FLIGHT TRANSPORTATION LABORATORY REPORT R87-8

A FLEXIBLE SCHEDULING ENVIRONMENT USING DYNAMIC EXCEPTION HANDLING

by

Peter H. Appel

May, 1987

FLIGHT TRANSPORTATION LABORATORY REPORT R87-8

A FLEXIBLE SCHEDULING ENVIRONMENT USING DYNAMIC EXCEPTION HANDLING

by

PETER H. APPEL

ABSTRACT

A generalized environment which facilities the development of various types of schedules has been developed. This environment includes an extensive user interface in which a graphics terminal and a mouse is used to display and manipulate schedules. A common data structure is used for representing schedules; this data structure is flexible enough to allow for the different amounts of information required by different scheduling problems.

In developing algorithms to be used in conjunction with this scheduling environment, an attempt has been made to address the situation that has faced automated airline scheduling systems in the past – that there are such a large number of pieces of information specific to each scheduling situation that it is hard to incorporate this "knowledge" into a deterministic algorithm. The solution that is proposed can be called *dynamic exception handling.* This system will allow an individual to communicate these specific piece of information to the automated system in a way that will allow the special cases to be handled with a minimum loss of efficiency.

Acknowledgements

While I have been doing this research, many people have provided advice, encouragement, and inspiration. I would like to take this opportunity to thank a few of them. First, I would like to thank Professors Robert Simpson and Dennis Mathaisel, who taught me a great deal about airline scheduling and gave me many ideas for my research. I would like to thank Lyman Hazelton for many enlightening conversations about Artificial Intelligence and countless other topics. Dr. John Pararas and Mark Kolb always seemed to be there when I had a question about the implementation of my system, and I thank them for their patience.

I am grateful to Dr. Kenneth Sealy of the London School of Econmics, who introduced me to many topics within the field of transportation in general and the airline industry in particular, encouraging further work in the field that has led me to many fascinating pursuits. Finally, I would like to thank my parents, who always believed in me.

Contents

Chapter 1

Introduction

Scheduling problems – whether they involve aircraft rotations, airline crews, or other items in transportation and other fields, have many characteristics in common. When developing computer software to solve these problems we can take advantage of these similarities. Ideally, the developer of scheduling algorithms should be able to work in an environment in which those characteristics common to many scheduling problems serve as a basis and in which there are sufficient tools and sufficient flexibility to develop a wide variety of scheduling systems.

Within the field of airline scheduling, there have been several types of approaches to the problems of optimizing aircraft schedules. Ideally, there would be a "black box" which – when fed in information about demand for service, characteristics of routes and of aircraft, and other relevant factors – would produce the optimal schedule, maximizing some factor such as total profit from the service. However, no such system has been developed, because of the number of barriers facing its development. Some of these problems are discussed by Etschmaier and Mathaisel [3] One problem is that it is hard to express the objectives or an airline in purely monetary terms – there are often many other goals that must be addressed. In addition, the constraints to the scheduling process are also hard to define – factors that deal with interactions between people (such as in regulatory agencies) are hard to quantify. Perhaps one of the greatest barriers to a large automated system which encompasses all constraints and objectives of the scheduling process is that all of these details are continually changing. Whereas it might theoretically be possible to create a process which incorporated a very large number of details about how a schedule should be constructed, the process of maintaining the day-to-day accuracy of this information might prove more complicated than the development of the system itself.

From within the fields of operations research and network analysis there have emerged several tools which can perform schedule optimizations given a limited, definable set of constraints. These include the application of network flow algorithms such as the out-of-kilter algorithm to a set of arcs representing flight segments constrained by a set of costs, representing factors such as revenue on a flight. Other heuristic algorithms have been developed with the goal of reducing the number of aircraft required to serve a given schedule. However, no model currently in place is able to consider every possible constraint and objective of the scheduling process.

The fact that a computer cannot, thus far, solve the entire scheduling problem certainly does not preclude it it from assisting in the process. In its manual form, scheduling is a time-consuming, tedious process. A typical manual scheduling situation might involve large wall charts on which slips of paper representing flight segments are placed horizontally on the chart, and then continually manipulated to try to conform to the numerous constraints to the process – some documented in reference books and others stored in the minds of the humans who have been preparing the schedules for years. Changes to the schedule can create an extensive amount of work if they, as is typical, affect a large number of flight segments.

The response to this situation has been the development of interactive tools on computer workstations to simulate and automate the manual scheduling process. Deckwitz [2] has developed software on graphics-oriented workstations, in which flight segments are represented graphically on the display and can be manipulated with a mouse. Once a system was developed in which aircraft flight schedules could be developed, represented, and manipulated, further steps were taken to incorporate optimization algorithms into this process. Van Cotthem [6] developed a system in which the out-of-kilter algorithm is incorporated into the interactive scheduling workstation; this system was designed to be used at the rescheduling stage.

At this point, it is worthwhile to determine – based on the scheduling systems that have been developed in the past – what characteristics a future scheduling system should possess. Two important characteristics seem apparent. First, a system should have the the flexibility to handle a large group of scheduling problems. In the airline situation, the scheduling process has several distinct phases and there are several resources to be assigned; a scheduling system which was oriented toward use in many of these situations would be desirable. Second, the system should address the problem of how to deal with those constraints and objectives which are difficult to define in standard mathematical programming or mathematical optimization formulations.

Chapter 2

The Representation Of Transportation Schedules

2.1 Reasons For Generalization Of The Problem

Several pieces of software have been developed which solve various schedule optimization problems; much of the code in these programs has been devoted to schedule representation as opposed to schedule optimization. In order to incorporate an optimization algorithm into a scheduling process, it is first necessary to define the appropriate data structures containing information and constraints which serve as input to the algorithm. One then must develop schedule manipulation software (or modify existing software) which will manipulate data in this form. Finally, the schedule manipulation software must be interfaced with the algorithm to be incorporated.

The existence of this large amount of overhead hinders the development of, and experimentation with, schedule optimization algorithms. Ideally, the schedule representation component of the software could be generalized and in turn easily interfaced with prototypes of schedule optimization algorithms. This would require a common data structure which would be used to represent various different types of structure. A generalized data structure to represent transportation scheduling systems is presented herein.

2.2 A Generalized Data Structure For Transportation Scheduling

In examining the different types of scheduling problems we seek to solve, we observe that they all involve scheduling similar objects, and we shall call the object that we are scheduling an *event*. Events possess certain basic components. They occur at a particular time and at a particular location. The third basic component is the type of event, such as an aircraft arrival or the start of a college class.

Beyond the basic pieces of information that define an event, there can be a variable number of auxiliary attributes which provide further information about the event. These attributes might include information about how much the event can be shifted in time, directives to the scheduling process about how other events can be scheduled relative to it, and any other information that might be relevant to the scheduling situation. Because the number of attributes can vary depending upon the scheduling problem being solved (and, within one scheduling situation, depending upon the particular event), it is useful to represent the set of attributes as a list of variable length.

The event data structure can be represented as shown in figure 2.1.

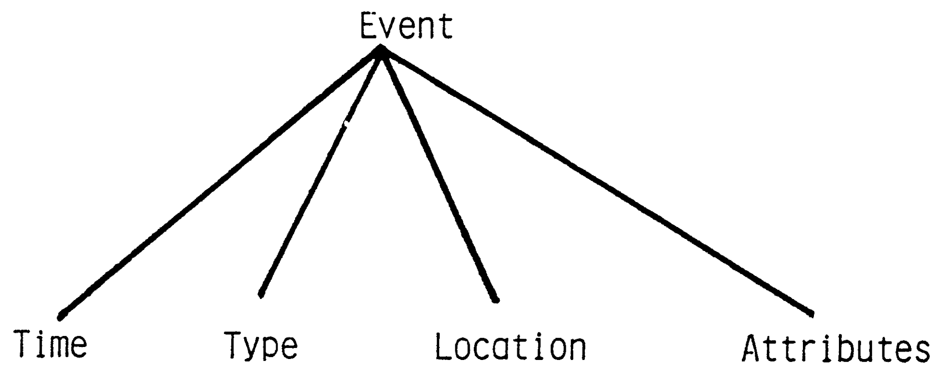


Figure 2.1: The Event Data Structure

The representation of this data structure in the Prolog language is as follows:

```
event(Time,Type,Location,Attributes)
```

where *Time* is simply a numeric representation of the time, Location is a character string representation of the location (for example an airport station name), and *Attributes* is a list, containing a set of pairs of the form

```
[ Attribute_Name , Attribute_Value ]
```

In the field of transportation scheduling, at which this generalized data structure is aimed, events usually occur in pairs. For example, the pair could be the departure and arrival of an aircraft or the beginning and the end of a crew overnight stay. For this reason, we develop the second major basic data structure, called a *link*. Links contain two events; from the point of view of network theory, a link can be thought of as an arc between the two event nodes. In the aircraft flight scheduling example, a link can be thought of as a flight segment. Links can contain attributes which may include directives to the scheduling process. For example, if the two events were an aircraft departure at one station and its arrival at another, the attributes might include the type of aircraft, constraints on passengers or crew or aircraft types, or any other information involving a flight. Attributes can also be used to store the assignments of resources, such as aircraft, to the link. The Prolog language representation of a link is:

```
        link(Event1, Event2, Attributes)
```

where *Event1* and *Event2* are instantiations of the event data structure as described above, and Attributes is a list of the same form as the event attribute list.

Once we have defined these two data structures which we believe can be adapted to scheduling problems in many applications, as we develop algorithms we can develop appropriate ways of grouping the events and links together to optimize the algorithm. However, these two data structures will provide the core on which the scheduling environment, graphical output, user input, and development of algorithms will be based. A large set of scheduling algorithms can be thought of as taking a set of links as input, performing some operations upon the set of links – such as assigning resources to them or modifying the time at which they are scheduled – and returning a new, modified set of links in the same format.

Chapter 3

An Interactive Graphics Scheduling Environment

The data structure that has been described serves as an effective framework for representing events to be scheduled by automated processes. However, it is equally important to have a system which allows these candidates for scheduling to be created and manipulated by a user. The reasons for this are twofold: first, it is necessary to create a set of links as input to an automated scheduling algorithm, and second, it is often important to have an environment in which schedules can be manually manipulated.

As described earlier, a link is represented as a triplet containing two events and a set of attributes. The two events must be instantiated, but the set of attributes has an arbitrary length that could be zero. All the event information that is necessary for the scheduling algorithm fits into this triplet, and therefore it would be quite possible for a scheduler to submit all this data in textual format. One could, for example, submit requests to an aircraft scheduling problem for the following three flights out of station BOS:

```
link(event(0900,D,bos,[]),event(1000,A,jfk,[]),[[pax,85]]).

link(event(1000,D,bos,[]),event(1200,A,ord,[]),[[pax,54]]).

link(event(1100,D,bos,[]),event(1300,A,mia,[]),[[pax,70]]).
```

which are flights to JFK, ORD, and MIA with no event attributes but each containing a link attribute called *pax* which describes the expected number of passengers on the flight. The input data file for the flight links on which a scheduling algorithm will be run will contain a list of objects of this format, with varying types of attribute lists depending on the amount of specific information is to be submitted to the scheduling process.

The challenge for the designer of a graphics oriented system is to provide an environment in which links as described above can be created and manipulated in a way that is much easier and more illustrative of the scheduling situation than a textual list of links. Previous scheduling workstations have addressed this problem using the mouse-based graphics environment of the Apple Lisa microcomputer. Building upon the previous graphics-based scheduling environments, an environment has been developed on the TI Explorer which is a prototype of a flexible system adaptable to many different scheduling problems.

In transportation scheduling, there are two basic pieces of information most crucial to the scheduling of events: time and location. In some situations, it is most useful to look at events from the point of view of what time they occur and in others it is most useful to look at them from the point of view of where they occur. For this reason, the interactive scheduling environment has two main scheduling displays: the Time Chart Display and the Station Display. The

Time Chart Display shows all events to be scheduled on one large time chart, with location information relegated to the role of auxillary pieces of information associated with the graphic depiction of the event. The station display groups events together by location and is therefore useful for isolating the repercussions of the schedule as a whole at individual locations.

3.1 The Time Chart Display

In the time chart display, links are represented as horizontal bars along a metered time chart. While the horizontal positioning of the bars is based on the time at which the events occur, the vertical positioning could relate to a number of different factors, depending on the scheduling application involved.

Using the mouse, a user can access pop-up menus which allow any change to be made to the scheduling display. If the mouse button is pressed when the mouse is on a link object, the pop-up menu with options involving an individual link appears. If the mouse button is pressed at any other point on the time chart display, the main time chart options menu appears.

The main pop-up menu on a blank time chart display is shown in figure 3.1. The first option on this menu allows for the creation of a link object; upon selection of this option a window appears (figure 3.2) which allows the user to describe the time and location information about a link. Once this minimal information has been filled in, the graphic link object appears. No attribute information is necessary for the initial creation of a link. Using the mouse, the user can move the link object to the appropriate location in the schedule, and an attribute whose value represents the vertical position the link appears on the display is set to the appropriate value.

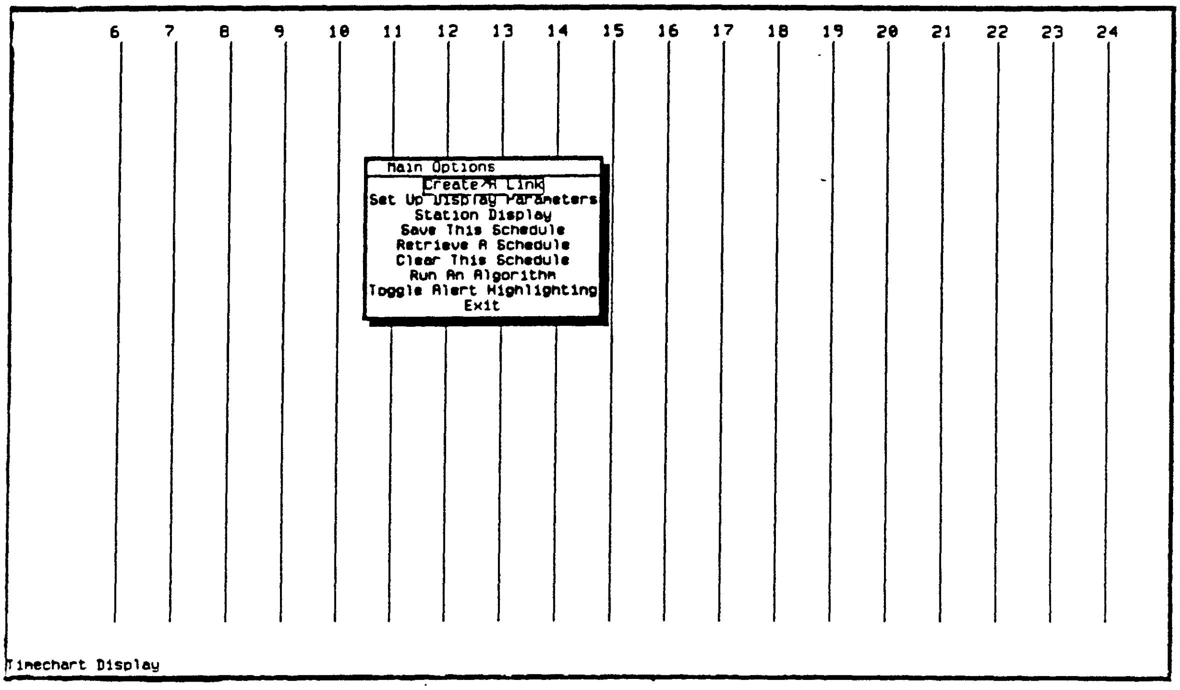
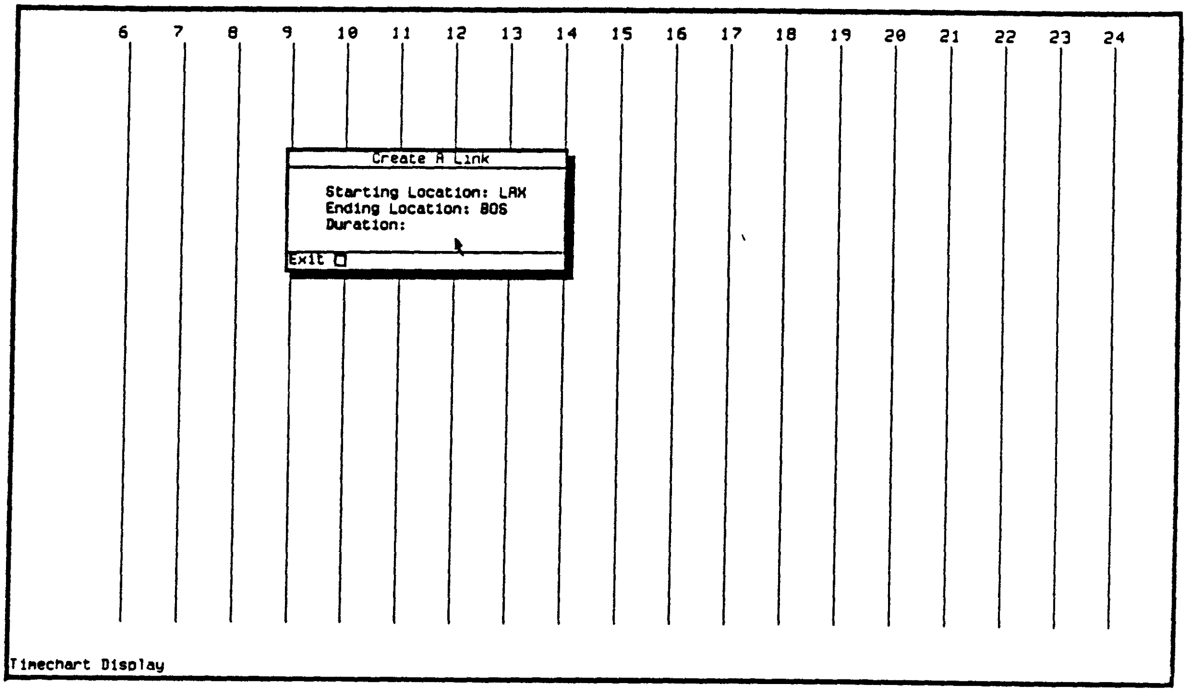


Figure 3.1: Time Chart Display With Main Menu



Starting Location: LAX
Ending Location: BOS
Duration:

Figure 3.2: Link Creation

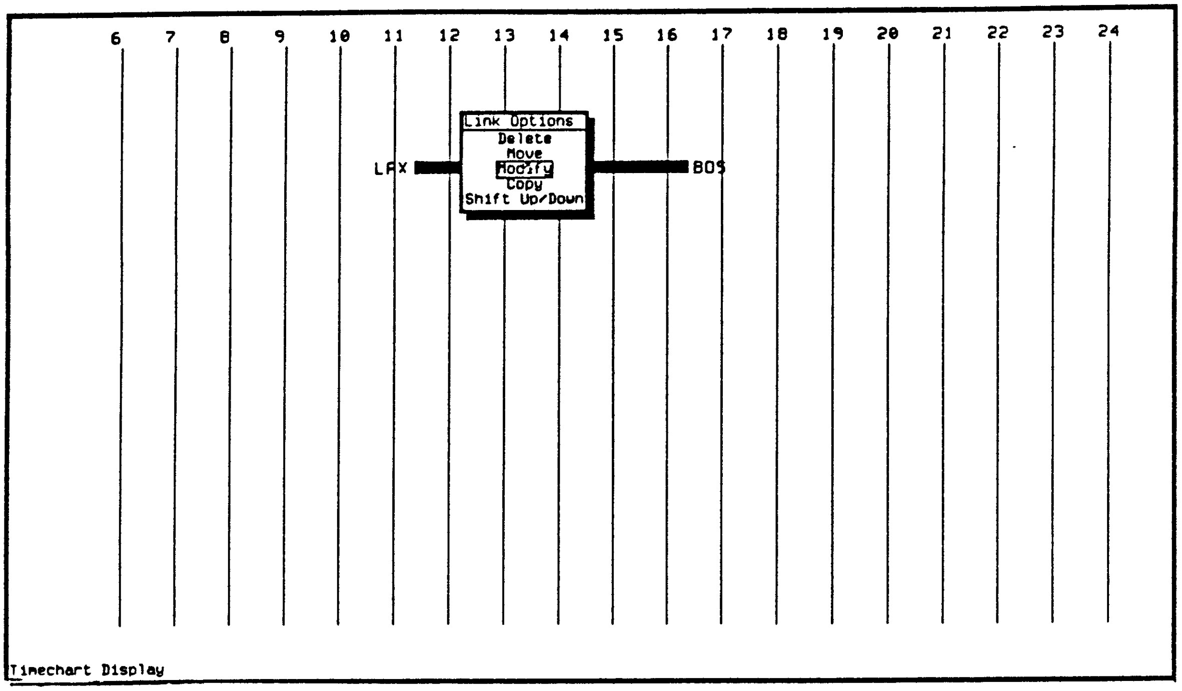
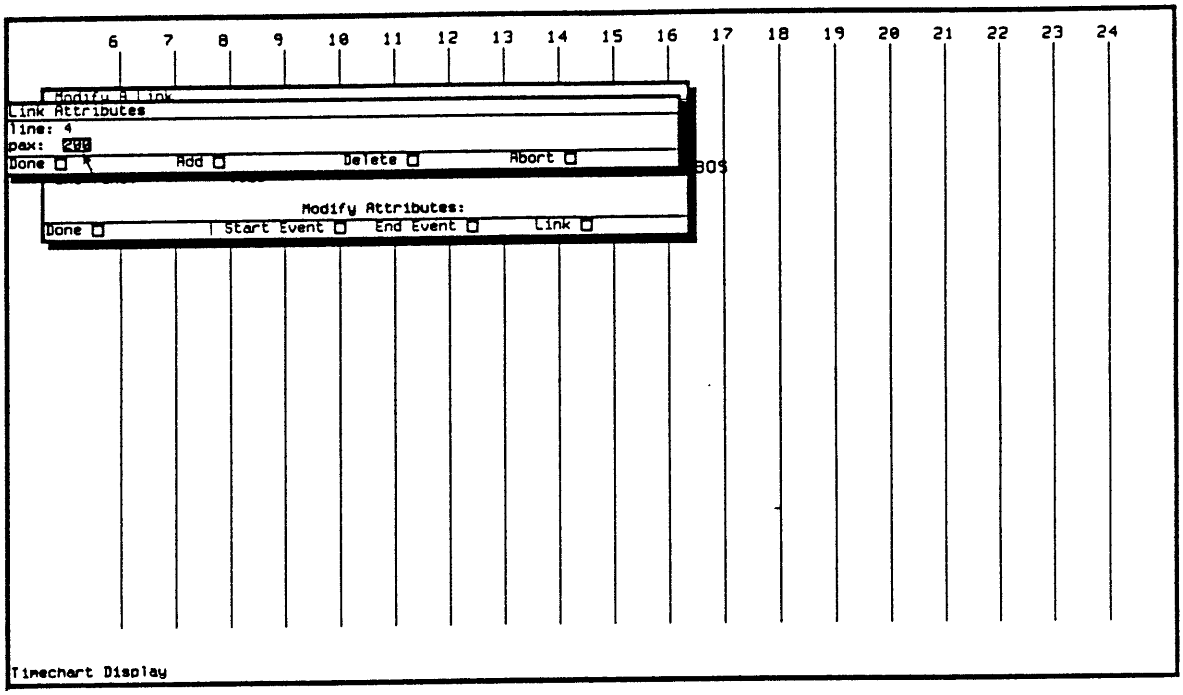


Figure 3.3: Link Options Menu

3.1.1 Link Operations

Once a link object has been created, it can be manipulated with any one of the five options on the link menu (figure 3.3). The Delete option can be used to remove the link from the display, the Move option can be used to change the horizontal (time) position of the link and/or the link's vertical position. When a link is moved to a new position, the values for time and the vertical positioning attribute are updated to their appropriate new values.

The Modify option allows the user to change any piece of information about a link. Selecting the Modify option brings up a window in which the basic information about a link – the time and location information about the link's two events – can be changed. This windcw also includes the options of modifying

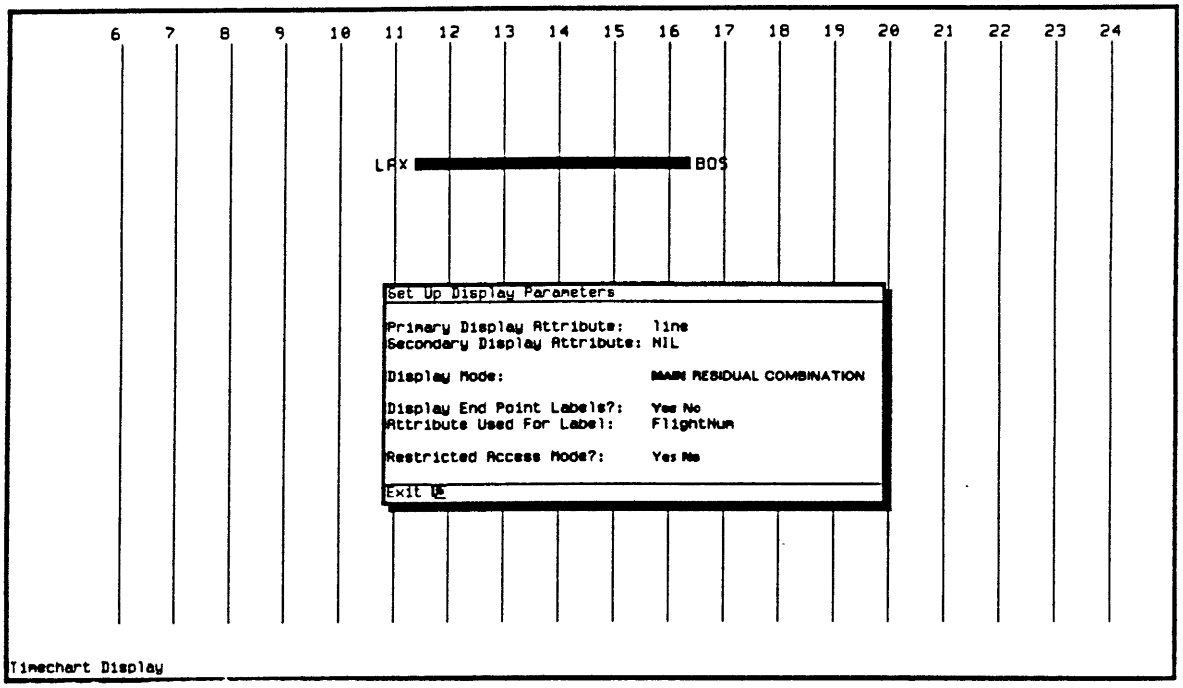


line: 4
pax: 200

Figure 3.4: Attribute List Editing

either of the attribute lists associated with the link's events or modifying the attribute list associated with the link as a whole. If modification of an attribute list is selected, a window appears in which the attribute list is shown and the user can extend it or adjust it in a quick and flexible manner (figure 3.4).

The Copy option allows the user to make a copy of a link object and then use the mouse to move it to a new position on the display. Since, in transportation scheduling, many items to be scheduled are simply identical versions of each other placed at varying locations in a time frame, it is useful to be able to define a link – giving its location, duration, and any set of attributes – and be able to transfer all this information to another link whose only difference might be that its start and end events occur at a different time.



Set Up Display Parameters

Primary Display Attribute: line
Secondary Display Attribute: NIL

Display Mode: [✓ MAIN] RESIDUAL COMBINATION

Display End Point Labels?: [✓ Yes] No
Attribute Used For Label: FlightNum

Restricted Access Mode?: Yes [✓ No]

Figure 3.5: Display Parameters Menu

3.1.2 Schedule Display Modes

Within the timechart display, there is a set of parameters, called *Display Parameters*, which govern how the various attributes of the links are used to display them on the screen. Through the use of the Display Parameters menu (figure 3.5), the user can set up the graphical display in a way that provides flexibility in viewing and manipulating the data.

The first parameter that can be set is the *Display Mode*. There are three display modes: main, residual, and combination. Through the use of these three modes, appropriates displays can be set up for development of schedules, viewing of assignments made by the schedules, and viewing of those items which could not be displayed.

When in the *Main Display Mode*, the vertical position of a link is based on the value of a given link attribute, called the *Primary Display Attribute*. In the system's most basic form, when the user is simply creating links and using the mouse to position them on the display, this attribute represents only the physical position of the link on the display, and in the default case it is called *line*. This attribute is the first attribute to be created in a given link, as it is automatically placed into the attribute list when the link is placed on the display. For example, if the user creates a new link and uses the mouse to place it on the top line of the display, the attribute list for that link becomes:

```
[ [ line , 1 ] ]
```

which is the list whose only element is the pair [line , 1].

If the user runs a scheduling algorithm which assigns a value to a certain attribute for each link (or or a given set of links), the main display mode can be used to show the schedule of assigned links. For example, if the algorithm is an aircraft rotation scheduling algorithm which assigns to a given set of links a rotation number associated with attribute *rotation*, the main display mode can be used to show which links are assigned to which aircraft rotations. Each line of the display will contain all the links assigned to a given resource, and a set of labels in the left margin will display the name of the corresponding resource (in this case, vehicle) to which the links on that line have been assigned.

As an example, figure 3.6 shows a time chart display on which nine links have been entered. This display uses the default set of parameters, which calls for the main display mode with the primary attribute set to be *line*. We now consider the case in which there are two aircraft available, and we attempt to assign these

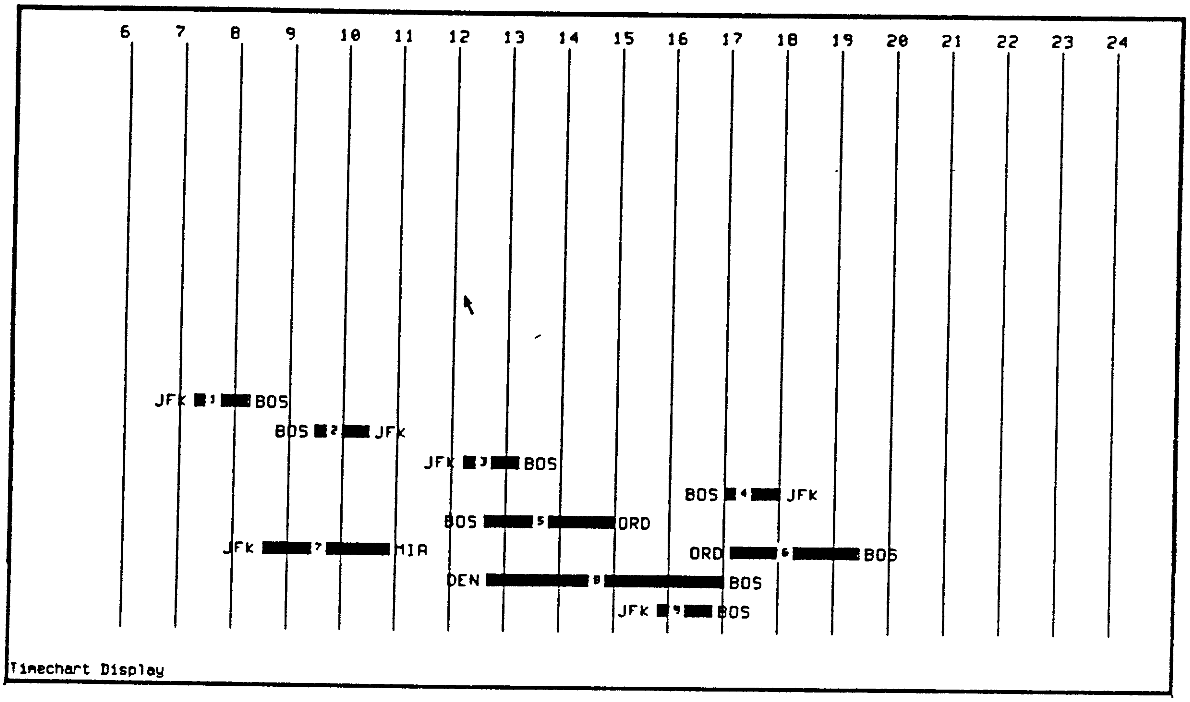
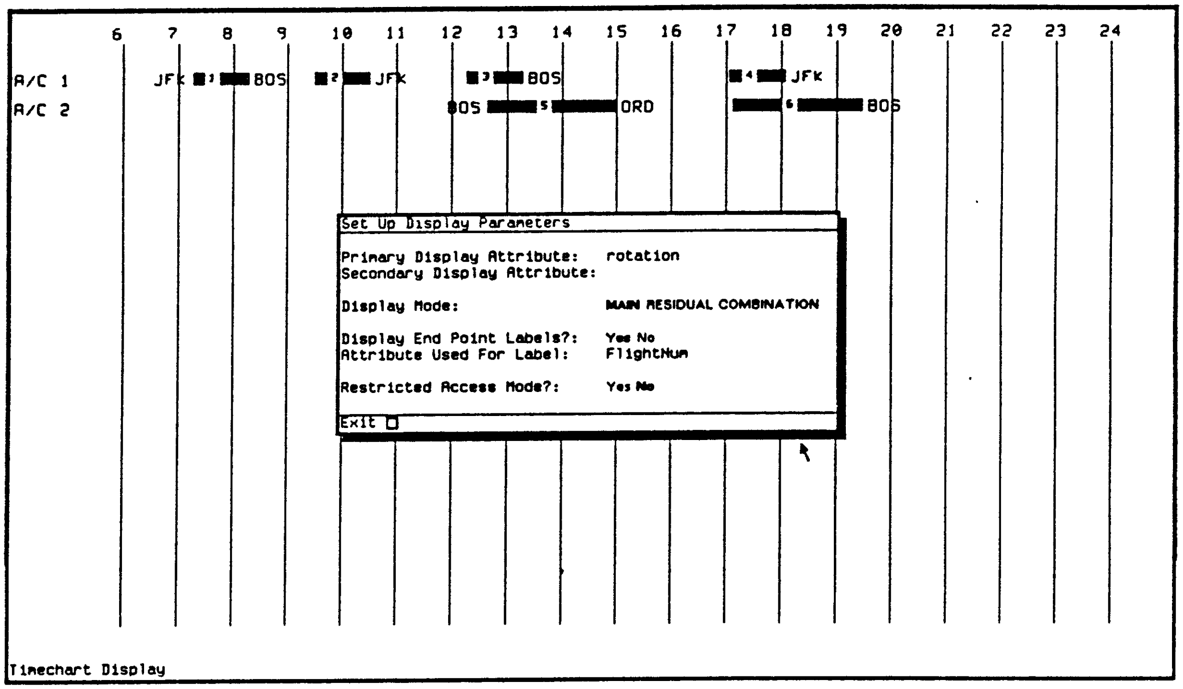


Figure 3.6: Main Display Mode; Primary Display Attribute Is *line*



Set Up Display Parameters

Primary Display Attribute: rotation
Secondary Display Attribute:

Display Mode: [✓ MAIN] RESIDUAL COMBINATION

Display End Point Labels?: [✓ Yes] No
Attribute Used For Label: FlightNum

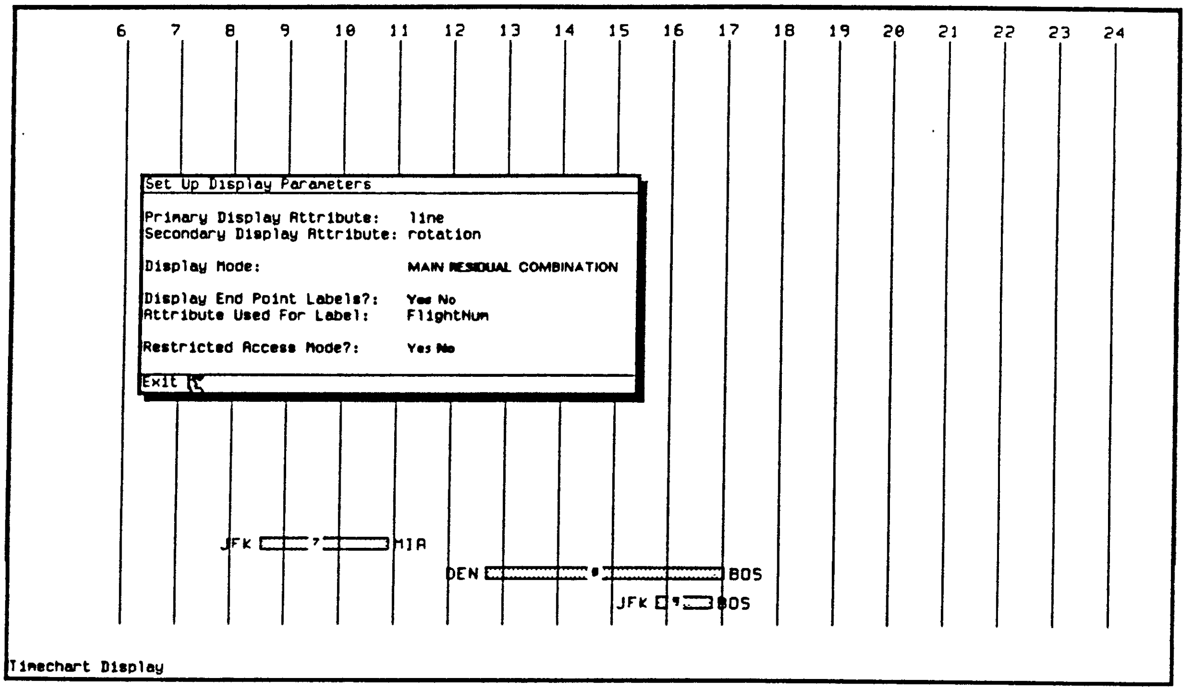
Restricted Access Mode?: Yes [✓ No]

Exit □

Figure 3.7: Main Display Mode; Primary Display Is *rotation*

links to appropriate aircraft. When the assignments have been made, we wish to see which links have been assigned to which aircraft. Thus, we set the primary display attribute to be *rotation*, and display the links in the main display mode. The results are shown in figure 3.7.

However, in addition to the display of the assignments that have been made, another important result of the run of the scheduling procedure is this set of links that have not been assigned to any resource. To look at this set of links, we make use of the ***Residual Display Mode.*** This mode uses the values of two attributes: in addition to the primary display attribute, a ***Secondary Display Attribute*** is now assigned. The primary display attribute again contains the value indicating the position at which the link is to be displayed. However, the



Set Up Display Parameters

Primary Display Attribute: line
Secondary Display Attribute: rotation

Display Mode: MAIN [✓ RESIDUAL] COMBINATION

Display End Point Labels?: [✓ Yes] No
Attribute Used For Label: FlightNum

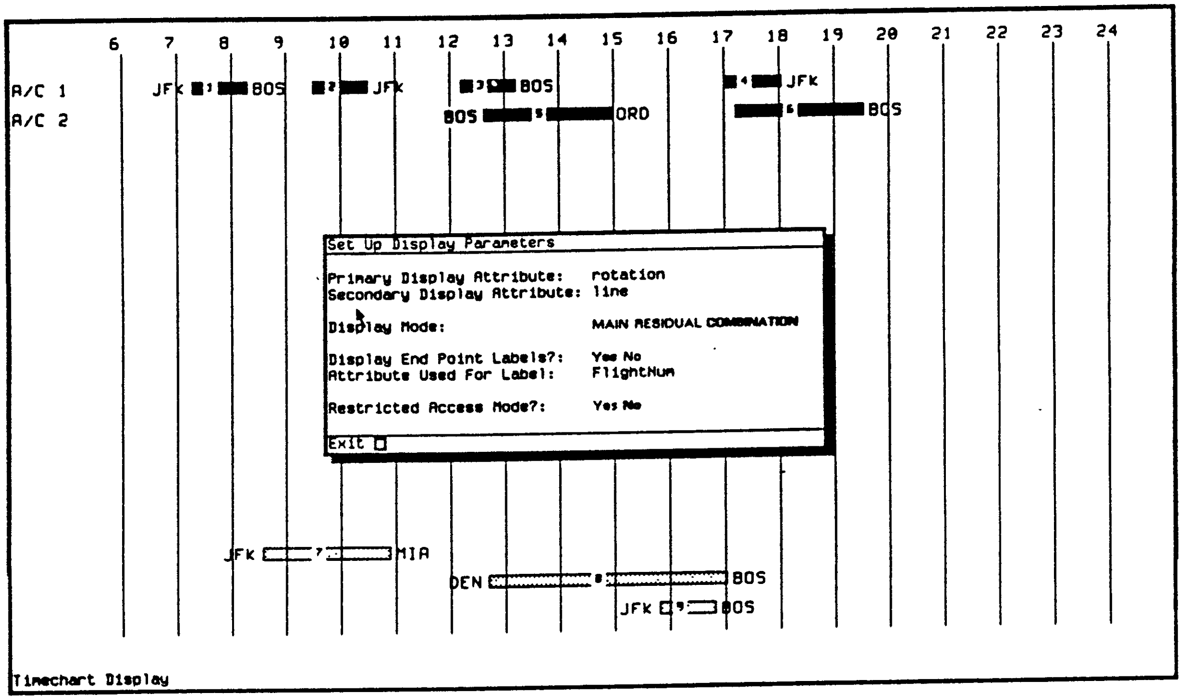
Restricted Access Mode?: Yes [✓ No]

Timechart Display

Figure 3.8: Residual Display Mode; Primary Display Attribute Is *line* and Secondary Display Attribute Is *rotation*

link is only displayed if there has been no assignment made for the secondary display attribute on that link. For example, if we wish to look at the "rejection list" for a rotation scheduling process – the set of links to which no rotation was assigned – we set the primary display attribute to be *line* and the secondary display attribute to be *rotation*. The time chart will then display all links for which no rotation has been assigned, and the vertical position will be determined based on the value of *line*. In nine-flight example, the list of unassigned flights is shown using the residual display mode in figure 3.8.

The third display mode is the *Combination Display Mode*. This mode can be used to see both the list of assigned links and the rejection list on the same



Set Up Display Parameters

Primary Display Attribute: rotation
Secondary Display Attribute: line

Display Mode: MAIN RESIDUAL [✓ COMBINATION]

Display End Point Labels?: [✓ Yes] No
Attribute Used For Label: FlightNum

Restricted Access Mode?: Yes [✓ No]

Exit □

Figure 3.9: All Links Are Displayed Using Combination Display Mode

display. For each link, if there is a value for the primary display attribute, it is displayed with its vertical position determined by the value of that attribute, and if there is no value for the primary display attribute but there is a value for the secondary attribute, it is displayed in another shading with the position determined by the value of the secondary attribute. Using this mode, the schedule can be set up such that the links are developed in one section of the screen, and after the assignments have been made all links that have been scheduled move to the other section of the screen and are shaded differently. The user is then able to manipulate those links on the "rejection list", possibily inserting them into the schedule. An example of the combination display mode in which all of the links in our previous example are displayed is shown in figure 3.9.

In addition to setting the display mode which indicates how attribute values affect positioning and shading of links, this menu can be used to control labeling of links. Normally, a link is displayed as a horizontal bar on the display. There is a display parameters which allows the user to decide whether to display labels on the end points of the link (for a flight link from whose start event is at location BOS and whose end event is at location LAX, it is useful to display these labels, but for a ground link whose start and end locations are the same, it might be more useful and less cluttered simply to display the bar). Another display parameter contains the name of an attribute whose value is used to label the bar. It may be useful in some cases, for example, to label flight bars with the flight number, while it might be useful in other cases to label flight bars with the name of the crew that has been assigned to that flight.

The display parameters menu also includes the option of selecting *Restricted Access Mode.* In certain situations in which schedules are manipulated, it is required that the basic information about times and locations be held constant, while certain attributes, such as resource assignments, are variable. If this is this case, the system can be put into Restricted Access Mode, in which – for any link – only the attribute lists can be modified. By modifying attributes, the user can change the resource assignments or exception calls for individual links. However, by changing the attribute lists alone the user is unable to change a link's time or location in the schedule.

3.1.3 Data File Input and Output

The next two options on the main time chart menu allow the user to save schedules to files and retrieve them from files. Schedules are saved in a form which is simply the textual representation of the data structure described earlier.

The structures which are stored in the data file are thus readily available for use as data to applications in Lisp, Prolog, or any other programming language. In addition, enhancements can be made to the schedules while they are in this data file; as long as the structure of the data remains intact, these enhancements will be preserved when the data is read back in to the interactive environment.

The Save and Retrieve facilities, when used in conjunction with the three available display modes, allow for easy separation of distinct pieces of a schedule. When the Save command is executed, the links which are stored in the data file are only those which are directed to be displayed on the screen by the given display mode. For example, if a rotation number has been assigned to a subset of the links in the system, and the display mode has been set to the main mode with *rotation* as the primary attribute, the links to be stored in the data file after a subsequent Save command would only be those links to which a rotation had been assigned. Thus, a data file containing only the links which are to be served can be kept separate from another data file which might contain only the links which were denied service. Combined with the facility allowing the user to clear the schedule, these options allow for quick manipulation of subsets of links.

3.1.4 The Algorithm Interface

The next option available on the main menu is to run an algorithm. This option provides the link between the graphics scheduling environment and an extendable set of algorithms, which can be written in any language. The logic of this interface is as follows. We begin with a set of links that are shown via the graphics display. When a scheduling algorithm is selected and run, it takes as input this set of links and changes some attributes of those links (for example, to assign a new rotation number to each link). Therefore, running the algorithm results in the creation of a new, albeit similar, set of links. When the execution of the algorithm is complete, the user is given the option of either displaying the resulting set of links on the screen or storing them away to a file.

Because of the straightforward nature of this interface, any piece of software which can take as input a set of links and produce as output another set of links can easily be incorporated into this scheduling system. The algorithms that are implemented, which will be described later, are written in Prolog and in Lisp, but they could be written in any language available on the workstation as long as they conform to this structure.

3.2 The Station Display

In transportation scheduling problems it is quite useful to see the schedule from the dual points of view of location and time. With this in mind, the station display of the interactive environment has been developed. In the station display, a location (or station) is represented as a vertical bar from the top to the bottom of the display, with a label which shows the name of the station at the top of the bar. There can be several stations displayed on the screen at one time. The vertical axis in the station display represents the time of day, starting with the earliest times at the top of the chart and continuing to the latest times at the bottom.

Each link, as we recall, consists of two events – a start event and an end event. In a small scale example, it might be useful to represent the link as a line from the station bar of the start event at the vertical position of the start time to the station bar of the end event at the vertical position of the end time.

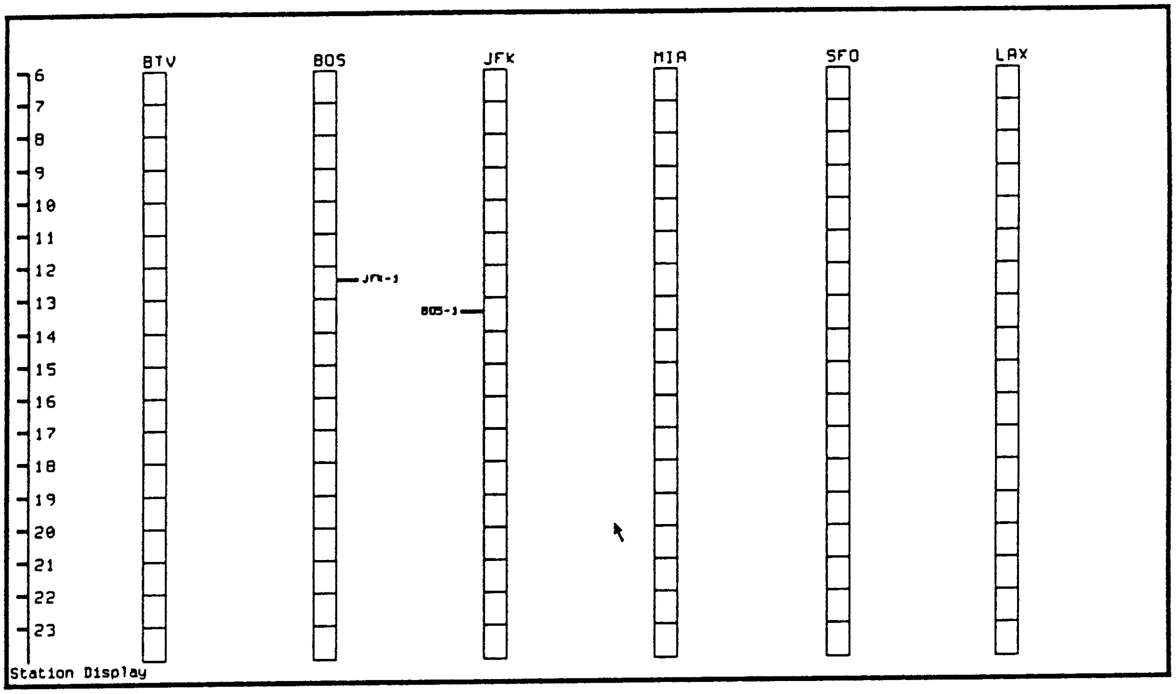
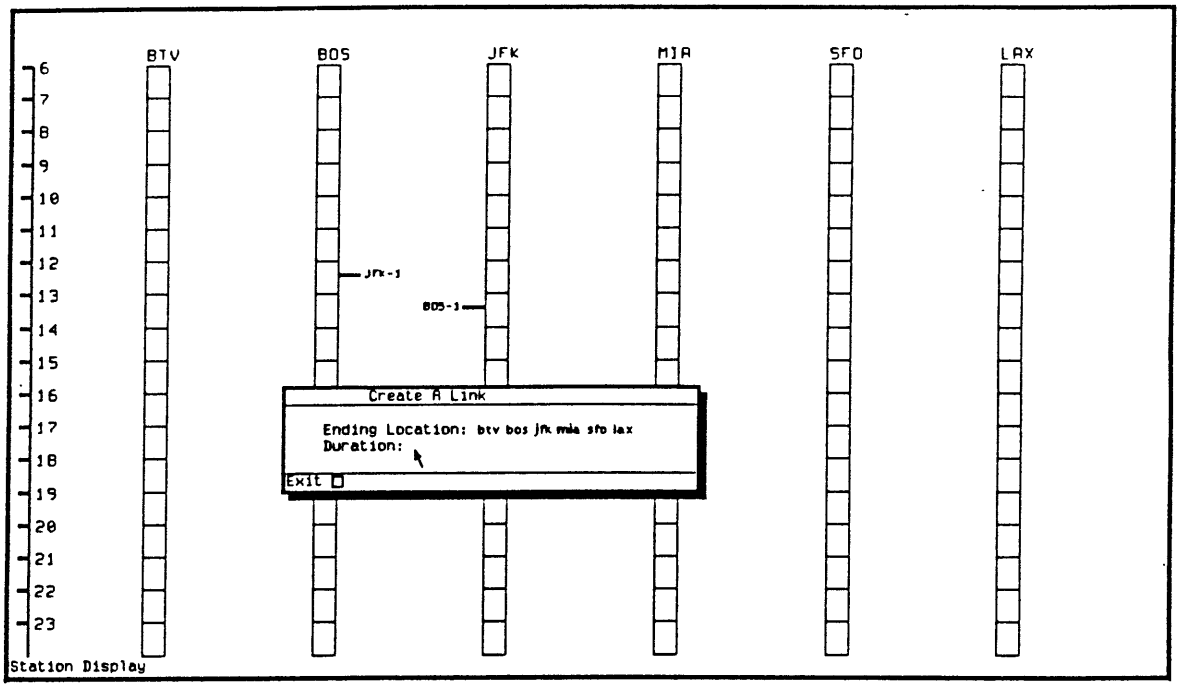


Figure 3.10: A Single Link On The Station Display

In a large, diverse schedule, however, these lines would significantly clutter up the display and would make it so that only a small number of stations and links could be displayed on the screen. We can modify this system to represent a link as two short line segments, one corresponding to each of the link's events. A line segment emanating from the right side of a station bar can represent a start event (in the airline flight case, a departure) and a line segment emanating from the left side of a station bar can represent an end event, each of which has a vertical position corresponding to the appropriate time of day. An example of the station display in which there is a single link departing from Boston and arriving at New York's JFK airport appears in figure 3.10. When the station display appears on the screen, there are three cursor locations at which the mouse button can be depressed to bring up a menu of possible operations. These areas



Ending Location: btv bos jfk mia sfo lax
Duration:

Figure 3.11: Link Creation In The Station Display

are: on one of the station bars, on an existing link event object, or somewhere else on the display.

Pressing the mouse button while the cursor is on a station directs the system to add a link either arriving at or departing from that station. In the station display, the assumption is made that all links created while a given set of station bars are on the screen are links between two of the displayed stations. Thus, when a user chooses to add either a departure from or arrival at a given station, a menu appears in which the other station can be selected from a list of currently displayed stations. Using this facility, a set of links between a subset of stations can quickly and easily be created. Figure 3.11 illustrates the facility for creating a link in the station display.

When a link is created, it is displayed as two separate event objects. To perform an operation on this link, either one of its component line segments can be selected by the mouse. When this occurs, the Link Options pop-up menu appears, with analogous options to the Link Options of the time chart display. However, since this display consists of separate objects representing each of the two component events, it is useful to be able to modify each event separately. This feature is implemented as the "Slide Event" option; when this option is selected, one individual event object can be moved up or down on the schedule without affecting the other component event of its link.

If the mouse button is depressed when the cursor is somewhere other than on a station bar or on a link event object, the main station display menu appears, as shown in figure 3.12. The same main options are available on the station display are are available on the time chart display; saving the schedule from this point creates a data file of the exact same format as saving a schedule in the time chart display, and once saved the schedule can be retrieved on either display. After running an algorithm, seeing its results on the station display is sometimes more appropriate than seeing the results on the timechart display; while algorithms whose results involve assigning links to resources, such as aircraft rotation scheduling algorithms, might be more appropriately displayed on the time chart display where one horizontal line can represent a given resource, when running algorithms whose results involve the positioning of events at given stations, it is more illustrative of the results to show them on the station display.

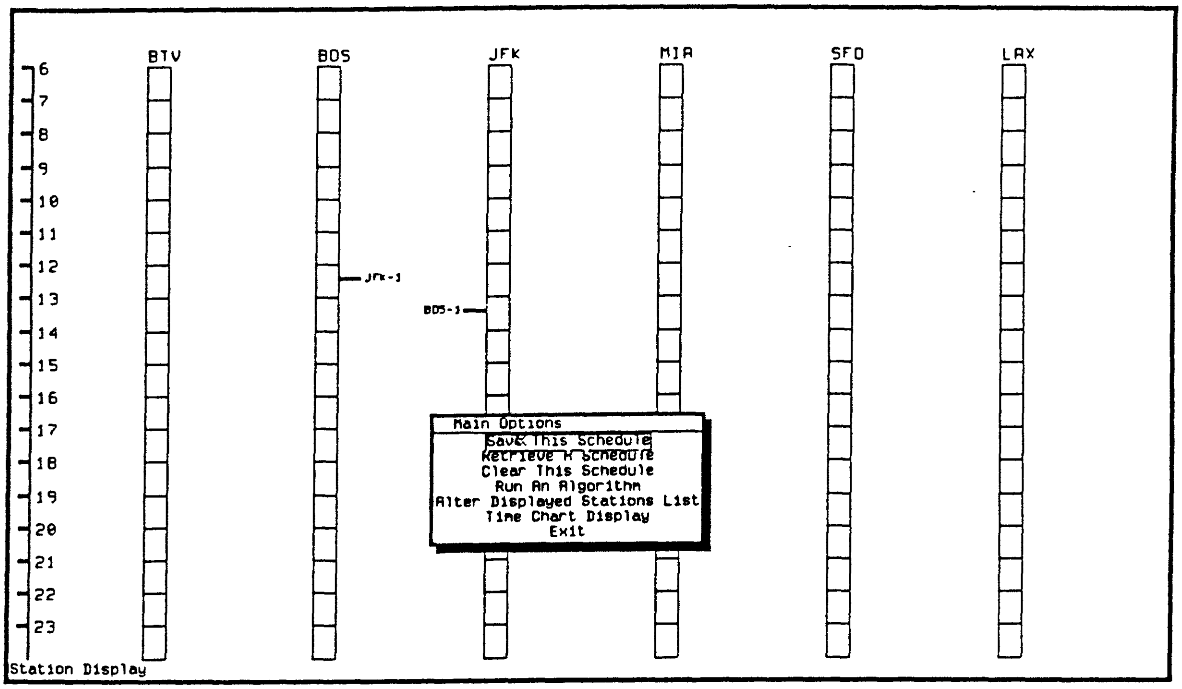


Figure 3.12: Main Station Display Menu

Chapter 4

Dynamic Exception Handling

4.1 Exceptions In The Scheduling Process

Certain deterministic computer-based algorithms have been able to create schedules which bring the usage of resources such as fleets and crews close to an optimal level, but in many cases the algorithms are not able to account for pieces of information crucial to the scheduling process that a human "expert" would be aware of. The result of this is that the schedules that are produced by some computer algorithms, while adequate for the most part, contain pieces that are unacceptable for one reason or another.

The challenge is to incorporate these "exceptions" into the otherwise automated scheduling process. Existing computer-based scheduling systems, while often based on a particular underlying scheduling algorithm, are able to take into account constraints on the scheduling process by incorporating "flags", which signify information which can alter the course of a scheduling process.

For example, we can consider a set of executive jets which for which a daily schedule is prepared to accommodate a set of requests for point-to-point service.

An automated scheduling algorithm which sets out to solve this problem will have a particular goal (such as to maximize the number of requests that are served) and a certain amount of supporting information (such as priorities for each of the requests) that control the order in which requests are served and provides a basis for comparing the "value" of possible schedules. Through an algorithm which attempts to reach the given goal while making use of the supporting information, a schedule can be produced which assigns as many requests as possible to the available resources. For this schedule to be workable, it often must meet certain very specific criteria. For example, a certain executive of particularly high priority might want to be the only passenger on each of his or her flights. Therefore there must be a "flag" on this executive's request, called "no other passengers allowed" that indicates that when this request is assigned to a vehicle, no other requests can be assigned to that vehicle, even if there are several empty seats available. It might also be possible that the executive prefers a particular aircraft in the fleet and would always like to fly on that vehicle if possible. So there might be a "preferred vehicle" flag. Each time the scheduling process is considering making a particular assignment of link to a resource, these flags would have to be checked to make sure all of these requirements, or as many as possible of the preferences, are being met.

Another example of an exception involves maintenance scheduling. There might be a requirement that a certain aircraft return to the base at a certain time of day in order to be serviced. Thus, even if the optimization process dictated that an aircraft be used for service to one point, the maintenance exception might dictate that it be flown to another point.

In actuality, the potential exists for there to be a very large number of these flags. In other words, although the process of assigning links to resources can be based on a general algorithm, there can be a very large number of exceptions to the process. A computer program written to handle this set of exceptions would have to check for the existence of each of these flags when making assignments, and there would have to be appropriate code to deal with each case. The scheduling program would grow in size and complexity as more and more exceptions are uncovered.

It is because of the existence of a very large number of exceptions in many scheduling problems that in many cases the process of scheduling, whether for executive jets or for college classrooms or any other resource, cannot be adequately represented on a computer. It is therefore performed by a human expert, someone who has done the scheduling for years and keeps tracks of all of the idiosyncrasies of and exceptions to the scheduling process.

4.2 A Model For Exception Representation

A solution to this problem involves dividing the job of the automated scheduling system into two basic components: first, the standard algorithm which, given a goal and a sets of resources to assign, has a process for optimally scheduling those resources, and second, the set of exceptions or arbitrary constraints that must be taken into consideration when preparing the schedule. The first piece, the algorithm, is a fixed piece of computer code, written to solve a restricted scheduling problem. The second piece, the set of exceptions, should be dynamic. If the set of exceptions must be hard-wired into the code of the algorithm, it becomes much more difficult for the automated scheduling system to readily adapt to changing situations. If the set of exceptions can be quickly and easily modified and augmented by those experts doing the scheduling, the system becomes more useful for larger set of possible situations.

To visualize how these exceptions will actually be incorporated into a deterministic optimization algorithm, it is useful to think of the decision making process as divided into two parts: "generate" and "test". A given restricted problem might have a finite set of feasible solutions. A deterministic algorithm such as FIFO (first-in-first-out) or LIFO (last-in-first-out) performs the task of determining which of these feasible solutions, or pieces of a solution, is "best", based on a restricted set of guidelines. This determination is the "generate" step.

However, a given solution which is generated by a restricted set of guidelines might be, for some reason, unacceptable. In other words, the simple guidelines that led to the solution might not have taken into account factors which were significant to the result. It is for this reason that the "test" step exists. During this step of the process, the feasible solutions which were generated by the given set of guidelines are checked for their legitimacy.

The Prolog language serves as a useful environment for developing scheduling algorithms which incorporate both of these components. Prolog code is in the form of a set of rules in a database. These rules can consist of a sequential series of instructions, and therefore serve as a program analogous to code written in a language such as Lisp, C, or FORTRAN. Thus, a Prolog program can be written to implement a given scheduling algorithm. However, these rules can also be added, modified, and removed at or before execution time, and it is through this capability that the dynamic set of exceptions can be implemented.

The handling of the exceptions is an important part of the decision making process of the algorithm. The link between the exception handler and the rest of the interactive environment is implemented in such a way as to make the creation and adjustment of exceptions a basic operation in designing a schedule. An exception reference is a form of an attribute. As discussed earlier, attribute lists can come as components of links or they can be components of individual events. Likewise, exceptions can relate to individual events or to links as a whole. An example of an exception relating to an event is minimum ground time. After a particular arrival event, there might be a requirement that the vehicle stay on the ground for two hours before the next departure event. An example of a link exception is the requirement that one particular link be assigned to the same aircraft as a certain other link.

The implementation of exceptions is in the form of a connection between the data in the attribute lists of links and events and rules in the Prolog database. When a particular link (e.g. request for service) has an exception associated with it, there is an item in the attribute list with the name of the exception as its name and a parameter to that exception as its value. For example, the name of the exception might be "Required Connection" and the value might be a flight number, and this exception would imply that the link to which it is attached must be connected on the next leg to a certain other leg with the given flight number.

When an exception is named in the attribute list of one or more links or events, it is necessary to define a simple Prolog rule which handles that exception. These exceptions take the form of "logical predicates" based on a comparison between either two events or two links. For example, the predicate to handle the "Minimum Ground Time" exception, an event exception, would take as arguments the amount of time that the flight must be on the ground and the links involved in the connection. It would succeed if the time difference between the two events involved was greater than the parameter, the specified minimum ground time.

Exceptions fall into two categories: binding exceptions, called *constraints*, and non-binding exceptions, called *suggestions*. Constraints are exceptions which require a certain predicate to succeed before a scheduling decision can be made. For example, if the minimum ground time exception is identified as a constraint, the scheduling procedure would not allow an assignment to be made between two links which did not have the requisite intervening ground time. The search algorithm in the scheduling process would go on to try to find another assignment that could be made in place of that one.

To assign a constraint to a particular event or link, an item is added to the attribute list identified by a "c-" and followed by the name of the rule to be called. For example, if a minimum ground time of 30 is to be a constraint, the attribute name might be:

```
c-min_gtime
```

and the attribute value would be 30. There would then be a predicate in the database called "min_gtime" which would succeed if and only if this ground time condition is satisfied, as described above.

The other type of exception is a non-binding exception called a *suggestion*. When an exception of this sort is placed on a link, the flow of the scheduling process is not altered. The placement of this exception directs the scheduling system to alert the user when the condition noted in the exception is reached. For example, following a given flight link, there might be a suggestion that the aircraft stay on the ground for 45 minutes before the next departure, but this might not be an absolute requirement. To enter this non-binding exception into the scheduling process, the user would add an attribute to the arrival event of the flight link with the name

```
s-min_gtime
```

and the value of 45 minutes. This indicates a suggestion which is tied to the min_gtime rule in the Prolog rule base. If the ground time condition is violated in the course of the resource assignment, this fact is recorded, and stored on the attribute list of the appropriate link or event as an attribute called an "Alert". When the execution of the algorithm is complete, the set of alerts that have been triggered during the scheduling process are highlighted on the interactive scheduling display, and the user is then able to take any appropriate corrective action.

4.3 The Rule Base

A rule, as defined in conjunction with the exception handler, can be thought of as a Boolean function which returns true or false ("succeeds" or "fails") depending on the values of its parameters. We shall define two types of rules: event rules and link rules. Event rules pertain to two events, such as the arrival event for one link and the departure event for another, and succeed or fail based on certain criteria involving those two events. Link rules operate in a similar manner except they can examine the entire link.

4.3.1 The Structure of Prolog Predicates

It is first worthwhile to explain the format of a Prolog predicate. A simple Prolog predicate is a logical statement upon which other logical statements can be based. For example, we might wish to state that an airplane is a type of vehicle. A Prolog predicate representing this fact might be written as follows.

```
vehicle(airplane).
```

which might indicate that there is a class of objects called *vehicle* and one such object is called an *airplane*. We might wish to make further statements, such as:

```
vehicle(hot_air_balloon).
vehicle(car).
vehicle(bicycle).
vehicle(helicopter).

has_an_engine(car).
has_an_engine(airplane).
has_an_engine(helicopter).
```

We now have created two categories, called *vehicle* – which is the class containing objects which are vehicles, and *has_an_engine* – which is the class containing objects which have engines. We have also defined some objects which fall into those categories. We might also wish to assert that there are some objects which can fly or be flown. To do this, we can make the assertions:

```
can_fly(helicopter).
can_fly(airplane).
can_fly(hot_air_balloon).
```

In all of the above Prolog statements, we have made explicit statements unrelated to any other logical inferences. Now, if we want to use these statements to make other logical inferences, we can use conditional Prolog predicates. For example, the following is a predicate defining an engine powered aircraft.

```
engine_powered_aircraft(Object) :-
        vehicle(Object),
        has_an_engine(Object),
        can_fly(Object).
```

The above statement says that a given object is an engine powered aircraft if that object is a vehicle, has an engine, and can fly. The ":-" in the above lines represents the "if" statement. We can think of *engine_powered_aircraft* as a rule, and "Object" as the parameter to the rule, the value of which will determine whether or not the rule succeeds. A rule can have any number of parameters, and can call any number of other rules to determine whether or not it succeeds. Parameters do not need need to be instantiated (contain values) when the rule is called; the value of the parameter will be provided during execution. For example, calling the rule

```
engine_powered_aircraft(Object).
```

will succeed, and define Object to be a particular value such that the logical statement that Object is an aircraft will hold true.

4.3.2 Exception Handler Predicates

A rule to be used in conjunction with the exception handler is a Prolog predicate with three parameters. In an event rule, two of the parameters are events; in a link rule, two of the parameters are links. The other parameter in an exception handler rule is a value which is used to evaluate the rule.

As an example, consider the rule to determine whether the minimum ground time between an arrival and a subsequent departure has been met. This rule can be defined as follows:

```
min_gtime(Min,EventA, EventD) :-
        event_time(EventA,TimeA),
        event_time(EventD,TimeD),
        time_difference(TimeD,TimeA,GTime),
        GTime >= Min.
```

In this case, Min refers to the number of minutes that an aircraft is required to be on the ground, EventA refers to an arrival event, and EventD refers to a departure event. The *event_time* function, when passed an event in the first parameter, returns the time at which the event occured in the second parameter. The *time_difference* function, when given two times, returns the number of minutes between them. The last line indicates that the entire rule will succeed if the ground time, as determined in the previous lines of the rule, is equal to or exceeds the specified minimum. If this is not true, this particular test will fail. Link rules are defined in a similar way, except the data provided to them includes everything about two entire links instead of just two particular events.

4.4 An Aircraft Rotation Scheduling Algorithm Which Incorporates Dynamic Exception Handling

The selection of aircraft rotations is an interesting problem in which to demonstrate the use of dynamic exception handling. Given a set of flight segments, we are faced with – for each station involved – a set of flights which arrive at that station and a set of flights which depart from that station. For each departing flight, we wish to determine which aircraft should be used; the flights that arrived and became "ready" to use prior to that flight and have not yet departed will provide a set of aircraft from which to choose.

The situation facing the rotation scheduling algorithm for a given station is pictured in figure 4.1. In this case, there are three arrivals into the station and three departures from it. The rotation scheduling process must generate a set of "turns" – matchings between incoming flights and outgoing flights. After each arrival, the number of aircraft available at the station is indicated. Between any pair of zeros (times in which there are no aircraft available at the station), there will be a "cluster" of positive numbers. Within this cluster, there are a finite number, NP, of ways to create a "turn pattern". This finite number is given to be the product of all the positive numbers; in this example NP = 1 * 2 * 2 = 4.

There are a number of different strategies for determining which cluster of matchings should be made. The LIFO (Last-In-First-Out) strategy, in which departures would be matched with the last incoming arrival, would lead to longer

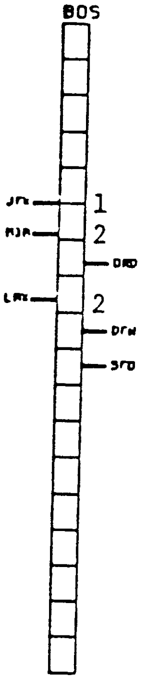


Figure 4.1: Arrivals And Departure At Station BOS

ground times for the aircraft which might be desirable for maintenance and servicing. The FIFO (First-In-First-Out) strategy would lead to more uniformly distributed ground stays. In practice, there are many factors which dictate which solutions are better than others. So, while FIFO might generally lead to a reasonable solution, there are several factors that could make a given solution more desirable than another. These factors might include maintenance scheduling requirements, crew scheduling constraints, or any arbitrary reason that a given incoming flight should be matched with a given outgoing flight.

4.4.1 A Description Of The Algorithm

A First-In-First-Out rotation scheduling algorithm has been developed and interfaced with the interactive scheduling environment. An system of dynamic exception handling has been incorporated into this implementation. A description of this implementation follows.

Organization Of The Data

There are three sets of data that serve as the input to the aircraft rotation scheduling algorithm. First, there is a set of links, as generated in the interactive scheduling environment. Second, there is a set of available aircraft to which to assign to the links. Third, there is the name of the attribute which will – in each link – contain the tail number of the vehicle to which the link will be assigned.

The first step in the rotation generation process is to organize the data into a set of arrival and departure lists for each station. The program iterates through the list of links. When each link is processed, a pointer to the link is inserted on two separate ordered binary trees – the tree representing the departure list at station from which the link departs, and the tree representing the arrival list at the station at which the link arrives. When each link has been processed, the ordered binary trees are converted to sequential arrival and departure lists.

Generation Of Turns

When all arrival and departure lists have been created, a set of turns can be determined for each station. At this point we are concerned only with matching incoming links to outgoing links, not with assigning those links to vehicles. We begin the process of assigning turns for a station by examining the arrival and departure lists. If either the departure list is empty or the arrival list is empty, we stop because all possible assignments have been made. Otherwise, we select the first link on the arrival list and the first link on the departure list and attempt to make a turn. These two links become our "candidates" for a match.

The Testing Process: Consultation Of The Rule Base

We have generated a possible matching; it is now necessary to test to see if the matching is legal. This testing process involves checking each associated constraint to see if any violations exist. If at any point in this process a reason is found that the matching cannot be made, this process "fails" and the generation of another turn can be attempted.

The testing process begins with the "default" set of tests. This is the set of rules to which all turns must conform. These might include a set of "default event rules" and a set of "default link rules". An example of a default event rule might be that for any turn to be valid, the departure event must occur at least thirty minutes after the arrival event. If this were the only default rule, then the default predicate would be listed as follows:

```
        default_event_rules(EventA,EventD) :-
                min_gtime(30,EventA,EventD).
```

This indicates that, given arrival event *EventA* and departure event *EventD*, *default_event_rules* will succeed if the minimum ground time rule succeeds given those two events and a value of 30. The *default_link_rules* predicate would be referenced after *default_event_rules*, and is defined similarly.

Once the default rules for a given turn have been satisfied, the next step in the testing process is to check for exceptions. The first exceptions that are sought are any exceptions that might be tied to the arrival and departure events. A function called *make_constraints_and_suggestions_lists* is applied to the attribute list of each event, producing the list of exceptions associated with each event. These exceptions are then examined in the following order:

1. Check constraints in the Arrival Event
2. Check constraints in the Departure Event
3. Check suggestions in Arrival Event
4. Check suggestions in the Departure Event

The list of constraints in the arrival event, we recall, are the list of those attributes of the arrival event which were preceded by the symbol *c-*. Following this symbol is the name of the rule to be called, and in the value field of the attribute is the value with which to call the rule. In *check_event_constraints*, we convert each constraint entry in the attribute list to a reference to a Prolog rule, calls it with the associated value, and only succeed if each rule that is specified is satisfied. For example, if one of the constraints on the arrival event's attribute list specified that following the arrival, the aircraft must remain on the ground for 120 minutes, and the constraint was listed as follows:

```
[ c-min_gtime , 120 ]
```

then one of the predicates which must be satisfied for the test as a whole to succeed would be the call to

```
min_gtime(120,EventA,EventD).
```

As specified in steps (1) and (2) above, the *check_event_constraints* rule is called for the attribute lists of both the arrival event and the departure event; any rule listed in either event's attribute list is called with reference to both events.

If any of the above constraints have not been satisfied, the attempted match of the two links will fail, and the testing process will halt. If everything has been satisfied at this point, a check is made for non-binding constraints (suggestions). At this point *check_event_suggestions* is called. Because suggestions are non-binding, this call will always succeed; the only action that will be taken during its execution would be to generate "alerts" and add them to the appropriate attribute list.

The *check_event_suggestions* step is similar in structure to its constraint-checking counterpart, *check_event_constraints*. The difference lies in the course of action taken when the rule specified in the attribute list is not satisfied. When the constraint rule was not satisfied, the entire process failed and was terminated. When a suggestion rule is not satisfied, an "Alert" is generated. The attribute list in which the suggestion was specified is augmented with a new attribute. The name field of this new attribute is set to be "Alert" and the value field is set to be the name of the rule which was not satisfied. For example, if the initial attribute list contained the suggestion

```
[ s-min_gtime , 120 ]
```

and subsequently the predicate

```
min_gtime(120,EventA,EventD)
```

was not satisfied, then a new attribute would be generated, as follows:

```
[ Alert , min_gtime ]
```

This is an attribute whose name is "Alert" and whose value is *min_gtime*.

When the four steps have been taken to check the event attributes of the two candidates for a match, the same four steps are then taken to check the link attributes. The difference is that the information that is used by the exception-handler mechanism now refers to the link as a whole. For example, if there is a constraint on the link which is incoming to the station which indicates that it cannot be matched with an outgoing link headed for destination X, the information required by the exception handler involves not only the departure event of the outgoing link, but also its subsequent arrival event. Once again, if any constraint called specified by a link attribute fails, the entire process is terminated. If a suggestion made in one of the two link's attribute lists is not satisfied, the alert which is generated is placed on the attribute list of the link which contained the suggestion.

Once the testing process is complete, one of two results has been reached: either the match has succeeded or it has failed. If the match succeeded, both the incoming link and the outgoing link involved in the turn are removed from the event list, and the process continues with the next events on the lists. If the match failed, further matches are attempted with subsequent events at the station.

Assignment of Vehicles

When all possible turns have been made at a given station, the process resumes at another station. When every station has been processed, each link on the event lists will contain – on its attribute list – an entry for the rotation number. This will not be an actual tail number, but rather a pointer to an uninstantiated value. In other words, all that is known is which links are assigned to the same rotation as which other links. The next step is to assign tail numbers to the rotations.

The list of links is traversed in order of time, with the link whose departure event occurs at the earliest time processed first. A tail number from an aircraft available at the station from which this link departs is assigned to that link; this assignment in turn instantiates the rotation number attribute of all links which have been assigned to the same vehicle. We then turn to the next link. If the rotation attribute for this link has already been instantiated, no assignment is made; otherwise, the instantiation is made as before. The process continues until the rotation number attribute for each link has been instantiated. At this point, the rotation assignment process is complete.

4.4.2 Examples Of The Rotation Assignment Process

As an example of aircraft rotation scheduling, let us consider a small example in which we wish to assign six flight legs to appropriate aircraft. We assume that there is a set of aircraft available at various stations around the system, and that we wish to minimize the number of those aircraft which are actually used. The time chart representation of the six flight legs we shall consider is shown in figure 4.2. Examining the station display representation of these six flight legs, in figure 4.3, we quickly see that the issue at hand involves generating "turns" for station BOS. There are three flights coming in to the station, and three flights departing from it. To minimize the number of aircraft used, we wish to match incoming flights to BOS to corresponding outgoing flights from BOS.

The standard First-In-First-Out scheduling rule would dictate the following matching:

1. Match the first flight into BOS (flight #1 from BTV) to the first flight out (flight #4 to JFK).
2. Match the second flight into BOS (flight #2 from JFK) to the second flight out (flight #5 to SFO).
3. Match the third flight into BOS (flight #3 from MIA) to the third flight out (flight #6 to MIA).

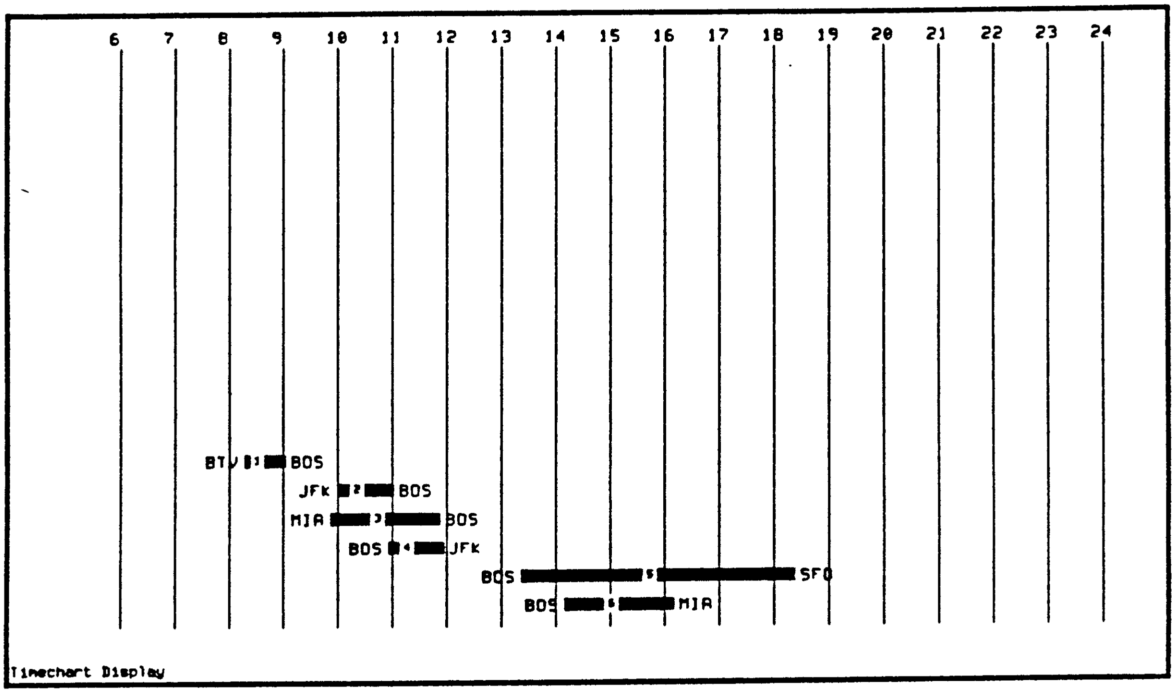


Figure 4.2: Time Chart Representation Of Six Links To Be Assigned To Aircraft

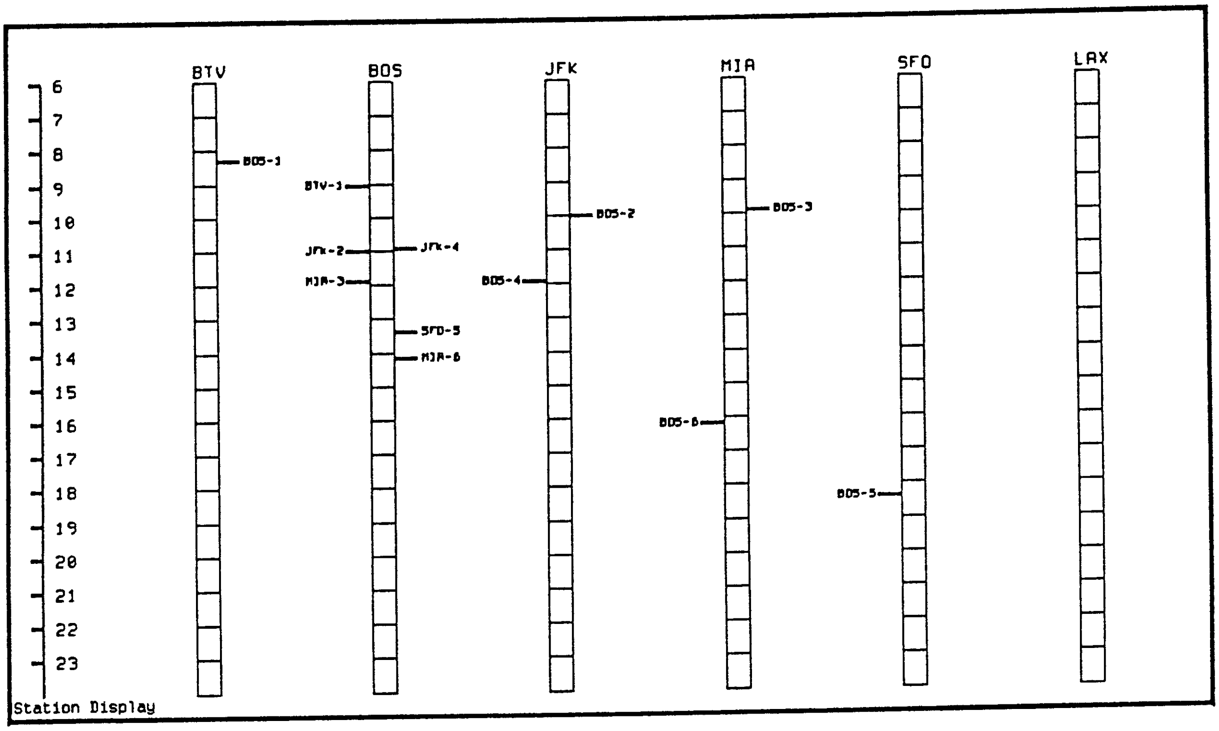


Figure 4.3: Station Display Representation Of Six Links To Be Assigned To Aircraft

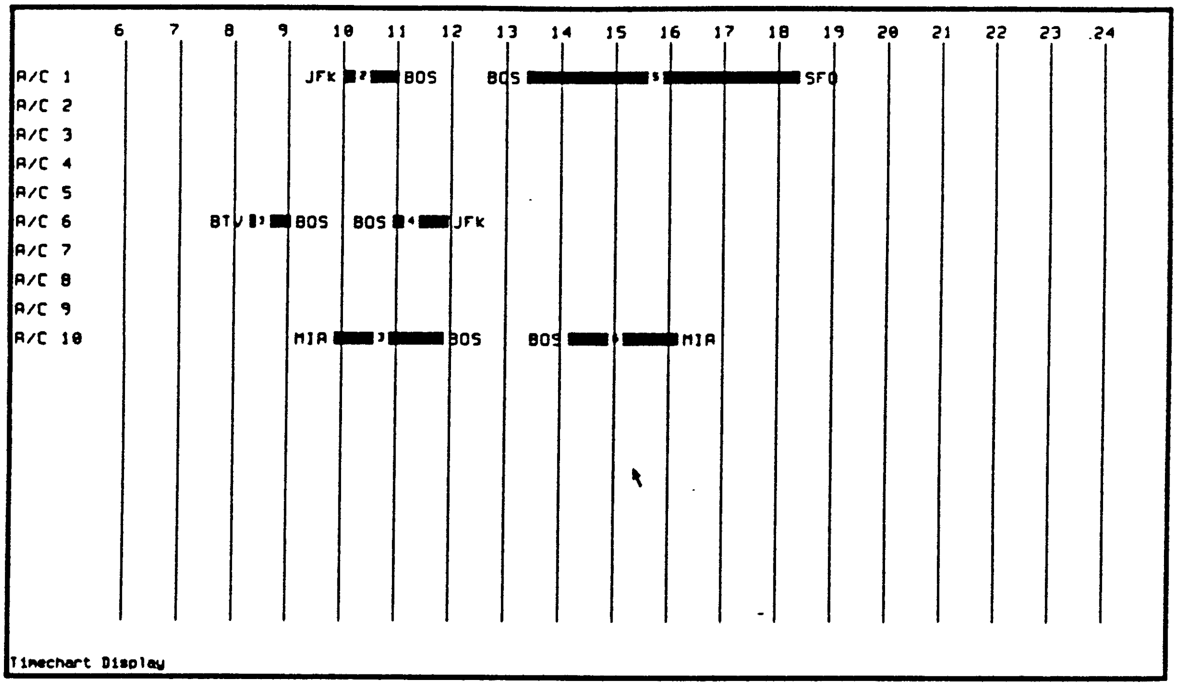
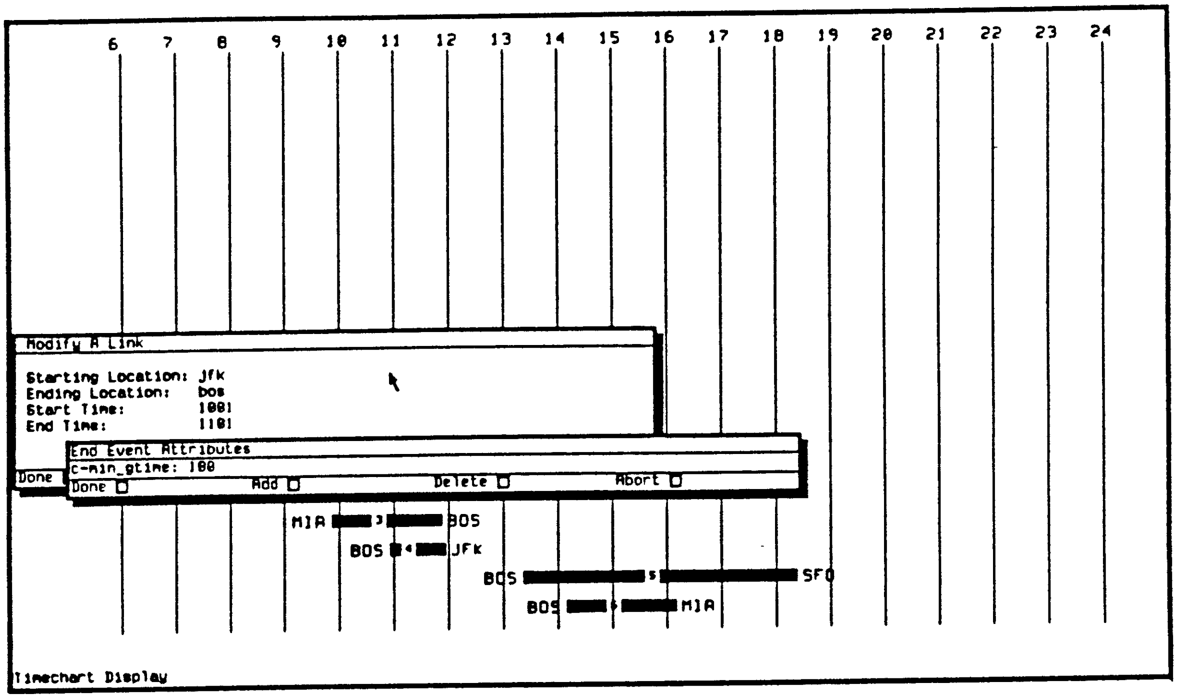


Figure 4.4: Vehicle Assignments Made By FIFO

Running the FIFO algorithm within the interactive environment, and then assigning the resulting flight sequences to aircraft selected from the ten available aircraft, produces the aircraft assignment display shown in figure 4.4.

Suppose, now, that there was some reason that flight #1, from BTV to BOS, should stay on the ground at BOS for much longer than the default minimum ground time of 30 minutes. This requirement could, for example, involve a maintenance operation that must be performed at BOS. Regardless of the reason for this requirement, it is an exception to the scheduling process. To indicate that this exception exists, we press the mouse button with the cursor on flight #2, and pull up the Modify menu. From this menu, we select the option to modify the End Event Attributes (indicating that following the arrival event,



Starting Location: Jfk
Ending Location: bos
Start Time: 1001
End Time: 1101

c-min_gtime: 100

Figure 4.5: An Exception Is Placed On Flight #2

there must be a certain amount of time, for example three hours, before the following departure event occurs). When we select the "add" option, we can enter the attribute *c-min_gtime* with a value of 180 minutes. The attribute list display after this addition is shown in figure 4.5.

Following the placement of this exception, we again run the algorithm. The first candidate for matching is the pair of the first arrival (flight #1 from BTV) with the first departure (flight #4 to JFK). This match succeeds, and the turn is created. The next candidate for matching is the pair #2 from JFK and #5 to SFO. When the arrival event for flight #2 is checked against the departure event for flight #5, the minimum ground exception rule associated with flight #2's arrival event is triggered. The predicate is referenced, and because there is

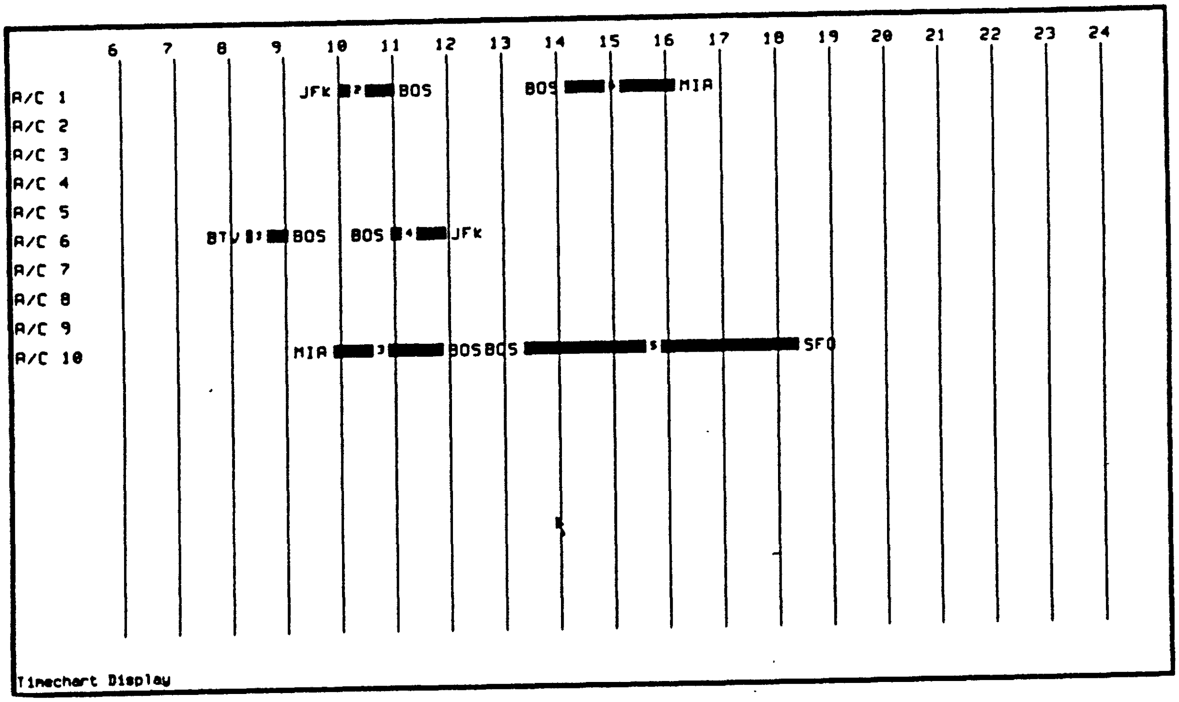
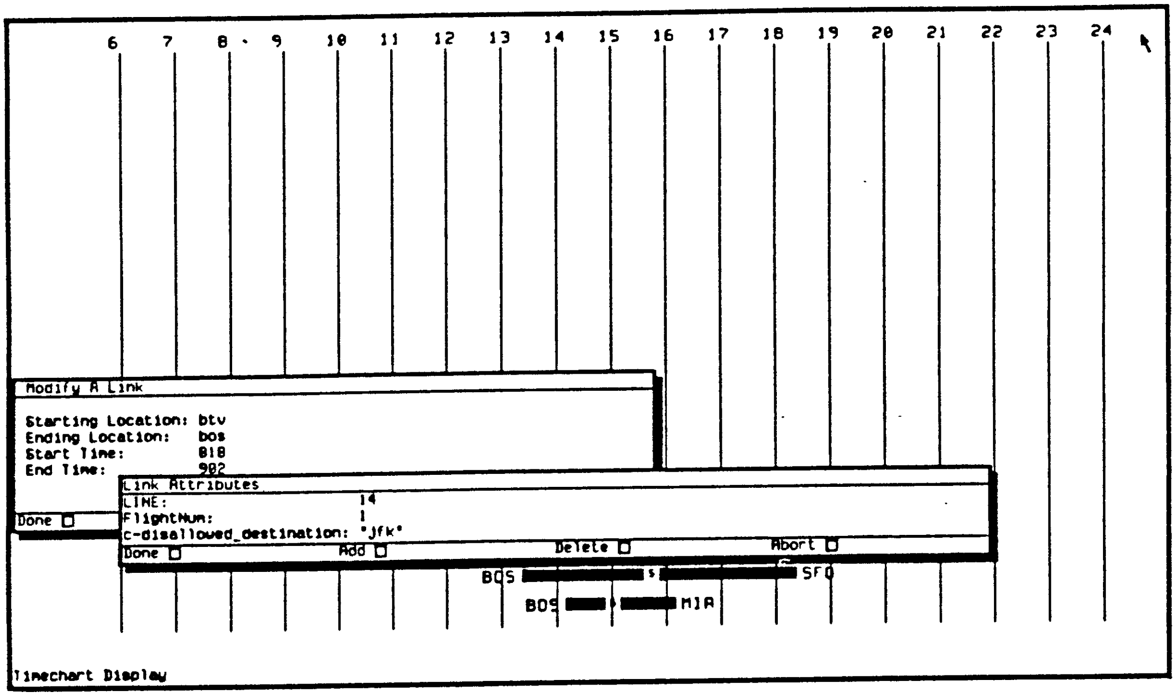


Figure 4.6: Vehicle Assignments With Exception On Flight #2's Arrival Event

not a ground time of 180 minutes between the arrival event and the departure event, the predicate fails. Thus, this match cannot be made.

The process continues with the same arrival and the next departure on the list, flight #6 to MIA. Once again, the exception handler is triggered, but this time the predicate succeeds, because there is an ample amount of ground time between flight #2's arrival event and flight #6's departure event. This match goes on to succeed, and the third arrival, flight #3, is then legally matched with the remaining departure, flight #5. The vehicle assignment display resulting from this execution is shown in figure 4.6

Another example of an exception is a disallowed destination. For a given link, there might be a rule that states that the link may not be connected to a



Modify A Link

Starting Location: btv
Ending Location: bos
Start Time: 818
End Time: 902

Done □

Link Attributes

LINE: 14
FlightNum: 1
c-disallowed_destination: "jfk"

Done □ Add □ Delete □ Abort □

Figure 4.7: Placement Of A Link Exception On Flight #1

flight leg which is headed for a particular destination. This exception is a link exception as opposed to an event exception, because the information required for evaluating the exception involves more than merely the arrival and departure events. The placement of this link constraint, on flight #1, is shown in figure 4.7. This exception states that flight #1 cannot be connected to a link which arrives at station JFK.

Following the placement of this exception, the algorithm is again executed. When the first turn pair – flight #1 and flight #4 – is examined, the link exception is triggered on the link for flight #1. The predicate is not satisfied and thus the match fails, and the next departure, #5, is legally paired with arrival #1. The second arrival cannot be matched with flight #4 because flight

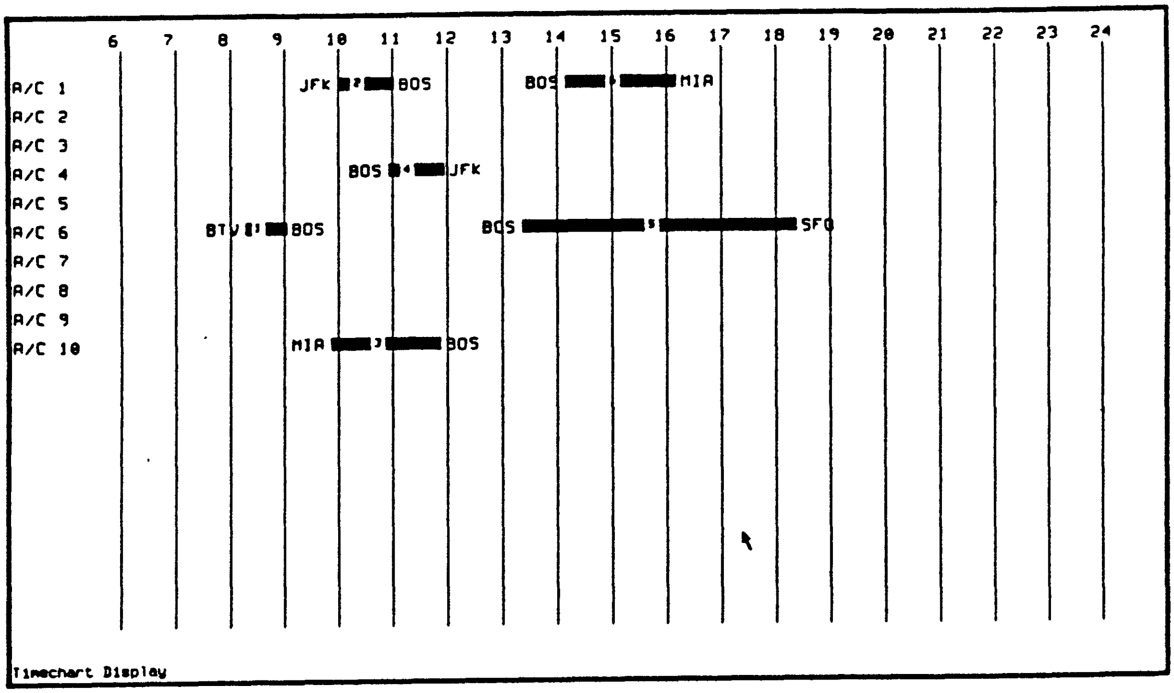
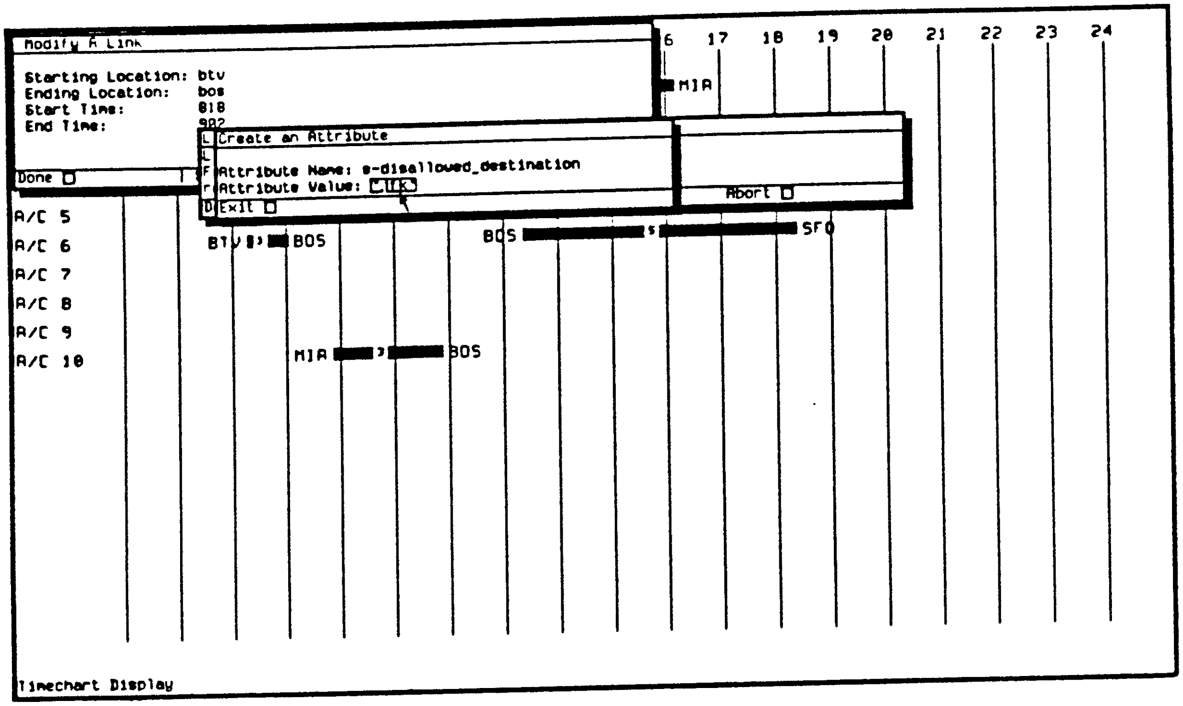


Figure 4.8: Vehicle Assignments With Link Constraint On Flight #1

#4 departs before flight #2 arrives, and thus the default minimum ground time rule fails. Thus, flight #2 is matched with the next unassigned departure, flight #6. The remaining unassigned departure cannot be assigned to the remaining unassigned arrival, so no more assignments are made, and one more aircraft is required to serve these flight legs than in the previous example. The vehicle assignments are shown in Figure 4.8

We now consider non-binding constraints, called *suggestions*. It is possible that some of the exceptions, such as the previous one, merely reflect desirable situations but in – in certain cases – do not necessarily have to be adhered to. It might, however, be important to at least know exactly when these exceptions are violated. For example, in the previous case, the disallowed destination might



Modify A Link

Starting Location: btv
Ending Location: bos
Start Time: 818
End Time: 902

Create an Attribute

Attribute Name: s-disallowed_destination
Attribute Value: [illegible]

Figure 4.9: A Link Suggestion Is Placed On Flight #1

not be a binding constraint. It therefore can be stated as a suggestion, as shown in figure 4.9. When the algorithm is executed with the link exception on flight #1 stated as a suggestion instead of as a constraint, the assignments are made as they were in an unconstrained environment. However, during the matching of flight #1 with flight #4, the suggestion exception is triggered, and an "Alert" is placed on the link in which the exception was included, flight #1. Following the execution of the algorithm, the Alert on flight #1 is indicated with a pointer, as shown in figure 4.10.

Once a optimization program has been executed, there may be a number of alerts that are highlighted with a pointer. To examine the reason for the placement of an Alert, the user simply uses the attribute editing feature to look

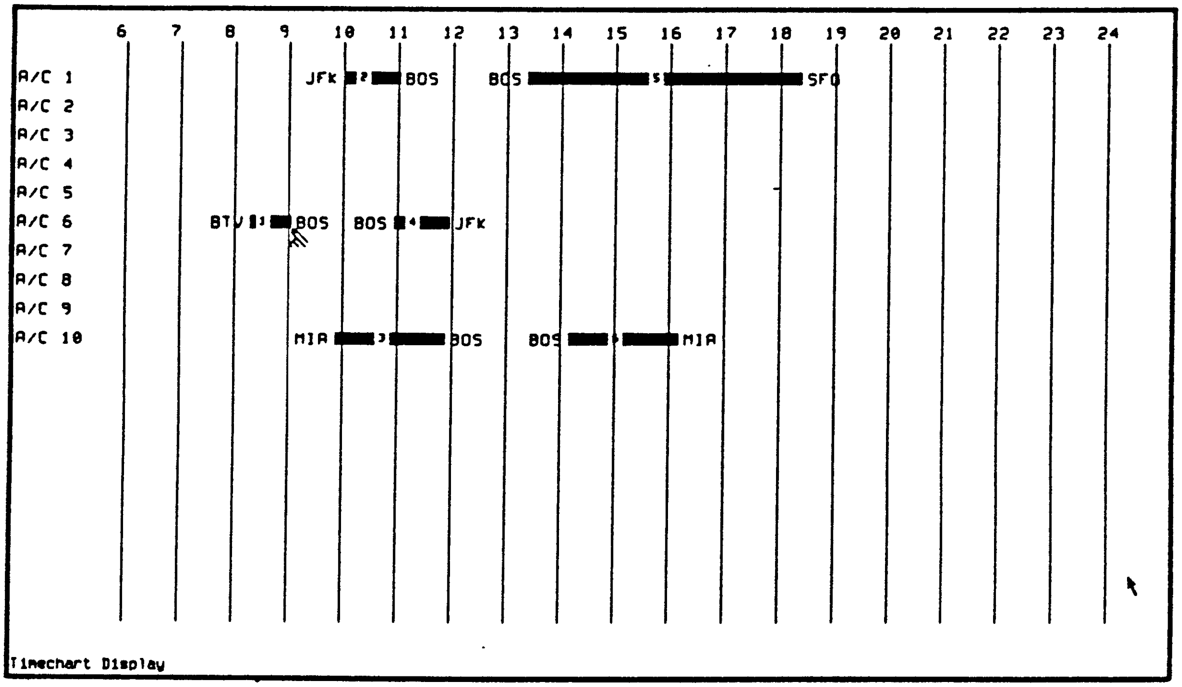
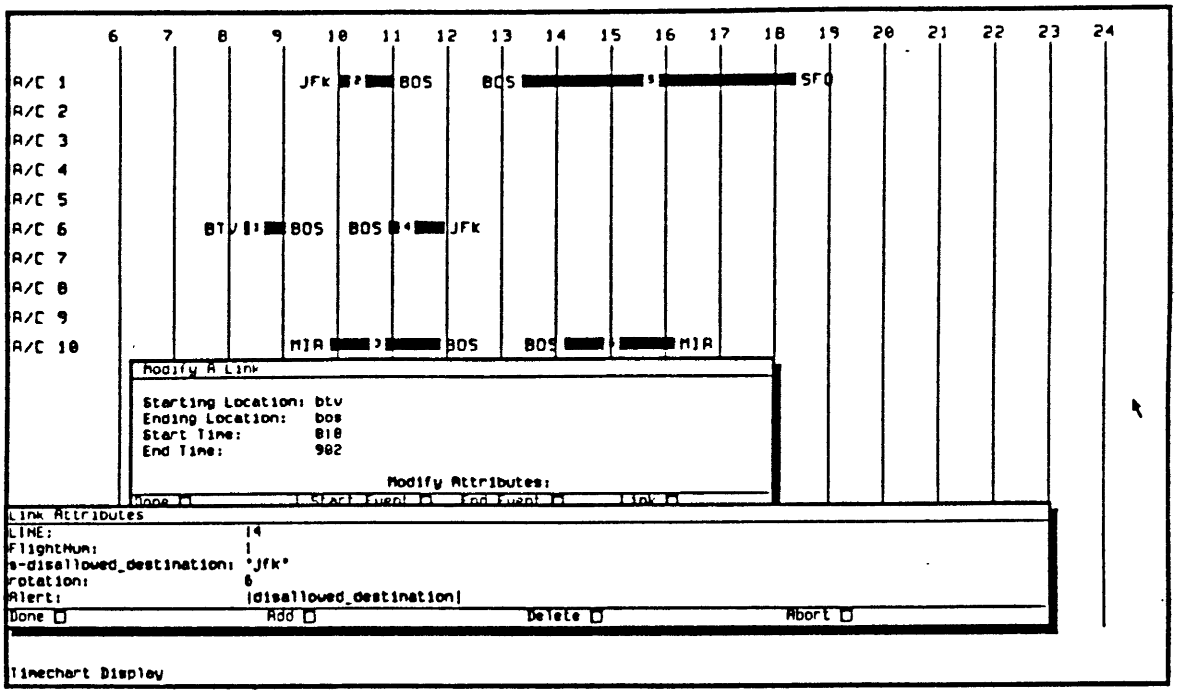


Figure 4.10: An Alert Is Indicated On Flight #1



```
Starting Location: btv
Ending Location:   bos
Start Time:        818
End Time:          902

                    Modify Attributes:
```

```
LINE:                         14
FlightNum:                    1
s-disallowed_destination:     "jfk"
rotation:                     6
Alert:                        |disallowed_destination|
```

Figure 4.11: Attribute List For Flight #1 Contains An Alert

at the appropriate attribute list. In this example, the attribute list for flight #1 is brought up, and shown in figure 4.11.

Chapter 5

Incorporation Of Other Scheduling Algorithms

5.1 Reducta: A Fleet Reduction Algorithm

The type of algorithm described so far involves assigning resources, in this case aircraft, to links which represent requested flight segments. There, the position of each link in the schedule is fixed, and the only thing that the automated system does is to add elements to the attribute list. We now turn to a type of algorithm in which the schedule is not fixed and is the goal of the algorithm to optimize the schedule, by shifting flight segments in the schedule to improve aircraft utilization.

An automated system for optimizing fleet utilization has been interfaced with the interactive scheduling environment; this system is based on a series of fleet size reduction algorithms that were developed at the Flight Transportation Laboratory at M.I.T [4] The original program, called REDUCTA, was written in Fortran and operated on batch data. Through the use of the interactive scheduling environment with variable length attribute lists, the implementation of the fleet size reduction algorithm allows a great deal more flexibility in setting constraints on individual flight segments and the graphics displays allows the user to clearly see what modifications have been made in the schedule to improve vehicle usage. The algorithm is written in Prolog; as this language is more commonly used for symbolic and artificial intelligence types of applications, it is interesting to see the issues involved with using Prolog for developing a more numerical heuristic algorithm.

The fleet reduction problem takes as input a set of service requests, and for each, a range of departure times. The goal is to minimize the number of aircraft required to serve the set of requests. As each service consists of an origin location and a destination location, this set of service requests translates into a set of arrival and departure events at each station. These arrival and departure lists are shown graphically in the Station Display. If, for each arrival event, there is a certain minimum ground time after which the aircraft used for that flight can be used for a subsequent departure, we can compute a "ready time" for each arrival event, which is the actual time that the vehicle arrived adjusted forward by the minimum ground time. Likewise, we can create a set of ready times for departure events. We can define NA to be the number of aircraft on the ground after each event, with NA increasing by one after each arrival and NA decreasing by one after each departure. The number of vehicles which "overnight" at each station can be called NAC.

Figure 5.1 shows a sequence of arrival and departure events at station BOS, and the two tabular columns to its right show the value of NA after each event with two possible values of NAC. The first goal is to find the minimal value for NAC such that there are always zero or more aircraft at the station. Once this is done, the algorithm finds those events at which the NA value is zero,

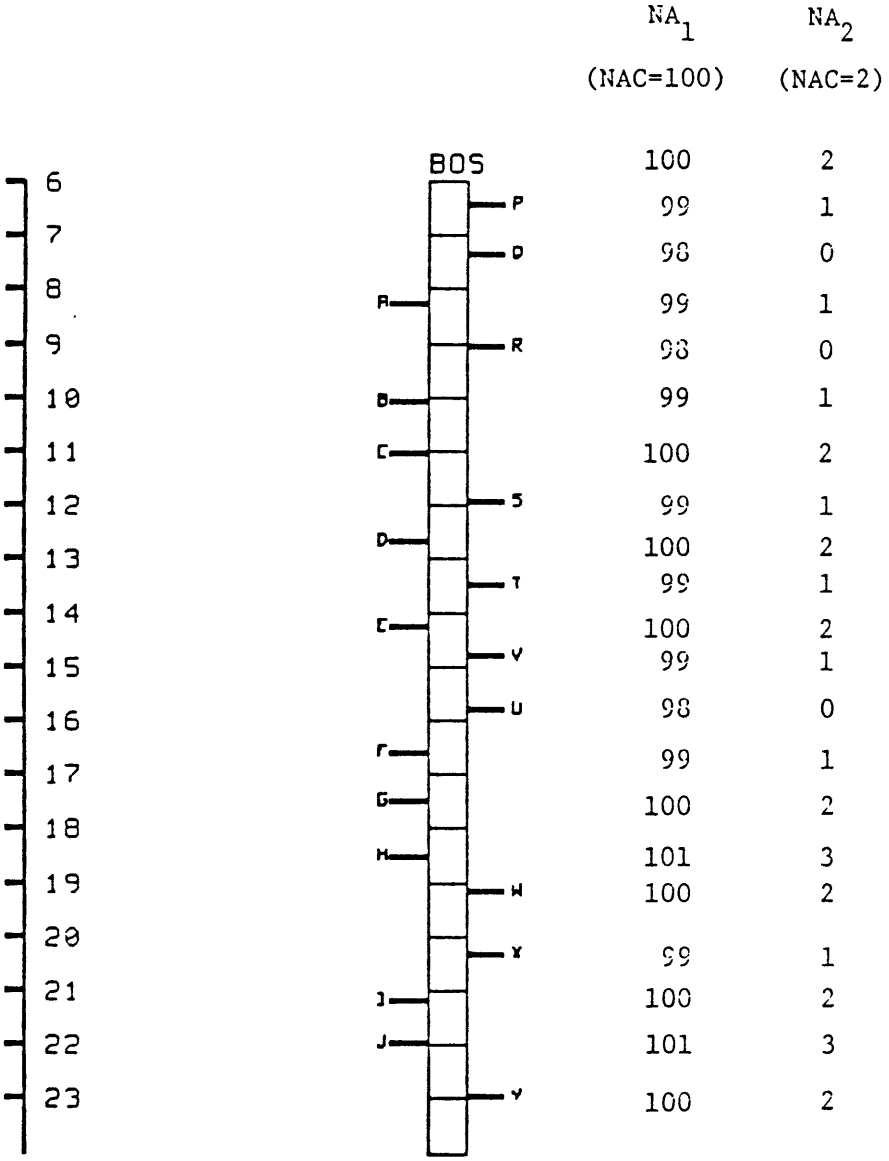


Figure 5.1: Aircraft Available Following Events At BOS

and attempts to increase those NA values by interchanging arrival events with departure events. If all of the "zero events" are eliminated, the value of NAC for the station is decreased, and the number of aircraft required by the schedule as a whole is therefore reduced. This process continues, iterating through each station, until all possible zero events are eliminated. Much of the processing involved in the execution of this algorithm involves verifying that shifts can be made in the schedule. Each arrival and departure event is linked to some other event at another station, and thus before a given event is shifted, it must be verified that the number of required aircraft will be not be increased at another station.

The Prolog language implementation of the fleet reduction algorithm within the interactive scheduling environment makes use of a set of dynamic attributes in addition to the time and location information inherent to a link. These attributes define the time window and ground time associated with the service requests. Associated with the Reducta code is a rule base which define default time windows and ground times; if there are no specific values for these attributes associated with the links, the defaults are used.

The following link is an example of a flight which can move up or down by 20 minutes, and which must remain on the ground 45 minutes after it lands.

```
link(event(1000,d,bos,[]),
     event(1100,a,jfk,[c-min_gtime,45]),
     [[timewindow_up,20],[timewindow_down,20]]).
```

Two two link attribute *timewindow_up* and *timewindow_down* indicate the amount of time that the link can shift up and down, and the arrival event attribute *c_min_gtime* contains the amount of time that the aircraft must remain on the ground after landing.

The Prolog program *reducta* takes a list of links and processes through several steps to produce a new chain with the updated departure and arrival times. The first step is the initialization step which converts the links as defined in the interactive environment into a set of event lists which can be used by the Reducta algorithm. Next, the set of event lists are evaluated to discover how many aircraft are required at each station, and subsequently, how many are required in the schedule as a whole. When the number of required aircraft is determined, the *iterate_removing_zeros* step is executed, attempting, for each station, to remove the "zero events" in order to reduce the number of aircraft at that station. Following this step is a re-computation of the number of required aircraft, and if the number of aircraft has been reduced in the previous step, the "zero-removal" repeats.

The first major issue that evolved in the implementation of this algorithm involved the choice of a data structure in which to store the ordered event lists. Whereas in a resource assignment problem, the times and locations of events are fixed and therefore the order in which they are stored in the data structure can be a constant, in the Reducta situation the order of events will continually be changing. Therefore, to store events in a linear list, as in the aircraft rotation assignment algorithm, would be inefficient because the list would have to be reconstructed rather frequently. Random access arrays are not implemented in standard Prolog because individual pieces of information cannot be changed once they are instantiated.

An efficient data structure for representing ordered event lists in which the order of events might be frequently changed is a fully balanced binary tree. This binary tree can simulate an array; each node is labeled by an index number. In the Reducta algorithm, any change of the order of events involves swapping two adjacent events in the event list (i.e., sliding an arrival before a departure). In a balanced binary tree, these two event will likely be in the same sub-tree; in a large number of cases they will be be children of the same parent node. To create a new tree in which the order of these two events has been swapped, we need only to replace the sub-tree which contained these two events. The pointer to the rest of the original tree can be carried over to the new tree.

Once a set of event lists has been created in which the number of aircraft available after each event has been determined, the *remove_station_zeros* step attempts, for each station, to shift the appropriate events to remove the "zero departures". The procedure which does this represents the heart of this algorithm, and is outlined here.

The first step taken by *remove_station_zeros* is to determine, for each event occuring at the given station, the "shift limits" for that event. If the event is an arrival, the amount by which the link can be shifted upwards is constrained by the top of its time window, with the provision that the the link's departure event (at some other station) cannot be shifted before some arrival event so as to create a zero departure. In other words, no shift can be made at one station which increases the number of aircraft required at another station. The amount by which a departure can be shifted downwards is limited by a similar criteria. The amount by which a link can be shifted downwards is constrained by the end of its time window, with the provision that the link's arrival event at a downstream station is not shifted past some other departure event to create a zero departure.

Once the "shift limits" have been determined, the next program iterates through the zero departures, attempting to remove each one. The ways in which a zero departure can be removed including the following types of shifts:

1. Shift a later arrival ahead of the zero departure
2. Shift the zero departure past a subsequent arrival
3. Shift a later arrival up as much as possible and shift the zero departure past it.

These steps are attempted in the order shown; it is preferable to move flights earlier in the schedule than later. This allows a greater amount of flexibility in using aircraft later in the day. When a particular shift is selected for a link, the link is bubbled up or down (as appropriate) through the ordered event lists at the current station and at the other affected station. While the event is moving through the list, the values for NA (number of aircraft required after each event) must continually be updated so they are accurate for the next iteration.

These three types of schedule modifications are illustrated in the following examples. In each of them, we are examining station JFK in an attempt to reduce NAC, the number of aircraft required in the morning at that station, from one to zero. The time window for each link is set at thirty minutes up or down, and the minimum ground time is set at twenty minutes.

In figure 5.2, the arrival at JFK from BOS occurs just after the departure to Miami, and thus two aircraft are required for these two flights. However, because there are no limitations on shifts other than the thirty-minute time windows, the

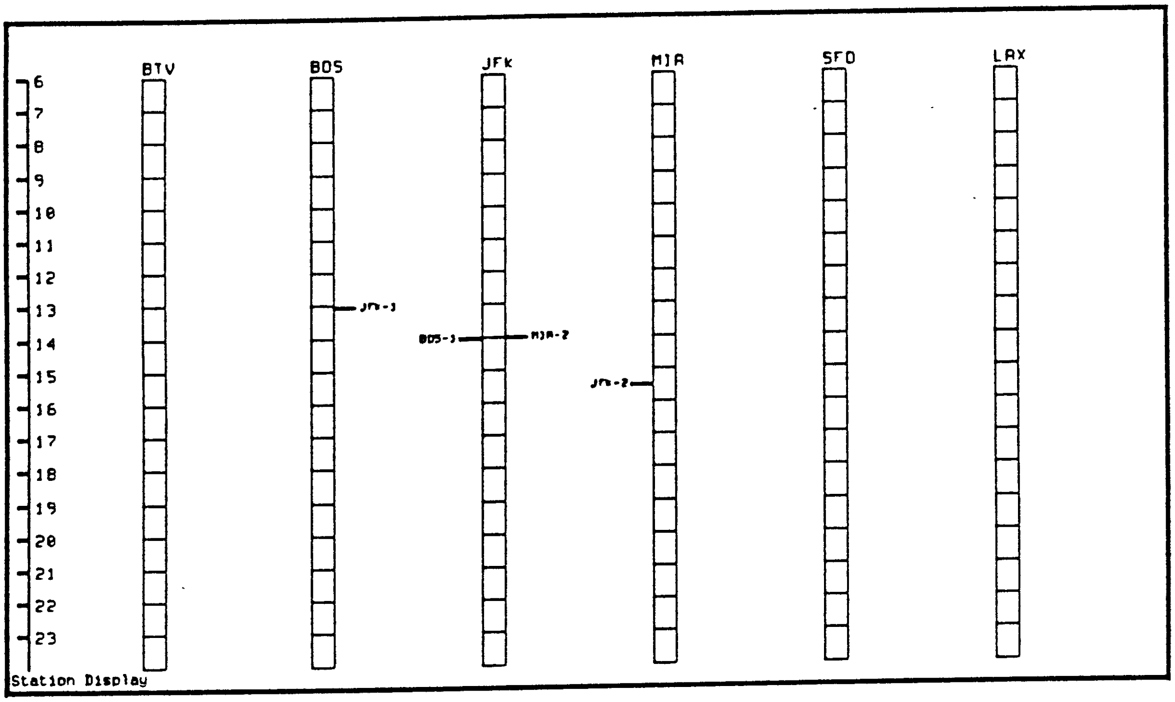


Figure 5.2: Two Aircraft Are Required For Events At JFK

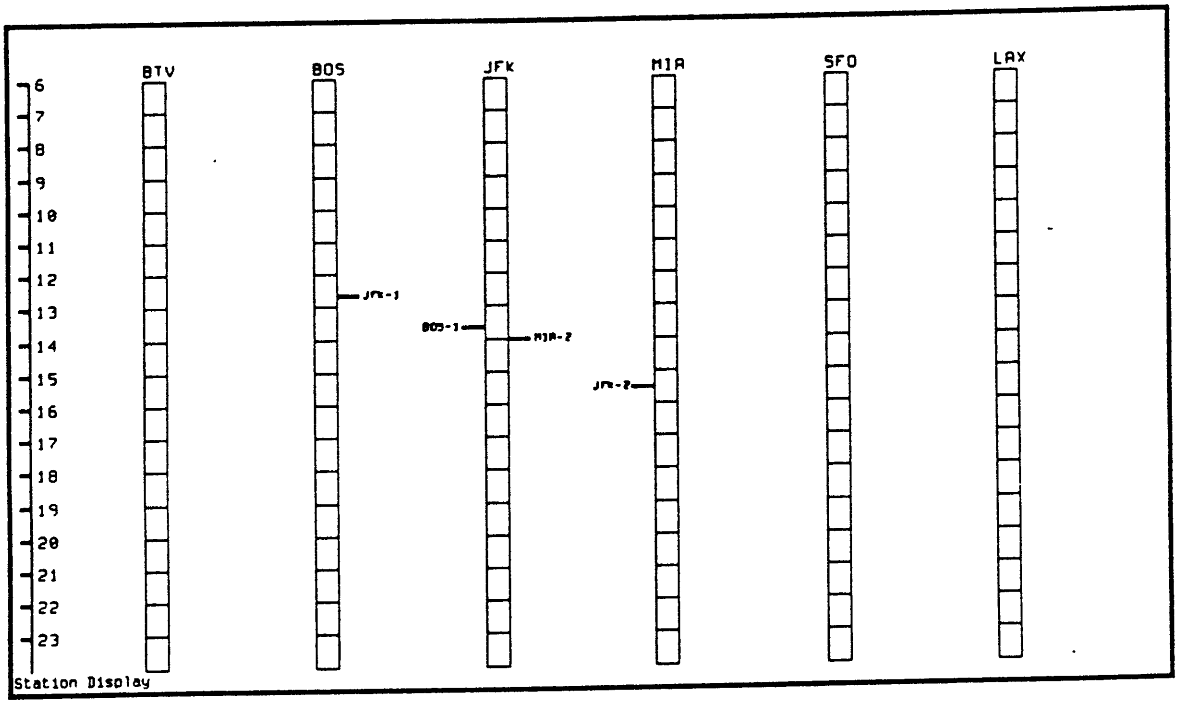


Figure 5.3: Shift In Arrival From BOS Reduces Aircraft Requirement

arrival from Boston can be shifted to twenty minutes before the departure for Miami, and as shown in figure 5.3, there is one less aircraft required at station JFK. In Figure 5.4, a similar situation occurs, except that the flight from Boston is limited in its upward shift because, at Boston, the departure is constrained to be twenty minutes after the arrival from BTV. Thus, shift type 2 is executed, shifting the departure for MIA twenty minutes past the arrival from Boston, to achieve the timetable shown in figure 5.5.

A third situation is shown in figure 5.6 , in which the arrival from Boston

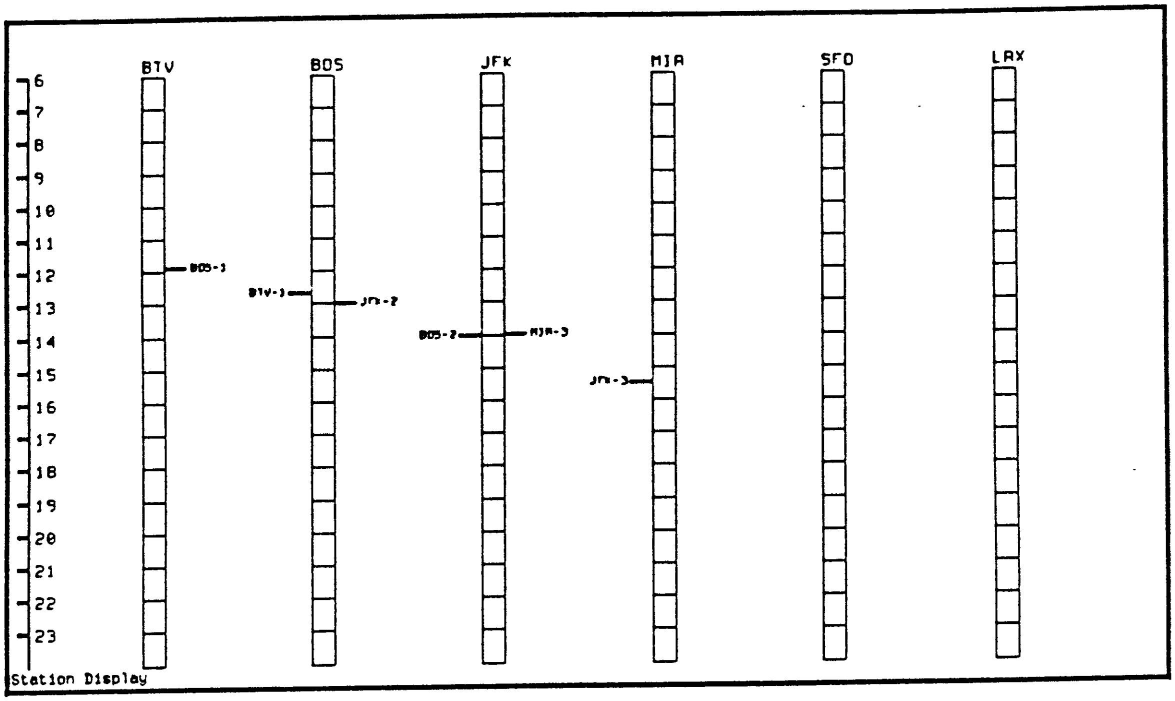


Figure 5.4: Arrival At JFK From BOS Is Constrained In Upward Shift

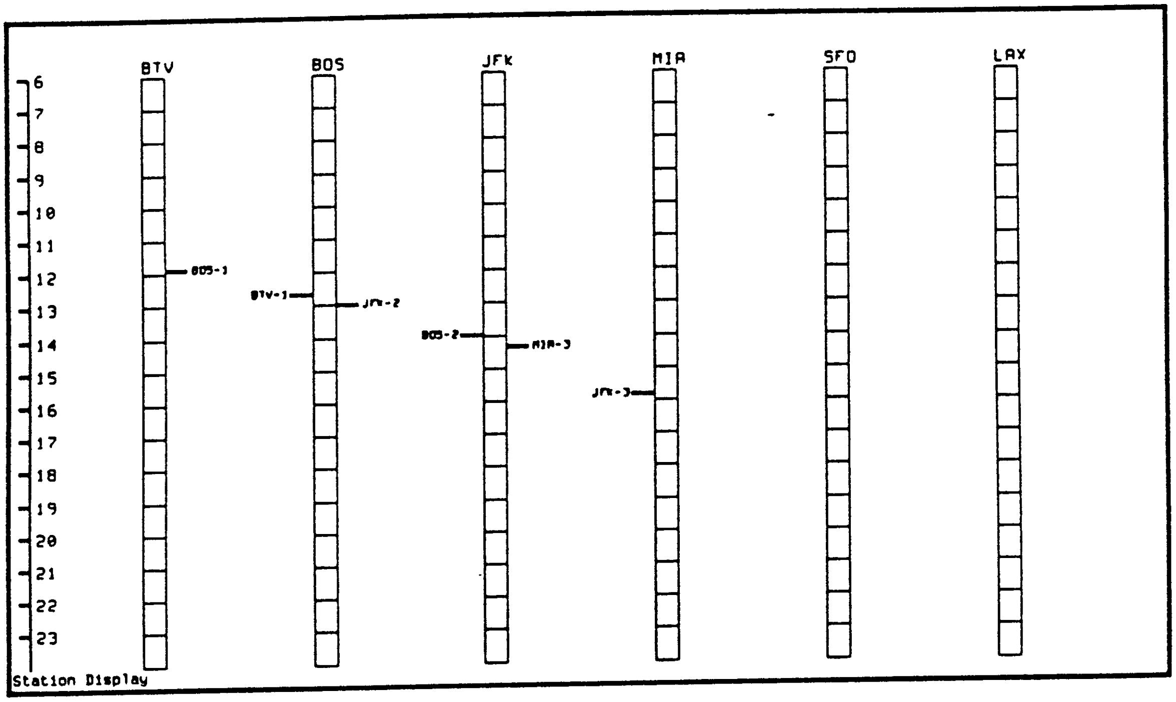


Figure 5.5: Shift In Departure To MIA Reduces Aircraft Requirement

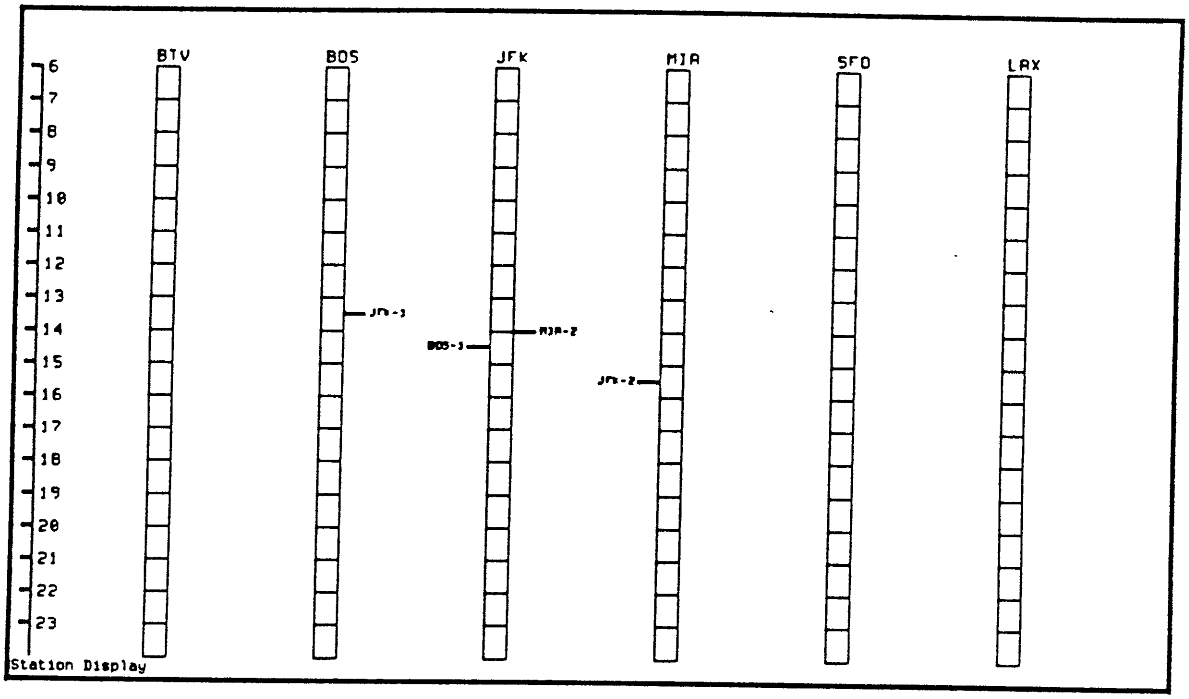


Figure 5.6: A Single Shift Will Not Reduce Required Aircraft

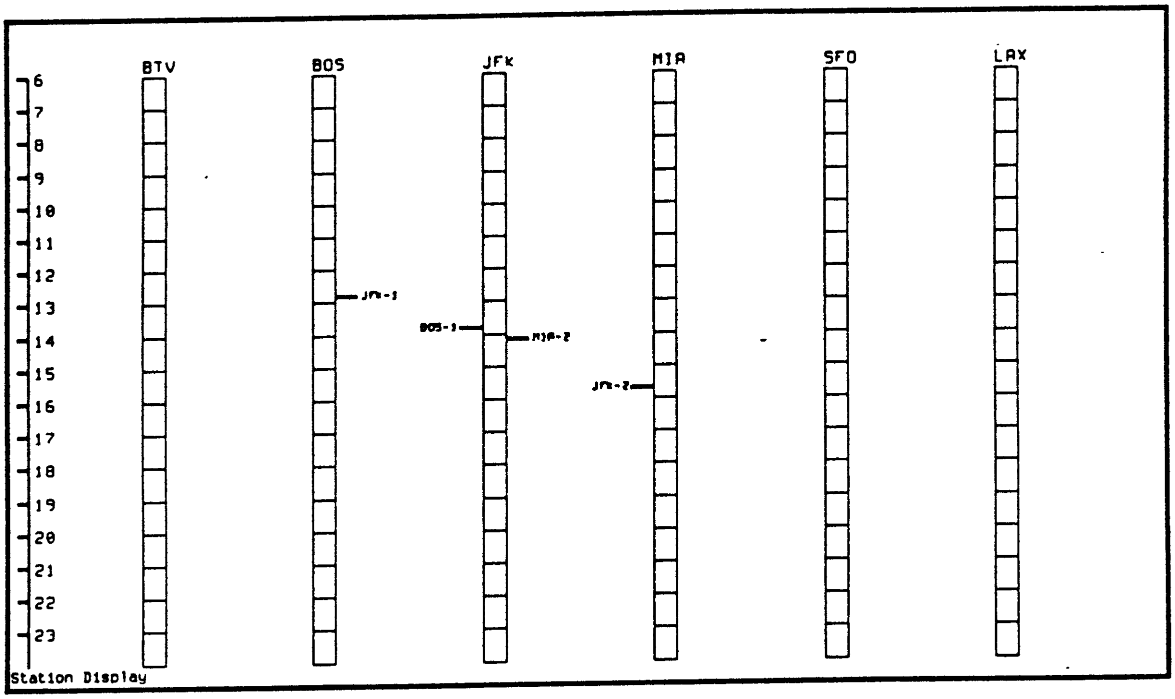


Figure 5.7: Arrival and Departure Events At JFK are Shifted

initially is one half-hour past the departure for Miami. In order for the arrival to end up twenty minutes before the departure, both the arrival and departure must be shifted. The arrival is shifted up as far is it can, to the top of its time window, and the departure is shifted down the remaining required amount (figure 5.7).

When one zero is eliminated, the procedure of removing zeros at the station continues from the next zero departure. If all zero departures in a station are eliminated, the number of aircraft required at the station decreases by one. If no more zeros can be eliminated at the station, the procedure backtracks to the state prior to commencement of *remove_station_zeros* for that station. The algorithm then attempts the same process at another station.

When all stations have been examined, the iteration is complete. If the procedure has been successful in reducing the number of aircraft required in the schedule as a whole, another iteration is attempted. This process continues until no improvements in fleet utilization can be made.

The time window, as used in the Reducta algorithm, could reflect the period of time during the day in which a particular demand exists for the flight. During peak hours, the time window for a particular demand level might be small, while during other hours time windows might be larger. For this reason, time windows can be dynamically altered for individual links. There is a default time window size which can be set in the rule base, but if the attributes *timewindow_up* or *timewindow_down* are set for an individual flight, this will override the default. A similar situation exists for minimum ground times. The minimum ground times can be set in the same way as they are set in the implementation of the fleet assignment algorithm.

5.1.1 Example Of The Reducta Process

As example of the Reducta process, consider the schedule shown in figure 5.8. In its current form, twelve aircraft are required to "overnight" at the six stations shown when there is a minimum ground time requirement of twenty minutes before departure. When we run the Reducta algorithm, one iteration through each of the six stations makes appropriate shifts to reduce the aircraft requirement to 8 aircraft, as shown in figure 5.9. The resulting schedule is much

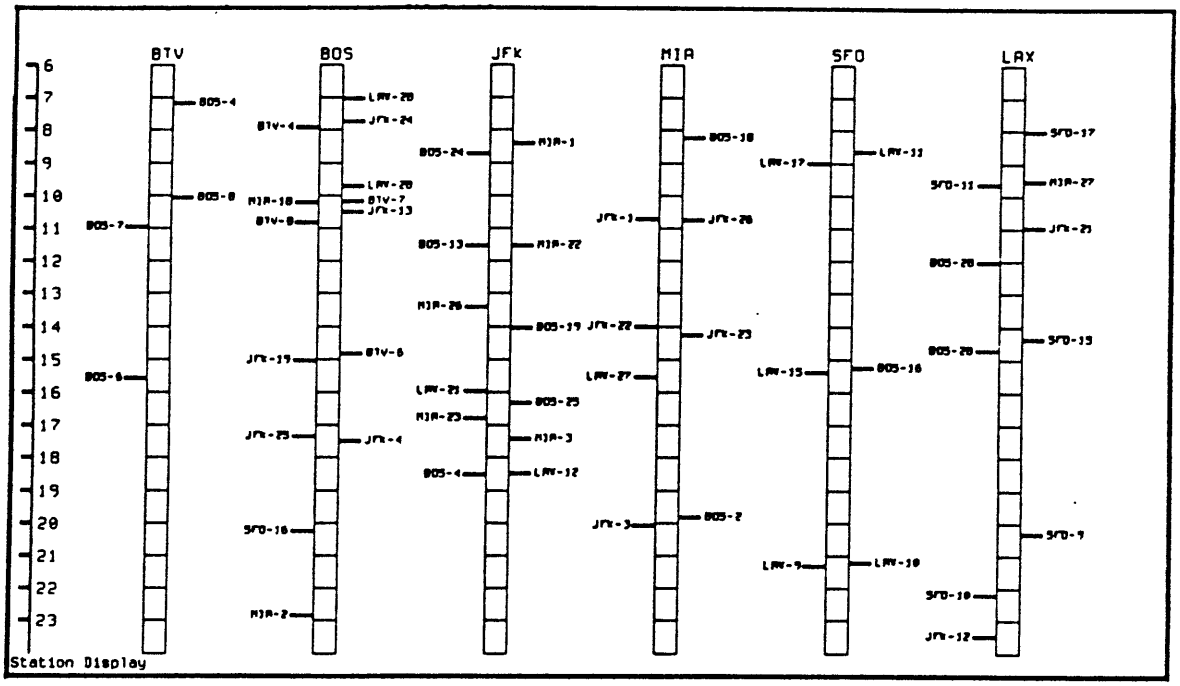
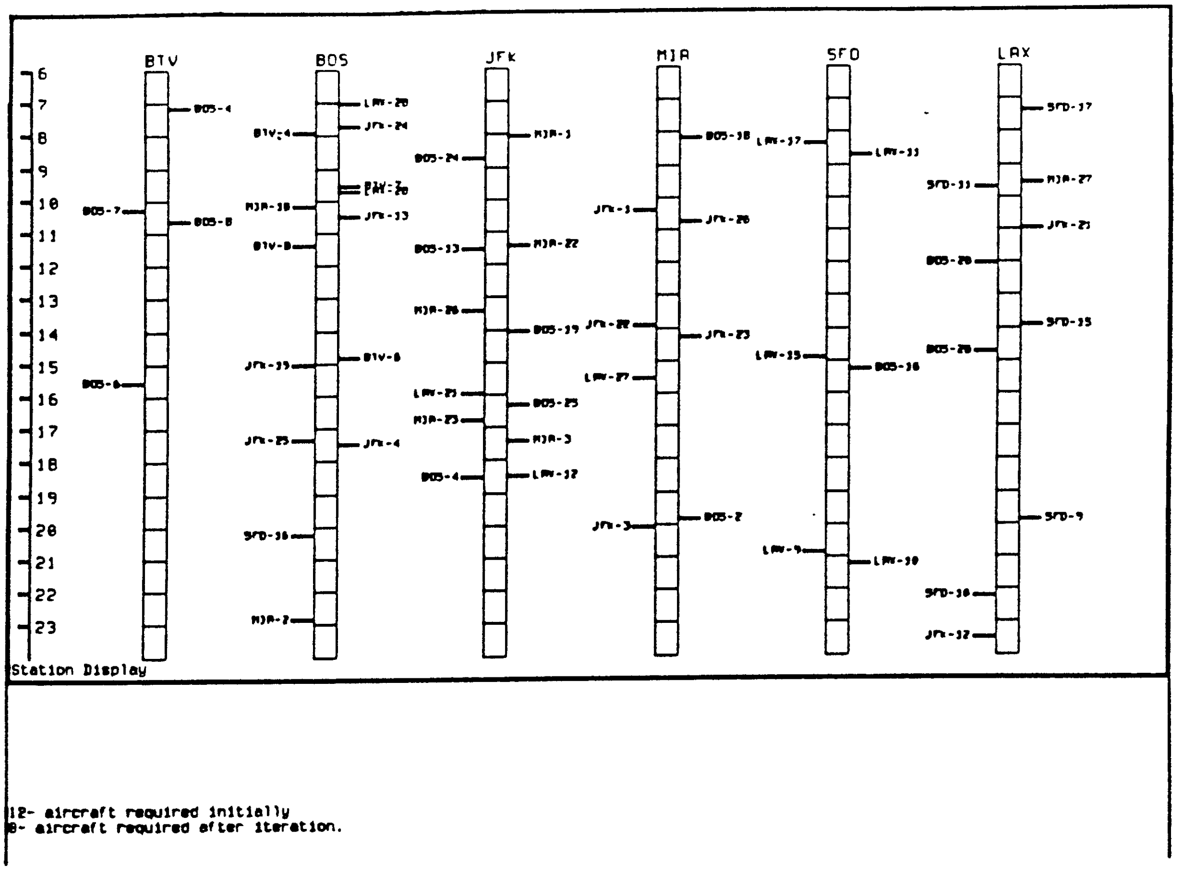


Figure 5.8: Schedule Requiring Twelve Aircraft To Be Available



12- aircraft required initially
8- aircraft required after iteration.

Figure 5.9: Reducta Reduces Aircraft Requirement To Eight

closer to a situation of alternating arrivals and departures, which is the sequence which would optimize fleet utilization.

5.2 Gate Scheduling

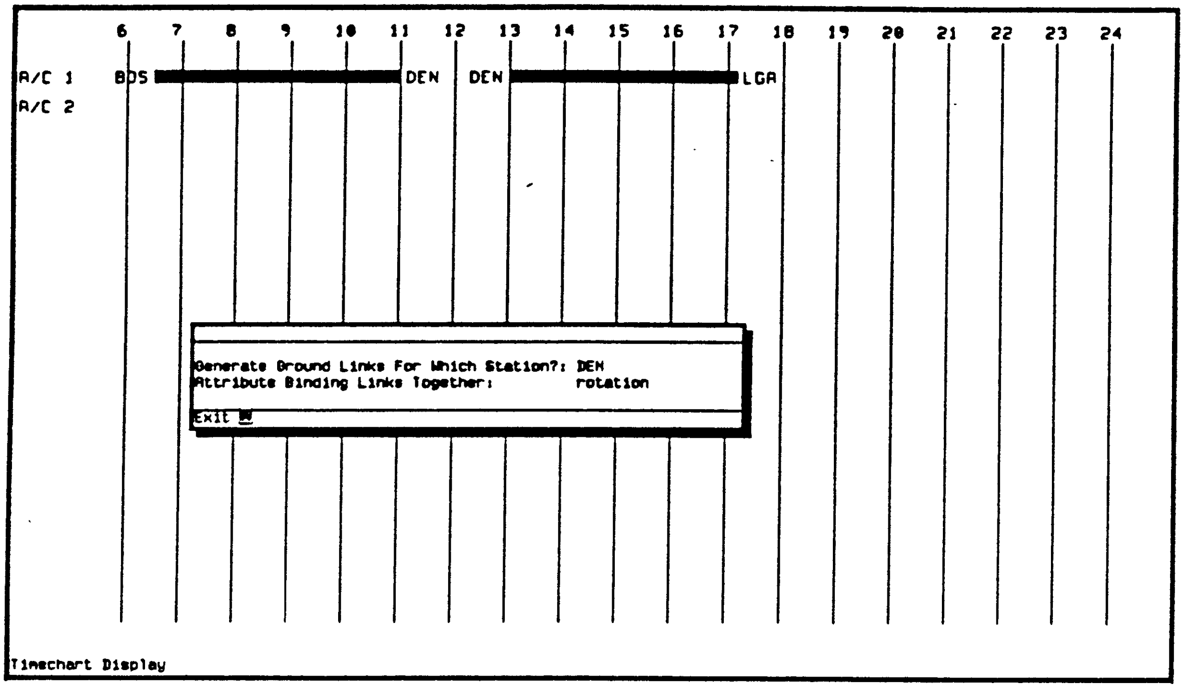
We now turn to the problem of assigning airport gates. The input to this problem consists of a set of links which represent the time that a vehicle spends on the ground, and our goal is to assign a gate for each of these "ground links". As with our previous examples, we are assigning resources to links which consist of two events and a set of attributes. However, while in the flight scheduling examples the link consisted of a departure event followed by an arrival event, in the gate scheduling example, a link consists of an arrival event following by a departure event. Both of these events will occur at the same location.

The first step in the gate assignment process is to prepare the input to the problem – the set of ground links. If there are two adjacent flight links which will be assigned to the same aircraft, for in which the first link arrives at a given airport and the second link departs from that same airport, we can use the arrival event of the first link and the departure event of the second link to form a new link, called a ground link.

As an example, let us consider the two flight links displayed in figure 5.10. The first flight arrives in station DEN at 11:00 A.M. The second flight departs from DEN two hours later, at 1:00 P.M. The time between 11:00 and 1:00 is our ground time; during this time the vehicle will be occupying an aircraft gate. In order to schedule the assignment of this gate resource, we create a ground link consisting of the first flight's arrival event and the second flight's departure event. To create this ground link, we specify the name of the attribute which "binds the links together"; in other words, the resource that the two links have in common such that they share the same ground link. In this case that attribute is *rotation*, referring to the aircraft rotation number to which each link is assigned. The ground link created from the above two links is shown in figure 5.11.

To illustrate the airport gate scheduling problem, consider the set of links shown in figure 5.12. Each line consists of a flight itinerary; on each itinerary there is a flight arriving and departing from station SLC, which is the "hub". To examine the gate activity at station SLC, we convert each pair of links arriving at and departing from SLC into ground links, producing the thirteen ground links shown in figure 5.13.

Given a set of gates and a set of ground links, there could be a number



```
Generate Ground Links For Which Station?: DEN
Attribute Binding Links Together:         rotation
Exit
```

Figure 5.10: Two Flight Links Which Are Assigned To Vehicle #1

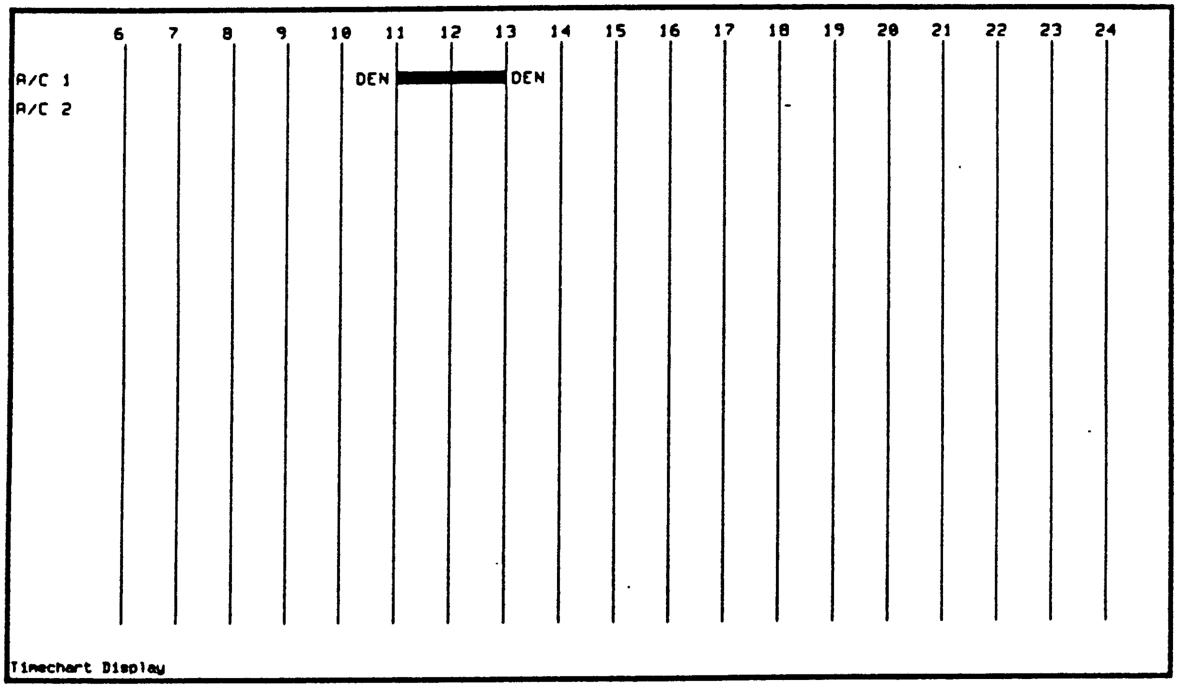
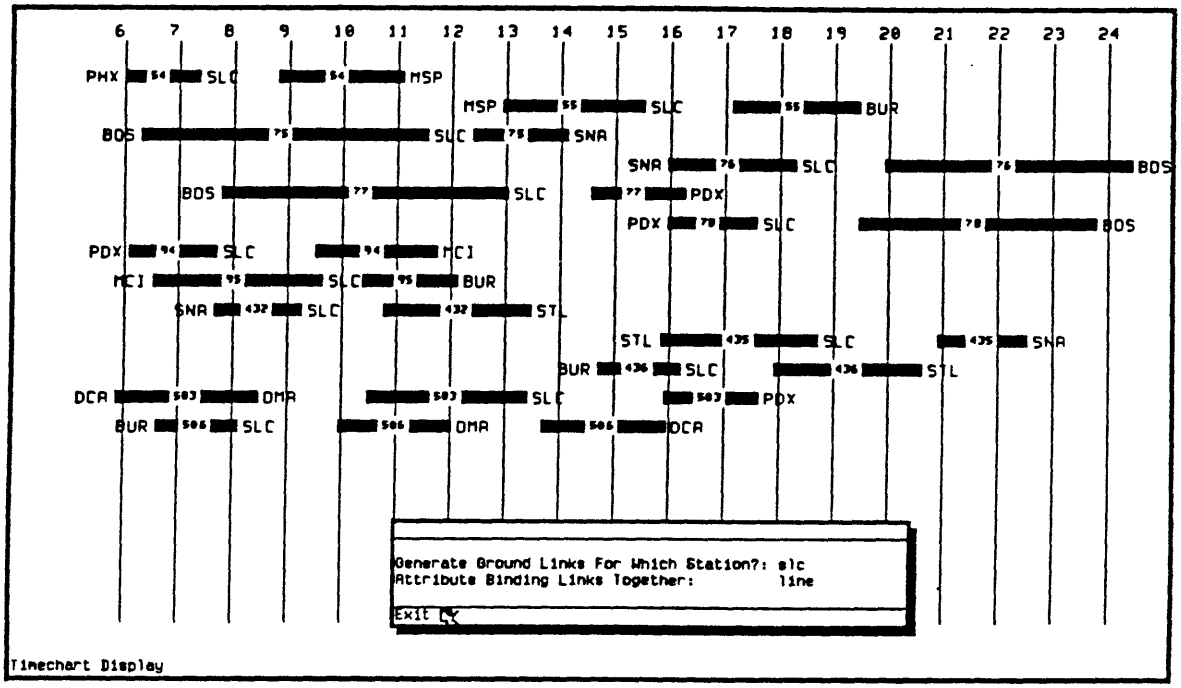


Figure 5.11: Ground Link Created From Previous Two Flight Links



Generate Ground Links For Which Station?: slc
Attribute Binding Links Together: line

Figure 5.12: Flights Centered Around A Hub At SLC

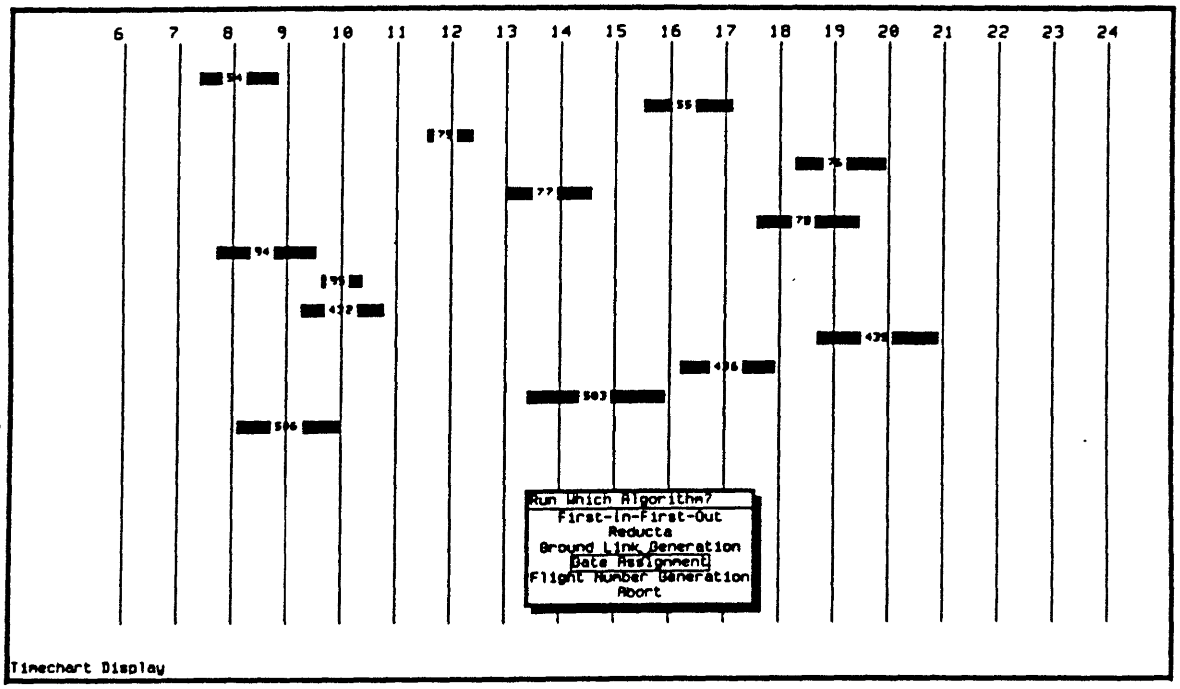


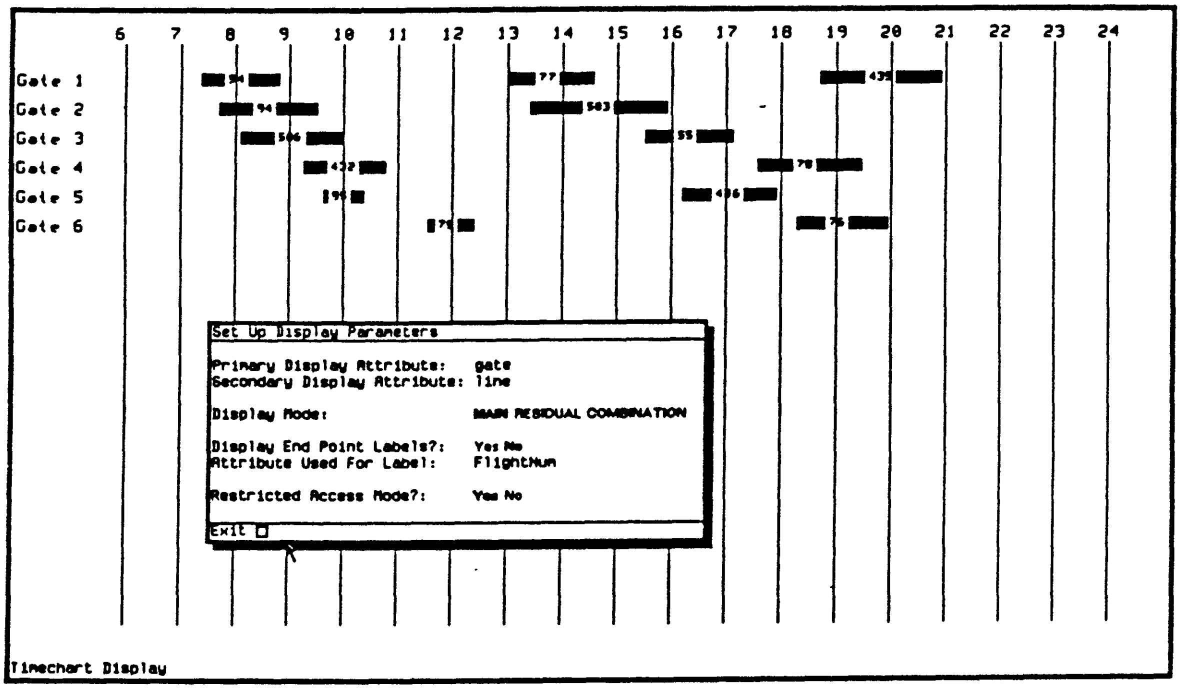
Figure 5.13: Ground Links At Station SLC

of objectives when assigning the gates to the links. One objective might be to minimize the amount of times that gate assignments need to be rescheduled due to delays; in order to achieve this objective we might wish to maximize the amount of time between any two usages of a gate. Another objective could be to minimize the number of gates that are used; this would provide a quite different schedule than that produced by the previous case.

The gate scheduling algorithm implemented within this system is based on the rule stating that each incoming aircraft is assigned to the gate which has been available for the longest amount of time. This rule serves to cut down on the disruption in the schedule due to delays; it also serves to maximize the usage of gates.

As an example of the gate assignment process, we consider the assignment of the thirteen ground links at station SLC to a set of six available gates. Using the stated objective, the algorithm produces the assignments shown in figure 5.14. Each ground link now contains a link attribute called *gate*, which indicates the gate to which the ground link has been assigned.

It is likely that there will be other factors which would serve to alter the flow of the gate scheduling process. First, the vehicle type might affect the assignment of a vehicle to a gate – for example, certain gates might only be able to handle narrow body aircraft. Second, there might be a desire to assign certain ground links to gates which are near the gates which handle certain other ground links so as to ease connections. Through the use of data contained the the ground link attribute lists, this and other information can be incorporated into the gate scheduling process. Any information that is relevant to the gate assignment process can be passed along from the flight scheduling process; the



Set Up Display Parameters

```
Primary Display Attribute:   gate
Secondary Display Attribute: line

Display Mode:                MAIN RESIDUAL COMBINATION

Display End Point Labels?:   Yes No
Attribute Used For Label:    FlightNum

Restricted Access Mode?:     Yes No
```

Exit □

Figure 5.14: Ground Links Assigned To Gates At Station SLC

attribute list of a ground link includes all information contained in the attribute lists of the two component flight links.

5.3 Other Scheduling Problems

The implementations of the gate scheduling process along with the fleet reduction vehicle assignment algorithms serve to demonstrate the flexibility of the interactive scheduling environment for dealing with a wide variety of transportation scheduling problems. Two of these algorithms – aircraft rotation scheduling and gate scheduling – fall into the category of resource assignment. That is, given a set of links and given a set of available resources, we wish to assign the resources to the links.

The resource assignment problem comes up in many areas of transportation scheduling, particularly in aircraft scheduling. The generalized interactive scheduling environment, which allows for a dynamic set of attributes (for example, resources) to be associated with a link, is well suited to many of these resource assignment problems. One very important resource allocation problem is the crew scheduling problem. Given a set of links that have been assigned to aircraft, we wish to assign crews to the links while following a detailed set of constraints. There are limits on such factors as the number of hours that a crew member can fly in a given day or a given month, and these time constraints are usually much more limiting than the constraints on the vehicle assignment process.

The interactive scheduling display can be used to display crew assignments in much the same way in which it used to show aircraft assignments and gate assignments. A crucial aspect of the generalized scheduling system is that all resource allocation information pertaining to a link can be stored along with the link, but to solve any individual scheduling problem, only the pieces of information required for that individual problem need to be used in the algorithm and plotted on the display.

The environment could also be used for real-time operations, such as to handle rescheduling of flights if aircraft suddenly became unavailable. In this case, we would be dealing with more than just the "scheduled" time for each flight; we would also be dealing with estimated and actual times of departure and arrival. The link attributes, in this case, could indicate the amount of estimated and actual delays with reference to the scheduled departure and arrival times for the link; in addition they could include other information about what actually occured on a flight, such as the number of passengers on board.

5.4 Conclusions

The interactive scheduling environment presented herein attempts to address several problems facing transportation schedulers. The prototype that has been developed illustrates a number of points relating to the role of computer graphics and artificial intelligence in the field of scheduling.

We first address the fact that schedulers are often attempting to solve many different scheduling problems which – while they involve different resources, time frames, constraints and objectives – have a great deal in common in their structure. The generalized system for representing schedules shows that many of these different problems can be solved within the same framework and using the same user interfaces. For example, the same display that is used to show the assignment of aircraft rotations to flights is used to show the assignment of arriving flights to airport gates. The underlying data structure allows the developer of optimization algorithms for these and other problems to quickly adapt the system to whichever problem is being addressed. Thus, the overhead of preparing input and analyzing output for many different pieces of software to solve several related scheduling problems can be significantly reduced.

We then address the fact that many of constraints and objectives of schedule development are hard to quantify. A system of exception represention and handling is presented; this system allows a deterministic scheduling algorithm to be enhanced with additional rules which improve the solution. The framework of incorporating a dynamic rule base with a hard-wired algorithm allows for a great deal of flexibility in the specification and adjustment of those rules which are important to the scheduling process. This system allows scheduling software to take advantage of efficient algorithms from within the fields of network analysis and operations research while not giving up the ability to take advantage of hard-to-quantify information. Because the exception-handling rules are only evaluated in the small number of cases in which they are referenced, the process only loses a minimal amount of efficiency. If certain "exceptions" appear in a large number of cases, the process does become less efficient and it becomes worthwhile to incorporate the exception into the "basic" algorithm.

A flexible rule base, such as that developed in Prolog for the exception handlers presented herein, can be a very powerful tool in developing expert systems to solve scheduling problems. The system that is presented has been developed in the Lisp and Prolog languages. These languages are especially well suited to problems which are not rigidly structured and make use of a flexible rule base. It is possible that future expert systems might move further away from traditional scheduling optimization algorithms than the Dynamic Exception Handling approach developed here. The system herein is not driven by the knowledge in the rule base, it is driven by a traditional deterministic algorithm and enhanced – in certain specific cases – by the dynamic exception rules. Developing a large rule base which could simulate the complete decision-making process of an expert scheduler would be a lofty, but extremely worthwhile, goal.

Regardless of structure of the computerized decision making process, it is important to have an environment which allows a human to easily develop and manipulate schedules. The generalized system of using attributes of pieces of the schedule to determine the way the schedule is displayed makes the system adaptable to a number of different situations. The mouse-driven graphics systems presented incorporates many features which make this process easier. The system of pop-up windows and menus which allow the user to quickly pinpoint and alter individual pieces of the schedule is an important component of this flexible system; computer operating systems which include dedicated graphic window manipulation systems are becoming prevalent and therefore it is becoming possible to develop environments such as this one on many different types of computers.

Bibliography

[1] Clocksin, W.F. and C.S. Mellish. *Programming in Prolog.* New York: Springer-Verlag, 1981.

[2] Deckwitz, Thomas A. "Interactive Dynamic Aircraft Scheduling." MIT Flight Transportation Laboratory Report R84-5, June 1984.

[3] Etschmaier, Maximilian M. and Dennis F.X. Mathaisel. "Aircraft Scheduling: The State Of The Art." Proc XXIV. AGIFORS Symposium, Strasbourg, France, 1984.

[4] Simpson, Robert W. "Computerized Schedule Construction for a VTOL Airbus Transportation System." *J. Aircraft*, Vol. 5, No. 3, May-June 1968, pp. 299-305.

[5] Simpson, Robert W. and Dennis F.X. Mathaisel "Automation of Airlift Scheduling for the Upgraded Command and Control System of Military Airlift Command." MIT Flight Transportation Laboratory, June 1984.

[6] Van Cotthem, Jan. "Interactive Dynamic Aircraft Scheduling and Fleet Routing with the Out-Of-Kilter Algorithm." MIT Flight Transportation Laboratory Report R84-5, June 1984.