Station, automatically making adjustments during job execution time when faced with unexpected device failures or last-minute user requests. It combines algorithmic and heuristic search techniques, sophisticated pattern-matching capabilities, and a flexible scheduling strategy adaptable to many different applications. SCS was implemented using Inference's Automated Reasoning Tool (ART™) expert system shell augmented with custom LISP and C code; it runs on a Digital™ VAX computer.

SCS addresses deadline scheduling problems char- acterized by:

1. *Continuous, indivisible jobs* — Once started, a job will not be preempted before completion.

2. *Negligible or constant set-up times between jobs* — Set-up times are usually a trinary function of two jobs and one resource, yielding a time. For example, **SETUPTIME(A,B,R) = 10** means that it will take 10 minutes to "reset" resource R between the end of job A and the start of job B. SCS requires that all set-up times are either negligible (in which case they can be ignored) or relatively constant (in which case they can be automatically added to the duration of each assignment).

3. *Low-capacity resources* — Each resource has a relatively low capacity, generally five jobs or less. SCS takes approximately 1.5 times longer to schedule a (λ+1)-capacity resource than a λ-capacity one.

4. *Partial order of job prioities* — Each job has its own scheduling priority, and no job may preclude a higher-priority job from being scheduled. In other words, it is better to schedule one job of priority 100 than 10 jobs of priority 90.

The subclass of scheduling problems handled by SCS is actually very common. Virtually any domain requiring "appointments" — from a doctor's office, to dinner reservations, to library books, to a college computer terminal room, to communications equipment aboard the Space Station — is a domain in which SCS is applicable.

3.1 SCS Data Structures

Within SCS, jobs are represented by *service-requests*, resources by *tables*, and allocations by *blocks*. Service-requests and tables, along with certain *scheduling parameters*, make up the input to SCS. No two service-requests or two blocks may be exactly alike. The lone output of the system are the blocks that make up the schedule. This section contains the formal definitions of the input and output specifications of SCS.

All SCS time specifications are in military format with discrete one-minute increments. A time interval is specified as an ordered pair (t_1, t_2), inclusive of its startpoint but not inclusive of its endpoint. For example, "(0800, 0900)" specifies 60 one-minute units of time, the first unit beginning at 8:00 AM and the 60th beginning at 8:59 AM. While this is a somewhat inelegant convention, no perfect way exists to model time as an ordered set of discrete elements (Allen, 1983).

3.1.1 Service-Requests

A service-request is a 4-tuple,

$$\sigma = (j, \pi, T, \rho)$$
$$\Sigma = \{\sigma_1, ..., \sigma_m\}$$
$$\mathbf{J = \{j(\sigma_i) \mid 1 \le i \le m\}}$$

where j is the *job* represented by σ; π is an ordered pair, (π_p,π_i), specifying *priority*; T is a set of 4-tuples, $\mathbf{T = \{T_1,T_2,...\}}$, $\mathbf{T_i = (t_\sigma, (\Delta_+,\Delta_-), MIN\Delta, d)}$ representing the *time constraints*; and ρ is the *resource set*, $(\rho_1,\rho_2,...)$, required to perform s.

Each service-request corresponds to exactly one job, $j \in J$. However, a job may be represented by multiple service-requests, each with a different π, T, and/or ρ. Once a job is successfully scheduled by SCS, all alternative requests for that job are automatically deactivated.

π_p and π_i are called the *scheduling priority* and *inertia value* of σ. Scheduling priority is used only during the initial scheduling phase, and it represents the relative importance of σ in comparison to other service-requests. If and when a job is successfully scheduled, the inertia value specifies how difficult it is to "bump" that job during the rescheduling phase.

T, the time constraint for s, is itself a set of 4-tuples. Each $T_i \in T$ consists of a *start time* (t_s); the allowable negative and positive *offsets* from the start time, (Δ_-,Δ_+); a boolean flag (MINΔ) which, when set, specifies that the request should be scheduled as close as possible to t_s; and the job's *duration* (d). In standard terminology, the release time for σ is $(t_\sigma - \Delta_-)$ and the deadline for σ is $(t_\sigma + \Delta_+ + d)$. Each T_i in T signifies an equally acceptable time interval for scheduling the job.

Finally, ρ specifies the resource set for σ. Because resources are represented as "tables" within SCS, ρ is expressed as a nonempty set of table names or table classes (see Section 3.1.2). For σ to be successfully scheduled, all tables in ρ must be available at the specified time; otherwise, no resources are allocated and σ is deactivated.

An example of how to specify SCS service-requests is given in Figure 3-1.

3.1.2 Tables and Table Classes

A table is a triplet such as

$$\rho = (r, \pi_\rho, \lambda)$$
$$P = \{\rho_1, ..., \rho_n\}$$
$$\mathbf{R = \{r(\rho_i) \mid 1 \le i \le n\}}$$

where r is the *resource* represented by ρ, π_ρ its *reduction priority*, and λ its *capacity*. Every resource, r_i, has exactly one corresponding table, ρ_i, and vice versa.

The reduction priority, π_ρ, is used during the latter stages of scheduling when SCS assigns a fixed start time and resource set to each job. The higher a table's reduction priority, the more likely its corresponding resource will be used continuously in the final schedule. λ represents the resource's capacity and is specified as a positive integer.

For efficiency, similarly used resources can be collectively expressed as *table classes*. Whenever a service-request specifies a table class, T, $(P \supseteq T)$, in its resource set, SCS will automatically generate N new requests $(N = |T|)$ with the table class replaced by each $\tau \in T$. This allows a user to issue general resource requests such as "one room large enough to hold 20 people" rather than "Either Room B or Room C or Room D or..."

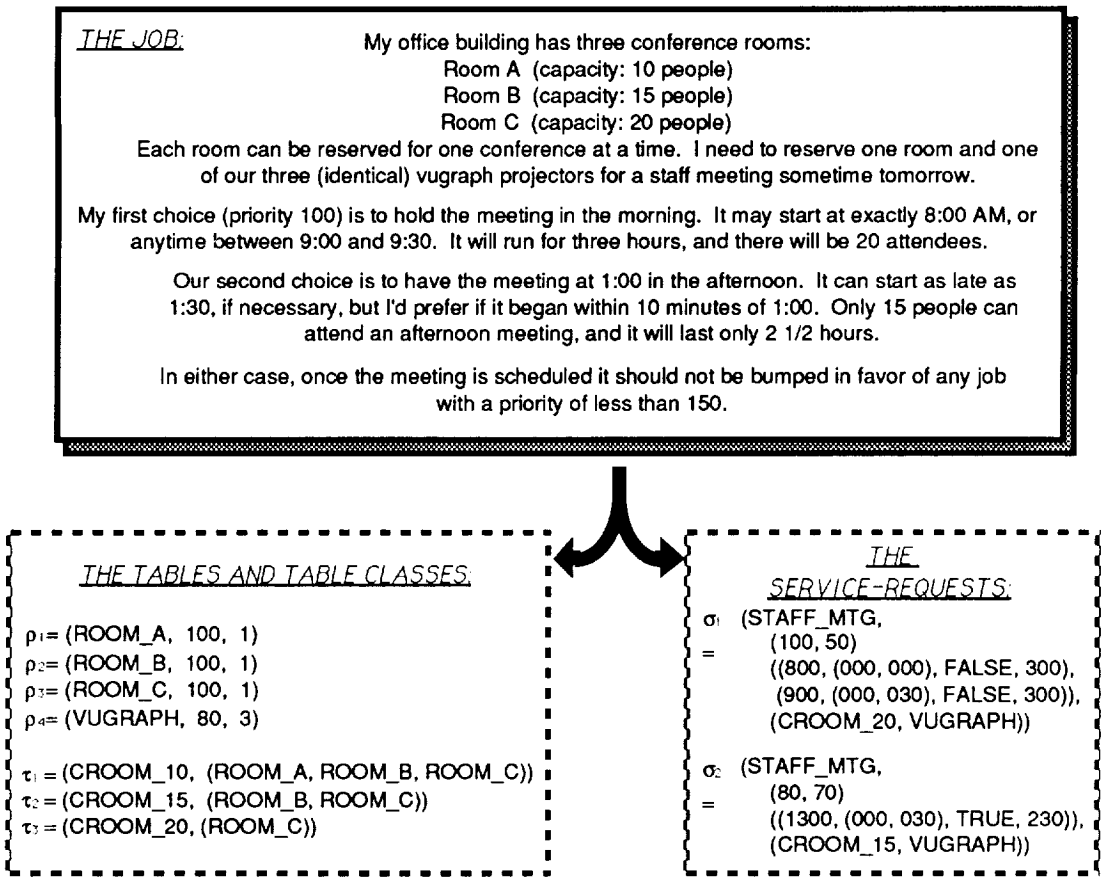


Figure 3-1. Service-request and table specifications for the "Staff Meeting" scenario.

An example of how to specify SCS tables and table classes is shown in Figure 3-1. Note that the capacity of each conference room is 1, not 10, 15, or 20. While a room may be large enough to seat up to 20 people, it still can support only one meeting at a time.

3.1.3 Blocks

The output from SCS is a set of blocks representing the mapping of tables to service-requests (i.e., resources to jobs). Blocks are expressed as a 5-tuple:

$$\beta = (\sigma, T_\sigma, P_\sigma, t_w, t_c)$$
$$B = \{\beta_1, \ldots, \beta_p\}$$

where σ is the service-request associated with β,
T_σ ($\in T(\sigma)$) is the time constraint of σ corresponding to the block
P_σ ($\in P(\sigma)$) is the set of tables in which the block is present
t_w is the *window* of time for the block
t_c is the *critical time* of the block.

The window of a block, t_w, is expressed as an ordered pair (t_{ws}, t_{we}). The length of the window is at least as long as the duration (d) of T_σ. In addition, the window is a subinterval of T_σ's release time and deadline.

The critical time, t_c, specifies the subinterval of t in which the block, if chosen for the final schedule, will definitely be in use. It, too, is expressed as an ordered pair, (t_{cs}, t_{ce}), where $t_{cs} = t_{we} - d$ and $t_{ce} = t_{ws} + d$. If the length of t_w is greater than or equal to two times the block's duration, d, then the block has no critical time, and t_c is expressed simply as "NONE."

To better illustrate critical times, consider a service-request that specifies a release time of 800 hours, a deadline of 1100 hours, and a duration of 200 hours. Such a request could be scheduled from either 800 to 1000, or from 900 to 1100, or from 845 to 1045, etc. No matter which start and

end times are chosen, however, the request **must** be in execution between 900 and 1000. Therefore, the block generated from this request would have $t_W(\beta)$ = (800, 1100) and $t_C(\beta)$ = (900, 1000).

The specific types of blocks found in SCS include the following: A *fixed block* is one in which $t_W = t_C$; that is, its window length is exactly equal to its job's duration. A *critical block* is one in which $t_C \neq$ "NONE"; similarly, a *noncritical block* has t_C = "NONE". A *split block* is a special type of noncritical block in which $(t_{We} - t_{WS}) = 2d$. Two blocks are *alternatives* if they share a common job ($\beta_1 \neq \beta_2$ & $j(\beta_1) = j(\beta_2)$), while a *unique* block is one with no alternative.

A generalized definition of schedule can now be given as "any conflict-free set of blocks." A schedule is called *complete* if it consists of only fixed, unique blocks. Schedules that are not complete are *partial* (that is, they contain at least one block which is nonunique and/or nonfixed). Partial schedules are converted to complete ones by assigning fixed start times and resource sets to each job and removing superfluous blocks; this process is called *reduction*. Examples of SCS blocks are given in Figure 3-2.

3.1.4 Notational Conventions

Notational conventions for representing tables and blocks can be defined pictorially. Conflicts, overloaded resources, block alternatives, etc., are much more noticeable when displayed graphically rather than as a textual list of n-tuples.

Figure 3-3 introduces the notational conventions used to represent blocks and tables. The two distinct formats for representing blocks are *standard notation* and *critical notation*. *Standard notation,* which highlights duration and delta time, is best suited for displaying SCS scheduling states. *Critical notation* highlights a block's critical time (or lack thereof) and is useful for identifying conflicts and overloads.

Figure 3-4 shows two blocks, β_1 and β_2, graphed within table ρ_3 ("ROOM_C"), using critical notation.

3.2 SCS Operation

The two operational phases of SCS are the Scheduling Phase and Rescheduling Phase. Its five distinct modes of execution are called Pre-Processing, Placement, Allocation, Completion, and Deallocation. Section 3.2.1 discusses the scheduling strategy used by SCS in each of its various system states.

3.2.1 Scheduling Phase vs. Rescheduling Phase

During the Scheduling Phase, the SCS generates an initial schedule from a static set of service-requests and tables. Then, SCS switches to the Rescheduling Phase in which additions and changes to the initial schedule may be made. Although they are mutually exclusive, both phases share a common set of rules, data structures, computational states, search strategies, and terminology. The most important distinction between the two is that in the Scheduling Phase SCS **creates** a new schedule, while in the Rescheduling Phase SCS **adjusts** an existing schedule.

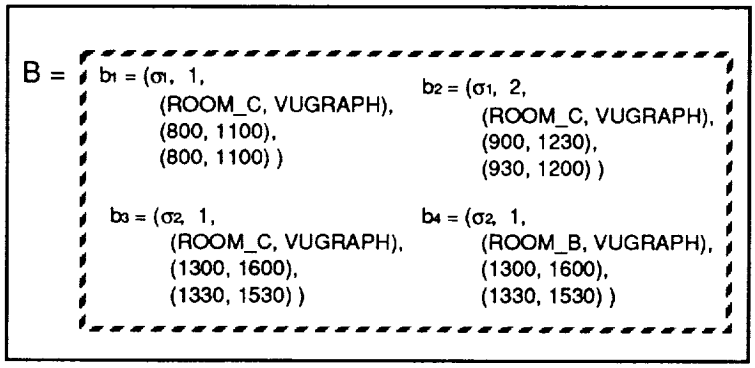


Figure 3-2. SCS blocks corresponding to the "Staff Meeting" scenario.

Scheduling phase is run once, and only once, for any given set of input. After the initial schedule is generated and has been accepted by the user, SCS automatically switches to the Rescheduling Phase. Note that the Scheduling Phase operates on a static set of input data. If the user wishes to make any changes to the input set while SCS is actively generating a schedule, two choices are available: either wait until for Rescheduling Phase and make the changes then, or halt the system, adjust the input, and start the system over.

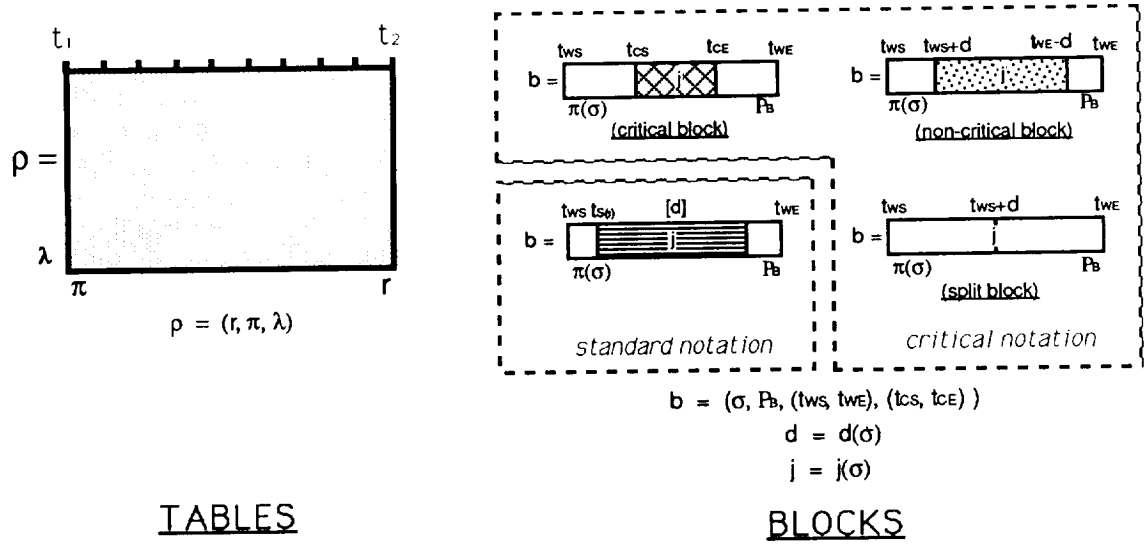


Figure 3-3. Notational conventions.

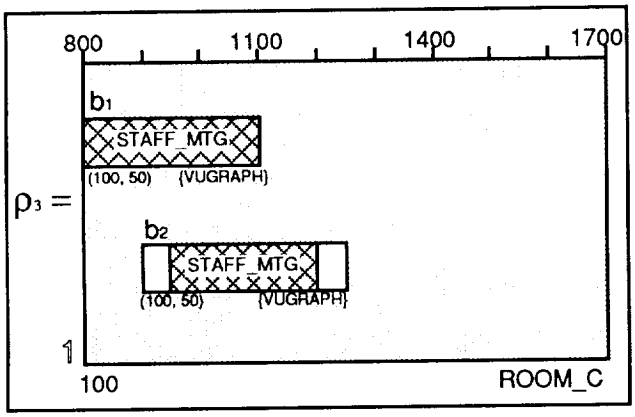


Figure 3-4. Graphing table ROOM_C in the "Staff Meeting" scenarios.

SCS is normally inactive during the Rescheduling Phase, and returns to an active state only when the set of service-requests (Σ) or tables (P) changes. The important feature of the Rescheduling Phase is that it makes *nondisruptive scheduling adjustments*, meaning that it will keep the original schedule as intact as possible during rescheduling. This feature is particularly important for the Space Station because any changes in the C&TS schedule may have a ripple effect on the schedules of other systems (e.g., payload deployment, laboratory experiments, astronauts' personal agendas, etc.).

SCS's projected role for the Space Station is its running in the Scheduling Phase once per evening to generate the C&TS schedule for the next day. SCS remains active in Rescheduling Phase throughout the day to handle last-minute service-requests, device failures, newly activated resources, etc.

3.2.2 Modes of Execution

SCS utilizes a five-step scheduling strategy, with each step known as a *mode of execution*. The SCS system state can be specified as an ordered pair of phase and mode:

$$S = S_p \times S_m$$
$$= \{\text{SCHEDULING, RESCHEDULING}\} \times \{\text{PRE-PROCESSING, PLACEMENT, ALLOCATION, COMPLETION, DEALLOCATION}\}$$

If the operational phases (S_p) define the **goal** of SCS (creating a new schedule or adjusting an existing one), then the five modes of execution (S_m) define the **approach** that SCS uses to reach its goal.

SCS's scheduling strategy may be described as a sophisticated generate-and-test algorithm. In summary, one unprocessed service-request is selected and translated into blocks, which in turn are entered in the appropriate tables. Next, SCS analyzes each table to determine when and where conflicts are present. It then uses a collection of algorithmic, heuristic, and blind-search

rules to resolve each conflict. Finally, SCS decides whether the new partial schedule is "valid" (i.e., the current request is successfully scheduled, no previously scheduled job has been displaced, and the partial schedule is reducible to some final schedule) or "invalid". In the latter case, SCS restores each modified table to its previous state and marks the current request as "unschedulable." This process is repeated until each job has been processed, at which point SCS fixes a time interval and resource set for each job.

The key to this strategy is that each intermediate partial schedule must be reducible to some final complete schedule such that every job in the former is also in the latter. That is, if you arbitrarily fix any one nonfixed block in the partial schedule, then one method to reduce it to a final schedule must still be available without displacing any jobs. Reduction is defined in more detail in Section 3.2.2.3 on Allocation Mode. Figure 3-5 shows the state transition diagram of SCS.

3.2.2.1 Pre-Processing Mode

Pre-Processing Mode is the first computational state of SCS, the user's input specifications are received and translated to ART™ relations. Unlike the other four execution modes, Pre-Processing Mode is never called explicitly. Rather, it remains in a wait state until new input is received from the user. It then activates, interrupts the current execution mode, processes the new input, and returns to the wait state.

3.2.2.2 Placement Mode

The main computational state of SCS is called Placement Mode. In this mode, service-requests are translated into blocks, and resource conflicts are detected and resolved. Figure 3-6 shows the transition diagram for Placement Mode and its seven substates: Selection, Block Generation, Resolution, Acceptance, Restoration, Rejection, and Displacement.

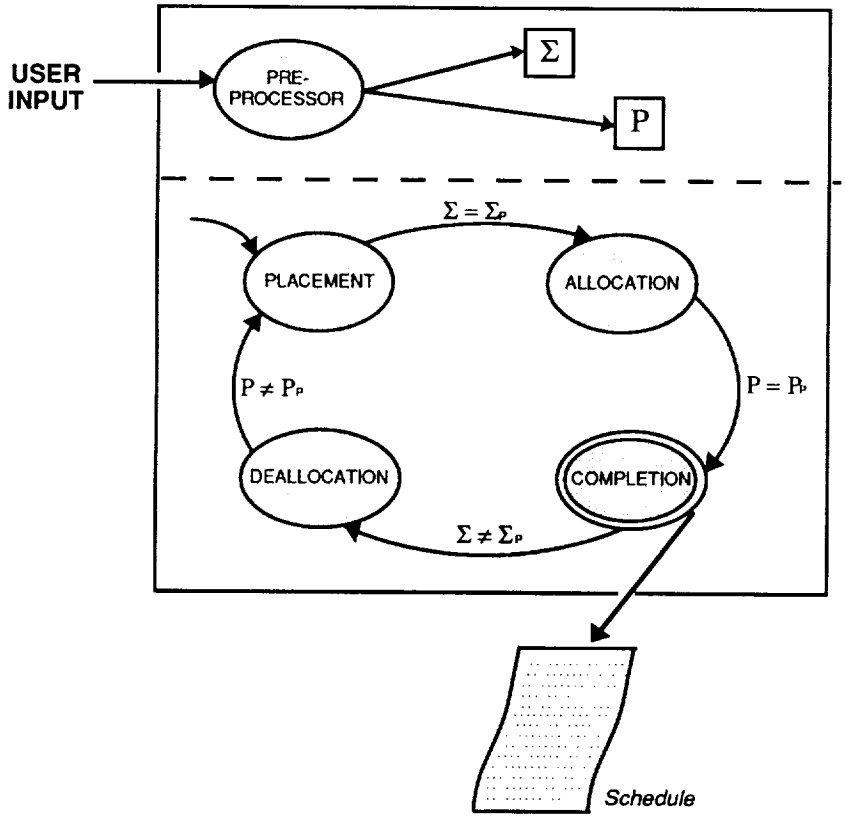


Figure 3-5. State transition diagram for SCS.

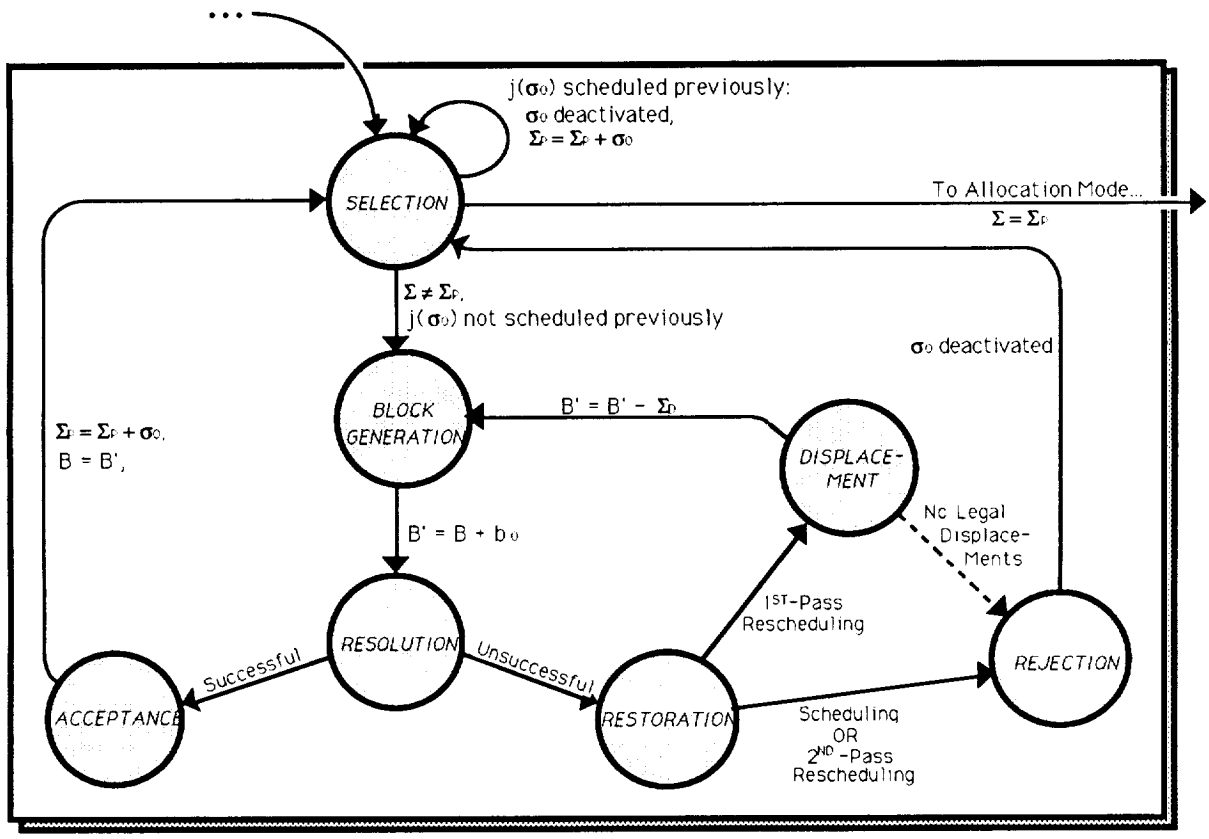


Figure 3-6. Substrate transition diagram for Placement Mode.

SCS operates on one service-request at a time from the *agenda* of unprocessed requests. Recall that the relative scheduling priority, π_p, of each service-request defines a partial order on Σ, and that this partial order determines scheduling order. In the Selection substate, SCS selects the *current-service-request* (represented by σ_0) at random from the set of requests at the top of the agenda. Though using the maximum duration of each request as the second discriminator seems reasonable when selecting σ_0, no improvement using this approach was observed during system testing.

The Block Generation substate performs two functions. First, it checks whether the job corresponding to σ_0 is already present in the partial schedule. Formally, it checks if $\exists\ (\beta \in B)\ (j(\beta) = j(\sigma_0))$. If so, SCS deactivates the request and selects a new σ_0. Otherwise, SCS generates β_0, the set of blocks defined by σ_0, and sets $B' = B \cup \beta_0$. One service-request may generate dozens of alternative blocks for a job, but these will be reduced to, at most, one fixed block in the final schedule.

In the Resolution substate, SCS detects and resolves any conflicts in B′. This substate is by far the most complex component of SCS and is discussed in detail in Section 3.3, "Conflicts and Resolution."

For the current-service-request to be successfully scheduled, B′ must meet two criteria after all conflicts are resolved. First, σ_0 must be represented by at least one block in B′. Second, every job represented by a block in B must also be represented in B′. Formally, $\exists\ (\beta \in B')\ (\sigma(\beta) = \sigma_0) \wedge \forall\ (\beta \in B)\ \exists\ (\beta' \in B')\ (j(\beta') = j(\beta))$. If both criteria are met, then the new partial schedule is accepted.

If either of the two criteria are not met — that is, either σ_0 is not represented in B′, or it "bumped" a job that had been scheduled in B — then the new partial schedule is invalid and the previous partial schedule is restored. The next state transition is dependent on whether SCS is in Scheduling or Rescheduling Phase. In the latter case, the Displacement Substate will attempt

to displace blocks from B so that σ_0 can be successfully scheduled (see Section 3.2.3 "Rescheduling Strategy"). If SCS is in Scheduling Phase, or if displacement has already been attempted for σ_0, then the Rejection substate deactivates σ_0 and returns control to the Selection substate.

3.2.2.3 Allocation Mode

When Placement Mode is complete, SCS switches to Allocation Mode. Here, the partial schedule specified by B is reduced to a final, complete one. Each job is assigned a fixed start time and resource set, and its alternative blocks are removed.

Just as Placement Mode processes Σ sequentially according to each request's scheduling priority, Allocation Mode processes P sequentially according to each table's reduction priority. The higher the value of π_ρ for a table, the more likely its corresponding resource will be in continuous use during job execution time.[3] The *current-table* being reduced is denoted ρ_0. Note, however, that the reduction of ρ_0 may cause changes in other tables not yet reduced (because the same block is often present in multiple tables).

The reduction strategy used by Allocation Mode is based loosely on the Fox-Kempf Principle of Opportunism. A huge number of complete schedules may be derivable from one partial schedule, B, thus indicating a heuristic reduction strategy. Consider, though, that certain jobs in J may be represented by only one fixed block (call it $\beta_1 \in B$) in the partial schedule. Because SCS has no option on scheduling this job, and because every effort must be made to keep resources in continuous use, the system should try to find another block, β_2, that can either start when β_1 ends, or end when β_1 starts.

Fixing β_2 next to β_1 creates a *chain* of length two. SCS's reduction strategy is to first extend any existing chains in ρ_0 as long as possible. When no chains are extendible, SCS tries to create a new chain from the remaining set of nonfixed blocks in ρ_0. Only "Minimize-Delta" blocks are exempt from this process; these are fixed as close as possible to their requested start times before any attempt at chaining begins.

When all blocks in B are fixed and unique, Allocation Mode halts and the final schedule is presented to the user. SCS enters Rescheduling Phase (if it is not there already) and waits for new user input in Conclusion Mode.

3.2.3 Rescheduling Strategy

SCS's rescheduling philosophy is to make adjustments to the current schedule with as few disruptions as possible. If a service-request appears on the agenda during Rescheduling Phase — either because the user just issued it, or because one of its allocated resources suddenly became unavailable, or because it was bumped from the final schedule by another job — SCS will first try to schedule it without disturbing any existing blocks. Failing that, SCS will displace certain lower-priority blocks to "squeeze" the new request into the schedule. These bumped jobs are, in turn, placed on the agenda and rescheduled.

Certain jobs will naturally increase in priority once they become part of the final schedule. On the Space Station, for example, the astronauts will arrange their personal schedules according to the daily job schedule they receive each morning. Even a relatively minor job, such as a noncritical scientific experiment, may require extensive preparation time.

The *inertia* of a service-request, $\pi_i(\sigma)$, specifies the difficulty of displacing σ during Rescheduling Phase. Inertia is specified as a nonnegative increment to the request's scheduling priority and σ' cannot bump σ unless $\pi_p(\sigma') > \pi_p(\sigma) + \pi_i(\sigma)$.

The Placement Mode of the Rescheduling Phase will perform two separate attempts to schedule a request on its agenda. During the first pass, this mode operates exactly as in Scheduling Phase unless, and until, the Rejection substate is

3. The reason why SCS strives to keep resources in continuous use stems from the Space Station domain. To conserve electricity, each device in the Communications and Tracking System is turned off when not in use. All such devices must undergo a "power-up procedure" before being switched back into operation, and this procedure can be relatively expensive in terms of electricity.

reached. Instead of deactivating σ_0, SCS transfers control to a seventh Placement Mode substate called Displacement. Here, SCS creates a *displacement set,* B_D ($B \supseteq B_D$), for σ_0. B_D includes all blocks in B which (1) conflicted with a block $\beta \in B_0$ during first-pass Resolution, and (2) have a rescheduling inertia less than β's scheduling priority. SCS then sets $B = B - B_D$ and returns to the Block Generation substate.

If σ_0 is still unschedulable after the second pass, B_D is wholly restored and control is passed to the Rejection substate as usual. If σ_0 is successfully scheduled, however, then a total restoration of the displacement set will be impossible. Instead, SCS will attempt to restore each block in B_D individually, beginning with the one having the highest inertia. Call the set of unrestorable blocks B_D'. SCS will reactivate all service-requests corresponding to a block in B_D' and place them on the agenda, where they will be rescheduled accordingly.

The Deallocation Mode, also performed during the Rescheduling Phase, is strictly administrative in purpose. In Deallocation, SCS sets P_p, the set of processed tables, equal to the empty set. This ensures that the Allocation Mode, when it is rerun, will process and reduce every table in P.

3.3 Conflicts and Resolution

Conflicts in scheduling and their resolution are an innate part of creating almost any type of schedule. The following example introduces just how SCS addresses such conflicts and resolves them.

Recall the Staff Meeting scenario of Figure 3-1. The first service-request on the agenda, σ_1, requests one 20-seat conference room and one vugraph projector for a 3-hour period, beginning at either 8:00 AM or sometime between 9:00 and 9:30 AM. SCS will generate the two blocks described by this request, then skip over σ_2 because its job is already represented in B.

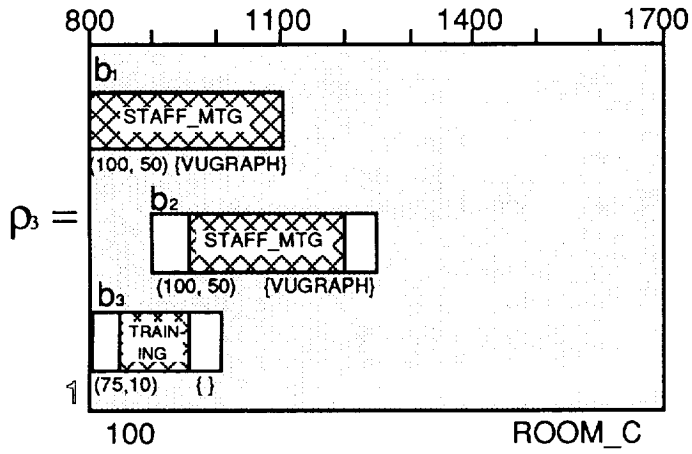
Now assume that σ_3 ($j(\sigma_3)$ = "TRAINING"; $\Pi(\sigma_3)$ = (75, 10)) requests a 20-seat conference room for 1.5 hours starting sometime between 8:00 and 8:30 AM (see Figure 3-7). Only ROOM_C seats 20 people, so this training session will either be held in that room or not held at all. STAFF_MTG has already reserved ROOM_C for most of the morning, but has requested more time than it actually needs. The question is whether a method exists to satisfy the requirements of both jobs.

This is the basis of SCS's *conflict-resolution* scheduling strategy. When the new B_0 is added to B, a number of time intervals will likely have a resource reserved for more jobs than the resource can legally support. The goal of the Resolution substate is to detect and resolve all the conflicts that might prevent B from being reducible to a complete final schedule.

The *Conflict Set* for B is denoted $X = \{\chi_1,\ldots,\chi_n\}$ where each conflict is an ordered triple:

$$\chi_i = (\beta, (t_1,t_2), \Psi).$$

χ_i may be read "block β cannot be scheduled between interval t_1 to t_2 while all blocks in Ψ are



$\chi_1 = (b_2, (900, 930), \{b_3\})$
$\chi_2 = (b_1, (830, 930), \{b_3\})$
$\chi_3 = (b_3, (800, 1000), \{b_1\})$
$\chi_4 = (b_3, (930, 1000), \{b_2\})$

$\chi' = (b_3, (930, 1000), \{b_1, b_2\})$

Figure 3-7. STAFF_MTG vs. TRAINING conflict.

present in B." For every χ_i, the n blocks in $\Psi(\chi_i)$ must represent n distinct jobs.

3.3.1 Conflicts and Decidability

A conflict can be detected as follows: add up the number of distinct jobs present in a table at any time and then check if the sum exceeds the table's capacity. Note, however, that "table" is not one of the parameters in the conflict triplet. Conflicts are a property of blocks alone, and a block may be present in more than one table. In fact, two conflicts found in different tables are often combined to form a third conflict.

Certain types of conflicts are harmless. Figure 3-8a shows a two-block table that can clearly be reduced to a final schedule, despite the presence of an overload at (t_1,t_2). However, the conflict in Figure 3-8b is definitely harmful. One of its two blocks will have to be eliminated if the table is to be reducible. The first type of conflict can be labeled "safe" and the other "dangerous."

A question now arises, does a simple algorithm exist that decides whether any given χ is safe or dangerous? Apparently not. While many conflicts, such as the two in Figure 3-8, are easily decidable, some appear to require nothing short of trial-and-error.

SCS classifies conflicts into three categories, $X = X_S \cup X_D \cup X_U$ (see Figure 3-9). Dangerous conflicts ($\chi \in X_D$) are resolved algorithmically, either by restricting the width of $\beta(\chi)$ or removing it entirely. Similarly, safe conflicts ($\chi \in X_S$) are ignored for the moment, although later changes to the schedule may cause a safe conflict to become dangerous. Any conflict that cannot be easily proven safe or dangerous is called "undecidable" ($\chi \in X_U$).

3.3.2 Dangerous Conflicts

A decision as to the type of conflict is basically a problem of pattern matching. Templates can be defined that describe a certain class of conflict in its simplest form. A pattern matcher would then try to find matches for these templates in a heavily crowded schedule. This type of problem is well suited for a rule-based implementation such as ART™.

Fortunately, the majority of dangerous conflicts fall into three easily defined classes. The most common type is a *first-order conflict.* This conflict occurs in table ρ when the critical times of $\lambda(\rho)$ blocks overlap each other, and these in turn overlap another block. The four conflicts χ_1-χ_4 in Figure 3-7, as well as the one in Figure 3-8a, are first-order conflicts.

A simple *second-order conflict* is shown in Figure 3-10b. Here, noncritical block 4 overlaps the critical times of blocks 1 through 3, but no first-order conflicts are present. Block 5 overlaps block 4 at both points t_A and t_B. Upon examination, it is apparent that block 5 must either (a) start after t_A or (b) end before t_B if block 4 is to be schedulable.

Third-order conflicts (see Figure 3-10c) are closely related to second-order conflicts, except that in this example block 4 is a critical block rather

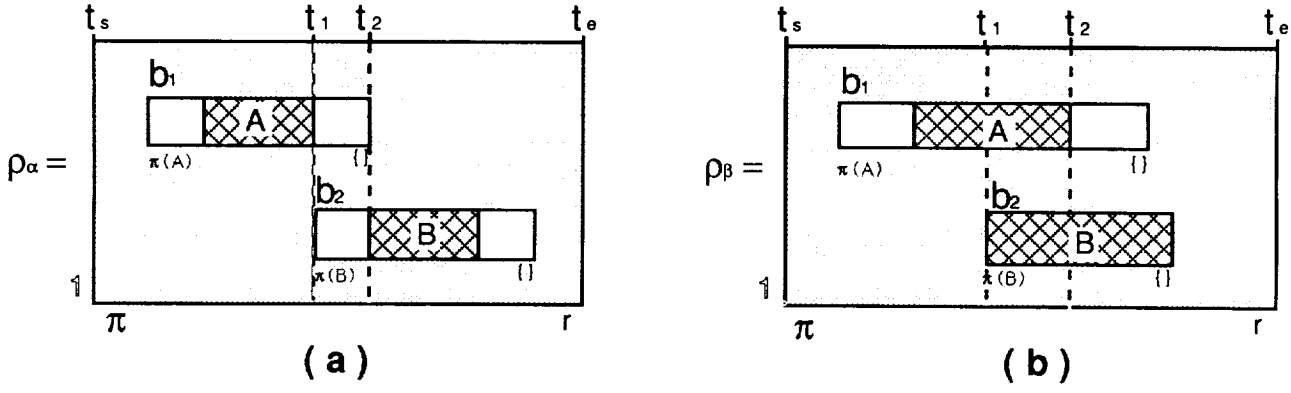


Figure 3-8. Safe vs. dangerous conflicts.

C-2

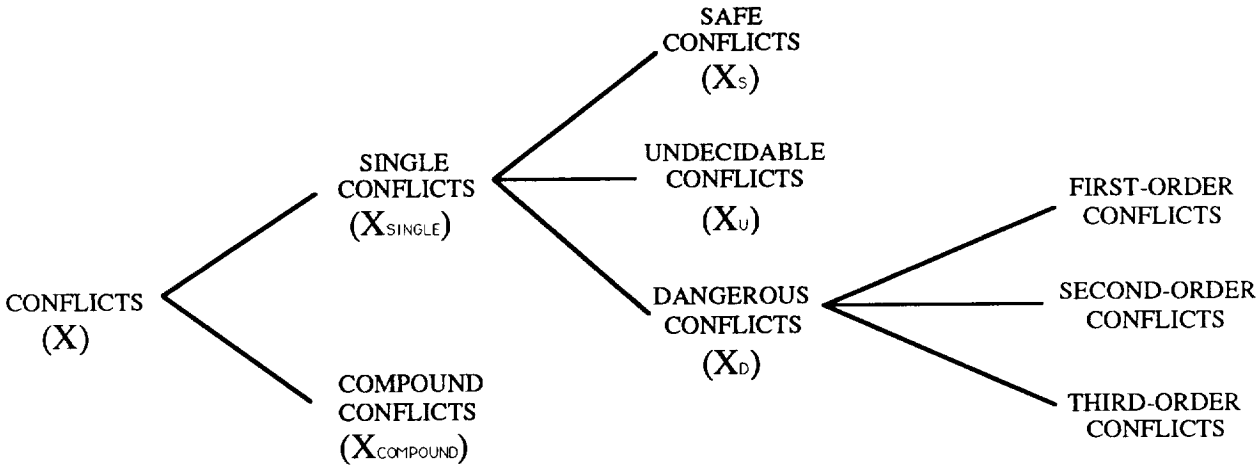


Figure 3-9. SCS conflict classes.

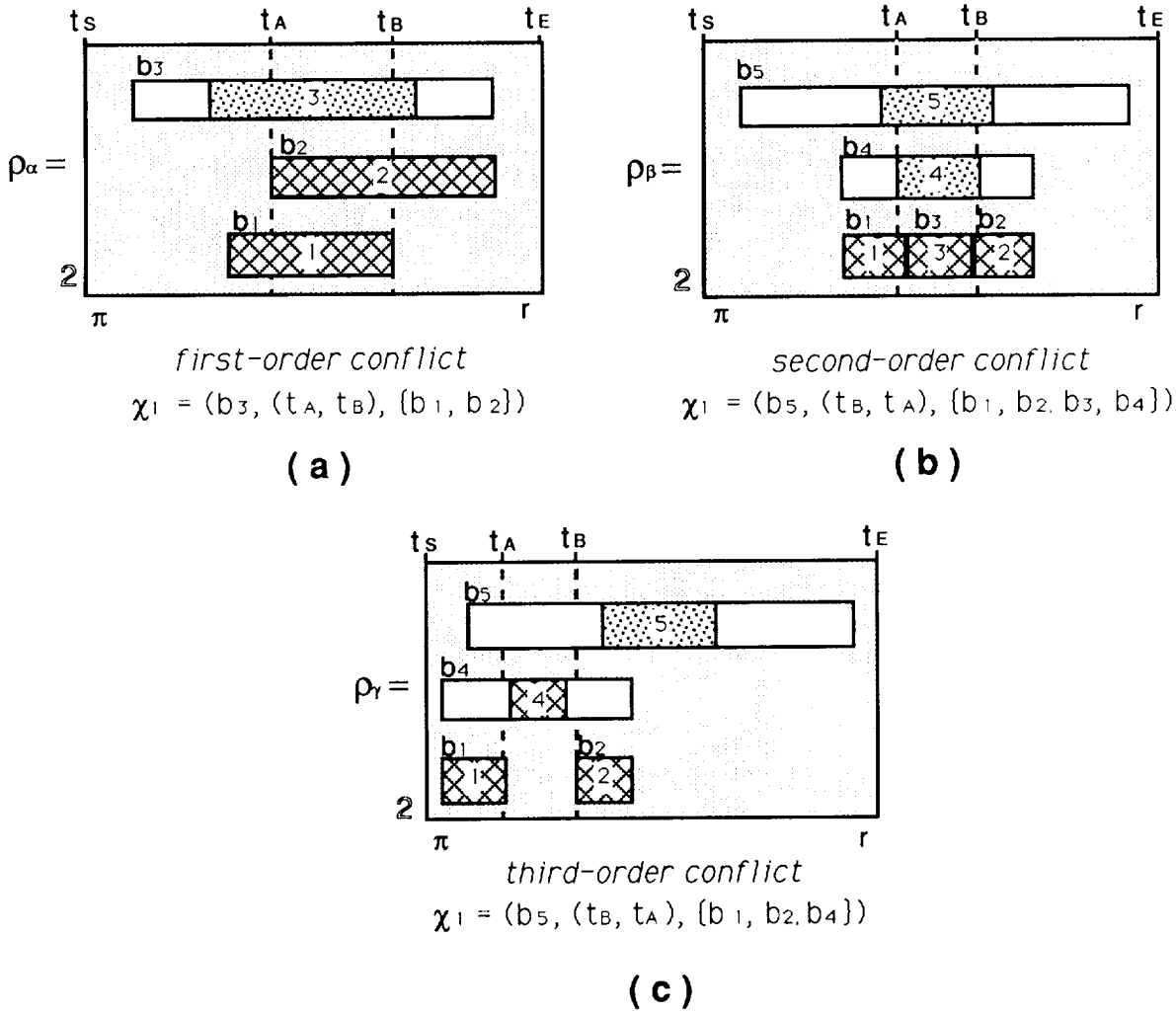


Figure 3-10. Dangerous conflict classes recognized by SCS.

than a noncritical one. Again, note that no first-order conflicts are present and that block 5 must start after t_A if block 4 is to be scheduled.

Both second- and third-order conflicts have $t_2(\chi)$ less than $t_1(\chi)$. It seems strange to say that a conflict is present "between 10 AM and 9 AM". However, if one considers $t_1(\chi)$ to be the latest safe time that $\beta(\chi)$ can end, and $t_2(\chi)$ to be the earliest safe time that $\beta(\chi)$ can start, then this order of specifying times is consistent for all three conflict classes. Note also that the job duration of $\beta(\chi)$ must be greater than $(t_1(\chi) - t_2(\chi))$ minutes for any conflict to be valid.

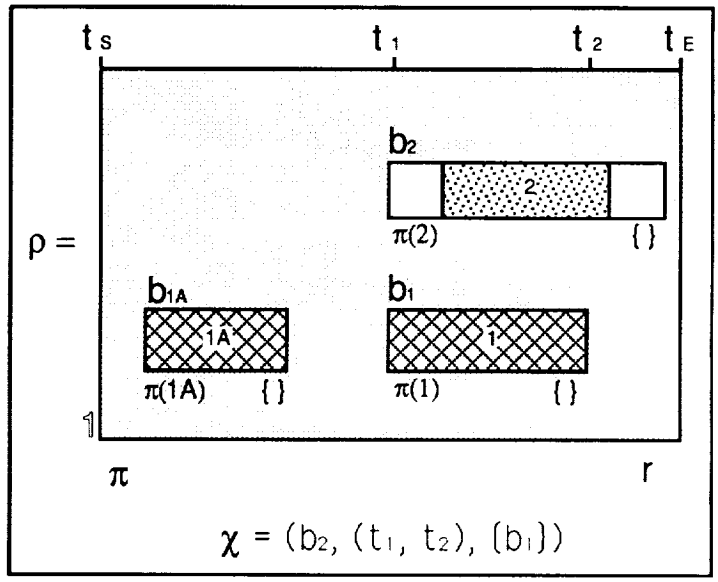


Figure 3-11. An "open" dangerous conflict.

3.3.3 Operations on Conflicts

As described previously, the presence of a dangerous conflict simply means that a certain block cannot be scheduled concurrently with certain other blocks. How such a conflict should be resolved, or even if it should be resolved, is not always clear.

Consider Figure 3-11. This table clearly contains a dangerous first-order conflict, χ = (2, (t_1, t_2), {1}). Does this mean that block 2 must be restricted to start after t_2? Not necessarily. If blocks 1 and 1A are alternatives, that is, j(1) = j(1A), two options are available: block 2 can be restricted, or block 1 can be removed. Which option is "correct" depends on the service-requests yet to be processed.

A dangerous conflict with more than one possible resolution is called an *open* conflict. The Fox-Kempf Principle of Least Commitment calls for the decision on resolving open conflicts to be delayed as long as possible. A *closed* conflict is one which has only one possible resolution, in which case SCS can make the necessary adjustment immediately.

Now consider Figure 3-12 which has two dangerous first-order conflicts: χ_1 and χ_2. Assume j(1A) = j(1B). Both conflicts are open when examined separately, but notice that if block 2 is scheduled across time t_2, then neither block 1A nor 1B is schedulable. A new closed conflict has appeared from the intersection of two open ones: χ' = (2, (t_2, t_2), {1A, 1B}).

Combining two open conflicts in this manner yields a *compound conflict* (represented χ'). Compound conflicts are always dangerous, though they may be open or closed. They differ from single conflicts in that two or more blocks in $\Psi(\chi')$ may represent the same job. A complex set of rules governs when and how two conflicts may be combined.

The criteria for determining whether any conflict, single or compound, is open or closed can now be addressed. Given χ = (β, (t_1, t_2), Ψ), if no job in $\Psi(\chi)$ has an alternative not contained in $\Psi(\chi)$, then χ is closed.

SCS resolves closed conflicts by moving the offending block completely out of the conflict interval. Specifically, for any closed conflict χ = (β, (t_1, t_2), Ψ), SCS will try to split β into two new blocks: one running from $t_{WS}(\beta)$ to $t_1(\chi)$, the other from $t_2(\chi)$ to $t_{WE}(\beta)$. Of course, if a new block is not wide enough to support j(β), then it is removed from B.

3.3.4 Undecidable Conflicts and Resolution States

As stated earlier, SCS is unable to make decisons concerning certain dangerous conflict classes. Most of these occur in tables having an overabundance of noncritical blocks. Figure 3-13 illustrates one such example. None of the three

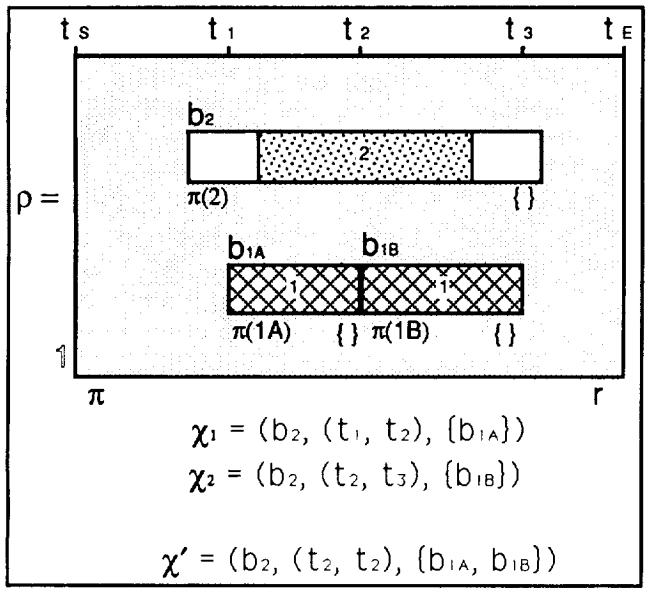


Figure 3-12. A "closed" compound conflict.

known conflict orders are present, but no method exists to reduce this table without removing one of the 13 blocks.

Although a scheduler should ideally not make firm scheduling decisions until absolutely necessary, SCS requires that all partial schedules be reducible at the conclusion of each Resolution substate. SCS must, therefore, assume that all undecidable conflicts are dangerous. At this point, SCS can still heuristically adjust the schedule to minimize the chance that a later service-request will be precluded unnecessarily. However, SCS's first priority is to schedule the current service-request, and it might become necessary to make some restrictive decisions to squeeze σ_0 into B.

The obvious first step is to do nothing about undecidable conflicts until all closed conflicts have been detected and resolved. Resolving one closed conflict often greatly decreases |X| because it may eliminate a block that was part of many other conflicts.

Step two is to safely eliminate an undecidable conflict, either by removing the overload condition that is causing it or by converting it into a safe one. Simply nudging a block out of a conflict interval or eliminating it completely, if it has an alternative that is not part of any conflict, often eliminates an undecidable conflict. SCS uses a set of heuristics to choose an effective, relatively nondisruptive adjustment, then checks if any existing conflicts may be combined or closed as a result.

If these methods fail, step three is to do whatever is required to successfully schedule σ_0, no matter what effect this may have on the scheduling of future requests. SCS checks the most promising search paths looking for any successful and reducible partial schedule.

The longer SCS requires to eliminate X_U, the less flexibility SCS has to deal with later service-requests. Consequently, the more conflict classes that are decidable, the better quality schedule SCS will produce. However, the execution time of SCS is directly proportional to the number of decidable conflict classes.

In the Staff Meeting example discussed at the beginning of this section, SCS recognizes four dangerous conflicts when σ_3's blocks are added to B (χ_1 to χ_4; see Figure 3-7). A fifth, compound conflict (χ_1') is generated by merging χ_3 and χ_4. Three of these are closable (χ_1, χ_2, χ_1'), and the resulting partial schedule is shown in Figure 3-14.

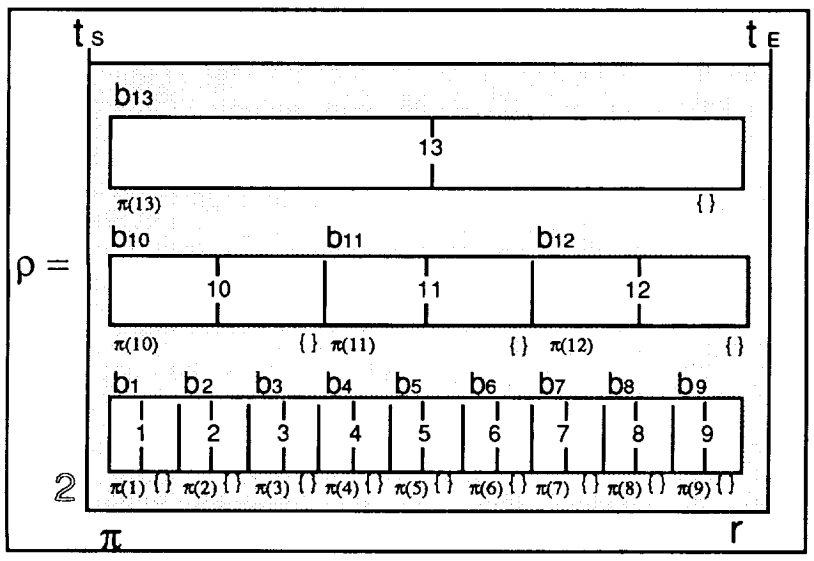


Figure 3-13. An undetectable dangerous conflict.

4.0 IMPLEMENTATION

SCS has been implemented on a Digital VAX-8650 mainframe under the VMS operating system. SCS contains over 300 ART production rules, and 1000 lines of custom C and LISP code. The user supplies an ASCII file containing definitions of the service-requests, tables, and scheduling parameters.

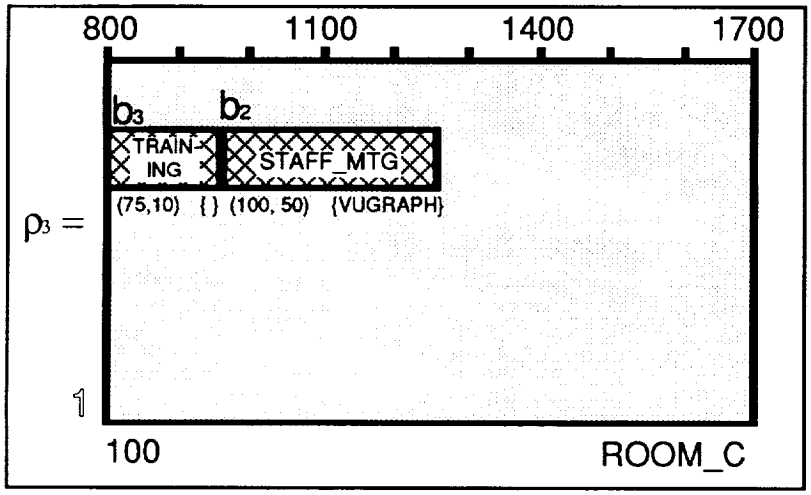


Figure 3-14. The "Staff Meeting" scenario of Figure 3-1 with all conflicts resolved.

A graphical user interface to SCS has also been developed using a Digital VT341 color terminal (see Figure 4-1). Tables and blocks are represented in graphical format, with the user having the option of displaying or suppressing critical times. Mousing on a corresponding graphic obtains information on tables or blocks. A column of mouse icons along the right edge of the screen allows the user to enter commands to SCS.

The user interface may be run as a coprocess or parent process to SCS. The scheduler generates a series of one-line ASCII messages that notifies the interface program when a significant action has been performed.

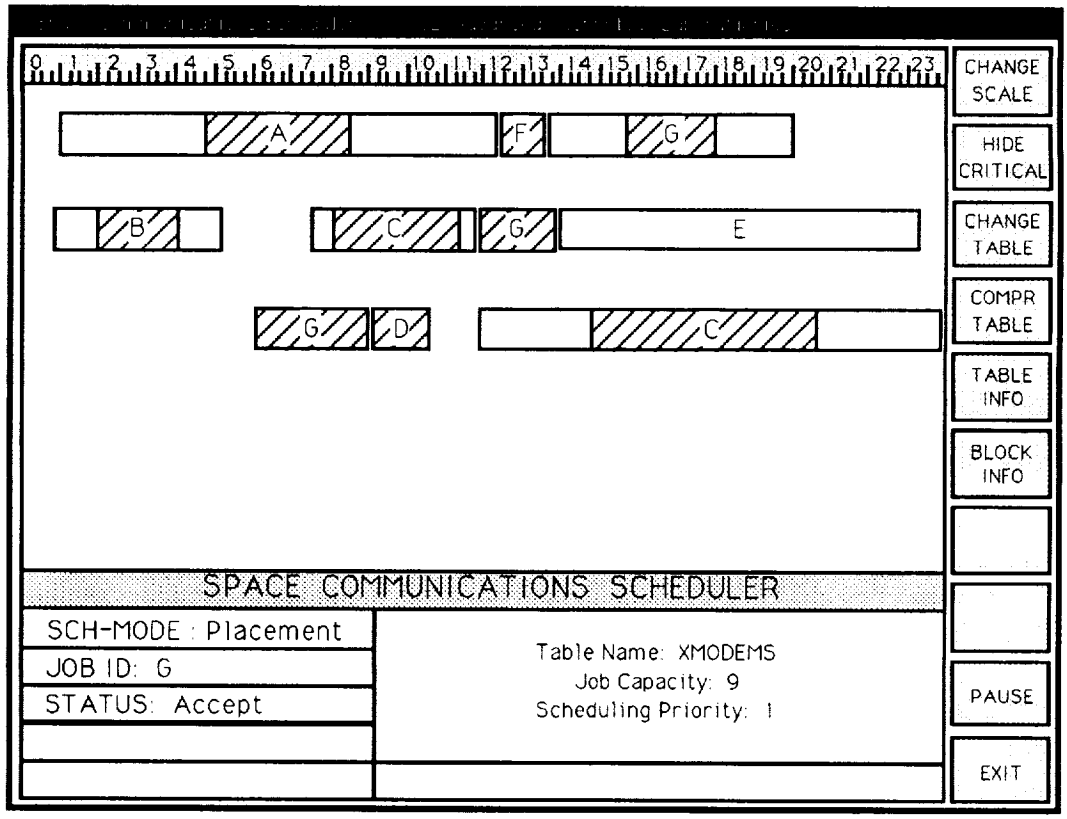


Figure 4-1. SCS user interface.

These actions include creation of a table, creation of a new block, adjustment or deletion of an existing block, selection of a new current-service-request, or a change in the system's phase or mode. The user interface acts on these messages sequentially, adjusting the display to reflect the new system state.

5.0 CONCLUSIONS AND FUTURE WORK

The conflict-resolution scheduling strategy of SCS works quite well within SCS's limited scheduling subclass. Empirical data indicates that SCS operates in low-order polynomial time for |P| (the number of tables) and |J| (the number of distinct jobs). It appears, however, to be exponential for MAX_λ, the maximum capacity of any $\rho \in P$ (hence the requirement for low-capacity resources).

Future development work may address variable-length job durations ("I need a conference room for between three and four hours"), non-reusable resources, and resource sets with nonidentical time constraints ("I need a conference room for two hours, and a vugraph projector for the first half-hour"). Another useful feature would be to allow inertia values to be specified as a function of time, based on the theory that it is better to reschedule a job ten hours before its scheduled start time rather than just ten minutes beforehand. SCS may also be translated into C or Ada to improve speed and facilitate software verification. Before any translation can occur, however, an efficient means for detecting dangerous conflicts is needed because SCS will no longer have access to ART™'s powerful pattern-matching facility.

SCS could also be extended to allow temporal restrictions between jobs. For example, a specification could be made that job j_1 may not begin executing until j_2 halts. Restrictions could also be specified at the service-request level ("if j_1 is scheduled via σ_1, then σ_2 must run concurrently with it"), or they could define one job to be contingent on another ("If j_1 is scheduled before 0800 hours, then do not schedule j_2"). Allen [9] lists 13 possible relationships between time intervals that could be used as the bases for temporal restrictions.

REFERENCES

Garey, M.R., and D.S. Johnson; *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York, NY: W.H. Freeman and Co., 1979.

Martin, C.F., and R.S. Poling; "Fast, Dynamic Programming Selection Algorithm for the Job-Shop Problem with Job Availability Intervals", GE Technical Information Series, No. 78CIS012, 1978.

Marcus, R.; "An Application of Artificial Intelligence to Operations Research", *Communications of the ACM*, Vol. 27, No. 10; pp. 1044-1047; October1984.

Deitel, H.M.; *An Introduction to Operating Systems.* Reading, MA: Addison-Wesley, 1984.

Tanenbaum, A.S.; *Operating Systems - Design and Implementation.* Englewood Cliffs, NJ: Prentice-Hall Inc.,1987.

Nillson, N.J.; *Principles of Artificial Intelligence.* Palo Alto, CA: Tioga Publishing Co., 1980.

Hayes-Roth, B., F. Hayes-Roth, S. Rosenschein, and S. Cammarata; "Modeling Planning as an Incremental, Opportunistic Process", *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*; IJCAI-79, Tokyo, pp. 375-383, 1979.

Fox, B.R., and K.G. Kempf; "Complexity, Uncertainty and Opportunistic Scheduling", *Proceedings of the Second Conference on Artificial Intelligence Applications. Washington, DC*: IEEE Computer Society Press, pp. 487-492, 1985.

Allen, J.F.; "Maintaining Knowledge About Temporal Intervals", *Communicatons of the ACM*, Vol. 26, No. 11; pp. 832-843, November 1983.